

ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ В .NET

Марк Симан



MANNING



PITPER

Аннотация:

Внедрение зависимостей позволяет уменьшить сильное связывание между программными компонентами. Вместо жесткого кодирования зависимостей (например, драйвера какой-либо базы данных), внедряется список сервисов, в которых может нуждаться компонент. После этого сервисы подключаются третьей стороной. Такой подход обеспечивает лучшее управление будущими изменениями и решение проблем в разрабатываемом программном обеспечении. Данная книга рассказывает о внедрении зависимостей и является практическим руководством по их применению в приложениях .NET. Издание содержит основные шаблоны внедрения зависимостей, написанные на «чистом» C#. Кроме того, рассмотрены способы интеграции внедрений зависимостей со стандартными технологиями Microsoft, такими как ASP.NET MVC, а также примеры применения фреймворков StructureMap, Castle Windsor и Unity.

Книга "Внедрение зависимостей в .NET" демонстрирует основные DI паттерны на обычном языке C#, поэтому вы в полной мере осознаете, как работает механизм внедрения зависимостей, кроме того она охватывает интеграцию с такими стандартными технологиями компании Microsoft, как ASP.NET MVC, и учит вас, как использовать такие DI фреймворки, как Structure Map, Castle Windsor и Unity.

Внедрение зависимостей – это прекрасный способ уменьшить сильное связывание между компонентами программного обеспечения. Вместо жесткого кодирования зависимостей, например, указания драйвера базы данных, вы внедряете перечень сервисов, которые могут понадобиться компоненту. Затем сервисы соединяются при помощи стороннего компонента. Эта методика позволяет вам лучшим образом справляться с будущими изменениями и другими возникающими в вашем программном обеспечении сложностями.

Книга "Внедрение зависимостей в .NET" знакомит вас с механизмом внедрения зависимостей и предоставляет практическое руководство по применению этого механизма .NET приложениях. Книга демонстрирует основные паттерны на обычном языке C#, поэтому вы в полной мере поймете, как работает механизм внедрения зависимостей. Далее вы научитесь интегрировать DI с такими стандартными технологиями компании Microsoft, как ASP.NET MVC, и использовать такие DI фреймворки, как Structure Map, Castle Windsor и Unity. К концу книги вы будете свободно применять эту мощную методику в рамках повседневной разработки .NET приложений.

Эта книга написана для разработчиков на языке C#. Не требуется никакого первоначального опыта работы с механизмом внедрения зависимостей или DI фреймворками.

Что внутри:

- Множество примеров на языке C#
- Каталог DI паттернов и анти-паттернов
- Примеры использования как фреймворков компании Microsoft, так и DI фреймворков с открытым исходным кодом

[Появление механизма внедрения зависимостей](#)

[Каталог DI](#)

[DI самостоятельно](#)

[DI-контейнеры](#)

Появление механизма внедрения зависимостей

Механизм внедрения зависимостей (Dependency Injection или DI) принадлежит к списку самых неправильно воспринимаемых концепций объектно-ориентированного программирования. Эта путаница широко распространена и касается терминологии, целей и механики. Должен ли этот механизм называться внедрением зависимостей, инверсией управления (Inversion of Control) или даже сторонним подключением (Third-Party Connect)? Является ли целью DI всего лишь поддержка модульного тестирования или же имеется более широкая цель? DI – это тоже самое, что и Service Locator? Нужен ли DI-контейнер?

Существует множество публикаций блогов, журнальных статей, конференционных презентаций и т.д., в которых обсуждается механизм DI, но, к сожалению, во многих из них используется противоречивая терминология и даются неправильные советы. Это справедливо и за рамками указанных источников, и даже такие крупные и влиятельные деятели, как компания Microsoft, тоже вовлечены в эту путаницу.

Дело не должно обстоять таким образом. В этой книге я представляю и использую непротиворечивую терминологию, которую, я надеюсь, примут и другие. В большинстве случаев я принимал и разъяснял существующую терминологию, определенную другими, но временами я добавлял некоторую терминологию, которая ранее не существовала. Это чрезвычайно помогло мне в выделении спецификаций области применения и границ DI.

Одной из основных причин противоречивости и неправильных советов является тот факт, что границы DI довольно нечеткие. Где заканчивается DI и начинаются другие концепции объектно-ориентированного программирования? Думаю, невозможно провести разграничительную линию между DI и другими аспектами написания качественного объектно-ориентированного кода. При обсуждении DI нам приходится вовлекаться в другие концепции такие, как SOLID и Clean Code. Не думаю, что я могу достоверно писать о механизме DI, не затрагивая при этом некоторых из этих других тематик.

Первая часть книги поможет вам осознать позицию DI по отношению к другим аспектам разработки программного обеспечения – заявляя, таким образом, о нем на всеуслышание.

Первая глава предоставляет вам быстрый обзор DI, охватывая его цели, принципы и преимущества, а также предоставляя набросок области ее применения для остальной части книги. Если вы хотите узнать, что такое DI и почему этот механизм должен вас заинтересовать, то это как раз то, с чего вам следует начать. Содержание главы предполагает, что у вас нет никаких первичных познаний DI, но даже если вы уже знали о ней, вы все еще можете иметь желание прочитать ее – может оказаться, что ее содержание отлично от того, что вы ожидали.

Первая глава сфокусирована на общей картине представления и не вдается в детали. Глава 2, с другой стороны, всецело зарезервирована под крупный пример. Понимается, что этот пример даст вам намного больше конкретного понимания DI. Он разделен на две части и сформирован практически в виде комментария. Для того чтобы противопоставить DI другим "традиционным" стилям программирования, глава сначала демонстрирует типичную, сильно связанную реализацию шаблонного приложения, а впоследствии заново реализует его с помощью DI.

Третья и конечная глава части 1 вводит понятие DI-контейнера и объясняет, как оно вписывается в общую картину представления DI. Я обсуждаю DI в общих понятиях и, несмотря на то, что я предоставляю примеры кода, которые демонстрируют, как работает типичный DI-контейнер, целью главы не является объяснение деталей конкретного API. Главная цель главы 3 – показать, что DI-контейнер является довольно полезным, необязательным инструментом. Вполне допустимо использовать DI без использования DI-контейнера, поэтому части 2 и 3 более или менее игнорируют DI-контейнеры и вместо этого обсуждают DI, не затрагивая контейнеры. Далее в части 4 мы возвращаемся к понятию DI-контейнера с целью анализа четырех конкретных контейнеров.

Часть 1 определяет контекст всей остальной книги. Она нацелена на читателей, которые не имеют первичных познаний DI, но опытные специалисты, использующие DI, могут также приобрести полезные знания, просматривая главы с целью получить понимание терминологии, используемой в рамках всей книги. К концу части 1 вы должны приобрести прочное понимание словаря и общих понятий, даже если некоторые конкретные детали все еще слегка расплывчаты. Ничего страшного – книга становится более конкретной на протяжении ее чтения, поэтому части 2, 3 и 4 должны будут ответить на вопросы, которые, скорее всего, появятся у вас после прочтения части 1.

[1. "Дегустационное меню" механизма внедрения зависимостей](#)

[2. Комплексный пример](#)

[3. DI-контейнеры](#)

1. "Дегустационное меню" механизма внедрения зависимостей

Меню:

- Неправильное понимание механизма внедрения зависимостей
- Цель механизма внедрения зависимостей
- Преимущества механизма внедрения зависимостей
- Когда необходимо применять механизм внедрения зависимостей

Вы могли слышать о том, что процесс приготовления беарнского соуса (*sauce béarnaise*) достаточно сложен. Даже большинство из тех людей, которые регулярно занимаются приготовлением пищи, никогда не пытались его приготовить. Это позорно, поскольку соус очень вкусный (его традиционно подают вместе с бифштексом, но он также отлично сочетается с белой спаржей, яйцами-пашот и другими блюдами). Некоторые прибегают к таким его заменителям, как готовые соусы или растворимые смеси, но нет ничего более восхитительного истинного беарнского соуса.

О пределение.

Беарнский соус – это эмульсионный соус, приготовленный из яичного желтка и масла и приправленный эстрагоном, кервелем, луком-шалотом и уксусом. Этот соус не содержит воды.

Самой большой проблемой при приготовлении беарнского соуса является то, что он может и не получиться – соус может загуститься или расслоиться, и если это случится, вы не сможете его восстановить. Процесс приготовления соуса занимает примерно 45 минут, поэтому проваленная попытка означает, что у вас уже не будет времени на вторую попытку.

С другой стороны, любой шеф-повар умеет готовить беарнский соус. Это входит в их обучение и, как они скажут вам сами, это не трудно. Вам не нужно быть профессиональным поваром, чтобы приготовить этот соус. Всякий, кто учился его готовить, по крайней мере, один раз терпел неудачу, но как только вы научитесь его готовить, у вас он будет получаться всегда.

Я думаю, что механизм внедрения зависимостей – это нечто вроде беарнского соуса. Полагают, что он сложен, и поэтому немногие его применяют. Если вы попытаетесь его использовать и потерпите неудачу, то у вас, скорее всего, не будет времени на вторую попытку.

О пределение.

Механизм внедрения зависимостей (*Dependency Injection*) – это набор принципов и шаблонов проектирования программного обеспечения, которые дают нам возможность разрабатывать слабосвязанный код.

Несмотря на Страх, Неопределенность и Сомнения (FUD), окружающие DI, ему так же легко научиться, как и приготовлению беарнского соуса. Вы можете допускать ошибки во

время обучения, но как только вы овладеете методикой, вы уже больше никогда не потерпите неудачу при применении этого механизма.

В разделе вопросов и ответов, связанных с разработкой программного обеспечения, веб-сайта Stack Overflow демонстрируется ответ на вопрос "Как объяснить механизм внедрения зависимостей пятилетнему ребенку?". Самый высоко оцененный ответ, который был дан Джоном Маншем, предоставляет поразительно точную аналогию, нацеленную на (воображаемого) пятилетнего изыскателя:

Когда ты идешь к холодильнику и что-то самостоятельно достаем из него, ты можешь есть причиной возникновения проблем. Ты можешь оставить дверь открытой, ты можешь взять то, что мама или папа не хотели бы, чтобы ты брал. Ты даже можешь искать то, чего у нас и не было, или то, что уже закончилось.

Все, что от тебе следует сделать, – это сформулировать, что тебе нужно, "Мне нужно то-то, чтобы выпить во время ланча", и после этого мы будем следить за тем, чтобы у тебя было то, когда ты сядешь кушать.

В терминах разработки программного обеспечения с использованием технологий объектно-ориентированного программирования это означает: взаимодействующие классы (пятилетние дети) должны полагаться на инфраструктуру (родителей), которая предоставляет необходимые услуги.

Как продемонстрировано на рисунке 1-1, данная глава совершенно прямолинейна по своей структуре. Сначала я знакомлю с DI, в том числе с его целями и преимуществами. Несмотря на то, что я включаю в нее примеры, в целом в этой главе содержится меньше кода, нежели в любой другой главе этой книги.

Рисунок 1-1: Структура книги совершенно прямолинейна. Вы должны прочитать первый раздел перед тем, как прочитать следующий и т.д. Это может показаться очевидным, но некоторые из последующих глав этой книги менее прямолинейны по своей натуре.



Перед тем как познакомить вас с механизмом внедрения зависимостей, я буду обсуждать основную цель DI: удобство сопровождения. Это важно, поскольку очень легко неправильно понять механизм DI, если вы не подготовлены должным образом. Далее, после примера (Hello DI), я буду рассматривать преимущества и область применения DI, в сущности, прокладывая маршрутную карту этой книги. Когда вы закончите изучение этой главы, вы будете подготовлены к более продвинутым концепциям, обсуждаемым в остальной части книги.

Для большинства разработчиков механизм внедрения зависимостей может казаться слегка отсталым способом создания исходного кода и, по аналогии с беарнским соусом, в него включено слишком много Страх, Неопределенности и Сомнения. Для того чтобы изучить механизм внедрения зависимостей, вам для начала необходимо понять его цель.

[1.1. Написание поддерживаемого кода](#)

[1.2. Привет DI](#)

[1.3. Что внедрять, а что не внедрять](#)

[1.4. Область применения DI](#)

[1.5. Резюме](#)

1.1. Написание поддерживаемого кода

Каким целям служит механизм внедрения зависимостей? DI, сам по себе, не является целью, скорее, это средство достижения результата. В конечном итоге целью большинства технологий программирования является предоставление программного обеспечения, работающего настолько эффективно, насколько это возможно. Одним из аспектов этой цели является написание поддерживаемого кода.

За исключением тех моментов, когда вы будете писать прототипы или приложения, у которых никогда не будет релиза выше первого, вы вскоре обнаружите, что занимаетесь поддержанием и расширением существующего кода. Для того чтобы с таким кодом можно было эффективно работать, он должен быть настолько поддерживаемым, насколько это возможно.

Одним из множества способов создания поддерживаемого кода является использование *слабого связывания*. Еще в 1995 году, когда "Банда четырех" (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс) написала книгу "Паттерны проектирования" ("Design Patterns"), существовало универсальное знание:

Программируй, основываясь на интерфейсе, а не на классах.

Этот совет является не заключением, а скорее остроумной предпосылкой книги "Паттерны проектирования": он появляется на странице 18. Слабое связывание делает код расширяемым, а расширяемость делает его поддерживаемым.

DI – это не более чем технология, которая разрешает слабое связывание. Как бы то ни было, существует множество моментов недопонимания механизма внедрения зависимостей, и иногда они мешают истинному пониманию. Перед тем как приступить к изучению, вы должны забыть то, что (как вы думаете) вы уже знали.

Какие знания о DI нужно забыть

Подобно стереотипу, касающемуся голливудских восточных единоборств, вы должны забыть все, что вы знали раньше, прежде чем сможете чему-то научиться. Существует множество моментов недопонимания механизма DI, и если вы будете носить эти знания в себе, то вы неправильно поймете то, что вы прочитаете в этой книге. Вы должны очистить свой разум, чтобы понять механизм DI.

Существует, по крайней мере, четыре общих мифа, касающихся механизма внедрения зависимостей:

- DI имеет отношение только к "позднему связыванию" (*late binding*).
- DI имеет отношение только к модульному тестированию (*unit testing*).
- DI – это вид абстрактной фабрики (Abstract Factory) на "стероидах".
- Для механизма DI необходим DI-контейнер.

Несмотря на то, что ни один из этих мифов не является правдой, они, тем не менее, широко распространены. Мы должны развеять их до того, как приступим к изучению механизма внедрения зависимостей.

Позднее связывание

В данном контексте под **поздним связыванием** понимается возможность заменять части приложения без необходимости перекомпиляции кода. Приложение, которое позволяет добавление-вставку сторонних компонентов (таких, как Visual Studio), является одним из примеров позднего связывания.

Еще один пример – стандартное программное обеспечение, которое поддерживает различные исполняемые среды. У вас может быть приложение, которое может запускаться больше, чем на одном движке базы данных: например, такое, которое поддерживает как Oracle, так и SQL Server. Для поддержки такой возможности остальная часть приложения может обращаться к базе данных посредством интерфейса. База кода может предоставлять различные реализации этого интерфейса с целью обеспечения доступа к Oracle и SQL Server соответственно. Для контроля над тем, какая реализация должна применяться для данной инсталляции, может использоваться опция конфигурации.

Тот факт, что механизм внедрения зависимостей имеет отношение только к такому виду сценариев, является повсеместным заблуждением. Это понятно, поскольку DI разрешает такой сценарий, но ошибочно думать, что зависимость симметрична. То, что DI разрешает позднее связывание, еще не означает, что он имеет отношение только к сценариям позднего связывания. Как это продемонстрировано на рисунке 1-2, позднее связывание является всего лишь одним из аспектов механизма внедрения зависимостей.

Рисунок 1-2: Позднее связывание разрешено механизмом внедрения зависимостей, но полагать, что DI применим только в сценариях позднего связывания значит принимать ограниченное представление более широкой перспективы.



Late binding

Если вы думали, что механизм внедрения зависимостей имеет отношение только к позднему связыванию, то вам необходимо забыть об этом. Механизм DI делает намного больше, чем просто разрешает позднее связывание.

Модульное тестирование

Некоторые люди думают, что механизм внедрения зависимостей имеет отношение только к поддержке модульного тестирования. Это неверно, несмотря на то, что механизм DI определенно является важной составляющей поддержки модульного тестирования.

По правде говоря, мое первоначальное знакомство с механизмом DI произошло во время борьбы с определенными аспектами технологии **разработки через тестирование** (*Test-Driven Development* или *TDD*). Во время этой борьбы я познакомился с DI и узнал, что остальные использовали этот механизм для того, чтобы поддерживать некоторые сценарии, похожие на те, к которым я обращался.

Даже если вы не пишете модульные тесты (если вы этого не делаете, то вам следует начать прямо сейчас), механизм DI все еще уместен, благодаря всем остальным преимуществам, которые он предлагает. Утверждение о том, что механизм DI имеет отношение только к поддержке модульного тестирования, подобно утверждению о том, что DI имеет отношение только к поддержке позднего связывания. Рисунок 1-3

демонстрирует, что, несмотря на то, что это уже другое представление, это представление такое же ограниченное, как и изображенное на рисунке 1-2. В данной книге я сделал все возможное, чтобы продемонстрировать вам общую картину.

Рисунок 1-3: Хотя предположение о том, что модульное тестирование является единственной целью механизма DI, – это уже другое представление, нежели то, которое касалось позднего связывания, оно все равно является ограниченным представлением более широкой перспективы



Если вы думали, что механизм внедрения зависимостей имеет отношение только к модульному тестированию – забудьте об этом. Механизм DI делает намного больше, чем просто разрешает модульное тестирование.

Абстрактная фабрика "на стероидах"

Возможно, самым опасным заблуждением является то, что механизм DI включает в себя некоторого рода универсальную абстрактную фабрику, которую мы можем использовать для создания экземпляров необходимых нам зависимостей.

Во введении к данной главе я писал, что "взаимодействующие классы ... должны полагаться на инфраструктуру ..., которая предоставляет необходимые услуги".

Какими были ваши первоначальные соображения по поводу этого предложения? Думали ли вы об инфраструктуре, как о некоторого рода сервисе, к которому вы могли бы обратиться, чтобы получить необходимую зависимость? Если это так, то вы не одиноки в своих мыслях. Многие разработчики и архитекторы думают о механизме внедрения зависимостей как о сервисе, который может использоваться для указания местоположения других сервисов; этот сервис имеет название *Service Locator* (сервис-локатор), но он является полной противоположностью DI.

Если вы думали о механизме DI, как о сервис-локаторе, – а именно, универсальной фабрике – вам нужно об этом забыть. Механизм внедрения зависимостей является противоположностью *Service Locator*; это способ структурирования кода таким образом, чтобы нам никогда не нужно было обязательно запрашивать зависимости. В противном случае мы принуждаем пользователей возмещать их.

DI-контейнеры

Тесно связано с предыдущим заблуждением мнение о том, что для механизма DI необходим DI-контейнер. Если бы вы придерживались предыдущего, ошибочного мнения о том, что DI включает в себя *Service Locator*, то было бы легко прийти к выводу о том, что DI-контейнер может взять на себя ответственность за *Service Locator*. Это имеет место быть, но это никак не является тем, как мы должны использовать DI-контейнер.

DI-контейнер – это необязательная библиотека, которая может упростить процесс создания компонентов при регистрации приложения, но это не обязательный способ.

Когда мы создаем приложения без использования DI-контейнера, мы называем это *Poor man's DI*; для этого требуется больше работы, но несколько другого рода, при которой нам не нужно идти на компромисс при использовании какого-либо принципа механизма внедрения зависимостей.

Если вы думали, что для механизма DI нужен DI-контейнер, то забудьте об этом. Механизм внедрения зависимостей – это набор принципов и паттернов, а DI-контейнер – это полезный, но необязательный инструмент.

Вы можете думать, что, хотя я и изложил четыре мифа о механизме внедрения зависимостей, мне, тем не менее, нужно предоставить неопровержимое доказательство против каждого из этих мифов. Это верно. В некотором смысле, вся эта книга является большим аргументом против этих общих заблуждений.

По моему мнению, забывание некоторых фактов жизненно необходимо, поскольку люди имеют склонность к тому, чтобы пытаться модифицировать те факты, которые я рассказываю им о механизме DI и совмещать это с тем, что, по их мнению, они уже знали о механизме внедрения зависимостей. Когда такое случается, много времени тратится прежде, чем они, в конце концов, осознают, что некоторые из их основных предпосылок не верны. Я хочу поделиться с вами этими знаниями. Поэтому, если вы сможете, попытайтесь прочитать эту книгу так, будто вы ничего не знаете о механизме DI.

Давайте предположим, что вы ничего не знаете о механизме внедрения зависимостей и о его целях и начнем с того, что рассмотрим то, что механизм DI делает.

Осознание цели DI

Механизм DI – это не конечная цель, это средство достижения результата. Механизм внедрения зависимостей разрешает слабое связывание, а слабое связывание делает код более поддерживаемым. Это совершенное утверждение, и, несмотря на то, что я мог бы отослать вас за подробностями к таким прочно установившимся авторитетам, как "Банда четырех", я думаю, что справедливо будет объяснить вам, почему это именно так.

Разработка программного обеспечения все еще остается довольно новой профессиональной сферой, поэтому в большинстве случаев мы все еще находимся в процессе разгадывания того, как реализовать хорошую архитектуру. Тем не менее, некоторые личности, имеющие опыт в более традиционных профессиях (например, конструирование), давно это разгадали.

Проверка в дешевом отеле

Если вы останавливаетесь в дешевом отеле, то можете столкнуться со зрелищем, подобно тому, которое продемонстрировано на рисунке 1-4. На этом рисунке продемонстрирован фен, дружелюбно предоставленный отелем для вашего удобства, но, по-видимому, администрация отеля не верит, что вы оставите фен следующему гостю: прибор напрямую присоединен к стенной розетке. Несмотря на то, что шнур достаточно длинный для того, чтобы предоставить вам определенную степень подвижности, вы не можете взять фен с собой. По-видимому, управление отеля решило, что стоимость замены украденных фенов достаточно высока и оправдывает то, что в противном случае явно является худшей реализацией.

Рисунок 1-4: В комнате дешевого отеля вы можете найти фен, подсоединенный к стенной розетке напрямую. Это эквивалентно использованию универсальной практики написания сильно связанного кода.



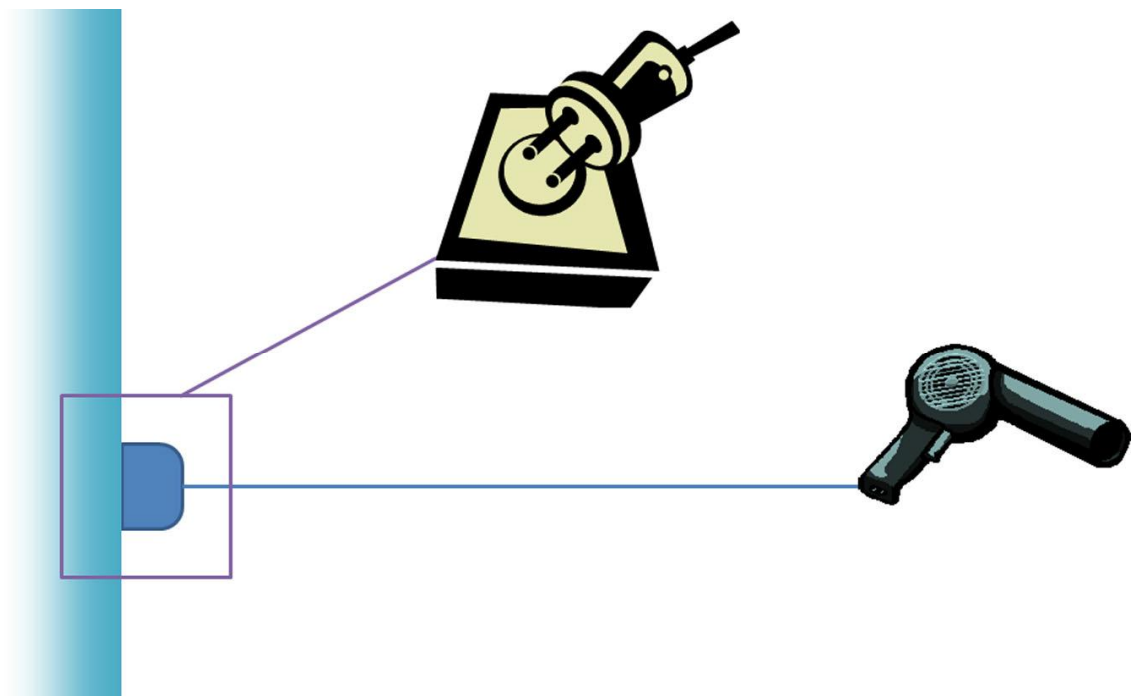
Что происходит, когда фены прекращают работать? Отелю приходится ввывать квалифицированного профессионала, который умеет решать эту проблему. Для ремонта жестко встроенного фена им придется отключить питание во всей комнате, делая ее при этом временно бесполезной. Затем специалист будет использовать специальные инструменты для того, чтобы тщательно отсоединить фен и заменить его на новый. Если вам повезет, то специалист не забудет снова включить питание в комнате и проверить, работает ли новый фен... Если вам повезет. Знакома ли вам вообще эта процедура?

Так вы бы приступали к работе с сильно связанным кодом. В этом сценарии фен сильно связан со стеной, и вы не сможете с легкостью модифицировать одного без влияния на другого.

Сравнение электрической проводки с паттернами проектирования

Обычно мы не монтируем электрические устройства вместе, напрямую присоединяя кабель к стене. Вместо этого, как продемонстрировано на рисунке 1-5, мы используем вилки и розетки. Розетка определяет форму, к которой необходимо присоединить вилку. Если провести аналогию с проектированием программного обеспечения, то розетка – это интерфейс.

Рисунок 1-5: Посредством использования розеток и вилок фен можно слабо связать со стенной розеткой.



В противоположность жестко встроенному фену вилки и розетки определяют слабо связанную модель соединения электрических устройств. Поскольку вилка вставляется в розетку, мы можем комбинировать устройства различными способами. Что в особенности интересно, так это то, что многие из этих универсальных комбинаций можно сравнить с хорошо известными принципами и паттернами проектирования программного обеспечения.

Во-первых, мы больше не стеснены рамками фенов. Если вы самый обычный читатель, то я бы предположил, что вам больше нужен компьютер, чем фен. Это не проблема: мы выдернем из розетки фен и подключим компьютер к той же розетке, как это показано на рисунке 1-6.

Рисунок 1-6: Используя розетки и вилки, мы можем заменить первоначально используемый фен из рисунка 1-5 на компьютер. Это соответствует принципу замещения Лисков.

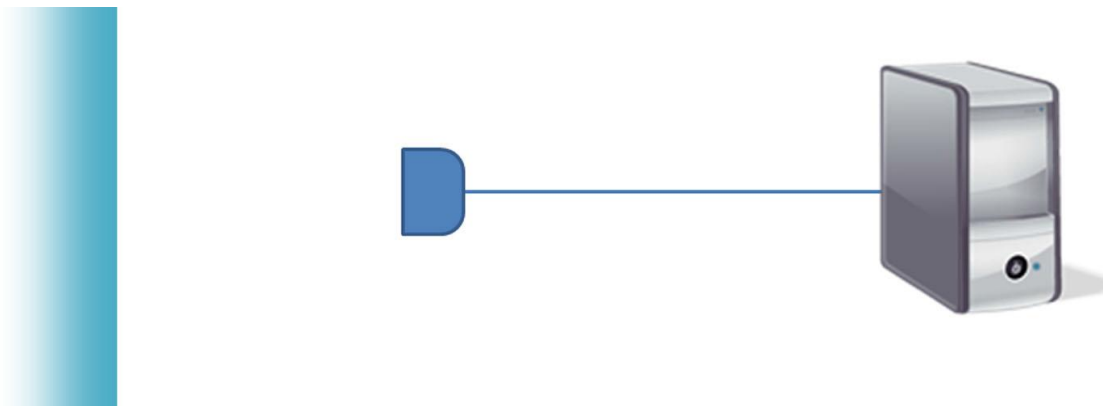


Удивительно, что понятие розетки стало использоваться на десятилетия раньше, чем появились компьютеры, и до сих пор она также является существенным для компьютера прибором. Первоначальные разработчики розеток, вероятно, не могли предсказать появление персональных компьютеров, но, поскольку их конструкция такая универсальная, можно столкнуться с потребностями, которые первоначально не рассматривались. Возможность замены одной конечной детали без смены другой схожа с центральным принципом проектирования программного обеспечения, который носит название **принцип замещения Лисков**. Этот принцип утверждает, что мы могли бы заменить одну реализацию интерфейса на другую, не разрушив при этом ни клиента, ни реализацию.

Что касается механизма внедрения зависимостей, принцип замещения Лисков – это один из самых важных принципов проектирования программного обеспечения. Это принцип, который дает нам возможность обращаться к потребностям, которые возникнут в будущем, даже если мы не можем предвидеть их сегодня.

Как это проиллюстрировано на рисунке 1-7, мы можем выдернуть шнур компьютера из розетки, если на данный момент нам не нужно его использовать. Даже если ничего не подключено к розетке, стена не взрывается.

Рисунок 1-7: Отключение компьютера не приводит ни к взрыву стены, ни к взрыву компьютера. Это можно приблизительно сравнить с паттерном **Null Object**.

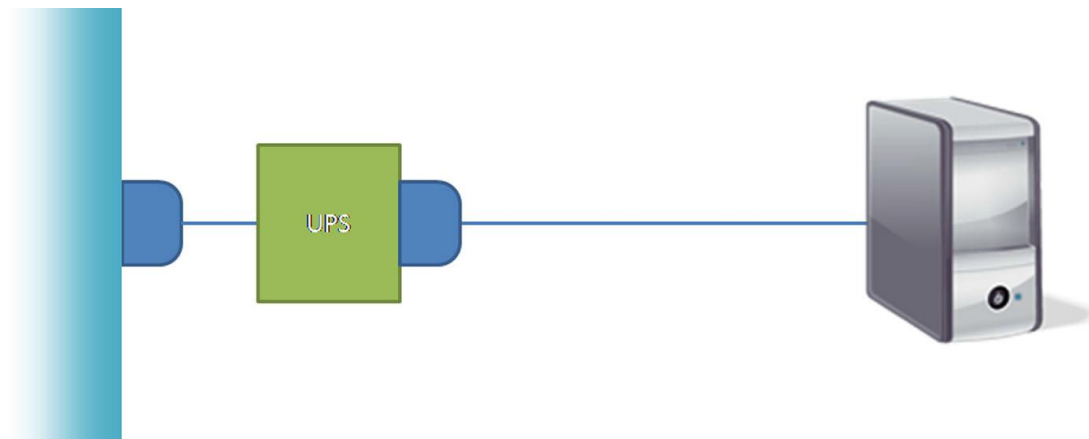


Если мы отсоединим компьютер от стены, то ни стена, ни компьютер не разрушатся (в действительности, если это портативная ЭВМ, то она может работать и на собственных

батарейках в течение некоторого времени). Тем не менее, что касается программного обеспечения, клиент часто ожидает, что сервис будет доступен. Если сервис был удален, то мы получаем `NullReferenceException`. Для того чтобы справиться с этой ситуацией, мы можем создать реализацию интерфейса, которая ничего не делает. Это паттерн проектирования, известный как **Null Object**, и он приблизительно соответствует отсоединению компьютера от стены. Благодаря тому, что мы используем слабое связывание, мы можем заменить существующую реализацию чем-то таким, что ничего не выполняет и при этом не приводит к проблемам.

Существует множество других вещей, которые мы можем сделать. Если мы живем по соседству со скачкообразным отключением электричества, то мы можем захотеть, чтобы наш компьютер продолжал работать и после отключения питания, подключив его для этого к системе бесперебойного питания (*Uninterrupted Power Supply*), как это продемонстрировано на рисунке 1-8: мы подключаем систему бесперебойного питания к стенной розетке, а компьютер к этой системе.

Рисунок 1-8: Можно воспользоваться системой бесперебойного питания для того, чтобы компьютер продолжал работать при отключении электричества. Это соответствует паттерну проектирования **Decorator**.

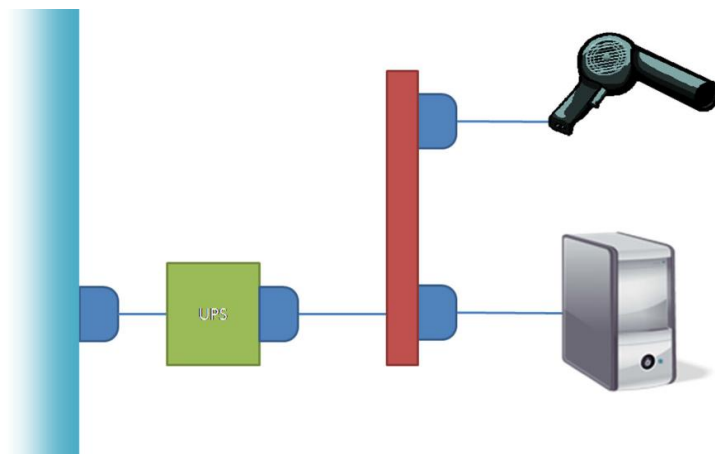


Компьютер и система бесперебойного питания служат разным целям. Каждый из них обладает самостоятельной ответственностью, на которую не может посягнуть другое устройство. Скорее всего, система бесперебойного питания и компьютер производятся двумя разными производителями, покупаются в разное время и подключаются в разное время. Как демонстрирует рисунок 1-6, мы можем запустить компьютер без системы бесперебойного питания (СБП), но мы также могли бы, возможно, использовать фен во время отключения электричества путем подключения его к СБП.

В проектировании программного обеспечения этот способ пересечения реализации другой реализацией того же самого интерфейса известен как паттерн проектирования **Decorator**. Он предоставляет нам возможность последовательно вводить новые возможности и сквозные сущности без необходимости заново переписывать и изменять огромные объемы существующего кода.

Еще один способ добавления новой функциональности в существующий код – комбинировать существующую реализацию интерфейса с новой реализацией. Когда мы соединяем несколько реализаций в одну, мы используем паттерн проектирования **Composite**. Рисунок 1-9 иллюстрирует то, как это соответствует подключению различных устройств к удлинителю.

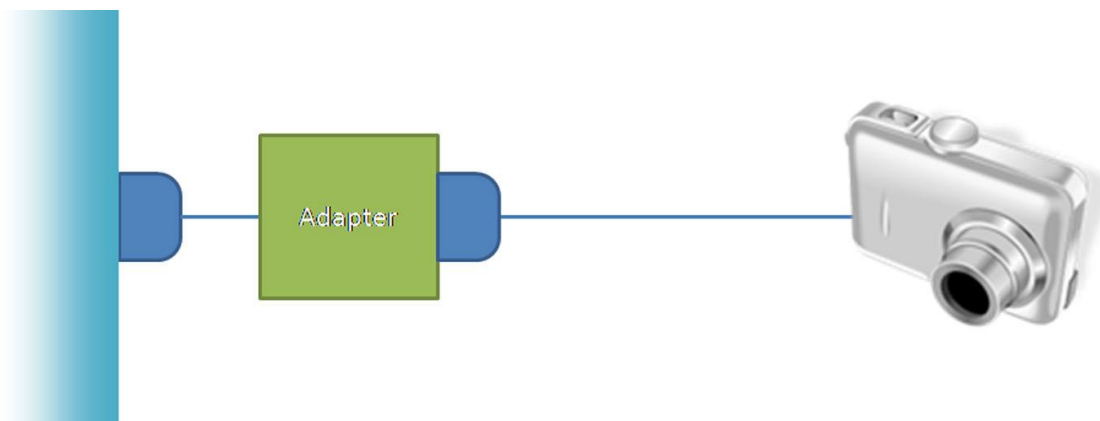
Рисунок 1-9: Удлинитель дает возможность подключать несколько устройств к одной стенной розетке. Это соответствует паттерну проектирования **Composite**.



Удлинитель имеет единственную вилку, которую мы можем вставить в единственную розетку, тогда как сам удлинитель предоставляет несколько розеток для множества устройств. Это позволяет нам подключать и отключать фен во время работы компьютера. Аналогично паттерн **Composite** облегчает процесс добавления и удаления функциональности посредством модификации набора составных реализаций интерфейса.

Ниже приведен окончательный пример. Мы иногда обнаруживаем, что вилка не подходит к определенной розетке. Если вы путешествовали в другую страну, то вы, скорее всего, замечали, что розетки во всем мире отличаются друг от друга. Если вы во время путешествия возите с собой что-то, например фотоаппарат, как показано на рисунке 1-10, то вам нужен адаптер для того, чтобы заряжать его. Соответственно, это паттерн проектирования с таким же именем – **Adapter**.

Рисунок 1-10: Во время путешествия нам часто нужно использовать адаптер для того, чтобы подключить устройство к иностранной розетке (например, чтобы перезарядить фотоаппарат). Это соответствует паттерну проектирования **Adapter**.



Паттерн проектирования **Adapter** работает так же, как и его физический тезка. Он может использоваться для соединения двух связанных, но разделенных интерфейсов друг с другом. Это в особенности полезно, когда у вас есть существующее стороннее API, которое вы хотите использовать в качестве экземпляра интерфейса, используемого вашим приложением.

По отношению к модели розетки и вилки удивительным является тот факт, что на протяжении десятилетий она является простой и универсальной моделью, и это доказано. После создания инфраструктура может использоваться кем угодно и адаптироваться к изменениям потребностей и непредсказуемым требованиям. Что еще более интересно, так это то, что когда мы связываем эту модель с разработкой программного обеспечения, все строительные блоки уже находятся на своих местах в виде принципов и паттернов проектирования.

Слабое связывание может сделать код более поддерживаемым.

Это самая простая часть. Программировать на основании интерфейса, а не реализации легко. Возникает вопрос, откуда берутся интерфейсы? В некотором смысле, это то, чему посвящена эта книга.

Вы можете создать новый экземпляр интерфейса так же, как вы создаете новый экземпляр конкретного типа. Код, подобный продемонстрированному ниже, не компилируется:

```
IMessageWriter writer = new IMessageWriter()
```

IMessageWriter: Программирование на основании интерфейсов

new IMessageWriter(): Не компилируется

У интерфейса нет конструктора, поэтому это невозможно. Экземпляр `writer` должен быть создан с помощью другого механизма. Механизм DI решает эту проблему.

С таким представлением цели механизма DI, думаю, вы готовы к примеру.

1.2. Привет DI

В традиции бесчисленных учебников по программированию давайте рассмотрим простое консольное приложение, которое выводит "Hello DI!" на экран. В этом разделе я продемонстрирую вам, на что похож код и вкратце обрисую некоторые ключевые преимущества, не вдаваясь при этом в детали, – в остальной части книги я рассмотрю это более детально.

Код приложения "Hello DI!"

Вероятно, вы привыкли видеть примеры "Hello World", которые пишутся в одну строку кода. В этой книге мы берем нечто чрезвычайно простое и делаем его сложным. Зачем? Мы скоро доберемся до этого, но сначала давайте посмотрим, как бы выглядел пример "Hello World" с использованием механизма внедрения зависимостей.

Партнеры

Чтобы понять структуру программы, мы начнем с рассмотрения метода `Main` консольного приложения, а затем я продемонстрирую вам взаимодействующие классы:

```
private static void Main()
{
    IMessageWriter writer = new ConsoleMessageWriter();
    var salutation = new Salutation(writer);
    salutation.Exclaim();
}
```

Программа должна вводить данные в консоль, поэтому она создает новый экземпляр `ConsoleMessageWriter`, который как раз и инкапсулирует эту функциональность. Он передает этого автора сообщения в класс `Salutation`, таким образом, экземпляр `Salutation` знает, куда записывать эти сообщения. Поскольку на данный момент все подсоединено должным образом, вы можете выполнить логику, что приведет к тому, что сообщение будет записано на экране.

Рисунок 1-11 демонстрирует взаимоотношения между партнерами.

Рисунок 1-11: Метод `Main` создает новые экземпляры как класса `ConsoleMessageWriter`, так и класса `Salutation`. `ConsoleMessageWriter` реализует интерфейс `IMessageWriter`, который используется `Salutation`. В сущности `Salutation` использует `ConsoleMessageWriter`, несмотря на то, что это не прямое использование не продемонстрировано.



Основная логика приложения инкапсулирована в классе `Salutation`, что продемонстрировано в следующем листинге.

Листинг 1-1: Класс Salutation

```
1. public class Salutation
2. {
3.     private readonly IMessageWriter writer;
4.     public Salutation(IMessageWriter writer)
5.     {
6.         if (writer == null)
7.         {
8.             throw new ArgumentNullException("writer");
9.         }
10.    this.writer = writer;
11.    }
12.    public void Exclaim()
13.    {
14.        this.writer.Write("Hello DI!");
15.    }
16. }
```

Строка 4: Внедряет зависимость

Строка 14: Использует зависимость

Класс `Salutation` зависит от пользовательского интерфейса под названием `IMessageWriter` и запрашивает экземпляр этого интерфейса через его конструктор. Этот процесс называется *внедрением через конструктор* (*Constructor Injection*) и описывается подробно в главе 4, которая также содержит более детальный анализ похожего примера кода.

Экземпляр `IMessageWriter` впоследствии используется в реализации метода `Exclaim`, который записывает соответствующее сообщение в зависимость.

`IMessageWriter` – это простой интерфейс, определенный следующим образом:

```
public interface IMessageWriter
{
    void Write(string message);
}
```

Он мог бы иметь другие элементы, но в этом простом примере вам нужен только метод `Write`. Этот интерфейс реализуется с помощью класса `ConsoleMessageWriter`, который метод `Main` передает в класс `Salutation`:

```
public class ConsoleMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

Класс `ConsoleMessageWriter` реализует `IMessageWriter` путем упаковывания класса `Console` из библиотеки базовых классов. Это простое приложение паттерна проектирования **Adapter**, о котором мы говорили в разделе "Осознание цели DI".

Возможно, вам интересно будет узнать о преимуществе замены однострочного кода на два класса и интерфейс, объем которых составляет 11 строк, и это вполне справедливо. Существует несколько преимуществ, которые можно получить, поступая таким образом.

Преимущества DI

Чем предыдущий пример лучше обычного однострочного кода, который мы обычно используем для реализации "Hello World" в C#? В этом примере механизм DI прибавляет издержек в размере 1,100%, но как только сложность кода возрастает от одной строки до десятков тысяч строк, эти издержки сокращаются и почти исчезают. Глава 2 предоставляет более сложный пример применения механизма внедрения зависимостей, и, несмотря на то, что этот пример все еще слишком прост по сравнению с реальными приложениями, вы должны заметить, что механизм DI менее навязчивый.

Я не виню вас в том, что вы можете найти предыдущий пример слишком надуманным, но обдумайте следующее: по своей сущности классический пример "Hello World" – это простая проблема с хорошо заданными и ограниченными требованиями. В реальном мире разработка программного обеспечения никогда не происходит таким образом. Требования изменяются и часто являются довольно расплывчатыми. Возможности, которые вам необходимо реализовывать, также стремятся к усложнению. Механизм внедрения зависимостей помогает решать такие вопросы путем разрешения слабого связывания. В частности мы получаем преимущества, перечисленные в таблице 1-1.

В таблице 1-1 я сначала указал такое преимущество как "позднее связывание", потому что, по моему опыту, это является самым важным для большинства людей. Когда архитекторы и разработчики не осознают преимуществ слабого связывания, это, главным образом, происходит, потому что они не рассматривают другие преимущества.

Таблица 1-1: Преимущества, получаемые при использовании слабого связывания. Каждое преимущество доступно всегда, но может быть по-разному оценено в зависимости от обстоятельств.

Преимущество	Описание	Когда оно полезно?
Позднее связывание	Сервисы могут меняться местами с другими сервисами.	Ценится в стандартном программном обеспечении, но, возможно, менее ценится в корпоративных приложениях, в которых исполняющая среда стремится к тому, чтобы быть хорошо определенной.
Расширяемость	Код можно расширять и использовать заново с помощью явно не запланированных способов.	Ценится всегда
Параллельная разработка	Код может разрабатываться параллельно.	Ценится в больших, сложных приложениях; но не так сильно в небольших, простых приложениях
Удобство сопровождения	Классы с явно определенными обязанностями легче поддерживать.	Ценится всегда
Тестируемость	Классы можно тестировать модульно.	Ценится только, если вы выполняете модульное тестирование (а вы действительно должны это делать)

Позднее связывание

При объяснении преимуществ программирования на основании интерфейсов и механизма внедрения зависимостей возможность заменять один сервис другим является самым преобладающим преимуществом для большинства людей, поэтому они стремятся взвесить все достоинства и недостатки, имея в виду только это преимущество.

Помните тот момент, когда я просил вас забыть все, что вы знали до этого, прежде чем начнете изучение? Вы можете сказать, что вы настолько хорошо знаете ваши потребности, что уверены в том, что вам никогда не придется заменять, скажем, вашу базу данных SQL Server чем-то еще. Тем не менее, потребности изменяются.

NoSQL, Windows Azure и аргументы в пользу композиции (composability)

Несколько лет назад я часто наталкивался на безразличные выражения, когда пыгался убедить разработчиков и архитекторов в преимуществах механизма внедрения зависимостей.

"Допустим, таким образом, вы можете заменить ваш реляционный компонент доступа к данным чем-то другим. Чем?" Существует ли какая-нибудь альтернатива реляционных баз данных?"

XML файлы никогда не казались подходящей альтернативой для крупномасштабных корпоративных сценариев. За последнюю пару лет все значительно изменилось.

О Windows Azure было объявлено на конференции Microsoft PDC 2008, и эта платформа многое сделала для того, чтобы убедить даже консервативные организации, которые используют только продукцию Microsoft, в том, что необходимо переоценить их позицию касательно хранилища данных. На данный момент существует реальная альтернатива реляционных баз данных, и мне приходится спрашивать людей только о том, хотели бы они, чтобы их приложение было "cloud-ready" приложением. Аргумент в пользу замещения на данный момент становится все весомее.

Связанное движение можно обнаружить во всей концепции NoSQL, которая моделирует приложения на основе ненормализованных данных — часто документо-ориентированных баз данных, но такие концепции, как Event Sourcing, также становятся все более важными.

В разделе "Код приложения "Hello DI!" вы не использовали "позднее связывание", поскольку вы явно создавали новый экземпляр `IMessageWriter` при помощи жестко-закодированного создания нового экземпляра `ConsoleMessageWriter`. Тем не менее, вы можете ввести "позднее связывание" путем изменения только одного фрагмента кода. Вам нужно всего лишь изменить следующую строку кода:

```
IMessageWriter writer = new ConsoleMessageWriter();
```

Для того чтобы разрешить позднее связывание, вы могли бы заменить эту строку кода чем-то вроде этого:

```
var typeName = ConfigurationManager.AppSettings["messageWriter"];  
var type = Type.GetType(typeName, true);  
IMessageWriter writer = (IMessageWriter)Activator.CreateInstance(type);
```

Посредством выпаскивания имени типа из конфигурационного файла приложения и создания из него экземпляра `Type` вы можете использовать рефлексии для создания экземпляра `IMessageWriter` во время компиляции без знания конкретного типа.

Чтобы выполнить это, вы указываете имя типа при настройке приложения `messageWriter` в конфигурационном файле этого приложения:

```
<appSettings>
  <add key="messageWriter"
    value="Ploeh.Samples.HelloDI.CommandLine.ConsoleMessageWriter, HelloDI" />
</appSettings>
```

Предупреждение

В этом примере для доказательства используются некоторые кратчайшие пути. В действительности на этот пример оказывает негативное влияние анти-паттерн **Constrained Construction**, который подробно рассматривается в главе 5.

Слабое связывание разрешает "позднее связывание", поскольку существует только одно место, где вы создаете экземпляр `IMessageWriter`. В связи с тем, что класс `Salutation` работает только по отношению к интерфейсу `IMessageWriter`, он никогда не замечает разницы.

В примере "Hello DI" "позднее связывание" будет предоставлять вам возможность писать сообщения другим адресатам, а не только в консоль – например, в базу данных или файл. Можно добавлять такие возможности, даже если вы явно не планировали их до этого.

Расширяемость

Успешное программное обеспечение должно быть способно к модификации. Вам может понадобиться добавить в него новые возможности, а также расширить существующие. Слабое связывание позволяет вам эффективно перестраивать приложение, подобно тому, как мы можем заново подключать электрические устройства при помощи вилок и розеток.

Давайте скажем, что вы хотите сделать пример "Hello DI" более безопасным, разрешая только авторизованным пользователям писать сообщения. Следующий листинг демонстрирует, как вы можете добавить эту возможность без изменения какой-либо существующей возможности: вы добавляете новую реализацию интерфейса

`IMessageWriter`.

Листинг 1-2: Расширение приложения "Hello DI" путем добавления возможности обеспечения безопасности

```
1. public class SecureMessageWriter : IMessageWriter
2. {
3.     private readonly IMessageWriter writer;
4.     public SecureMessageWriter(IMessageWriter writer)
5.     {
6.         if (writer == null)
7.         {
8.             throw new ArgumentNullException("writer");
9.         }
10.        this.writer = writer;
11.    }
12.    public void Write(string message)
```

```

13.  {
14.      if (Thread.CurrentPrincipal.Identity.IsAuthenticated)
15.      {
16.          this.writer.Write(message);
17.      }
18.  }
19.  }

```

Строка 14: Проверяет наличие авторизации

Строка 16: Записывает сообщение

Класс `SecureMessageWriter` реализует интерфейс `IMessageWriter` и в то же время использует его: он использует механизм внедрения зависимости через конструктор для того, чтобы запросить экземпляр `IMessageWriter`. Это стандартное приложение паттерна проектирования **Decorator**, о котором я упоминал в разделе "Осознание цели DI". Более подробно об этом паттерне мы поговорим в главе 9.

При реализации метода `Write` сначала проверяется, авторизован ли текущий пользователь. Только в этом случае этому пользователю разрешается записывать сообщения в поле `writer` с помощью метода `Write`.

Примечание

Метод `Write` в листинге 1-2 обращается к текущему пользователю через **Ambient Context** (окружающий контекст). Более гибкий, но в то же время немного более сложный вариант также мог бы предоставить пользователя посредством внедрения через конструктор.

Единственное место, где вам нужно будет изменить существующий код – это метод `Main`, поскольку вам нужно скомпоновать доступные классы несколько другим способом, нежели вы это делали до этого:

```

IMessageWriter writer =
    new SecureMessageWriter(
        new ConsoleMessageWriter());

```

Заметьте, что вы награждаете предыдущий экземпляр `ConsoleMessageWriter` классом `SecureMessageWriter`. В очередной раз класс `Salutation` не модифицируется, поскольку он использует только интерфейс `IMessageWriter`.

Слабое связывание позволяет вам писать код, который открыт для расширяемости, но закрыт для модификации. Это называется **принципом открытости/закрытости** (*Open/closed principle*). Единственное место, где вам нужно модифицировать код – в точке входа приложения, мы называем ее **Composition Root**.

`SecureMessageWriter` реализует возможность обеспечения безопасности в приложениях, в то время, как `ConsoleMessageWriter` обращается к пользовательскому интерфейсу. Это позволяет нам варьировать эти аспекты независимо друг от друга и компоновать их так, как нам нужно.

Параллельная разработка

Концепция разделения делает возможным разработку кода параллельными командами. Когда проект разработки программного обеспечения достигает определенного размера, становится необходимым разделить команду разработчиков на несколько команд легко управляемого размера. За каждой командой закрепляется ответственность за некоторую область приложения.

Для того чтобы разграничить ответственности, каждая команда будет разрабатывать один или более одного модуля, которые нужно будет интегрировать в законченное приложение. За исключением тех случаев, когда области деятельности каждой команды точно не зависят друг от друга, некоторые команды, скорее всего, зависят от функциональности, которая разрабатывается другими командами.

В приведенном выше примере благодаря тому, что классы `SecureMessageWriter` и `ConsoleMessageWriter` не зависят друг от друга напрямую, они могут разрабатываться параллельными командами. Момент, который им нужно будет согласовывать – это совместно используемый интерфейс `IMessageWriter`.

Удобство сопровождения

Как только ответственность каждого класса становится явно определенной и ограниченной, процесс сопровождения всего приложения облегчается. Это хорошо известное преимущество *принципа единственной ответственности* (*single responsibility principle*), который утверждает, что каждый класс должен обладать только единственной ответственностью.

Процесс добавления новых возможностей в приложение упрощается, поскольку совершенно ясно, где нужно выполнить изменения. Почти всегда нам даже не нужно изменять существующий код, но вместо этого мы можем добавлять новые классы и заново компоновать приложение. Здесь в игру снова вступает принцип единственной ответственности.

Поиск неисправностей также становится менее утомительным, поскольку область возможных виновников неисправностей сужается. Благодаря явно определенным ответственностям у вас всегда будет четкое осознание того, откуда нужно начинать поиск корневой причины появления проблемы.

Тестируемость

Для некоторых тестируемость является наименьшим поводом для беспокойства; для других это абсолютная потребность. Лично я принадлежу к последней категории людей: за мою карьеру я отклонил несколько предложений, потому что в них входила работа с определенными, нетестируемыми продуктами.

Определение

Приложение считается *тестируемым*, когда его можно тестировать помодульно.

Преимущество тестируемости, возможно, самое спорное из всех преимуществ, которые я перечислил ранее. Многие разработчики и архитекторы не практикуют модульное тестирование, поэтому они считают это преимущество в лучшем случае несущественным. Другие, такие как я, считают его существенным. Майкл Физерс даже дает определение

термину "унаследованное приложение" как любое приложение, которое не охвачено модульным тестированием.

Практически случайно слабое связывание разрешает модульное тестирование, потому что пользователи руководствуются *принципом замещения Лисков*. Они не заботятся о том, чтобы у их зависимостей были конкретные типы. Это означает, что мы можем внедрить *дублиры теста* (Test Doubles) в *тестируемую систему* (System Under Test (SUT)), как мы это видим в листинге 1-3.

Возможность подменять заданную зависимость заменителем конкретного теста – это побочный результат слабого связывания, но я предпочел указать эту возможность как отдельное преимущество, поскольку унаследованное значение отличается от первоначального.

Тестируемость

Термин "тестируемость" ужасно неопределенный, однако он широко используется в обществе разработчиков программного обеспечения, главным образом теми, кто практикует модульное тестирование.

В принципе любое приложение можно тестировать таким образом. Тесты могут выполняться людьми, которые используют это приложение, посредством их пользовательского интерфейса или какого-нибудь другого интерфейса, который предоставляет это приложение. Такие ручные тесты требуют больших временных и денежных затрат, поэтому наиболее предпочтительным является автоматизированное тестирование.

Существует множество различных видов автоматизированного тестирования, например, модульное тестирование, интеграционное тестирование, тестирование продуктивности, нагрузочное тестирование (*stress testing*) и т.д. Поскольку модульное тестирование имеет небольшое количество требований к исполняющим средам, оно является самым эффективным и сильным видом теста; часто в этом контексте и оценивается тестируемость.

Модульные тесты обеспечивают быструю ответную реакцию на состояние приложения, но модульные тесты можно писать только тогда, когда тестируемый модуль может быть явно изолирован от его зависимостей. Существует некоторая неоднозначность того, насколько в действительности детализирован модуль, но все соглашаются, что это действительно не является тем, что соединяет множественные модули. Возможность тестировать модули изолированно друг от друга очень важна для модульного тестирования.

Только когда приложение поддается модульному тестированию, его можно считать тестируемым. Самый безопасный способ обеспечения тестируемости приложения – это разрабатывать приложение при помощи технологии *тестирования через разработку* (TDD).

Необходимо отметить, что сами по себе модульные тесты не являются гарантией работоспособности приложения. Для того чтобы проверить, работает ли приложение так, как планировалось, все еще необходимы тесты всей системы или другие промежуточные виды тестов.

В зависимости от типа приложения, которое я разрабатываю, я могу заботиться, а могу и не заботиться о возможности выполнять "позднее связывание", но я всегда забочусь о тестируемости приложения. Некоторые разработчики не заботятся о тестируемости, но считают "позднее связывание" важным для разрабатываемого ими приложения.

Дублиеры теста

Это универсальная методика создания реализаций зависимостей, которые выступают в качестве заменителей реальных или планируемых реализаций. Такие реализации называются *дублерами теста*, и они никогда не будут использоваться в конечном приложении. Вместо этого они служат заполнителями для реальных зависимостей, когда они недоступны или не подходят для использования.

Для дублеров теста существует законченный язык паттернов и множество подтипов таких, как Stubs, Mocks и Fakes.

Пример: Модульное тестирование логики приложения "Hello"

В разделе "Код приложения "Hello DI!" вы видели пример приложения "Hello DI". Несмотря на то, что я сначала продемонстрировал вам конечный код, я, в действительности, разрабатывал это приложение при помощи тестирования через разработку. Листинг 1-3 демонстрирует самый важный модульный тест.

Примечание

Не волнуйтесь, если у вас нет опыта работы с модульным тестированием или динамическими *mock*-объектами. Они могут случайно всплывать на протяжении всей книги, но ни коим образом не являются обязательными для чтения.

Листинг 1-3: Модульное тестирование класса Salutation

```
[Fact]
public void ExclaimWillWriteCorrectMessageToMessageWriter()
{
    var writerMock = new Mock<IMessageWriter>();
    var sut = new Salutation(writerMock.Object);
    sut.Exclaim();
    writerMock.Verify(w => w.Write("Hello DI!"));
}
```

Для класса *Salutation* требуется экземпляр интерфейса *IMessageWriter*, поэтому вам нужно его создать. Вы могли бы использовать любую реализацию, но в модульных тестах могут быть очень полезными динамические *mock*-объекты – в данном случае вы используете *Mock*, но могли бы использовать и другие библиотеки или вместо этого свернуть свою собственную. Важная составляющая – обеспечение реализации *IMessageWriter* для конкретного теста с целью убедиться в том, что вы тестируете только один объект за раз; в настоящий момент вы тестируете метод *Exclaim* класса *Salutation*, поэтому вы не хотите, чтобы какая-нибудь производственная реализация *IMessageWriter* захламтила тест.

Чтобы создать класс *Salutation*, вы передаете *Mock*-экземпляр *IMessageWriter*. Поскольку *writerMock* – это экземпляр *Mock<IMessageWriter>*, свойство *Object* – это динамически создаваемый экземпляр *IMessageWriter*. Внедрение нужной зависимости посредством конструктора носит название "внедрение через конструктор".

После применения *тестируемой системы* (System Under Test (SUT)) вы можете использовать `Mock`, чтобы проверить, что метод `Write` был вызван с корректным текстом. При использовании `Moq` вы выполняете это путем вызова метода `Verify`, в качестве параметра которого задано выражение, которое определяет то, что вы запланировали. Если метод `IMessageWriter.Write` был вызван со строкой "Hello DI!", то вызов метода `Verify` завершается, но если метод `Write` не вызывался или вызывался с другим параметром, то метод `Verify` выдавал бы исключение и тест бы не выполнялся.

Слабое связывание предоставляет множество преимуществ: код становится проще разрабатывать, поддерживать, расширять, и он становится более тестируемым. Слабое связывание даже не особо и сложное. Мы программируем на основании интерфейсов, а не конкретных реализаций. Единственный основной недостаток – понять, как достать экземпляры этих интерфейсов. Механизм внедрения зависимостей дает ответ на этот вопрос путем внедрения зависимостей извне. Внедрение через конструктор – это наиболее предпочтительный метод внедрения зависимостей извне.

1.3. Что внедрять, а что не внедрять

Базы данных – это отличный пример типов BCL, которые являются неустойчивыми зависимостями: даже если LINQ to Entities – это технология, которая содержится в BCL, ее использование подразумевает реляционную базу данных.

В предыдущем разделе я описывал мотивационные силы, которые заставляют нас в первую очередь думать о механизме внедрения зависимостей. Если я убедил вас в том, что слабое связывание – это отличная идея, то вы можете захотеть делать все слабо связанным. В целом это хорошая идея. В те моменты, когда вам нужно решить, как упаковать модули, слабое связывание предоставляет особенно полезные рекомендации. Вам не приходится все абстрагировать и делать все подключаемым. В данном разделе я предоставляю вам некоторые средства, которые помогут вам принять решение о том, как моделировать ваши зависимости.

*Стандартная библиотека классов .NET (Base Class Library) состоит из множества сборок. Каждый раз при написании кода, который использует тип из сборки стандартной библиотеки классов, вы добавляете в ваш модуль зависимость. В предыдущем разделе я рассуждал на тему того, как важно слабое связывание, и насколько *программирование на основании интерфейсов* является основополагающим.*

Означает ли это, что вы не можете ссылаться ни на одну из сборок стандартной библиотеки классов и использовать их типы напрямую в приложении? Что если вам захочется применить `XmlWriter`, который определен в сборке `System.Xml`?

Вам не приходится обрабатывать все зависимости одинаково. Многие типы стандартной библиотеки классов можно использовать, не подвергая при этом опасности степень связанности приложения – но не все. Важно знать, как отличать типы, которые не представляют опасности, и типы, которые могут усилить степень связанности приложения. Сосредоточим наше внимание на последних.

Seams (Швы)

Везде, где мы решаем программировать на основании интерфейса, а не конкретного типа, мы вводим в приложение **Seam**. **Seam** – это место, где приложение монтируется из его составных частей, подобно тому, как куски материи сшиваются вместе с помощью швов. Также это место, где мы можем распаковать приложение и работать с модулями изолированно.

Пример "Hello DI", который я создавал в разделе "Hello DI", содержит Seam между классами `Salutation` и `ConsoleMessageWriter`, как это проиллюстрировано на рисунке 1-12. Класс `Salutation` не зависит напрямую от класса `ConsoleMessageWriter`; скорее, он использует интерфейс `IMessageWriter` для записи сообщений. Вы можете разобрать приложение на части в месте этого **Seam** и смонтировать приложение заново с другими составителями сообщений.

Рисунок 1-12: Пример "Hello DI" из раздела "Hello DI" содержит **Seam** между классами `Salutation` и `ConsoleMessageWriter`, потому что класс `Salutation` выполняет запись только при помощи абстракции интерфейса `IMessageWriter`.



Как только вы изучите механизм внедрения зависимостей, он поможет вам классифицировать ваши зависимости на стабильные и неустойчивые, но принятие решения о том, куда вставлять **Seams** вскоре станет для вас второстепенным. В следующих разделах будут более детально обсуждаться эти понятия.

Стабильные зависимости

Многие модули в стандартной библиотеке классов и за ее пределами не представляют никакой угрозы для степени модульности приложения. Они содержат функциональность, допускающую повторное использование, которую вы можете использовать для того, чтобы сделать ваш код более кратким.

Модули стандартной библиотеки классов всегда доступны в рамках вашего приложения, поскольку для его запуска необходим .NET Framework. Концепция параллельной разработки не применима к этим модулям, поскольку они уже существуют, и вы всегда можете повторно использовать стандартную библиотеку классов в других приложениях.

По умолчанию вы можете считать, что большинство (но не все) типов, определенных в BCL в качестве безопасных или стабильных зависимостей — я называю их *стабильными*, потому что они уже присутствуют там, склонны к обратной совместимости, а их вызов имеет детерминированные последствия.

Большинство стабильных зависимостей являются типами BCL, но и другие зависимости также могут быть стабильными. Важными критериями стабильных зависимостей являются перечисленные ниже критерии:

- Класс или модуль уже существует.
- Вы считаете, что новые версии не будут содержать разрушительных изменений.
- Рассматриваемые типы содержат детерминированные алгоритмы.
- Вы никогда не думаете о том, что вам придется заменять класс или модуль на что-то другое.

Как ни странно, DI-контейнеры сами по себе будут проявлять себя как стабильные зависимости, потому что они удовлетворяют всем критериям. Когда вы решаете взять за основу вашего приложения определенный DI-контейнер, вы рискуете споткнуться об этот выбор на протяжении всего жизненного цикла приложения; тем не менее, это еще одна причина того, почему вы должны ограничить использование контейнера в **Composition Root**.

Другие примеры приложений могут включать в себя специализированные библиотеки, которые инкапсулируют алгоритмы, имеющие отношение к вашему приложению. Если вы разрабатываете приложение, имеющее дело с химией, вы можете сослаться на стороннюю библиотеку, которая содержит химико-специфическую функциональность.

Обычно зависимости можно считать стабильными путем исключения: они стабильны, если они не являются неустойчивыми.

Неустойчивые зависимости

Введение в приложение **Seams** является дополнительной работой, поэтому вам следует это делать только при необходимости. Может существовать более одной причины необходимости изоляции зависимости за пределами **Seam**, но они тесно связаны с преимуществами слабого связывания, которое обсуждалось в разделе "Преимущества DI".

Такие зависимости можно узнать по их склонности создавать препятствия для одного или большинства из этих преимуществ. Они не стабильны, потому что они не обеспечивают достаточное основание для приложений, и по этой причине я называю их неустойчивыми зависимостями. Зависимость следует считать *неустойчивой*, если соблюдается любой из следующих критериев:

- Зависимость вводит требование, согласно которому необходимо установить и сконфигурировать исполняющую среду для приложения. Реляционная база данных является архитипическим примером: если мы не скроем реляционную базу данных за **Seam**, мы никогда не сможем заменить ее на какую-либо другую технологию. Это требование также усложняет процесс установки и запуска автоматизированных модульных тестов.

Базы данных – это отличный пример типов BCL, которые являются неустойчивыми зависимостями: даже если LINQ to Entities – это технология, которая содержится в BCL, ее использование подразумевает реляционную базу данных.

Другие необрабатываемые ресурсы такие, как очереди сообщений, веб-сервисы и даже файловые системы также попадают в эту категорию. Пожалуйста, отметьте, что это не сами конкретные .NET типы являются неустойчивыми, но скорее то, что они подразумевают об исполняющей среде.

Признаками этого вида зависимостей является отсутствие слабого связывания и расширяемости, а также невозможность тестирования.

- Зависимость еще не существует, но все еще разрабатывается. Очевидным признаком таких зависимостей является неспособность выполнять параллельную разработку.
- Зависимость не установлена на всех машинах организации, занимающейся разработкой. Это может быть аргументом в пользу дорогих сторонних библиотек или зависимостей, которые не могут устанавливаться на все операционные системы. Наиболее универсальный признак – невозможность тестирования.
- Зависимость содержит недетерминированное поведение. Это чрезвычайно важно в модульных тестах, поскольку все тесты должны быть детерминированными. Типичными источниками недетерминированности являются случайные числа и алгоритмы, которые зависят от текущих даты и времени.

Заметьте, что такие универсальные источники недетерминированности, как `System.Random`, `System.Security.Cryptography.RandomNumberGenerator` или `System.DateTime.Now` определены в **mscorlib**, поэтому вам не избежать ссылки на сборку, в которой они определены. Тем не менее, вам следует относиться к ним, как к неустойчивым зависимостям, потому что они склонны к разрушению тестируемости.

Неустойчивые зависимости являются центральным вопросом механизма внедрения зависимостей. Мы вводим **Seams** вприложение скорее для неустойчивых зависимостей, нежели для стабильных. Кроме того, они обязывают нас компоновать их при помощи механизма DI.

Теперь, когда вы поняли разницу между стабильными и неустойчивыми зависимостями, вы можете приступить к рассмотрению горизонтов области применения механизма внедрения зависимостей. Слабое связывание – это доминирующий принцип проектирования, поэтому механизм DI (как инструмент его реализации) должен быть в вашем коде повсюду. Не существует четкой границы между предметом механизма внедрения зависимостей и проектированием добротного программного обеспечения, но для того чтобы определить контекст остальной части книги, я быстро опишу, что какие вопросы охватывает механизм DI.

1.4. Область применения DI

Как мы видели в разделе "Hello DI", важный элемент механизма внедрения зависимостей – это вынесение различных ответственностей в отдельные классы. Одна из ответственностей, которую мы выносим в классы – это задача создания экземпляров зависимостей.

Как только класс уступает контроль над зависимостями, он отказывается от решения выбирать конкретные реализации. Тем не менее, как разработчики, мы приобретаем некоторые преимущества.

Примечание

Как разработчики мы приобретаем контроль путем удаления этого контроля из классов, которые применяют зависимости. Это применение *принципа единственной ответственности*: эти классы должны иметь дело только с их заданной областью ответственности, не беспокоясь при этом о том, как создаются зависимости.

Поначалу может показаться недостатком позволять классу отказываться от контроля над тем, какие объекты создаются, но, как разработчики, мы не теряем этот контроль – мы всего лишь переносим его в другое место.

Тем не менее, **композиция объектов** (*object composition*) является не единственным пластом управления, который мы удаляем, поскольку класс также теряет способность контролировать *время жизни* объекта. Когда в класс внедряется экземпляр зависимости, потребитель не знает, когда он был создан или когда он выйдет из области видимости. Во многих случаях это не представляет интереса для потребителя, но в некоторых случаях это важно.

Механизм внедрения зависимостей дает нам возможность управлять зависимостями универсальным способом. Когда пользователи напрямую создают и настраивают экземпляры зависимостей, каждый из них может делать это своим собственным способом, который может противоречить тому, как это делают остальные. Не существует способа, с помощью которого мы могли бы централизованно управлять зависимостями, а также способа, при помощи которого мы бы с легкостью обращались к сквозным концепциям. Благодаря механизму DI мы приобретаем возможность перехватывать каждый экземпляр зависимости и влиять на него до того, как он будет передан пользователю.

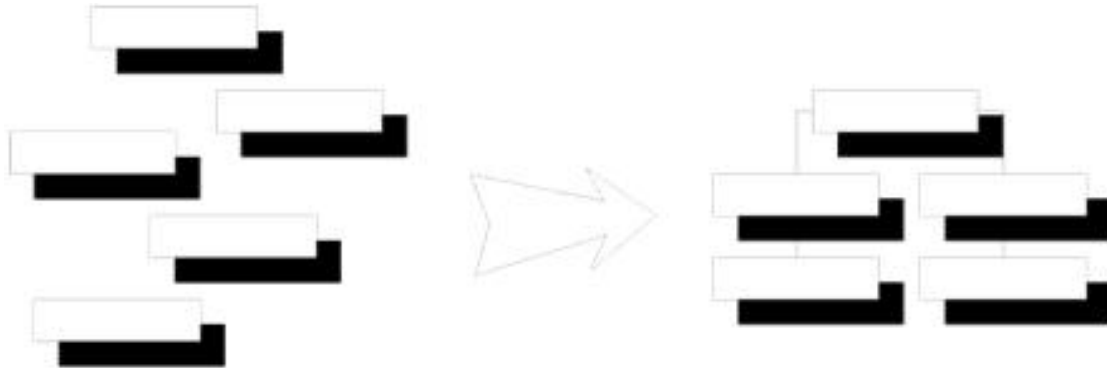
Благодаря механизму внедрения зависимостей мы можем компоновать приложения во время перехватывания зависимостей и контролирования их жизненного цикла. Композиция объектов, механизм перехвата и управление жизненным циклом – это три аспекта механизма внедрения зависимостей. Далее я вкратце раскрою эти аспекты; более детальное описание приводится в части 3 этой книги.

Композиция объектов

Чтобы воспользоваться результатами таких преимуществ, как расширяемость, позднее связывание и параллельная разработка, мы должны уметь компоновать классы в приложениях (см. рисунок 1-13). Такая композиция объектов часто является самой главной мотивацией для введения в приложение механизма внедрения зависимостей. Первоначально механизм DI был синонимичен композиции объектов; это единственный

аспект, обсуждаемый в оригинальной статье Мартина Фаулера относительно этого вопроса.

Рисунок 1-13: Композиция объектов означает, что модули в приложениях можно компоновать.



Существует несколько способов, с помощью которых мы можем компоновать классы в приложении. Когда я рассматривал позднее связывание, я использовал конфигурационный файл и несколько реализаций динамических объектов для того, чтобы вручную скомпоновать приложение из доступных модулей, но я также мог использовать и *технология конфигурирования в коде* или DI-контейнер. Мы вернемся к этим вопросам в главе 7.

Несмотря на то, что первоначальное значение механизма внедрения зависимостей тесно связано с композицией объектов, остальные аспекты также оказались уместными.

Жизненный цикл объектов

Класс, который отдает контроль над своими зависимостями, отказывается не просто от возможности выбирать определенные реализации абстракций. Он также отказывается и от возможности контроля над тем, когда создаются экземпляры зависимостей, и когда они выходят за рамки области применения механизма DI.

В .NET о большинстве из этих вещей заботится за нас **Garbage Collector**. Пользователь может иметь внедренные в класс зависимости и использовать их настолько долго, насколько того требует класс. Когда это происходит, зависимости выходят за рамки области применения механизма DI. Если на эти зависимости не ссылается больше ни один класс, то они как раз подходят для "сборки мусора" (*garbage collection*).

Что если два пользователя используют один и тот же вид зависимостей? Рисунок 1-14 иллюстрирует тот факт, что мы можем выбрать вариант внедрения отдельного экземпляра для каждого пользователя, тогда как рисунок 1-15 демонстрирует, что мы можем наоборот выбрать вариант, при котором единственный экземпляр будет совместно использоваться несколькими пользователями. Тем не менее, с точки зрения пользователей между этими вариантами нет никакой разницы. Согласно *принципу замещения Лисков* пользователь должен одинаково воспринимать все экземпляры данного интерфейса.

Рисунок 1-14: Для каждого из пользователей, совместно использующих один и тот же тип зависимостей, внедряется свой собственный приватный экземпляр.



Рисунок 1-15: индивидуальных пользователей, совместно использующих один и тот же тип зависимостей, внедряется один и тот же экземпляр.



Поскольку зависимости могут использоваться несколькими пользователями одновременно, единственный пользователь, вероятно, не может управлять их жизненным циклом. Поскольку управляемый объект может выходить за рамки области применения механизма DI и может быть удален сборщиком мусора (*garbage collector*), это не слишком важно, но когда зависимости реализуют интерфейс `IDisposable`, все становится намного сложнее.

В целом управление жизненным циклом является отдельным аспектом механизма внедрения зависимостей и настолько важным, что я выделил для него всю главу 8.

Отказ от контроля над зависимостью также означает отказ от контроля над ее жизненным циклом; нечто, находящееся выше в стеке вызовов, должно управлять жизненным циклом зависимости.

Перехват

Когда мы делегируем контроль над зависимостями стороннему компоненту, как это демонстрирует рисунок 1-16, мы также приобретаем возможность модифицировать эти зависимости до того, как мы передадим их в классы, которые используют эти зависимости.

Рисунок 1-16: Вместо того, чтобы внедрять первоначально запланированные зависимости, мы можем модифицировать эти зависимости посредством упаковывания их в другой класс до того, как передать их пользователю. Пунктирная стрелка указывает на направление действия – направление зависимости идет в обратном направлении.



В примере "Hello DI" я первоначально внедрял экземпляр `ConsoleMessageWriter` в класс `Salutation`. Затем, модифицируя пример, я добавил возможность обеспечения безопасности путем создания нового `SecureMessageWriter`, который всего лишь делегирует дальнейшую работу `ConsoleMessageWriter` при аутентификации пользователя. Это позволяет нам поддерживать *принцип единственной ответственности*.

Это возможно сделать, потому что мы всегда программируем на основании интерфейсов; помните, что эти зависимости всегда должны быть абстракциями. Что касается класса `Salutation`, то он не заботится о том, является ли используемый `IMessageWriter` экземпляром `ConsoleMessageWriter` или `SecureMessageWriter`. `SecureMessageWriter` может упаковывать `ConsoleMessageWriter`, который, однако, выполняет реальную работу.

Примечание

Перехват – это применение паттерна проектирования `Decorator`. Не переживайте, если вам не знаком паттерн проектирования **Decorator** – в главе 9 я предоставлю вам повторный курс, который всецело посвящен механизму перехвата.

Такие возможности перехвата переносят нас прямиком к *аспектно-ориентированному программированию (Aspect-Oriented Programming)* – тесно связанная с механизмом внедрения зависимостей тема, которая, тем не менее, выходит за пределы этой книги. Благодаря механизму перехвата мы можем применять такие сквозные механизмы, как авторизация, контрольные проверки, управление доступом, валидация и т.д. в хорошо структурированной манере, которая позволяет нам поддерживать концепцию разделения.

Механизм DI с точки зрения трех аспектов

Несмотря на то, что механизм внедрения зависимостей был задуман как набор паттернов, целью которых было решение проблемы композиции объектов, этот термин был в дальнейшем расширен и теперь также охватывает такие понятия, как жизненный цикл объектов и механизм перехвата. На сегодняшний момент я считаю, что механизм DI логично включает в себе все три аспекта.

Композиция объектов доминирует среди этих трех аспектов, поскольку без гибкой композиции объектов не было бы ни механизма перехвата, ни необходимости управлять жизненным циклом объектов. Композиция объектов доминировала на протяжении

большей части этой главы и будет продолжать доминировать на протяжении всей оставшейся книги, но мы не должны забывать об остальных аспектах. Композиция объектов предоставляет фундамент, а механизм управления жизненным циклом обращается к некоторым важным побочным эффектам, но пожинать плоды преимуществ механизма внедрения зависимостей мы начинаем, главным образом, когда переходим к механизму перехвата.

В части 3 я посвятил каждую главу отдельному аспекту, но в данной главе я предоставил обзор этих аспектов, поскольку важно знать, что на практике механизм внедрения зависимостей – это больше, чем просто композиция объектов.

1.5. Резюме

Механизм внедрения зависимостей по своей природе является средством достижения результата, а не целью. Это самый лучший способ разрешения слабого связывания, важной составляющей поддерживаемого кода. Преимущества, которые мы можем заполучить благодаря слабому связыванию, не всегда тотчас же очевидны, но становятся заметными с течением времени, когда возрастает сложность кода. Сильно связанный код, в конце концов, вырождается в "спагетти-код" (*Spaghetti Code*), между тем, как хорошо сконструированный, слабо связанный код может оставаться поддерживаемым. Для того чтобы достичь поистине "гибкого проектирования" (*Supply Design*), нужно не просто слабое связывание, но программирование на основании интерфейса является необходимым условием "гибкого проектирования".

Механизм DI – это больше, чем просто набор принципов и паттернов. Это, скорее, способ придумывания и конструирования кода, нежели средства и приемы – важный пункт слабого связывания, это и есть механизм внедрения зависимостей, и для эффективности он должен присутствовать в вашем коде повсюду.

Подсказка

Механизм внедрения зависимостей должен быть доминирующим. Вы не можете с легкостью подстраивать слабое связывание под существующий код.

Существует множество моментов неправильного понимания механизма DI. Некоторые люди думают, что он касается только таких узких проблем, как позднее связывание или модульное тестирование; несмотря на то, что эти аспекты проектирования программного обеспечения, безусловно, извлекают пользу из механизма внедрения зависимостей, область его применения намного шире. Абсолютной целью механизма внедрения зависимостей является удобство сопровождения кода.

В начале главы я утверждал, что вы должны забыть все, что вы знали до этого о механизме внедрения зависимостей. Это остается справедливым и для всей оставшейся части книги: вы должны очистить свой разум. В потрясающей публикации блога Николас Блумхардт пишет:

Словари ассоциативный массив – это одни из первых конструкций, о которых мы узнаем при разработке программного обеспечения. Легко увидеть аналогию между словарем и IoC-контейнером, который компонует объекты посредством использования механизма внедрения зависимостей.

Понимание механизма внедрения зависимостей как сервиса, смоделированного на основании строк словаря, напрямую ведет к анти-паттерну **Service Locator**. Вот поэтому я так много акцентировал ваше внимание на необходимости очистить ваш разум даже от самых основных предположений. В конце концов, говоря о словарях, мы говорим о некой субстанции, которая принадлежит к "рабелепному разуму программирования".

Цель механизма DI – сделать код поддерживаемым. Небольшие по размеру базы кода, подобные примеру "Hello World", являются по существу поддерживаемыми благодаря своему размеру; вот поэтому механизм внедрения зависимостей воспринимается в простых примерах как лишняя разработка. Чем больше по размеру становится база кода, тем более видимыми становятся преимущества механизма внедрения зависимостей. Я

посвятил следующую главу более значительному и сложному примеру для того, чтобы продемонстрировать эти преимущества

2. Комплексный пример

Меню:

- Комплексный пример
- Как не стоит поступать
- Как действовать правильно

Сообщение о том, что беарнский соус – это "эмульсионный соус, приготовленный из яичного желтка и масла", не привет вам способность готовить этот соус. Самый лучший способ обучения – это практика; пример часто может преодолеть разрыв между теорией и практикой. Полезно понаблюдать за тем, как профессиональный повар готовит беарнский соус, перед тем, как попытаться приготовить его самостоятельно.

Когда я знакомил вас в предыдущей главе с механизмом внедрения зависимостей, я предоставил вам высококлассный обзор для того, чтобы вы поняли его цель и основные принципы. Тем не менее, с помощью этого примера нельзя по достоинству оценить DI. Механизм внедрения зависимостей – это способ разрешения слабого связывания, а слабое связывание – это первый и самый главный эффективный способ справиться со сложностью.

Большинство программного обеспечения является сложным в том плане, что ему нужно обращаться ко многим сущностям одновременно. Помимо бизнес-сущностей (которые сами по себе могут быть сложными), программное обеспечение должно также обращаться к сущностям, связанным с вопросами обеспечения безопасности, диагностики, операций и расширяемости. Вместо того обращения ко всем этим сущностям в одном большом комке, слабое связывание позволяет нам обращаться к каждой сущности по отдельности. Проще обращаться к каждой сущности изолированно – но, в конечном счете, мы все еще должны компоновать этот сложный набор сущностей. Давайте рассмотрим сложный пример для того, чтобы лучше продемонстрировать механизм внедрения зависимостей.

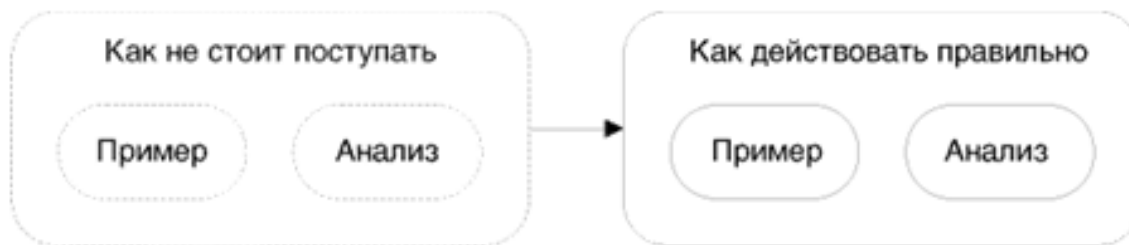
Предупреждение

Весь смысл слабо связанного кода – борьба со сложностью эффективным способом, и нам необходимы комплексные примеры для того, чтобы проиллюстрировать сложные сущности. Должно быть, вы предполагаете, что большинство примеров в этой книге являются сложными, и в них входят составные классы из составных библиотек. Сложность – это часть игры.

Я посвятил всю эту главу сложному примеру. Думаю, что это важно – сравнить слабо связанный код с более "традиционным", сильно связанным примером, поэтому в этой главе вы найдете одну и ту же возможность, реализованную двумя способами. Сначала я подготавливаю почву, демонстрируя, как легко писать сильно связанный код. Затем я реализую ту же функциональность при помощи механизма внедрения зависимостей. Вы можете пропустить пример сильно связанного кода, если хотите прямо сейчас увидеть некоторый слабо связанный код. После прочтения этой главы вы должны уже начать понимать, как можно использовать DI для того, чтобы создавать слабо связанный код.

Рисунок 2-1: Эта глава содержит две разновидности одного и того же примера. Сначала вы увидите, как легко писать сильно связанный код. Затем вы увидите, как писать то же самое приложение при помощи слабо связанного кода. Оба примера содержат как сам

пример, так и его анализ. Если вы хотите прямо сейчас увидеть слабо связанный код, то вы можете пропустить первый раздел



[2.1. Как не стоит поступать](#)

[2.2. Как действовать правильно](#)

[2.3. Расширение шаблонного приложения](#)

[2.4. Резюме](#)

2.1. Как не стоит поступать

Сама идея создания слабо связанного кода не является чрезвычайно противоречивой, но существует огромный разрыв между намерением и практикой. Перед тем как показать вам, как использовать механизм внедрения зависимостей для создания слабо связанных приложений, я хочу продемонстрировать вам, как легко все может пойти неправильным путем.

Общепринятой попыткой использования слабо связанного кода является создание многоуровневого приложения. Всякий может нарисовать схему трехуровневого приложения, и рисунок 2-2 доказывает, что я тоже могу это сделать.

Рисунок 2-2: Стандартная архитектура трехуровневого приложения. Это самая простая общепринятая разновидность архитектуры n -уровневого приложения, где приложение состоит из n -уровней, каждый из которых состоит из одного или более одного модуля. Некоторые разновидности n -уровневых схем будут иметь вертикальные блоки, которые обозначают уровни составного приложения. Такие схемы часто используются для представления таких сквозных сущностей, как обеспечение безопасности и вход в систему.



Создание трехуровневой схемы является обманчиво простым процессом, но само рисование схемы похоже на утверждение о том, что у вас будет беарнский соус с собственным стейком: это объявление намерения, которое не дает никаких гарантий относительно окончательного результата. У вас может получиться и что-то другое, как вы сами это скоро увидите.

Создание сильно связанного приложения

Существует более одного способа просмотра и проектирования гибкого и поддерживаемого сложного приложения, но архитектура n -уровневого приложения основывается на хорошо известном, практически опробованном подходе. Проблема в том, чтобы реализовать его корректно.

Вооружившись трехуровневой схемой, подобной той, что продемонстрирована на рисунке 2-2, вы можете приступить к созданию приложения.

Знакомство с Мэри Роуэн

Мэри Роуэн – профессиональный .NET разработчик, работающая на местного сертифицированного партнера компании Microsoft, который, в основном, занимается разработкой веб-приложений. Ей 34 и с программным обеспечением она работает уже 11 лет. Это делает ее одним из самых опытных разработчиков компании, и она часто выступает в роли наставника для младших разработчиков в дополнение к выполнению ее регулярных обязанностей в качестве старшего разработчика.

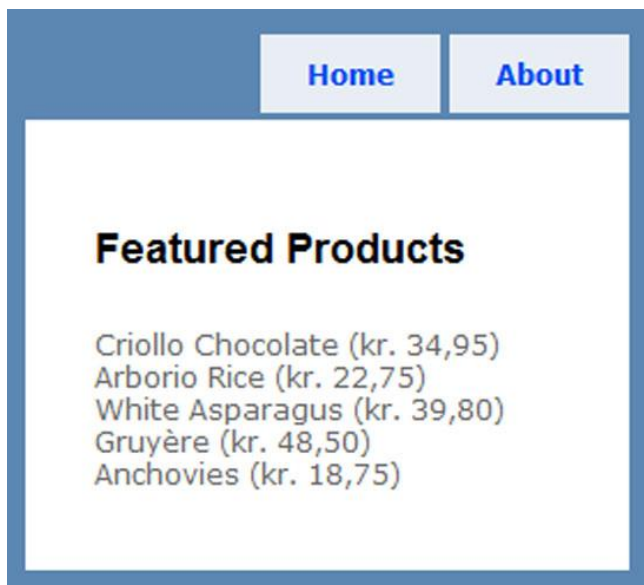
В целом Мэри получает удовольствие от своей работы, но ее расстраивает то, что основные этапы работы зачастую пропускаются, и это вынуждает ее и ее коллег работать в течение многих часов и недель, чтобы успеть выполнить все в срок. Она полагает, что должны существовать более эффективные способы создания программного обеспечения. Стремясь узнать больше информации об эффективности, она покупает множество книг по программированию – но у нее редко есть время на их прочтение.

Большинство своего свободного времени она тратит на мужа и двух дочерей. Мэри любит заниматься восхождением в горы. К тому же она увлеченный повар и уж точно знает, как приготовить настоящий беарнский соус.

Мэри попросили создать новое приложение электронной коммерции на платформе ASP.NET MVC и EntityFramework с SQL Server, используемым в качестве хранилища данных. Для того чтобы максимально увеличить модульность, это приложение должно быть трехуровневым.

Первой реализованной возможностью должен быть простой список рекомендуемых товаров, который выписывается из таблицы базы данных и отображается на веб-странице; пример продемонстрирован на рисунке 2-3. Если список просматривает привилегированный пользователь, то цена всех товаров должна быть снижена на 5 процентов.

Рисунок 2-3: Скриншот веб-приложения электронной коммерции, которое попросили создать Мэри. На нем изображен простой список рекомендуемых товаров и их цен ("kr." – это буквенный символ валюты для датских крон)



Давайте посмотрим, как Мэри реализует первую возможность приложения.

Уровень данных

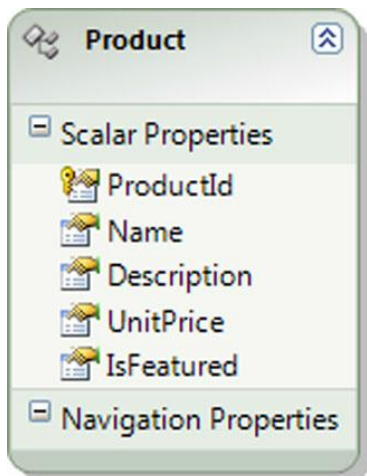
Поскольку Мэри нужно выпаскивать данные из таблицы базы данных, она решила начать с реализации уровня данных. Первый шаг – определить саму таблицу базы данных. Для создания таблицы продемонстрированной на рисунке 2-4, Мэри использует SQL Server Management Studio.

Рисунок 2-4: Мэри создает таблицу Product при помощи SQL Server Management Studio; альтернативные подходы включают написание T-SQL скрипта, или создание таблицы с помощью Visual Studio или с помощью каких-либо других средств

Product			
	Column Name	Data Type	Allow Nulls
🔑	ProductId	int	☐
	Name	nvarchar(50)	☐
	Description	nvarchar(MAX)	☐
	UnitPrice	money	☐
	Featured	bit	☐
			☐

Для реализации уровня "Data Access Layer" (уровень доступа к данным) Мэри добавляет в свое решение новую библиотеку. В Visual Studio она использует мастер Entity Data Model Wizard для того, чтобы сгенерировать целостную модель сущностей из базы данных, которую она ранее создала. Чтобы завершить оформление модели, она изменяет несколько названий, как это показано на рисунке 2-5.

Рисунок 2-5: Сущность Product, которая сгенерирована из таблицы базы данных Product, продемонстрированной на рисунке 2-4. Мэри изменила название столбца Featured на IsFeatured, а также изменила несколько названий в сгенерированномObjectContext (не продемонстрировано).



Примечание

Не стоит волноваться, если вы не знакомы с Microsoft EntityFramework. Подробная информация о реализации доступа к данным в *данном* контексте не является важной, поэтому вам нужно всего лишь суметь последовать данному примеру, даже если вы более знакомы с другой технологией доступа к данным.

Сгенерированные сущности `ObjectContext` и `Product` – это открытые типы, которые находятся в пределах одной и той же сборки. Мэри знает, что в дальнейшем ей нужно будет добавить в приложение больше возможностей, но добавление компонента доступа к данным, который необходим для реализации первой возможности, на этом завершено.

Рисунок 2-6 демонстрирует, как далеко Мэри зашла в реализации многоуровневой архитектуры, которая была представлена на рисунке 2-2.

Рисунок 2-6: До настоящего времени Мэри реализовывала уровень "Data Access Layer" (уровень доступа к данным) своего приложения. Уровни "Domain Logic Layer" (уровень доменной логики) и "User Interface Layer" (уровень пользовательского интерфейса) отодвинуты на второй план до тех пор, пока не будет реализована первая возможность.



Теперь, когда уровень доступа к данным реализован, следующий логичный шаг – реализация уровня доменной логики.

Доменный уровень

При отсутствии какой-либо доменной логики список товаров, предоставляемых с помощью сгенерированного `ObjectContext`, формально мог использоваться напрямую на уровне пользовательского интерфейса.

Предупреждение

За исключением простых приложений, содержащих отчеты, в приложениях всегда присутствует доменная логика. Сначала вы можете и не понять ее, но как только вы начнете узнавать предметную область приложения, его вложенные и скрытые правила и предположения постепенно всплывут. Реализация данной логики на уровне пользовательского интерфейса или на уровне доступа к данным приведет к печальным последствиям. Окажите себе честь и создайте сначала уровень доменной логики.

В требованиях к приложению Мэри говорится, что для привилегированных пользователей должен отображаться список цен с пятипроцентной скидкой. Мэри еще не поняла, как идентифицировать привилегированного пользователя, поэтому она попросила совета у своего коллеги Дженса:

Мэри: Мне нужно реализовать эту бизнес-логику так, чтобы привилегированный пользователь получал пятипроцентную скидку.

Дженс: Звучит легко. Просто умножь на 0,95.

Мэри: Спасибо, но я не об этом хотела тебя спросить. Я хотела спросить, как мне следует идентифицировать привилегированного пользователя?

Дженс: Понятно. У тебя веб-приложение или настольное приложение?

Мэри: Веб-приложение.

Дженс: Ясно, тогда ты можешь определить профиль пользователя и использовать свойство `IsPreferredCustomer`. Ты можешь получить профиль через `HttpContext`.

Мэри: Притормози-ка, Дженс. Этот код должен быть на уровне доменной логики. Это библиотека. Нет никакого `HttpContext`.

Дженс: О... (некоторое время думает). Я все еще считаю, что тебе нужно использовать возможность `Profile ASP.NET` для поиска значения по пользователю. Ты можешь потом передать значение в твою доменную логику как булево.

Мэри: Я не знаю...

Дженс: Это еще и даст тебе уверенность в том, что ты следуешь концепции разделения, потому что доменная логика не приходится иметь дело с обеспечением безопасности. Сама знаешь: Принцип единичной ответственности! Это самый быстрый способ достижения этого!

Мэри: Думаю, ты прав.

Предупреждение

Дженс базировал свой совет на своем техническом знании ASP.NET. Как только обсуждение вывело его из его же комфортной зоны, он стал отбиваться от Мэри при помощи тройной комбинации умных словечек. Знайте, что он не знает, о чем говорит: он неправильно использует концепцию разделения понятий, абсолютно неправильно использует **принцип единичной ответственности**, и он упоминает о том, что это самый быстрый способ, только потому что недавно слышал, как кто-то еще с энтузиазмом говорил об этом.

Вооружившись советом Дженса, Мэри создает новый проект библиотеки C# и добавляет класс с названием `ProductService`, который продемонстрирован в листинге ниже. Для того чтобы сделать класс `ProductService` компилируемым, она должна добавить ссылку на свою библиотеку `Data Access`, поскольку там определен класс `CommerceObjectContext`.

Листинг 2-1: Класс ProductService, добавленный Мэри

```
public partial class ProductService
{
    private readonly CommerceObjectContext objectContext;

    public ProductService()
    {
        this.objectContext = new CommerceObjectContext();
    }

    public IEnumerable<Product> GetFeaturedProducts(bool isCustomerPreferred)
    {
        var discount = isCustomerPreferred ? .95m : 1;
        var products = (from p in this.objectContext.Products
                        where p.IsFeatured
                        select p).AsEnumerable();
        return from p in products
               select new Product
               {
                   ProductId = p.ProductId,
                   Name = p.Name,
                   Description = p.Description,
                   IsFeatured = p.IsFeatured,
                   UnitPrice = p.UnitPrice * discount
               };
    }
}
```

Мэри счастлива, что смогла инкапсулировать технологию доступа к данным (LINQ to Entities), конфигурацию и доменную логику в классе `ProductService`. Она делегировала знания о пользователе вызывающему оператору путем передачи параметра `isCustomerPreferred`, и использует это значение для расчета скидки для всех товаров.

Дальнейшая обработка могла бы включать замену жестко-закодированного значения скидки (0, 95) на настраиваемое число, но на данный момент такой реализации достаточно. Мэри почти все сделала – остался только пользовательский интерфейс. Мэри решила, что это может подождать до следующего дня.

Рисунок 2-7 демонстрирует, как далеко Мэри зашла в реализации архитектуры, представленной на рисунке 2-2.

Рисунок 2-7: На данном этапе Мэри реализовала уровень доступа к данным и уровень доменной логики. Если сравнивать с рисунком 2-6, то здесь добавлен уровень доменной логики. Уровень пользовательского интерфейса все еще остается не реализованным



После реализации уровня доступа к данным и уровня доменной логики остается реализовать только уровень пользовательского интерфейса.

Уровень пользовательского интерфейса

На следующий день Мэри возобновляет свою работу над приложением электронной коммерции, добавляя новое ASP.NET MVC приложение в свое решение.

Примечание

Не стоит волноваться, если вы не знакомы с фреймворком ASP.NET MVC. Запутанные детали того, как функционирует MVC framework, не рассматриваются в этой книге. Важной составляющей является то, как используются зависимости, а это относительно нейтральная тема.

Ускоренный курс по ASP.NET MVC

ASP.NET MVC берет свое название от паттерна проектирования Model View Controller. В данном контексте самое важное, что необходимо понять, — это то, что, когда поступает веб-запрос, контроллер управляет этим запросом, потенциально используя модель (доменную) для того, чтобы затронуть этот запрос и сформировать ответ, который, в конце концов, отображается с помощью представления.

Контроллер — это обычно класс, который наследуется от абстрактного класса `Controller`. У него есть один или более методов действия, которые управляют запросами, например. Класс `HomeController` обладает методом `Index`, который управляет запросом страницы, отображаемой по умолчанию.

При возврате метода действия он передает итоговую модель в представление посредством экземпляра `ViewResult`.

Следующий листинг демонстрирует, как Мэри реализует метод `Index` класса `HomeController` для того, чтобы извлечь обработанные товары из базы данных и передать их в представление. Для того чтобы сделать этот код компилируемым, ей нужно добавить ссылки как на библиотеку `Data Access`, так и на библиотеку `Domain`, поскольку класс `ProductService` определен в библиотеке `Domain`, а класс `Product` определен в библиотеке `Data Access`.

Листинг 2-2: Метод `Index` в используемом по умолчанию классе контроллера

```
public ViewResult Index()
{
    bool isPreferredCustomer = this.User.IsInRole("PreferredCustomer");

    var service = new ProductService();
    var products = service.GetFeaturedProducts(isPreferredCustomer);
    this.ViewData["Products"] = products;

    return this.View();
}
```

Являясь частью жизненного цикла ASP.NET MVC, свойство `User` класса `HomeController` автоматически заполняется корректным объектом пользователя, поэтому Мэри использует его для того, чтобы определить, является ли текущий пользователь привилегированным.

Вооружившись этой информацией, она может вызвать доменную логику, чтобы получить список рекомендуемых товаров. Через некоторое время я вернусь к этому моменту, потому что здесь есть ловушка, но сейчас я позволю Мэри обнаружить эту ошибку самостоятельно.

Список товаров должен отображаться с помощью представления `Index`. Следующий листинг демонстрирует разметку для этого представления.

Листинг 2-3: Разметка представления `Index`

```
1. <h2>Featured Products</h2>
2. <div>
3.     <% var products = (IEnumerable<Product>)this.ViewData["Products"];
4.     foreach (var product in products)
5.     { %>
6.         <div>
7.             <%= this.Html.Encode(product.Name) %>
8.             (<%= this.Html.Encode(product.UnitPrice.ToString("C")) %>)
9.         </div>
10.    <% } %>
11. </div>
```

Строка 3: Получает товары, заполненные при помощи контроллера

ASP.NET MVC позволяет вам писать стандартный HTML-код с фрагментами императивного кода, который вставляется для получения доступа к объектам, созданным и переданным контроллером, который создал представление. В данном случае метод `Index` класса `HomeController` передал список рекомендуемых товаров ключу с названием `Products`, который Мэри использует в представлении для того, чтобы отображать список товаров.

Рисунок 2-8 демонстрирует, как Мэри реализовала архитектуры, показанную на рисунке 2-2.

Рисунок 2-8: На данный момент Мэри реализовала все три уровня приложения. Этот рисунок идентичен рисунку 2-2, но повторяется здесь для того, чтобы проиллюстрировать текущее состояние приложения Мэри.



Когда все три уровня находятся на своих местах, теоретически приложения должны работать, но только тест может проверить, действительно ли они работают.

Дымовой тест (Smoke test)

На данный момент Мэри реализовала все три уровня, поэтому пришло время понять, работает ли приложение. Она нажимает клавишу F5 и вскоре получает сообщение:

Указанное именованное соединение либо не найдено в конфигурации, либо не предназначено для использования в рамках EntityClient провайдера, либо не валидно.

Поскольку Мэри использовала конструктор по умолчанию `CommerceObjectContext` (продемонстрированный в листинге 2-1), то по смыслу ожидается, что строка соединения под названием `CommerceObjectContext` присутствует в файле `web.config`. Как я и упоминал в обсуждении листинга 2-2, эта имплицитность содержит в себе ловушку. За ночь Мэри забыла детали реализации ее доменного уровня. Код компилируется, но сайт не работает.

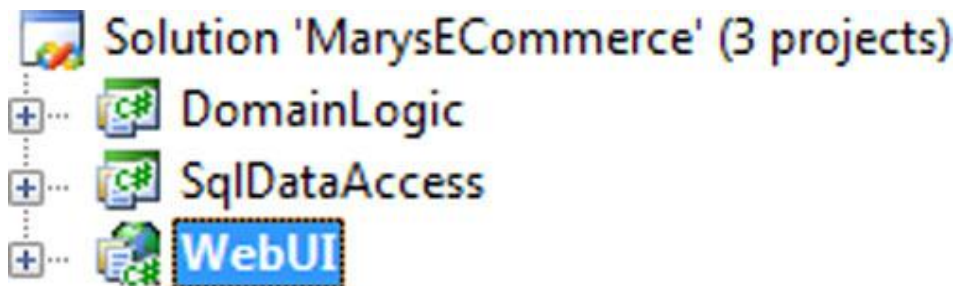
В этом случае фиксирование ошибки является прямолинейным. Мэри вставляет корректную строку соединения в файл `web.config`. При запуске приложения появляется продемонстрированная на рисунке 2-3 страница.

Возможность "Рекомендуемые товары" на данный момент реализована, и Мэри чувствует в себе уверенность и готовность реализовывать следующую возможность приложения. В конце концов, она последовала доказанным оптимальным методам и создала трехуровневое приложение.

Анализ

Удалось ли Мэри создать совершенное, многоуровневое приложение? Нет, не удалось – несмотря на то, что у нее наверняка были самые наилучшие намерения. Она создала три Visual Studio проекта, которые соответствуют трем уровням запланированной архитектуры, что продемонстрировано на рисунке 2-9. Для обычного наблюдателя такая реализация похожа на желаемую архитектуру, но как вы увидите, такой код является сильно связанным.

Рисунок 2-9: В веб-приложении электронной коммерции, созданном Мэри, присутствует по одному Visual Studio проекту для каждого уровня запланированной архитектуры – но является ли оно трехуровневым?



Visual Studio делает такой способ работы с решениями и проектами более простым и естественным. Если нам нужна функциональность из разных библиотек, то мы можем с легкостью добавить ссылку на нее и написать код, который создаст новые экземпляры

типов, определенных в этих библиотеках. Каждый раз при добавлении ссылки мы используем зависимость.

Диаграмма зависимостей

При работе с решениями в Visual Studio очень легко потерять путь перемещения важных зависимостей, поскольку Visual Studio показывает их вместе со всеми остальными ссылками проекта, которые могут указывать на сборки стандартной библиотеки классов .NET (BCL).

Для того чтобы понять, как модули приложения Мэри соотносятся друг с другом, мы можем нарисовать диаграмму зависимостей (см. рисунок 2-10).

Рисунок 2-10: Диаграмма зависимостей приложения Мэри, показывающая, каким образом модули зависят друг от друга. Стрелки указывают на зависимость модуля



Самым примечательным знанием, полученным из рисунка 2-10, является то, что библиотека пользовательского интерфейса (User Interface library) зависит как от доменной библиотеки (Domain library), так и от библиотеки доступа к данным (Data Access library). Кажется, что пользовательский интерфейс может обходиться в некоторых случаях без доменного уровня. Это предположение несет за собой дальнейшие исследования.

Анализ композиции

Главная цель создания трехуровневого приложения – разделить сущности. Нам бы хотелось отделить нашу доменную модель от уровня доступа к данным и уровня пользовательского интерфейса таким образом, чтобы ни одна из этих сущностей не засоряла доменную модель. Для больших приложений существенной является возможность работы с одной областью изолированно.

Для того чтобы оценить реализацию Мэри, мы можем задать простой вопрос:

Тест

Возможно ли использовать каждый модуль в изоляции от других модулей?

Теоретически мы должны иметь возможность компоновать модули так, как нам хочется. Возможно, нам понадобится написать новые модули для того, чтобы связывать

существующие модули вместе новыми и неожиданными способами, но в идеале, мы должны иметь возможность делать это без необходимости модифицировать существующие модули.

Примечание

Следующий анализ рассматривает вопрос того, можно ли заменять модули, но знайте, что это методика, которую мы используем для анализа композиции. Даже если нам никогда не нужно будет заменять модули, этот вид анализа раскрывает потенциальные вопросы, касающиеся связывания. Если мы обнаружим, что код сильно связан, то все преимущества слабого связывания будут потеряны.

Можем ли мы использовать модули приложения Мэри новыми и увлекательными способами? Давайте рассмотрим некоторые возможные сценарии.

Новый пользовательский интерфейс

Если приложение Мэри будет иметь успех, то заинтересованные стороны проекта захотят, чтобы она разработала полную клиентскую версию в системе построения клиентских приложений Windows – Windows Presentation Foundation (WPF). Возможно ли это сделать во время повторного использования доменного уровня и уровня доступа к данным?

При рассмотрении диаграммы зависимостей на рисунке 2-10 мы можем быстро определить, что ни один из модулей не зависит от пользовательского веб-интерфейса, поэтому можно удалить его и заменить на WPF пользовательский интерфейс.

Создание полной клиентской версии на базе WPF – это новое приложение, которое использует большинство своих реализаций совместно с первоначальным веб-приложением. Рисунок 2-11 иллюстрирует, как WPF приложению может понадобиться использовать те же самые зависимости, что и веб-приложение. Первоначальное веб-приложение может оставаться неизменным.

Рисунок 2-11: Замена пользовательского веб-интерфейса пользовательским WPF интерфейсом возможна, так как ни один из модулей не зависит от пользовательского веб-интерфейса. Первоначальный пользовательский веб-интерфейс на рисунке остается выделенным серым цветом для того, чтобы проиллюстрировать, что добавление нового пользовательского интерфейса не исключает первоначальный пользовательский интерфейс



Замена уровня пользовательского интерфейса наверняка возможна в реализации Мэри, поэтому давайте рассмотрим еще одну интересную декомпозицию.

Новый уровень доступа к данным

Представьте себе, что рыночные аналитики поймут, что для оптимизации прибыли приложение Мэри должно быть доступным в виде облачного приложения, размещенного на Windows Azure. В Windows Azure данные могут храниться в весьма масштабируемом сервисе Azure Table Storage Service. Этот механизм хранения данных основан на гибких data-контейнерах, которые содержат не свободные данные. Сервис не навязывает никакой конкретной схемы базы данных, и отсутствует соответствующая целостность.

Протокол, используемый для взаимодействия с Table Storage Service, – это HTTP, а наиболее очевидная .NET технология доступа к данным основывается на ADO.NET Data Services.

Такой тип базы данных иногда называют базы данных *ключ-значение*, и это несколько другой зверь, нежели реляционная база данных, доступ к которой осуществляется посредством Entity Framework.

Чтобы дать возможность использовать приложение электронной коммерции в виде облачного приложения, библиотеку доступа к данным можно заменить модулем, который использует Table Storage Service. Возможно ли это?

Из диаграммы зависимостей рисунка 2-10 мы уже знаем, что и библиотека пользовательского интерфейса, и доменная библиотека зависят от библиотеки доступа к данным, которая базируется на Entity Framework. Если мы попытаемся удалить библиотеку доступа к данным, то решение больше не будет компилироваться, поскольку отсутствует необходимая зависимость.

В большом приложении, состоящем из дюжины модулей, мы также могли бы попытаться удалить те модули, которые не компилируются, чтобы увидеть, что останется. Что касается приложения Мэри, то здесь очевидно, что нам пришлось бы удалить все модули, ничего при этом не оставляя.

Несмотря на то, что можно было бы разработать библиотеку Azure Table Data Access, которая имитирует API, используемое первоначальной библиотекой доступа к данным, нет ни одного способа, с помощью которого мы бы могли внедрить ее в приложение.

Приложение еще не настолько компонуемо, насколько того хотели бы заинтересованные стороны. Для разрешения облачных возможностей, которые максимально увеличивают прибыль, необходимо значительно переписать приложение, поскольку ни один из существующих модулей нельзя использовать повторно.

Рисунок 2-12: Попытка удаления реляционной библиотеки доступа к данным приводит к тому, что ничего не остается, потому что все остальные модули зависят от нее. Нет ни одного места, где мы могли бы дать доменной библиотеке указание использовать новую библиотеку Azure Table Data Access вместо первоначально используемой



Другие комбинации

Мы могли бы проанализировать приложение на возможность других комбинаций модуля, но это было бы спорным вопросом, поскольку мы уже знаем, что они не поддерживают важный сценарий.

Кроме того, не все комбинации имеют смысл. Мы могли бы поинтересоваться, можно ли заменить доменную модель другой реализацией. В большинстве случаев это было бы странным вопросом, потому что доменная модель инкапсулирует сердце приложения. Без доменной модели большинство приложений не имеют оснований для существования.

Анализ

Почему реализация Мэри не достигла желаемой степени компоновки? Не произошло ли это, потому что пользовательский интерфейс напрямую зависит от библиотеки доступа к данным? Давайте исследуем эту возможность более подробно.

Анализ диаграммы зависимостей

Почему пользовательский интерфейс зависит от библиотеки доступа к данным? Виновником этого является сигнатура метода данной доменной модели:

```
public IEnumerable<Product> GetFeaturedProducts(bool isCustomerPreferred)
```

Product: Раскрывает тип доступа к данным для клиентов

Метод `GetFeaturedProducts` возвращает последовательность товаров, но класс `Product` определен в библиотеке доступа к данным. Любой клиент, использующий метод `GetFeaturedProducts` должен ссылаться на библиотеку доступа к данным для возможности компиляции.

Можно изменить сигнатуру метода так, чтобы возвращалась последовательность типов, определенных в доменной модели. Такой подход был бы еще и более корректным, но это не решает имеющуюся проблему.

Давайте предположим, что мы разрушаем зависимость между библиотекой пользовательского интерфейса и библиотекой доступа к данным. Измененная диаграмма зависимостей выглядела бы сейчас так, как показано на рисунке 2-13.

Рисунок 2-13: Диаграмма зависимостей гипотетической ситуации, при которой разрывается зависимость пользовательского интерфейса от библиотеки доступа к данным



Позволило бы Мэри такое изменение заменить реляционную библиотеку доступа к данным такой библиотекой, которая инкапсулирует доступ к сервису Azure Table? К несчастью, не позволило бы, поскольку доменная библиотека все еще зависит от библиотеки доступа к данным. Пользовательский интерфейс, в свою очередь, все еще зависит от доменной модели, поэтому если мы попытаемся удалить первоначальную библиотеку доступа к данным, то в приложении ничего не останется.

Основная причина этой проблемы находится где-то в другом месте.

Анализ интерфейса доступа к данным

Доменная модель зависит от библиотеки доступа к данным, поскольку вся модель данных определена в этой библиотеке. Класс `Product` был сгенерирован, когда Мэри запускала мастер LINQ to Entities. Использование Entity Framework для реализации уровня доступа к данным может стать разумным решением.

Как бы то ни было, не используйте его напрямую в доменной модели.

Код, являющийся виновником возникшей проблемы, разбросан по классу `ProductService`. Конструктор создает новый экземпляр класса `CommerceObjectContext` и передает его в приватную переменную члена класса:

```
this.objectContext = new CommerceObjectContext();
```

Такой подход сильно связывает класс `ProductService` с библиотекой доступа к данным. Не существует разумного способа, с помощью которого мы могли бы перехватить этот фрагмент кода и заменить его чем-то еще. Ссылка на библиотеку доступа к данным жестко закодирована в классе `ProductService`.

Реализация метода `GetFeaturedProducts` использует `CommerceObjectContext` для того, чтобы вытянуть объекты `Product` из базы данных.

```
var products = (from p in this.objectContext.Products
                where p.IsFeatured
                select p).AsEnumerable();
```

Это только усиливает жестко закодированную зависимость, но на данном этапе повреждения уже нанесены. Все что нам нужно – это более подходящий способ компоновки модулей без использования такого сильного связывания.

Прочие вопросы

Перед тем как показать вам лучшую альтернативу, мне хотелось бы обратить ваше внимание на некоторые другие вопросы, связанные с кодом Мэри, к которым обязательно нужно обратиться.

- Кажется, что большинство доменных моделей реализуются в библиотеке доступа к данным. Несмотря на то, что тот факт, что библиотека доменной модели ссылается на библиотеку доступа к данным, является технической проблемой, то, что библиотека доступа к данным определяет такой класс, как класс `Product`, что является, в свою очередь, концептуальной проблемой. Открытый класс `Product` принадлежит к доменной модели.
- Оказавшись под влиянием Дженса, Мэри решила реализовать код, который определяет, является ли пользователь привилегированным покупателем в пользовательском интерфейсе. Тем не менее, то, каким образом покупатель идентифицируется в качестве привилегированного, является фрагментом бизнес-логики, поэтому это должно быть реализовано в доменной модели.

Аргумент Дженса в пользу концепции разделения и принципа единственной ответственности никак не оправдывает размещение кода в неправильном месте. Следование принципу единственной ответственности в рамках единичной библиотеки вполне возможно – это ожидаемый подход.

- Класс `ProductService` полагается на XML конфигурацию. Как вы видели при наблюдении за стараниями Мэри, она забыла, что ей пришлось поместить часть конфигурационного кода в ее файл `web.config`. Несмотря на то, что способность конфигурирования компилируемого приложения является важной, только окончательное приложение должно полагаться на конфигурационные файлы. Удобнее, когда повторно используемые библиотеки обязательно конфигурируются вызывающими их объектами.

В конце концов, конечный вызывающий объект сам по себе является приложением. На данном этапе все соответствующие конфигурационные данные можно прочитать из `.config` файла и загрузить при необходимости в основные библиотеки.

- Кажется, что представление (продемонстрированное в листинге 2-3) содержит слишком много функциональности. Оно выполняет расчеты и конкретное строковое форматирование. Такая функциональность должна быть перемещена в основную модель.

В следующем разделе я продемонстрирую вам более скомпонованный способ создания приложения с теми же самыми возможностями, которые создавала Мэри. Кроме того, я также обращаюсь к этим второстепенным вопросам.

2.2. Как действовать правильно

Механизм внедрения зависимостей (DI) можно было использовать для решения тех вопросов, которые мы обсуждали. Поскольку DI – это радикальное отклонение от того способа, с помощью которого Мэри создавала свое приложение, я не собираюсь его изменять. Вместо этого я собираюсь заново создать его с самого начала.

Вам не следует делать из этого решения вывод о том, что можно выполнить рефакторинг существующего приложения относительно DI; это можно сделать, но это очень сложно. По моему опыту для этого потребуется много раз выполнять рефакторинг.

Примечание

Пока я буду вести вас этому примеру, не волнуйтесь, если вы вдруг потеряетесь по пути. Механизм внедрения зависимостей очень сложный, и в нем множество элементов. Я выбрал этот пример, потому что он похож на реалистичный сценарий, но недостаток этого примера – это то, что он сложнее, чем выдуманный пример. В дальнейшем в этой книге я собираюсь погрузиться глубже в те сущности и методики, которые были введены здесь. После того, как вы прочитаете больше информации, вы всегда сможете вернуться обратно и заново прочитать этот раздел.

Многие люди называют механизм внедрения зависимостей инверсией управления (Inversion of Control). Иногда эти термины используются как взаимозаменяемые, но DI – это подмножество инверсии управления. На протяжении всей этой книги я буду последовательно использовать самый конкретный термин: DI. Если я буду иметь в виду инверсию управления, то я буду говорить об этом конкретно.

Внедрение зависимостей или Инверсия управления?

Термин *инверсия управления* (IoC) первоначально означал любой вид стиля программирования, в котором полноценный фреймворк или исполняющая среда контролировали ход выполнения программы. Согласно этому определению большинство программного обеспечения, разработанного на .NET Framework, использует принцип инверсии управления.

При написании ASP.NET приложения вы попадаете в жизненный цикл ASP.NET страницы, но вы ничем не управляете, всем управляет ASP.NET.

При написании WCF сервиса вы реализуете интерфейсы, помеченные атрибутами. Вы можете писать код сервиса, но, в конце концов, вы ничем не управляете, всем управляет WCF.

В настоящее время мы настолько привыкли работать с фреймворками, что уже не считаем их особенными, но эта модель уже не находится под полным вашим контролем. Это все еще может случиться и с .NET приложением – наиболее заметно при выполнении команд командной строки. Как только вызывается `Main`, ваш код приобретает полный контроль. Он контролирует ход выполнения программы, жизненный цикл, в общем все. Никаких особенных событий не происходит, и никакие переопределенные члены не вызываются.

До того момента, как механизм внедрения зависимостей приобрел свое название, люди начинали называть фреймворки, которые управляли зависимостями, *IoC-контейнерами*, и

вскоре смысл IoC постепенно сместился к конкретному значению: инверсия управления зависимостями. Будучи постоянным систематиком, Мартин Фаулер ввел термин *внедрение зависимости* для того, чтобы в частности сослаться на IoC в контексте управления зависимостями. С тех пор *внедрение зависимости* широко применяется в качестве самого корректного термина.

Короче говоря, инверсия управления – более широкий термин, который содержит в себе термин внедрение зависимостей, но не ограничивается им.

В контексте управления зависимостями инверсия управления в точности описывает то, что мы пытаемся выполнить. В приложении Мэри код напрямую управляет его зависимостями: когда ProductService требуется новый экземпляр класса CommerceObjectContext, он просто создает экземпляр с помощью ключевого слова new. Когда HomeController требуется новый экземпляр класса ProductService, он также создает новый экземпляр с помощью ключевого слова new. Приложение находится под тотальным контролем. Это может звучать сильно, но в действительности контроль ограничен. Я называю это анти-паттерном Control Freak (*прим. ред.* – термин в психологии характеризует человека, который пытается диктовать всем вокруг как по его мнению, должно все происходить. В переводе, с точки зрения анти-паттерна, упоминается как Руководитель-наркоман, но мы оставим английский вариант). Инверсия управления дает нам указание отпустить этот контроль и позволить еще кому-то управлять зависимостями.

Создание коммерческого приложения заново

При написании программного обеспечения я предпочитаю приступить к этому в наиболее значимом месте. Чаще всего это пользовательский интерфейс. Поэтому я работаю, добавляя еще больше функциональности до тех пор, пока возможность не осуществлена, и я могу перейти к следующей возможности. Эта методика "*снаружи-внутри*" помогает мне сфокусироваться на запрашиваемой функциональности без необходимости заново перестраивать приложение.

Примечание

Методика "*снаружи-внутри*" тесно связана с принципом YAGNI ("You Aren't Gonna Need It") – "Вам это не понадобится". Этот принцип подчеркивает, что необходимо реализовывать только требуемые возможности, и что реализация должна быть настолько простой, насколько это возможно.

Так как я всегда практикую разработку через тестирование (TDD), я начинаю писать модульные тесты, как только используемый мной подход "*снаружи-внутри*" начинает накатывать мне на создание нового класса. Несмотря на то, что я написал множество модульных тестов, чтобы создать этот пример, разработка через тестирование не требует реализации и использования механизма внедрения зависимостей, поэтому я не собираюсь демонстрировать эти тесты в моей книге. Если вам интересно, то эти тесты доступны в исходном коде, который приложен к этой книге.

Пользовательский интерфейс

Необходимым условием для списка рекомендуемых товаров является написание приложения, которое извлекает рекомендуемые товары из базы данных и отображает их в списке, как это показано на рисунке 2-3. Поскольку я знаю, что заинтересованным

сторонам проекта, главным образом, будет интересен визуальный результат, то лучше всего начать с пользовательского интерфейса.

Первое, что я делаю после открытия Visual Studio – добавляю в мое решение новое ASP.NET MVC приложение. Так как список рекомендуемых товаров должен продолжать титульную страницу, я начинаю с изменения `Index.aspx` таким образом, чтобы она содержала разметку, продемонстрированную в следующем листинге.

Листинг 2-4: Разметка `Index` представления

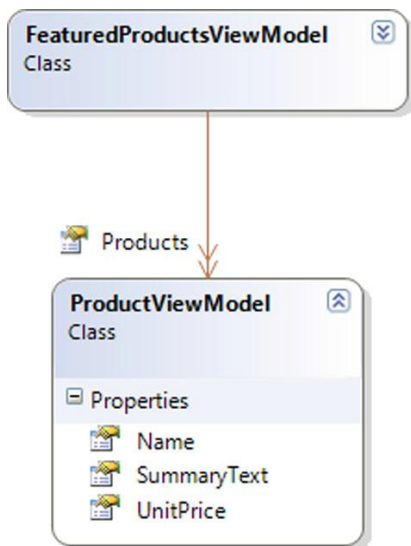
```
<h2>Featured Products</h2>
<div>
  <% foreach (var product in this.Model.Products)
  { %>
    <div><%= this.Html.Encode(product.SummaryText) %></div>
  <% } %>
</div>
```

Заметьте, насколько чище листинг 2-4 по сравнению с листингом 2-3. Первое усовершенствование – это то, что больше не нужно приводить элемент словаря к последовательности товаров до того, как итерация станет возможной. Я с легкостью выполнил это, разрешив странице `Index.aspx` наследоваться от `System.Web.Mvc.ViewPage<FeaturedProductsViewModel>`, а не от `System.Web.Mvc.ViewPage`. Это означает, что свойство `Model` страницы имеет тип `FeaturedProductsViewModel`.

Вся строка отображения товара берется прямо из свойства `SummaryText` этого товара.

Оба усовершенствования связаны с введением моделей конкретных представлений, которые инкапсулируют поведение представления. Эти модели являются РОСО-объектами (Plain Old CLR Objects). Рисунок 2-14 предоставляет обзор структуры таких объектов.

Рисунок 2-14: `FeaturedProductsViewModel` содержит список `ProductViewModels`. И `FeaturedProductsViewModel`, и `ProductViewModel` являются РОСО-объектами, что делает их в высокой степени доступными для модульного тестирования. Свойство `SummaryText` получено из свойств `Name` и `UnitPrice` для того, чтобы инкапсулировать логику отображения.



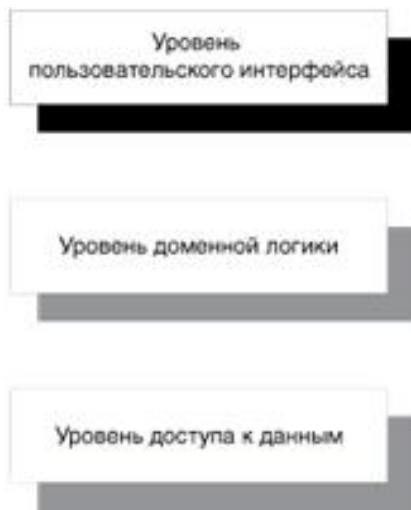
Чтобы код из листинга 2-4 был рабочим, HomeController должен возвращать представление с экземпляром FeaturedProductsViewModel. Например, на первом шаге он может быть реализован внутри HomeController, как это продемонстрировано ниже:

```
public ActionResult Index()
{
    var vm = new FeaturedProductsViewModel();
    return View(vm);
}
```

Это позволит веб-приложению выполняться без ошибок, но список рекомендуемых товаров всегда будет пустым. Предоставление списка рекомендуемых товаров – это задача доменной модели.

Рисунок 2-15 демонстрирует текущее состояние реализации архитектуры, показанной на рисунке 2-2.

Рисунок 2-15: На данном этапе реализован только уровень пользовательского интерфейса, осталось реализовать уровни доменной логики и доступа к данным. Сравним этот рисунок с рисунком 2-6, который демонстрирует успехи Мэри на этом же самом этапе. Одним из преимуществ того, что мы начинаем создание приложения с пользовательского интерфейса, является то, что мы уже получаем приложение, которое можно запускать и тестировать. Только на более позднем этапе, продемонстрированном на рисунке 2-8, Мэри приходит к тому моменту, когда она может запускать и тестировать свое приложение.



Несмотря на то, что пользовательский интерфейс теперь существует, он не представляет особого интереса. Список рекомендуемых товаров всегда пуст, поэтому мне нужно реализовать некоторую доменную логику, которая сможет обеспечить должный список товаров.

Доменная модель

Доменная модель – это обычная, несложная C# библиотека, которую я добавляю в свое решение. Эта библиотека будет содержать РОСО-объекты и абстрактные типы РОСО-объекты будут моделировать домен, в то время, как абстрактные типы обеспечивают абстракции, которые будут выступать в роли моих основных внешних записей в доменной модели.

Принцип программирования на основании интерфейсов, а не конкретных классов – это основа механизма внедрения зависимостей. Именно этот принцип позволяет нам заменять одну реализацию другой.

Интерфейсы или абстрактные классы?

Многие руководства по объектно-ориентированному проектированию фокусируются на интерфейсах как на главном механизме абстракций, в то время, как руководства по проектированию на базе .NET Framework поддерживают превосходство абстрактных классов над интерфейсами. Следует ли вам использовать интерфейсы или же все-таки абстрактные классы?

Относительно механизма внедрения зависимостей утешительный ответ на данный вопрос – не имеет значения, что вы будете использовать. Важно лишь то, что вы программируете относительно некоторого рода абстракции.

Выбор между интерфейсами и абстрактными классами важен в других контекстах, но не в данном. Вы заметите, что я использую эти понятия взаимозаменяемо; я часто использую термин "абстракция", чтобы объединить оба этих термина "интерфейсы" и абстрактные классы.

Я все еще руководствуюсь подходом "снаружи-внутри", поэтому я на некоторое время добавлю код в уровень пользовательского интерфейса. Некоторый добавленный мной код

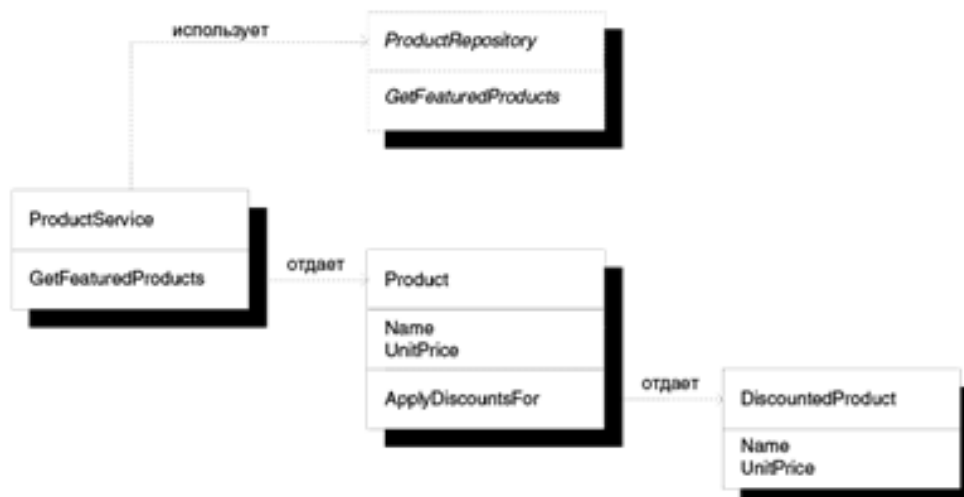
будет использовать типы из доменной модели. Это означает, что я добавлю ссылку на доменную модель из пользовательского интерфейса, как это делала Мэри. Все получится отлично, но я отложу анализ диаграммы зависимостей до раздела "Анализ слабо связанной реализации", чтобы я смог предоставить вам полную картину.

Универсальная абстракция доступа к данным обеспечивается паттерном **Repository**, поэтому я определяю абстрактный класс `ProductRepository` в библиотеке доменной модели.

```
public abstract class ProductRepository
{
    public abstract IEnumerable<Product> GetFeaturedProducts();
}
```

Готовый `Repository` будет иметь больше методов для поиска и изменения товаров, но следуя принципу "снаружи-внутри", я определяю только классы и члены, которые мне нужны для текущей задачи. Проще добавить функциональность в код, чем удалить все. Класс `Product` также реализован с самым минимальным количеством членов, что проиллюстрировано на рисунке 2-16.

Рисунок 2-16: Класс `Product` содержит только свойства `Name` и `UnitPrice`, поскольку это единственные свойства, которые нужны для реализации желаемой возможности приложения. `ApplyDiscountFor` применяет в случае необходимости предоставить скидку для пользователя и возвращает экземпляр класса `DiscountedProduct`. Абстрактный `GetFeaturedProducts` возвращает последовательность `Products`.



Метод `Index HomeController` должен использовать экземпляр `ProductService` для того, чтобы извлечь список рекомендуемых товаров, применить какую-либо скидку,

сконвертировать экземпляры `Product` в экземпляры `ProductViewModel`, затем добавить их в `FeaturedProductsViewModel`. Поскольку класс `ProductService` принимает в свой конструктор экземпляр `ProductRepository`, самое сложное – обеспечить его соответствующим экземпляром. Вспомните из анализа реализации Мэри, что создание зависимостей при помощи ключевого слова `new` является неправильным. Как только я это сделаю, я окажусь сильно связанным с типом, который я только что использовал.

Я собираюсь отказаться от контроля над зависимостью `ProductRepository`. Как продемонстрировано в следующем листинге, я лучше буду полагаться на что-нибудь другое, чтобы получить экземпляр с помощью конструктора `HomeController`. Этот паттерн называется `Constructor Injection` (внедрение через конструктор) – как и кем создается экземпляр не является заботой `HomeController`.

Листинг 2-5: `HomeController` с паттерном `Constructor Injection`

```
1. public partial class HomeController : Controller
2. {
3.     private readonly ProductRepository repository;
4.
5.     public HomeController(ProductRepository repository)
6.     {
7.         if (repository == null)
8.         {
9.             throw new ArgumentNullException("repository");
10.        }
11.        this.repository = repository;
12.    }
13.
14.    public ActionResult Index()
15.    {
16.        var productService = new ProductService(this.repository);
17.
18.        var vm = new FeaturedProductsViewModel();
19.
20.        var products = productService.GetFeaturedProducts(this.User);
21.        foreach (var product in products)
22.        {
23.            var productVM = new ProductViewModel(product);
24.            vm.Products.Add(productVM);
25.        }
26.        return View(vm);
27.    }
28. }
```

Строка 5: Внедрение через конструктор

Строка 11: Сохраняет внедренную зависимость для дальнейшего использования

Строка 16: Передает внедренную зависимость

Конструктор `HomeController` указывает, что каждый, кому захочется использовать класс, должен предоставить экземпляр `ProductRepository` (который, как вы, возможно, помните, является абстрактным классом). Ограждающее условие гарантирует выполнение путем генерирования исключения в случае, если используемый экземпляр пуст. Внедренная зависимость может быть сохранена и в дальнейшем безопасно использоваться другими членами класса `HomeController`.

В первый раз, когда я услышал о внедрении через конструктор, мне понадобилось много времени на то, чтобы осознать реальную пользу этого паттерна. Не приведет ли использование такого подхода к тому, что вся тяжесть управления зависимостями ляжет на другой класс?

Да, приведет – и в этом-то все дело. В *n*-уровневых приложениях мы можем вынести управление зависимостями любыми способами на самый верхний уровень приложения – в **Composition Root**. Это централизованное место, в котором могут компоноваться различные модули приложения. Это можно сделать вручную или же делегировать эту задачу DI-контейнеру.

HomeController делегирует большую часть своей работы классу ProductService, продемонстрированному в следующем листинге. Класс ProductService соответствует классу Мэри с тем же именем, но на данный момент он является подлинным классом доменной модели.

Листинг 2-6: Класс ProductService

```
1. public class ProductService
2. {
3.     private readonly ProductRepository repository;
4.
5.     public ProductService(ProductRepository repository)
6.     {
7.         if (repository == null)
8.         {
9.             throw new ArgumentNullException("repository");
10.        }
11.        this.repository = repository;
12.    }
13.
14.    public IEnumerable<DiscountedProduct> GetFeaturedProducts(IPrincipal
15.        user)
16.    {
17.        if (user == null)
18.        {
19.            throw new ArgumentNullException("user");
20.        }
21.        return from p in this.repository.GetFeaturedProducts()
22.               select p.ApplyDiscountFor(user);
23.    }
24. }
```

Строка 5: И снова используем Constructor Injection

Строка 14: Паттерн Method Injection

Строка 21-22: Используем обе внедренные зависимости для реализации поведения

Метод GetFeaturedProducts теперь принимает в качестве параметра экземпляр IPrincipal, который представляет собой текущего пользователя. Это еще одно отклонение от реализации Мэри, показанной в листинге 2-1, которая всего лишь принимала в качестве параметра булево значение, указывая, является ли пользователь привилегированным. Тем не менее, поскольку решение о том, является ли пользователем привилегированным, – это часть доменной логики, корректнее будет явно смоделировать

текущего пользователя в виде зависимости. Мы должны всегда соблюдать принцип *программирования на основании интерфейса*, но в данном случае мне не нужно ничего изобретать (как я это делал с `ProductRepository`), потому что в библиотеку стандартных классов .NET уже входит интерфейс `IPrincipal`, который представляет собой стандартный способ моделирования пользователей приложения.

Передача зависимости в метод в качестве параметра называется внедрением через параметры метода (Method Injection). И снова управление делегируется вызывающему оператору, подобно внедрению через конструктор. Несмотря на то, что детали могут варьироваться, основная технология остается той же.

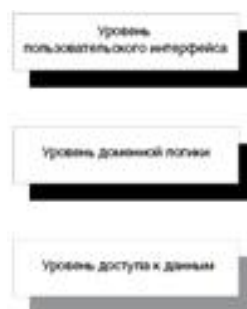
На данном этапе приложение все еще не работает. Осталось две проблемы:

- Отсутствует конкретная реализация `ProductRepository`. Эта проблема легко решается. В следующем разделе я буду реализовывать конкретный `ProductRepository`, который считывает рекомендуемые товары из базы данных.
- По умолчанию ASP.NET MVC предполагает, что контроллеры обладают конструкторами по умолчанию. Поскольку я ввел параметр в конструктор `HomeController`, MVC framework не знает, как создать экземпляр `HomeController`. Эта проблема может быть решена путем создания пользовательского `IControllerFactory`. То, как это делается, выходит за рамки данной главы, но этот вопрос будет обсуждаться в главе 8. Достаточно просто сказать, что эта пользовательская фабрика создаст экземпляр конкретного `ProductRepository` и передаст его в конструктор `HomeController`.

В доменной модели я работаю только с типами, определенными в рамках доменной модели (и библиотеки стандартных классов .NET). Сущности доменной модели реализуются в виде POCO-объектов. На данном этапе существует одна единственная представленная сущность, названная `Product`. Доменная модель должна уметь взаимодействовать с внешним миром (например, с базами данных). Эта необходимость смоделирована в виде абстрактных классов (например, `Repositories`), которые мы должны заменить конкретными реализациями перед тем, как доменная модель станет полезной.

Рисунок 2-17 демонстрирует текущее состояние реализации архитектуры, продемонстрированной на рисунке 2-2.

Рисунок 2-17: Уровень пользовательского интерфейса и уровень доменной логики уже реализованы, остается реализовать уровень доступа к данным. Сравните этот рисунок с рисунком 2-7, который демонстрирует успехи Мэри на данном этапе.



Доменная модель приложения еще не является объектно-ориентированной; чтобы завершить цикл, мне нужно реализовать только один абстрактный `ProductRepository`.

Доступ к данным

Как и Мэри, мне хотелось бы реализовать мою библиотеку доступа к данным при помощи технологии LINQ to Entities, поэтому я последую тем же шагам, которые она выполняла в разделе "Создание сильно связанного приложения" при создании модели. Главное отличие – модель и `CommerceObjectContext` теперь являются всего лишь деталями реализации; но с помощью них я могу создать реализацию `ProductRepository`, что продемонстрировано в следующем листинге.

Листинг 2-7: Реализация `ProductRepository` с помощью LINQ to Entities

```
1. public class SqlProductRepository : Domain.ProductRepository
2. {
3.     private readonly CommerceObjectContext context;
4.
5.     public SqlProductRepository(string connString)
6.     {
7.         this.context = new CommerceObjectContext(connString);
8.     }
9.
10.    public override IEnumerable<Domain.Product> GetFeaturedProducts()
11.    {
12.        var products = (from p in this.context.Products
13.                        where p.IsFeatured
14.                        select p).AsEnumerable();
15.        return from p in products
16.               select p.ToDomainProduct();
17.    }
18. }
```

Строка 16: Конвертирует в Domain Product

В приложении Мэри сгенерированная сущность `Product` использовалась в качестве доменного объекта, несмотря на то, что она была определена в базе данных. Так больше не происходит, потому что я уже определил класс `Product` в доменной модели. Когда я сгенерировал модуль, мастер создал для меня еще один класс `Product` и мне нужно выполнить конвертацию между этими классами. Рисунок 2-18 иллюстрирует, каким образом они определены в разных модулях. Существующий класс `Product` – это всего лишь деталь реализации, и я мог бы с легкостью сделать его внутренним, чтобы выразить его более явно.

Рисунок 2-18: И библиотека доменной модели, и библиотека доступа к данным определяют класс под названием `Product`. Доменный класс `Product` – это важный класс, который инкапсулирует доменную сущность – товар. Класс `Product` библиотеки доступа к данным – это всего лишь искусственный объект мастера Entity Framework. Его можно легко переименовать или сделать внутренним.



Примечание

Вы можете поспорить, что то, что EntityFramework не поддерживает неграмотные сущности, является его определенным дефектом (по крайней мере, не в версии .NET 3.5 SP1). Тем не менее, это вид ограничения, с которым вы обязательно столкнетесь в реальных проектах программного обеспечения.

Существующий `Product` задает конвертацию в доменный тип `Product`. Эта конвертация является обычным преобразованием соответствующих значений. Несмотря на то, что такая конвертация не относится конкретно к механизму внедрения зависимостей, я использую ее для более удобных измерений.

```
Domain.Product p = new Domain.Product();  
p.Name = this.Name;  
p.UnitPrice = this.UnitPrice;  
return p;
```

После реализации `SqlProductRepository` я теперь могу настроить ASP.NET MVC для того, чтобы внедрить экземпляр `SqlProductRepository` в экземпляры `HomeController`. Поскольку я более подробно буду рассматривать это в главе 7, я не буду демонстрировать это здесь.

Рисунок 2-19 демонстрирует текущее состояние архитектуры приложения, продемонстрированной на рисунке 2-2.

Рисунок 2-19: Все три уровня приложения на данный момент реализованы так, как это продемонстрировано на рисунке 2-2. Этот рисунок идентичен рисунку 2-2, но повторяется здесь для иллюстрации текущего состояния приложения. Рисунок также идентичен рисунку 2-8, который демонстрирует завершенное приложение Мэри.



Теперь, когда все корректно соединено вместе, я могу перейти к домашней странице приложения и получить такую же страницу, как продемонстрированная на рисунке 2-3.

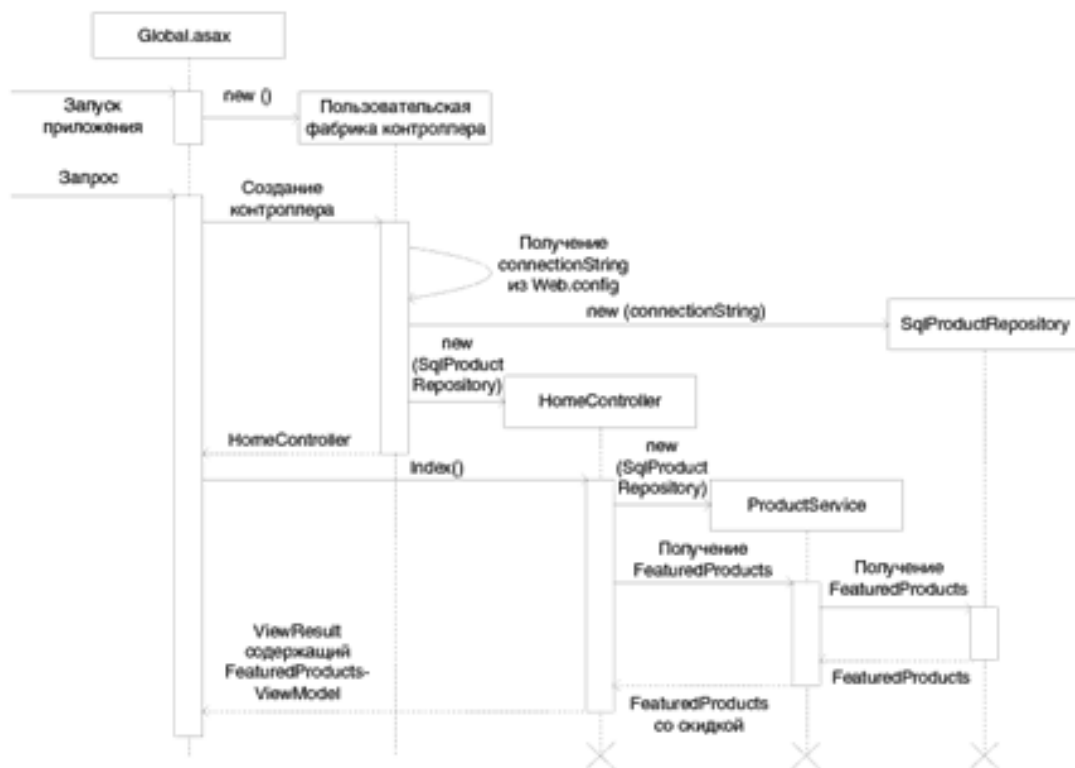
Анализ слабо связанной реализации

Предыдущий раздел содержал множество подробной информации, поэтому вряд ли вас удивит то, что вы потеряли общее представление о происходящем. В этом разделе я попытаюсь объяснить то, что произошло, с помощью более ясных терминов.

Взаимодействие

Классы каждого уровня взаимодействуют друг с другом напрямую или в абстрактной форме. Таким образом, они работают за пределами модулей, поэтому трудно будет проследить процесс их взаимодействия. Рисунок 2-20 иллюстрирует, как соединяются зависимости.

Рисунок 2-20: Взаимодействие между элементами, включенными в механизм внедрения зависимостей в коммерческом приложении. Заметьте, что экземпляр `SqlProductRepository` внедряется в `HomeController`, а затем спустя некоторое время через `HomeController` внедряется в `ProductService`, который, в конечном счете, использует его.



При запуске приложения код файла `Global.asax` создает новую пользовательскую фабрику контроллеров. Приложение сохраняет ссылку на фабрику контроллеров, поэтому, когда выполняется запрос страницы, приложение вызывает `CreateController` для фабрики. Фабрика ищет строку соединения в файле `web.config` и передает ее в новый экземпляр `SqlProductRepository`. Фабрика внедряет экземпляр `SqlProductRepository` в новый экземпляр `HomeController`, а затем возвращает этот экземпляр.

Затем приложение вызывает метод `Index` для экземпляра `HomeController`, заставляя его создать новый экземпляр `ProductService`, передавая экземпляр `SqlProductRepository` в его конструктор. `ProductService` вызывает метод `GetFeaturedProducts` для экземпляра `SqlProductRepository`.

В конце концов, возвращается `ViewResult` с заполненным `FeaturedProductsViewModel`, а ASP.NET MVC фреймворк находит и отображает корректную страницу.

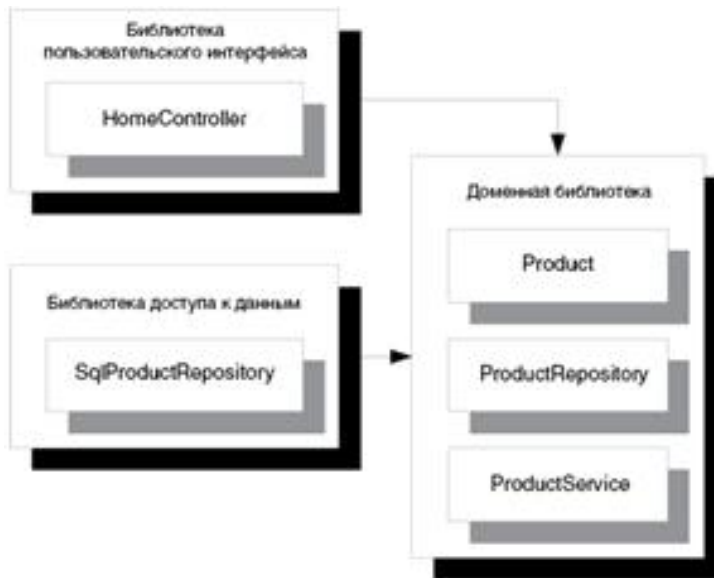
Диаграмма зависимостей

В разделе "Анализ" мы увидели, как диаграмма зависимостей может помочь нам проанализировать и понять степень гибкости, которую обеспечивает архитектурная реализация. Изменил ли механизм внедрения зависимостей диаграмму зависимостей приложения?

Рисунок 2-21 демонстрирует, что диаграмма зависимостей действительно изменилась. Доменная модель больше не имеет зависимостей и может выступать в роли автономного модуля. С другой стороны, у библиотеки доступа к данным теперь есть зависимость; в приложение Мэри у нее не было зависимостей.

Рисунок 2-21: Диаграмма зависимостей, которая демонстрирует пример коммерческого приложения при использовании механизма внедрения зависимостей. Самое

примечательное отличие – это то, что у доменной библиотеки больше нет зависимостей. Серые блоки внутри черных блоков демонстрируют примерные классы каждой библиотеки для того, чтобы дать вам представление о том, где какие классы должны быть. В каждой библиотеке присутствует больше классов, чем продемонстрировано на этом рисунке.



Это пробуждает наши надежды на то, что мы более плодотворно сможем ответить на первоначальные вопросы о композиции:

- Можем ли мы заменить пользовательский веб-интерфейс пользовательским интерфейсом WPF? Такая замена была возможна до этого и остается возможной в рамках нового проектирования. Ни библиотека доменной модели, ни библиотека доступа к данным не зависят от пользовательского веб-интерфейса, поэтому мы можем с легкостью поместить на его место что-то другое.
- Можем ли мы заменить реляционную библиотеку доступа к данным библиотекой, которая работает с сервисом Azure Table Service? В главе 3 я буду описывать, как приложение размещает и создает экземпляры корректного `ProductRepository`, поэтому на данный момент примите следующее как данность: библиотека доступа к данным загружается с помощью позднего связывания, а имя типа определяется в виде настройки приложения в файле `web.config`. Можно удалить текущую библиотеку доступа к данным и внедрить новую библиотеку, поскольку она также предоставляет реализацию `ProductRepository`.

Нельзя больше использовать текущую библиотеку доступа к данным изолированно, поскольку она теперь зависит от доменной модели. Во многих видах приложений это не является проблемой, но если заинтересованные стороны захотят, чтобы такая возможность была реализована, я могу решить эту проблему путем добавления еще одного уровня абстракции: с помощью извлечения интерфейса из `Product` (скажем, `IProduct`) и изменения `ProductRepository` для того, чтобы он работал с `IProduct`, а не с `Product`. Эти абстракции потом можно переместить в отдельную библиотеку, которая используется совместно библиотекой доступа к данным и доменной моделью. Это потребовало бы больших затрат времени, поскольку мне понадобилось бы писать код для преобразования `Product` в `IProduct`, но это все-таки возможно.

Благодаря проектированию на основании механизма внедрения зависимостей первоначальное веб-приложение может постепенно быть преобразовано в приложение Software + Services (доступ к программному обеспечению предоставляется заказчику через интернет) с богатым WPF интерфейсом и облачным движком хранилища данных. Единственное, что осталось от первоначальной работы – это доменная модель, но это целесообразно, поскольку доменная модель инкапсулирует все важные бизнес-правила, и по существу, нам следует ожидать, что она будет самым существенным модулем.

При разработке приложений мы вероятно, не можем предугадать каждое будущее направление, которое может нам понадобиться для того, чтобы получить товар, но это не проблема, поскольку мы можем хранить наши варианты открытыми для доступа. Механизм внедрения зависимостей помогает нам создавать слабо связанные приложения для того, чтобы мы могли заново использовать или заменять различные модули необходимыми модулями.

2.3. Расширение шаблонного приложения

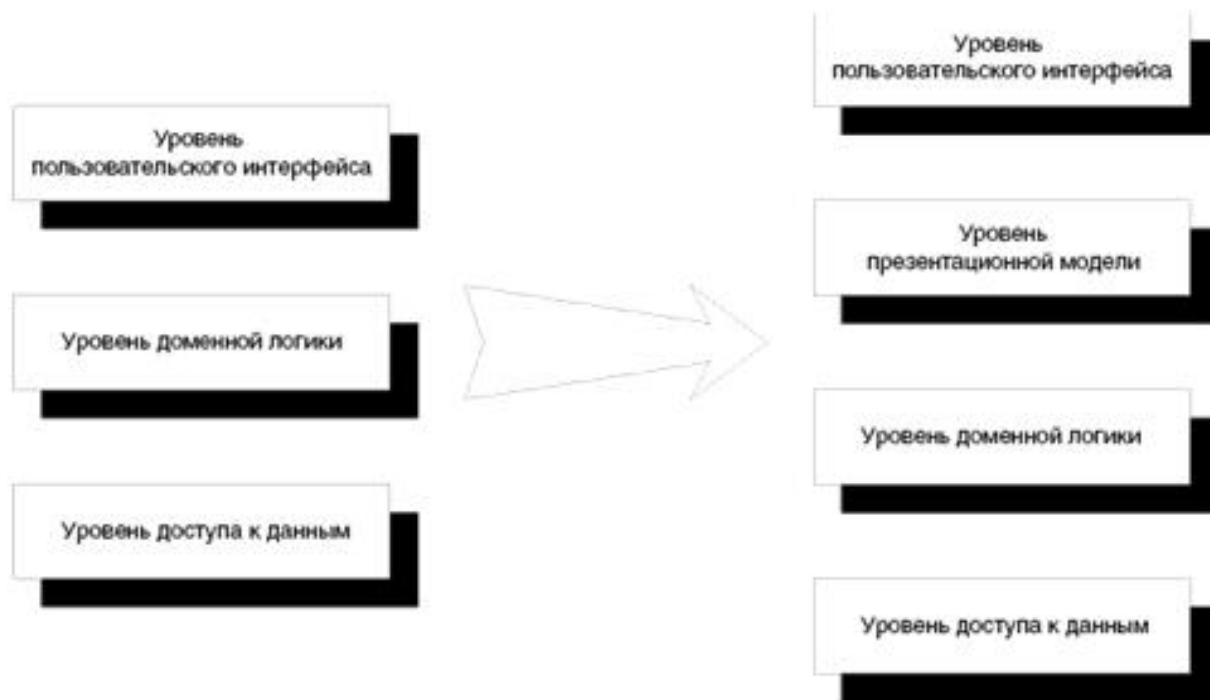
Чтобы поддержать оставшуюся часть книги и полностью продемонстрировать различные аспекты механизма внедрения зависимостей, мне понадобится расширить шаблонное коммерческое приложение. До настоящего момента я сохранял приложение настолько простым и небольшим по размеру, насколько это было возможно, чтобы осторожно ввести некоторые центральные сущности и принципы. Поскольку одной из главных целей механизма внедрения зависимостей является управление сложностью, нам нужно сложное приложение, чтобы полностью оценить его силу.

Я буду расширять приложение относительно двух аспектов: архитектурный рефакторинг и добавленные возможности.

Архитектура

До настоящего времени шаблонное приложение было трехуровневым приложением, но сейчас я хочу всунуть уровень презентационной модели между пользовательским интерфейсом и доменной моделью, как это показано на рисунке 2-22.

Рисунок 2-22: Уровень презентационной модели вставляется в шаблонное приложение для того, чтобы отделить логику представления от центральной части приложения.



Я переместил все контроллеры и модели представлений из уровня пользовательского интерфейса в уровень презентационной модели, оставив в уровне пользовательского интерфейса только представления (файлы `.aspx` и `.ascx`) и `Composition Root`.

Главная причина такого перемещения – отделить `Composition Root` от логики представления; таким образом, я могу продемонстрировать вам различные варианты конфигурационных стилей, оставляя при этом неизменной настолько большую часть приложения, насколько это возможно.

"Скромный" объект

Я разделяю приложение на уровень пользовательского интерфейса и уровень презентационной модели не только для образовательных целей; я регулярно проделываю это для всех разрабатываемых мной приложений, если они имеют пользовательский интерфейс.

Такое разделение обеспечивает понятную концепцию разделения логики представления (как ведут себя пользовательские интерфейсы) и логики отображения (как выглядят пользовательские интерфейсы). Благодаря такой концепции разделения вся логика помещается в уровень, где ее можно тестировать модульно, а вся разметка помещается в уровень, где может работать графический дизайнер, не боясь при этом что-то легко разрушить.

Цель – получить настолько небольшой по размеру императивный код, насколько это возможно на уровне пользовательского интерфейса, поскольку я не собираюсь писать никаких модульных тестов для этого уровня.

Ядро приложения, которое содержит только достаточный минимум кода для начальной загрузки самого себя, после чего оно делегирует всю остальную часть работы тестируемым модулям, называется скромным объектом (Humble object). В данном случае он содержит только представления и код начальной загрузки: Composition Root.

Помимо этой архитектурной модификации я также хочу добавить более расширенную возможность, нежели та, что мы рассматривали до настоящего момента.

Возможность добавления в корзину

Список рекомендуемых товаров знакомит нас только с ограниченным уровнем сложности: в сценарий, доступный только для чтения, вовлечен единственный репозиторий.

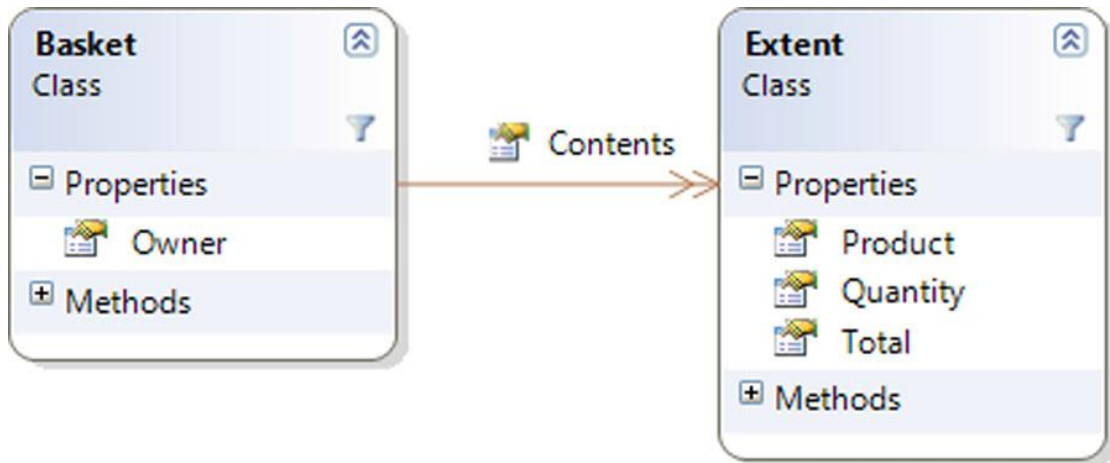
Следующий логичный шаг – ввести возможность добавления покупок в корзину. Рисунок 2-23 демонстрирует скриншот использования корзины покупок.

Рисунок 2-23: Впечатляюще скудная по возможностям корзина покупок в реорганизованном коммерческом шаблонном приложении.



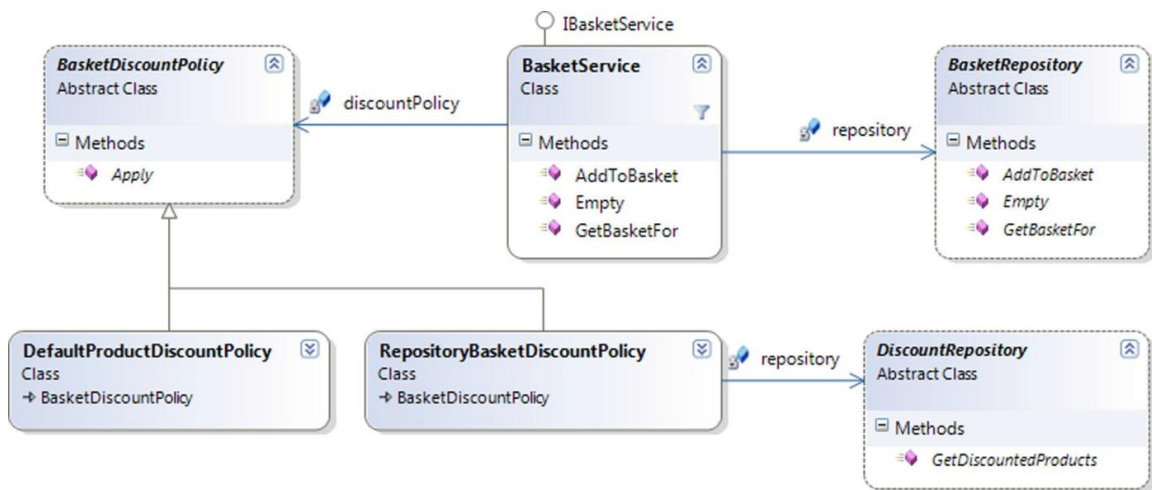
Чтобы обеспечить корзину покупок для каждого пользователя, мне понадобятся `Basket`, `BasketRepository` и хост поддерживаемых классов. Если вы похожи на меня, то вы захотите сначала увидеть класс `Basket`: рисунок 2-24 демонстрирует корзину и список ее элементов.

Рисунок 2-24: Корзина и ее содержимое, которое представляет собой список `Extent<Evaluated-Product>`. `Extent` представляет количество данного товара.



С точки зрения механизма внедрения зависимостей классы `Extent` и `Basket` не особенно интересны: они оба являются РОСО-классами без зависимостей. Большой интерес представляют классы `BasketService` и поддерживаемые классы, продемонстрированные на рисунке 2-25.

Рисунок 2-25: `BasketService` и поддерживаемые классы. `BasketService` может извлекать и вносить `Basket` для данного пользователя. `BasketService` использует `BasketRepository` для извлечения `Basket` и `BasketDiscountPolicy` с целью применения скидки (при необходимости)



`BasketService` может использоваться для извлечения пользовательского `Basket` и применения скидок. Он использует абстрактный `BasketRepository`, чтобы получить содержимое `Basket` и абстрактного `BasketDiscountPolicy` для дальнейшего применения скидок. Обе эти абстракции внедряются в `BasketService` посредством технологии внедрения через конструктор.

```
public BasketService(BasketRepository repository,
    BasketDiscountPolicy discountPolicy)
```

`BasketDiscountPolicy` может быть простой реализацией с жестко-закодированной стратегией, например, предоставление привилегированным покупателям пятипроцентной скидки, как мы уже видели в этой главе. Данная стратегия реализуется с помощью `DefaultProductDiscountPolicy`, в то время как более сложная реализация через данные обеспечивается `RepositoryBasketDiscountPolicy`, который уже использует абстрактный `DiscountRepository` для получения списка уцененных товаров. Эта абстракция еще раз внедряется в `RepositoryBasketDiscountPolicy` посредством внедрения через конструктор:

```
public RepositoryBasketDiscountPolicy(DiscountRepository repository)
```

Для управления всем этим я могу воспользоваться `BasketService`, чтобы распределить операции над `Basket`: добавление элементов, а также отображение и очистка `Basket`. Для выполнения этого `BasketService` необходимы как `BasketRepository`, так и `BasketDiscountPolicy`, которые (как вы догадались) передаются в него через его конструктор:

```
public BasketService(BasketRepository repository,
    BasketDiscountPolicy discountPolicy)
```

Для дальнейшего усложнения мне необходим ASP.NET MVC контроллер с названием `BasketController`, содержащий интерфейс `IBasketService`, который я снова внедряю через конструктор:

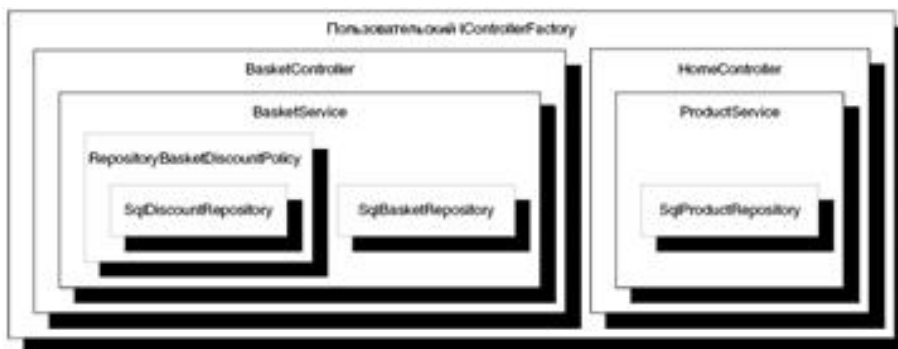
```
public BasketController(IBasketService basketService)
```

Как показывает рисунок 2-25, класс `BasketService` реализует `IBasketService`, это и есть используемая нами реализация.

`BasketController`, в конечном счете, создается при помощи пользовательского `IControllerFactory`, поэтому ей также потребуются эти абстракции.

Если по пути вы потеряли направление наших мыслей, то рисунок 2-26 демонстрирует диаграмму, которая иллюстрирует то, как зависимости komponуются в окончательном приложении.

Рисунок 2-26: Композиция шаблонного коммерческого приложения с добавленной возможностью отправки товаров в корзину, а также первоначальным списком рекомендуемых товаров на титульной странице. Каждый класс инкапсулирует свое содержание, и только `Composition Root` знает обо всех зависимостях.



Пользовательский `IControllerFactory` создает экземпляры `BasketController` и `HomeController`, предоставляя их вместе с соответствующими зависимостями. `BasketService`, например, использует переданный экземпляр `BasketDiscountPolicy`, чтобы применить стратегию скидок к корзине товаров:

```
var discountedBasket = this.discountPolicy.Apply(b) ;
```

Я не намекаю на то, что в данном случае переданный `BasketDiscountPolicy` — это экземпляр `RepositoryBasketDiscountPolicy`, который сам по себе является контейнером для `DiscountRepository`.

Это расширенное шаблонное приложение служит основой для большинства примеров кода оставшейся части книги.

2.4. Резюме

Поразительно легко писать сильно связанный код. Даже когда Мэри намеревалась написать трехуровневое приложение, это вылилось большей частью в монолитный фрагмент "спагетти-кода" (говоря о многоуровневом представлении, мы называем его "лазанья").

Одной из множества причин того, что писать сильно связанный код так легко, является то, что и возможности языка, и наши инструменты уже ведут нас в этом направлении. Если нам нужен новый экземпляр объекта, мы можем воспользоваться ключевым словом `new`, а если у нас отсутствует ссылка на требуемую сборку, Visual Studio облегчает процесс ее добавления.

Как бы то ни было, каждый раз при использовании ключевого слова `new` мы вводим сильное связывание.

Самый лучший способ минимизации использования ключевого слова `new` — использование паттерна проектирования Внедрение через конструктор всюду, где нам нужен экземпляр зависимости. Второй пример главы продемонстрировал, как заново реализовать приложение Мэри, программируя на основании интерфейсов, а не конкретных классов.

Внедрение через конструктор — это пример инверсии управления, поскольку мы инвертируем управление зависимостями. Вместо того чтобы создавать экземпляры при помощи ключевого слова `new`, мы делегируем эту ответственность стороннему компоненту. Как мы увидим из следующей главы, мы называем это место `Composition Root`. Это то место, в котором мы компонуем все слабо связанные классы приложения.

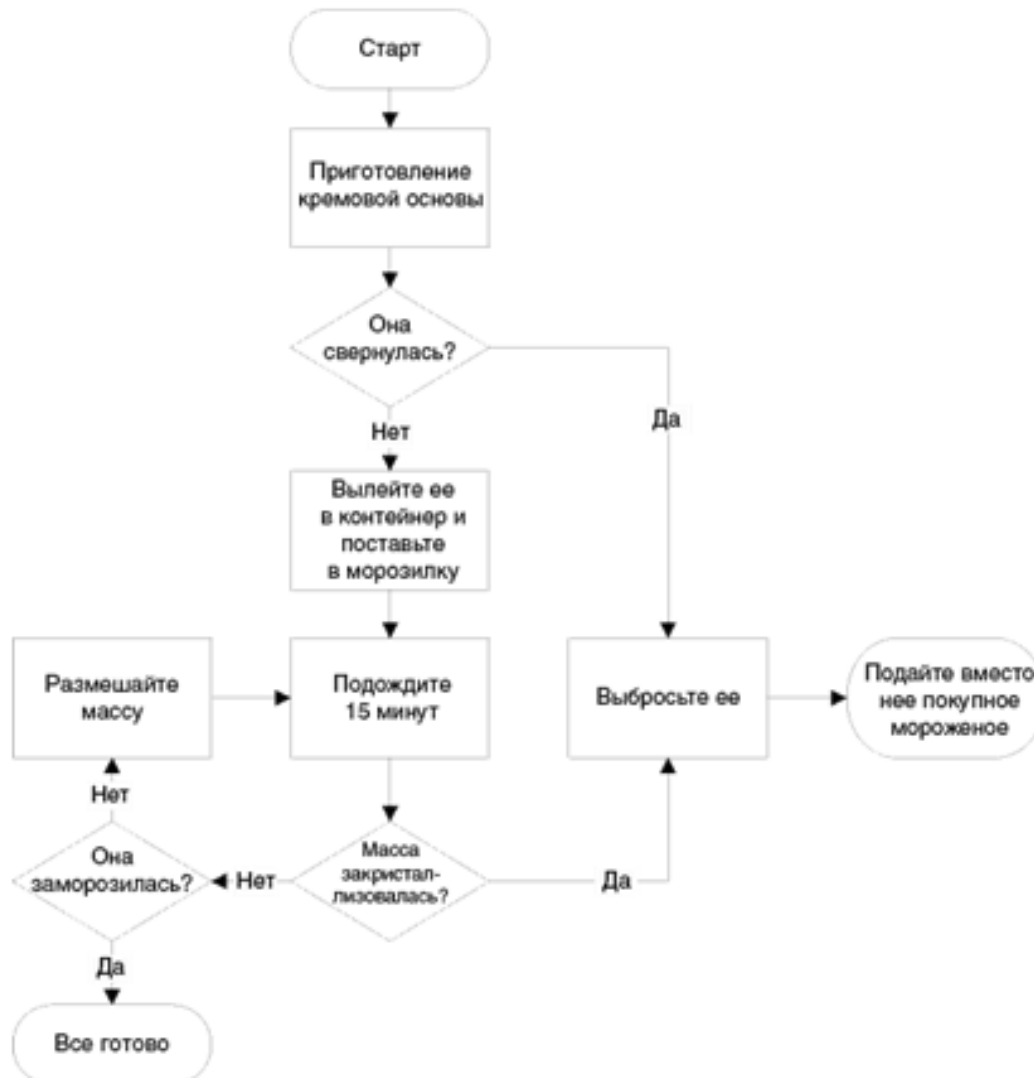
3. DI-контейнеры

Меню:

- XML конфигурация
- Использование кода в качестве конфигурации
- Автоматическая регистрация
- Composition Root
- Паттерн Register Resolve Release

Когда я был еще совсем ребенком, мы с мамой изредка готовили мороженое. Мы не делали это слишком часто, поскольку приготовление мороженого было трудоемким процессом, и очень сложно было получить в итоге настоящее мороженое. В случае если вы никогда не пробовали делать мороженое, рисунок 3-1 иллюстрирует процесс его приготовления.

Рисунок 3-1: Приготовление мороженого – трудоемкий процесс, включающий множество возможностей для совершения ошибок



Настоящее мороженое делается на кремовой основе, которая представляет собой легкий заварной крем, приготовленный из сахара, яичных желтков и молока или крема. Если перегреть эту смесь, то она будет сворачиваться. Если даже вам удалось этого избежать, следующая фаза приготовления несет за собой еще больше проблем. Если оставить кремовую смесь в морозилке и не трогать ее, то она будет кристаллизоваться, поэтому вам придется регулярно ее помешивать до тех пор, пока она не станет максимально густой. Только после этого вы получите хорошее, приготовленное в домашних условиях мороженое.

Несмотря на то, что приготовление мороженого – это медленный и трудоемкий процесс, если вы все же хотите его приготовить, и у вас есть все необходимые ингредиенты и оборудование, то вы можете воспользоваться выведенной мной технологией приготовления мороженого.

Сегодня, около 30 лет спустя, моя тёща делает мороженое с периодичностью, несравнимой с той, с которой мы с мамой делали мороженое в более юные годы – не потому что она любит делать мороженое, но потому что она пользуется технологией, которая ей помогает. Техника остается все той же, но вместо того, чтобы постоянно доставать мороженое из морозилки и перемешивать его, она пользуется электрической мороженицей, которая выполняет это за нее (см. рисунок 3-2).

Рисунок 3-2: Итальянская мороженица моей тещи

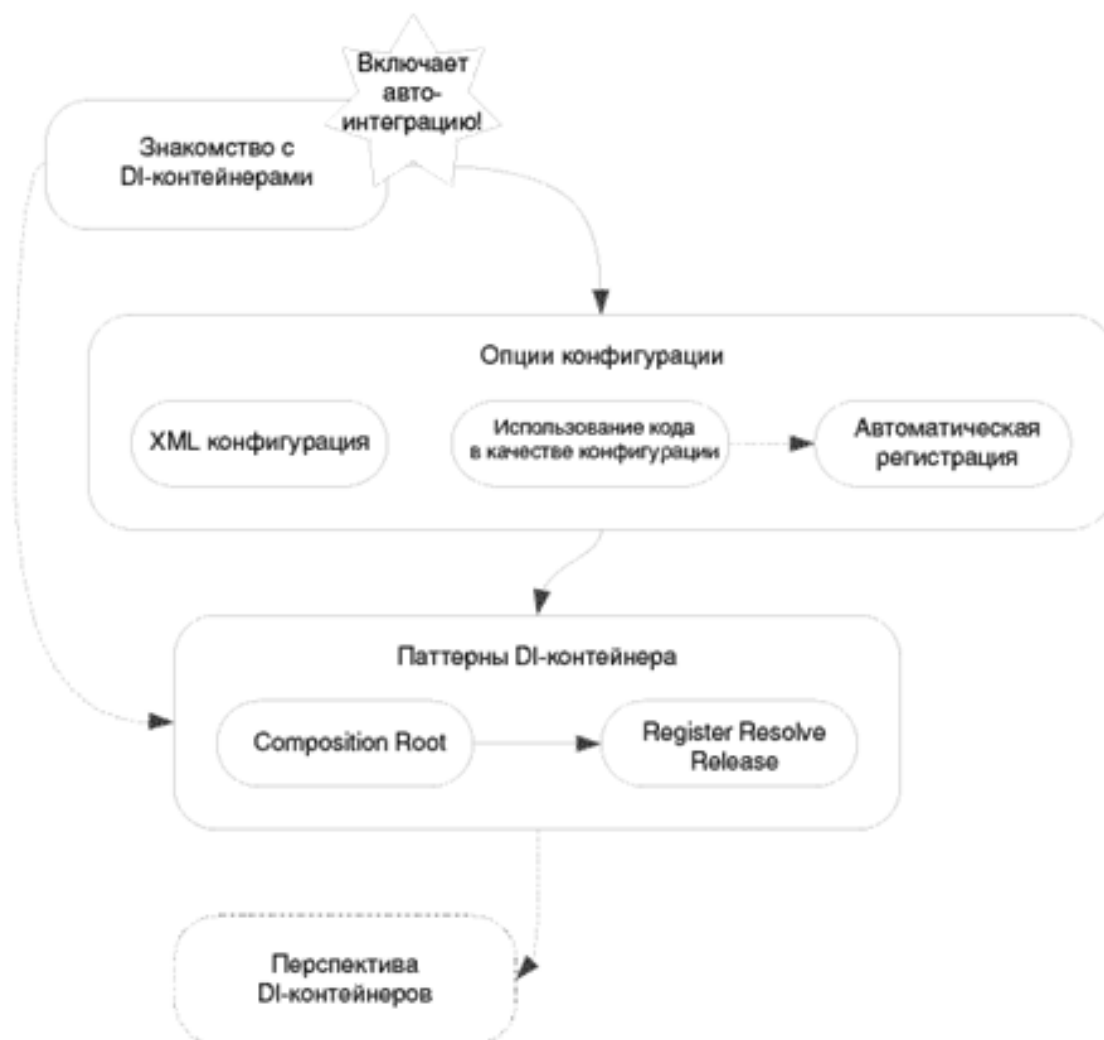


Механизм внедрения зависимостей – это первая и самая главная технология, но вы можете воспользоваться техническими средствами для облегчения процесса. В части 3 я буду описывать DI как технологию. Затем в части 4 мы рассмотрим технические средства, которые могут использоваться для поддержания технологии DI. Мы называем эти средства DI-контейнерами.

В этой главе мы рассмотрим DI-контейнеры как сущность: как они вписываются во всеобщую картину механизма внедрения зависимостей, рассмотрим некоторые паттерны и технологии, предусматривающие их использование, а также рассмотрим некоторые исторические факты, касающиеся DI. Попутно мы рассмотрим некоторые примеры.

Основной план-конспект главы проиллюстрирован на рисунке 3-3. Глава начинается с общего введения к DI-контейнерам, включая описание сущности под названием автоматическая интеграция, за которым следует раздел с описанием различных опций конфигурации. Вы можете читать о каждой из этих опций в отдельности, но думаю, было бы выгоднее, по крайней мере, прочитать о такой опции, как использование кода в качестве конфигурации, прежде чем читать об автоматической регистрации.

Рисунок 3-3: Структура этой главы. Раздел "Перспектива DI-контейнеров" не является обязательным для прочтения



Центральный раздел этой главы – это мини-каталог паттернов проектирования, называемых DI-контейнеры. Несмотря на то, что этот раздел руководствуется форматом каталога, описание паттерна Register Resolve Release (RRR) опирается на описание паттерна Composition Root, поэтому имеет смысл читать их последовательно. Вы можете пропустить раздел об опциях конфигурации и перейти прямо к описанию паттернов, но эти разделы лучше читать по порядку.

Последний раздел отличается от предыдущих. Он намного менее технический и фокусируется на том, как DI-контейнеры вписываются в экосистему .NET. Вы можете пропустить этот раздел, если вам не важен этот аспект.

Цель главы – предоставить вам хорошее понимание того, что такое DI-контейнер, и как он сочетается со всеми остальными паттернами и принципами данной книги; в некотором смысле, вы можете рассматривать эту главу как введение к части 4 данной книги. В ней мы будем говорить о DI-контейнерах в широком смысле, тогда как в части 4 мы будем говорить о конкретных контейнерах и их API.

Может показаться несколько странным то, что мы ведем разговор о DI-контейнерах здесь, в главе 3, а потом более или менее забываем о них на протяжении следующих шести глав, но для этого есть причина. В этой части книги я хочу обрисовать огромную картину механизма внедрения зависимостей, и важно, чтобы вы понимали, как DI-контейнеры

вписываются в эту схему. В части 2 и 3 я изредка буду демонстрировать вам некоторые примеры, в которые входит DI-контейнер, но в большинстве случаев я, в основном, буду придерживаться рассуждений. Принципы и паттерны, описанные в середине книги, могут быть применимы ко всем DI-контейнерам.

[3.1. Знакомство с DI-контейнерами](#)

[3.2. Конфигурирование DI-контейнеров](#)

[3.3. Паттерны DI-контейнеров](#)

[3.4. Перспектива DI-контейнеров](#)

[3.5. Резюме](#)

3.1. Знакомство с DI-контейнерами

DI-контейнер – это библиотека программных средств, которая может автоматизировать множество задач, включая компоновку объектов и управление их жизненным циклом. Несмотря на то, что можно написать весь необходимый код инфраструктуры с помощью *Root man's DI*, такой способ не добавляет большой значимости приложению. С другой стороны, задача компоновки объектов является всеобщей по своей натуре и может быть решена в один момент и сразу для всех; это так называемый *видовой субдомен*.

Определение

DI-контейнер – это библиотека, которая обеспечивает функциональность механизма внедрения зависимостей.

Примечание

DI-контейнеры также называют *IoC-контейнерами* или (намного реже) *Легковесными контейнерами*.

Несмотря на то, что вам нужно обращаться к инфраструктуре приложения, выполнение этого само по себе не добавляет никакой бизнес-значимости, поэтому лучше всего использовать общецелевую библиотеку. Она не отличается от реализации входа в систему или доступа к данным. Регистрация данных приложения – это проблема, к которой чаще всего обращается общецелевая библиотека управления логами. То же самое верно и для диаграмм компоновки объектов.

Предупреждение

Не ждите, что DI-контейнер волшебным образом превратит сильно связанный код в слабо связанный. DI-контейнер может сделать использование механизма внедрения зависимостей более эффективным, но приложение, в первую очередь, должно быть спроектировано с учетом DI паттернов и технологий.

В данном разделе я буду обсуждать то, как DI-контейнеры формируют диаграммы объектов и продемонстрирую вам некоторые примеры, чтобы дать вам общее представление того, на что должно быть похоже использование контейнера.

Контейнер "Hello"

DI-контейнер – это библиотека программных средств, похожая на любую другую библиотеку программных средств. Он предоставляет API, которое вы можете использовать для компоновки объектов. Формирование диаграммы объектов – это вызов единичного метода. Все DI-контейнеры необходимо конфигурировать перед использованием их для компоновки объектов, но я заново пересмотрю этот вопрос в разделе "Конфигурирование DI-контейнеров".

В этом разделе я продемонстрирую вам несколько примеров того, как DI-контейнеры могут преобразовывать диаграммы объектов для расширенного шаблонного приложения из раздела "Расширение шаблонного приложения". Для каждого запроса ASP.NET MVC фреймворк будет запрашивать экземпляр соответствующего типа *IController*, поэтому

вы должны реализовать метод, который использует DI-контейнер для формирования соответствующей диаграммы объектов.

Подсказка

Раздел "Построение ASP.NET MVC приложений" содержит подробную информацию о том, как формировать ASP.NET MVC приложения.

MVC фреймворк будет вызывать метод для экземпляра `Type`, который определяет нужный ему тип `Controller` (например, `HomeController` или `BasketController`), а вы должны возвращать экземпляр этого типа.

Эта функциональность может быть реализована для всех DI-контейнеров, рассматриваемых в части 4, но в данном разделе я продемонстрирую только несколько примеров.

Преобразование контроллеров для различных DI-контейнеров

Unity – это DI-контейнер с явно соответствующим паттерну API. Предполагая, что у вас уже есть экземпляр класса `UnityContainer` контейнера Unity, вы можете преобразовать экземпляр `Controller` из аргумента `controllerType` типа `Type`:

```
var controller = (Controller)this.container.Resolve(controllerType);
```

Вы будете передавать параметр `controllerType` в метод `Resolve` и получать экземпляр требуемого типа, полностью заполненный всеми подходящими зависимостями. Поскольку слабо-типизированный метод `Resolve` возвращает экземпляр `System.Object`, он должен быть отправлен в `Controller`.

Для таких случаев, когда вы уже на этапе проектирования знаете, какой тип необходим, существует также общая версия метода `Resolve`.

Многие DI-контейнеры обладают API, которое похоже на API контейнера Unity. Соответствующий код для Castle Windsor выглядит идентично коду Unity, несмотря на то, что экземпляр `container` будет уже экземпляром класса `WindsorContainer`. Остальные контейнеры имеют несколько другие названия – для StructureMap, например, предыдущий код будет выглядеть следующим образом:

```
var controller = (Controller)this.container.GetInstance(controllerType);
```

Единственное реальное отличие – метод `Resolve` называется `GetInstance`. Вы можете извлечь из этих примеров общий вид DI-контейнера.

Диаграммы преобразования объектов для DI-контейнеров

DI-контейнер – это движок, который преобразует и управляет диаграммами объектов. Несмотря на то, что DI-контейнер выполняет множество других функций, помимо преобразования объектов, преобразование объектов является центральной частью API любого контейнера. Предыдущие примеры демонстрируют, что у контейнеров для этих целей есть слабо-типизированный метод. Варьируя в названиях и сигнатурах, в общем случае он выглядит следующим образом:

```
object Resolve(Type service);
```

Как демонстрируют предыдущие примеры, поскольку возвращаемый экземпляр является типизированным, как например, `System.Object`, то вам часто приходится преобразовывать возвращаемое значение к необходимому типу перед тем, как его использовать.

Многие DI-контейнеры также для этих случаев предлагают общую версию, когда мы знаем, какой тип запрашивается во время компиляции. Они часто выглядят следующим образом:

```
T Resolve<T>();
```

Вместо применения аргумента `Type` метода такая перегрузка принимает типизированный параметр (`T`), который указывает на необходимый тип. Метод возвращает экземпляр `T`.

Большинство контейнеров выдают исключения, если не могут преобразовать необходимый тип.

Предупреждение

Сигнатура метода `Resolve` чрезвычайно влиятельная и универсальная. Вы можете запросить экземпляр любого типа, и ваш код все еще будет компилируемым. В действительности, метод `Resolve` устанавливает сигнатуру анти-паттерна `Service Locator`, и вам нужно действовать осторожно, чтобы не использовать свой DI-контейнер в качестве `Service Locator`.

Если мы рассмотрим метод `Resolve` изолированно, то он выглядит почти волшебным. С точки зрения компилятора, можно попросить его преобразовать экземпляры произвольных типов. Каким образом контейнер узнает, как формировать необходимые типы, включая все зависимости?

Он не знает этого, и вам придется сначала ему это разъяснить. Вы делаете это посредством *регистрации* или *конфигурирования*, и именно здесь вы преобразуете абстракции в конкретные типы – я вернусь к этому вопросу в разделе "Конфигурирование DI-контейнеров". Если у контейнера не будет подходящей конфигурации для того, чтобы полноценно сформировать необходимый тип, он будет выдавать описательное исключение. Например, `Castle Windsor` имеет следующие примерные сообщения-исключения:

- *Невозможно создать компонент "Ploeh.Samples.MenuModel.Mayonnaise", поскольку у него есть зависимости, которые необходимо выделить.*
- *Для Ploeh.Samples.MenuModel.Mayonnaise нужны следующие зависимости:*
- *Сервисы:*
- *– Ploeh.Samples.MenuModel.EggYolk, который не был зарегистрирован.*

В данном примере вы можете увидеть, что `Castle Windsor` не может преобразовать `Mayonnaise`, поскольку он не был настроен для работы с классом `EggYolk`.

Если контейнер корректно сконфигурирован, то он может преобразовывать даже сложные диаграммы объектов из требуемых типов. Если в конфигурации что-то упущено, то

контейнер может предоставить детальную информацию о том, что пропущено. В следующем разделе мы подробнее рассмотрим, как это делается.

Автоматическая интеграция

DI-контейнеры бурно развиваются на основе статической информации, собранной во всех классах, использующих технологию внедрения через конструктор. Используя рефлексия, они могут анализировать запрашиваемые классы и предугадывать, какие зависимости им нужны.

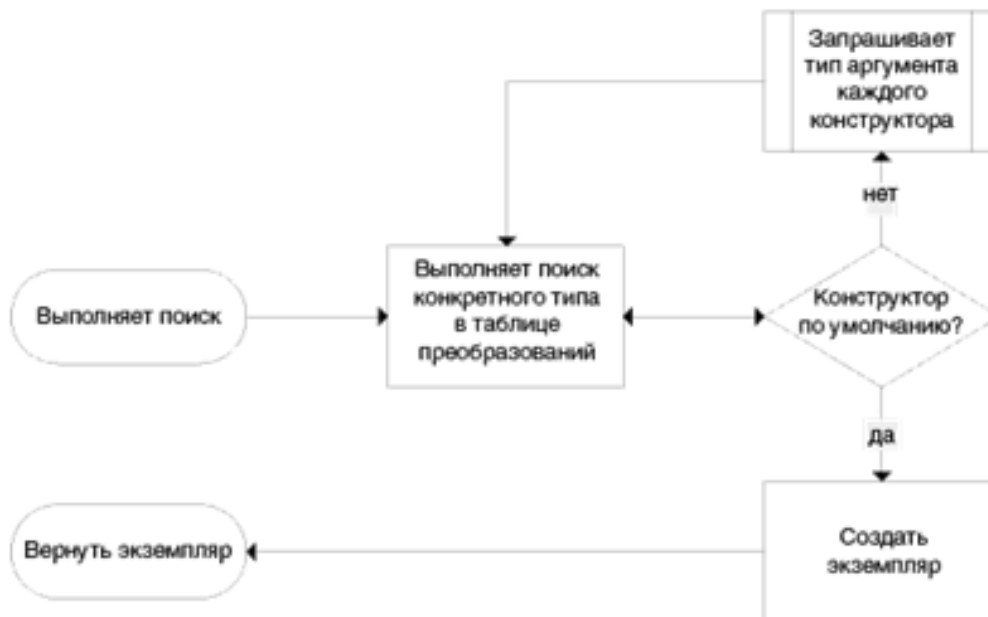
Некоторые DI-контейнеры понимают и паттерн Property Injection (внедрение зависимостей через свойства), но все они, по своему существу, понимают паттерн Constructor Injection и формируют диаграммы объектов путем сочетания их собственной конфигурации с информацией, извлеченной из конструкторов классов. Данный процесс называется автоматической интеграцией.

Определение

Автоматическая интеграция – это способность автоматически формировать диаграмму объектов из таблицы преобразований между абстракциями и конкретными типами.

Рисунок 3-4 описывает общий алгоритм, которому следует большинство DI-контейнеров, чтобы автоматически интегрировать диаграмму объектов. DI-контейнер будет использовать эту конфигурацию для того, чтобы найти соответствующий конкретный класс, совпадающий с запрашиваемым типом. Затем DI-контейнер использует рефлексия для изучения конструктора класса. Если существует конструктор по умолчанию, то DI-контейнер будет вызывать конструктор и возвращать созданный экземпляр.

Рисунок 3-4: Упрощенная последовательность действий для автоматической интеграции. DI-контейнер будет рекурсивно находить конкретные типы и изучать их конструкторы до тех пор, пока он не сможет создать целостное дерево объектов.



Если для конструктора нужны аргументы, то начинается рекурсивный процесс, во время которого DI-контейнер будет повторять этот процесс для каждого типа аргумента до тех пор, пока все конструкторы не будут просмотрены.

В разделе "Конфигурирование DI-контейнеров" мы подробнее рассмотрим то, как можно конфигурировать контейнеры, а сейчас самое главное – это понять, что центральной частью конфигурации является то, как различные абстракции преобразуются в конкретные классы. Все это звучит слегка теоретически (думаю, что слово "абстракция" не поможет), поэтому я думаю, что пример будет очень полезным.

Пример: Автоматическая интеграция BasketController

В этом примере я буду объяснять, как, в принципе, работает автоматическая интеграция. Пример не опирается ни на какой конкретный DI-контейнер, но вместо этого предоставляет обзор того, как контейнеры формируют диаграммы объектов.

Представьте себе, что вам нужно преобразовать экземпляр класса `BasketController`. Вы делаете это путем вызова метода `Resolve` для `typeof(BasketController)`. В итоге вам хотелось бы получить экземпляр `BasketController`, сформированный так, как это показано на рисунке 2-26. Чтобы достичь этого, вы должны сначала убедиться, что контейнер имеет корректную конфигурацию. Таблица 3-1 демонстрирует, как эта конфигурация преобразует абстракции в конкретные типы. Кроме того, я добавил столбец, который показывает, является ли абстракция интерфейсом или абстрактным базовым классом – с точки зрения DI-контейнера это очень важно, но я думал, что это поможет прояснить то, что происходит.

Таблица 3-1: Преобразование типов для обеспечения автоматической интеграции `BasketController`

Тип абстракции	Абстракция	Конкретный тип
Явный	<code>BasketController</code>	<code>BasketController</code>
Интерфейс	<code>IBasketService</code>	<code>BasketService</code>
Абстрактный класс	<code>BasketRepository</code>	<code>SqlBasketRepository</code>
Абстрактный класс	<code>BasketDiscountPolicy</code>	<code>RepositoryBasketDiscountPolicy</code>
Абстрактный класс	<code>DiscountRepository</code>	<code>SqlDiscountRepository</code>
Строка	<code>connString</code>	"metadata=res:/*/CommerceModel.csdl [...]"

Когда DI-контейнер получит запрос `BasketController`, первое, что он сделает – будет искать тип в его конфигурации. `BasketController` – это конкретный класс, поэтому он преобразуется в самого себя. Затем контейнер использует рефлексии для осмотра конструктора `BasketController`. Из раздела "Возможность добавления в корзину" вы можете помнить, что `BasketController` обладает единственным конструктором со следующей сигнатурой:

```
public BasketController(IBasketService basketService)
```

Поскольку этот конструктор не является конструктором по умолчанию, нам необходимо повторить этот процесс для аргумента конструктора `IBasketService`, если мы руководствуемся общей блок-схемой, изображенной на рисунке 3-4.

Контейнер ищет `IBasketService` в своей конфигурации и обнаруживает, что `IBasketService` преобразуется в конкретный класс `BasketService`. Единственный открытый конструктор `BasketService` имеет следующую сигнатуру:

```
public BasketService(BasketRepository repository,
    BasketDiscountPolicy discountPolicy)
```

Это все еще не конструктор по умолчанию, и теперь вы работаете с аргументами двух конструкторов. Контейнер осматривает каждый из аргументов по порядку, поэтому он начинает с класса `BasketRepository`, который согласно конфигурации преобразуется в `SqlBasketRepository`.

`SqlBasketRepository` обладает открытым конструктором со следующей сигнатурой:

```
public SqlBasketRepository(string connString)
```

Единственный аргумент конструктора – это строковый параметр под названием `connString`, который сконфигурирован таким образом, что обладает конкретным значением. Теперь, когда у контейнера есть подходящее значение, он может вызвать конструктор `SqlBasketRepository`. На данный момент контейнер успешно обрабатывал параметр `repository` конструктора `BasketService`, но ему понадобится придержать это значение ненадолго, поскольку ему также нужно позаботиться и о параметре `discountPolicy`.

Согласно конфигурации `BasketDiscountPolicy` преобразуется в конкретный класс `RepositoryBasketDiscountPolicy`, который обладает следующим открытым конструктором:

```
public RepositoryBasketDiscountPolicy(DiscountRepository repository)
```

Выполняя поиск `DiscountRepository` в своей конфигурации, контейнер обнаруживает, что он `DiscountRepository` преобразуется в `SqlDiscountRepository`, который обладает следующим конструктором:

```
public SqlDiscountRepository(string connString)
```

Ситуация совпадает с той, когда вы сталкивались с `SqlBasketRepository`. Аргумент `connString` преобразуется в конкретную строку соединения, которую контейнер может передать в конструктор.

На данный момент контейнер передает новый экземпляр `SqlDiscountRepository` в конструктор `RepositoryBasketDiscountPolicy`. Наряду с `SqlBasketRepository`, он теперь выполняет конструктор `BasketService` и вызывает его посредством рефлексии. В конце концов, он передает вновь созданный экземпляр `BasketService` в конструктор `BasketController` и возвращает экземпляр `BasketController`.

По существу это и есть то, как работает автоматическая интеграция, несмотря на то, что она является более сложной, чем описанный процесс. DI-контейнерам также необходимо заботиться об управлении жизненным циклом и, возможно, обращаться к технологии внедрения через свойства, а также к другим более специфичным необходимым условиям создания зависимостей. Основной момент – это то, что технология внедрения через конструктор статически объявляет требования зависимости о наличии класса, а DI-

контейнер использует эту информацию для того, чтобы автоматически интегрировать диаграммы сложных объектов.

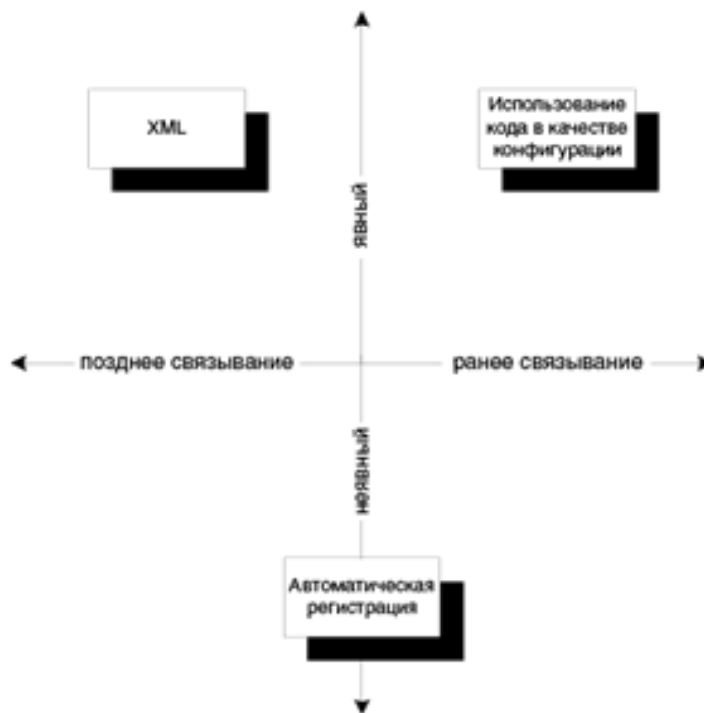
Как показывает пример, контейнер нужно сконфигурировать перед тем, как он сможет формировать диаграммы объектов. Регистрация компонентов может быть выполнена различными способами.

3.2. Конфигурирование DI-контейнеров

Несмотря на то, что метод `Resolve` — это место, где выполняются все действия, вы должны понимать, что потратите больше времени на конфигурацию API DI-контейнера. Преобразование диаграмм объектов, в конечном счете, — это вызов единственного метода.

DI-контейнеры поддерживают две или три общие опции конфигурации, продемонстрированные на рисунке 3-5. Некоторые из них не поддерживают автоматическую регистрацию, один из контейнеров не поддерживает технологию использования кода в качестве конфигурации, в то время как XML конфигурация используется повсюду. Большинство контейнеров позволяют вам сочетать несколько подходов в одном приложении.

Рисунок 3-5: Наиболее универсальные способы конфигурирования DI-контейнера, показанные относительно таких параметров, как ясность и степень связывания.



Эти три опции конфигурации имеют разные характеристики, которые делают их полезными в различных ситуациях. И XML конфигурация, и использование кода в качестве конфигурации являются явно определенными, поскольку они требуют, чтобы мы регистрировали каждый компонент в индивидуальном порядке. Автоматическая регистрация, с другой стороны, является более неявной, поскольку при этом подходе используются условные обозначения для регистрации набора компонентов с помощью единого правила.

При использовании кода в качестве конфигурации мы компилируем конфигурацию контейнера в сборку, тогда как XML конфигурация позволяет нам поддерживать позднее связывание, при котором мы можем изменять конфигурацию без необходимости заново компилировать приложение. В данном измерении автоматическая регистрация ниспадает где-то до середины, поскольку мы можем попросить просмотреть единственную сборку,

которая известна во время компиляции или, в противном случае, просмотреть все сборки в определенной папке.

В таблице 3-2 перечисляются преимущества и недостатки каждой опции.

Таблица 3-2: Опции конфигурации

Стиль	Описание	Преимущества	Недостатки
XML	Настройки конфигурации (часто в <i>.config</i> файлах) определяют преобразования.	Обеспечивает возможность замены без повторной компиляции. Высокая степень контроля.	Отсутствуют проверки во время компиляции. Очень подробный.
Использование кода в качестве конфигурации	Код явно определяет преобразования.	Проверки во время компиляции. Высокая степень контроля.	Не поддерживает возможность замены без повторной компиляции.
Автоматическая регистрация	Для определения местоположения подходящих компонентов и построения преобразований используются правила.	Обеспечивает возможность замены без повторной компиляции. Необходимы минимальные усилия. Помогает принудительным соглашениям сделать базу кода более логичной.	Частичные проверки во время компиляции. Наименьший контроль.

Традиционно DI-контейнеры начались с XML конфигурации, что также объясняет тот факт, что все контейнеры поддерживают данную опцию. Тем не менее, тенденцией настоящего времени является то, что эта возможность приуменьшается в пользу более условных подходов. Несмотря на то, что автоматическая регистрация – это самая современная опция, это не самое очевидное место для старта. Из-за своей неявности эта возможность может казаться более абстрактной, нежели другие более явные опции, поэтому я буду рассматривать каждую опцию в историческом порядке, начиная с XML конфигурации.

Конфигурирование контейнеров при помощи XML

Когда DI-контейнеры впервые появились в ранние 2000-е годы, все они использовали XML в качестве механизма конфигурации – с тех пор многое изменилось. Более частое использование XML в качестве механизма конфигурации в дальнейшем раскрыло тот факт, что такой подход изредка является самым лучшим.

XML является довольно подробным и хрупким. При конфигурировании DI-контейнера в XML вы идентифицируете различные классы и интерфейсы, но ваш компилятор не поддерживает возможность предупреждения вас о том, что вы что-то пропустили. Даже если имена классов корректны, нет никакой гарантии, что необходимая сборка будет находиться в исследуемом пути приложения.

Преимущество XML конфигурации – вы можете изменять поведение приложения без повторной компиляции. Это важно, если вы разрабатываете программное средство,

которое поставляется тысячам покупателей, потому что это дает им возможность самостоятельно настроить приложение. Как бы то ни было, если вы пишете внутреннее приложение или веб-сайт, в котором вы управляете средой развертывания, чаще всего бывает легче просто перекомпилировать и развернуть заново приложение в тех случаях, когда вам нужно изменить поведение.

Подсказка

Используйте XML конфигурацию только в тех случаях, когда вам очень хочется обеспечить позднее связывание. Во всех остальных случаях отдайте предпочтение стилю "код в качестве конфигурации" или автоматической регистрации.

DI-контейнер часто конфигурируется с помощью XML посредством направления его в конкретный XML-файл, но иногда он также может подхватить конфигурацию и из конфигурационного файла приложения. В следующих примерах используется последняя указанная опция.

Пример: Конфигурирование шаблонного коммерческого приложения с помощью XML

Поскольку контейнер Unity является одним из самых XML-центрированных DI-контейнеров, рассматриваемых в данной книге, имеет смысл использовать его для примера XML конфигурации.

В этом примере вы будете конфигурировать шаблонное коммерческое приложение из раздела "Расширение шаблонного приложения". Самая большая составляющая задачи – применить конфигурацию, которая законспектирована в таблице 3-1, но кроме того вы должны применить похожую конфигурацию для того, чтобы обеспечить композицию класса `HomeController`. Следующий листинг демонстрирует конфигурацию, необходимую для поднятия и запуска приложения.

Листинг 3-1: Конфигурирование Unity при помощи XML

```
1. <register type="IBasketService" mapTo="BasketService" />
2. <register type="BasketDiscountPolicy"
   mapTo="RepositoryBasketDiscountPolicy" />
3. <register type="BasketRepository" mapTo="SqlBasketRepository">
4.   <constructor>
5.     <param name="connString">
6.       <value value="CommerceObjectContext"
7.         typeConverter="ConnectionStringConverter" />
8.     </param>
9.   </constructor>
10. </register>
11. <register type="DiscountRepository" mapTo="SqlDiscountRepository">
12.   <constructor>
13.     <param name="connString">
14.       <value value="CommerceObjectContext"
15.         typeConverter="ConnectionStringConverter" />
16.     </param>
17.   </constructor>
18. </register>
19. <register type="ProductRepository" mapTo="SqlProductRepository">
20.   <constructor>
21.     <param name="connString">
22.       <value value="CommerceObjectContext"
23.         typeConverter="ConnectionStringConverter" />
```

```

24.         </param>
25.     </constructor>
26. </register>
27. <register type="CurrencyProvider" mapTo="SqlCurrencyProvider">
28.     <constructor>
29.         <param name="connString">
30.             <value value="CommerceObjectContext"
31.                 typeConverter="ConnectionStringConverter" />
32.         </param>
33.     </constructor>
34. </register>

```

Строка 4-9: Определяет строку соединения

Строка 1: Простое преобразование

Как вы можете видеть даже из этого простого листинга кода, XML конфигурация достаточно подробна. Простые преобразования, подобные преобразованию интерфейса `IBasketService` в класс `BasketService`, легко выражаются с помощью простого элемента `register`.

Тем не менее, как вы, возможно, помните, некоторые конкретные классы принимают строку соединения в качестве входных данных, поэтому вам необходимо определить, каким образом находится значение этой строки. Что касается Unity, вы можете сделать это, указав, что вы используете пользовательский тип конвертера под названием `ConnectionStringConverter`. Этот конвертер будет искать значение `CommerceObjectContext` среди стандартных строк соединения `web.config` и возвращать строку соединения с этим именем.

Оставшиеся элементы повторяют эти два паттерна.

Поскольку Unity может автоматически преобразовывать запросы в конкретные типы, даже если отсутствуют явные регистрации, вам не нужно применять XML элементы для `HomeController` и `BasketController`.

Загрузка конфигурации в контейнер выполняется при помощи вызова единственного метода

```
container.LoadConfiguration();
```

Метод `LoadConfiguration` загружает XML конфигурацию из листинга 3-1 в контейнер. После размещения конфигурации контейнер теперь может преобразовывать запросы в `HomeController` и в другие.

Остальные DI-контейнеры также поддерживают XML конфигурацию. Точная XML-схема отличается для каждого контейнера, но всеобщая структура остается аналогичной.

Предупреждение

Как только ваше приложение будет вырастать в размерах и усложняться, тоже самое будет происходить и с вашим конфигурационным файлом, если вы используете конфигурационную композицию. Он может стать настоящим камнем преткновения, поскольку этот файл моделирует такие сущности кода, как классы, параметры и тому подобное, но без преимуществ компилятора, опций отладки и т.д. Конфигурационные

файлы будут становиться хрупкими и непрозрачными с точки зрения наличия ошибок, поэтому используйте данный подход только, если вам необходимо позднее связывание.

Из-за таких недостатков, как подробность и хрупкость, вам следует воспользоваться другими альтернативами для конфигурирования контейнеров. Технология использования кода в качестве конфигурации схожа с XML конфигурацией по глубине детализации и по концепции, но очевидно использует код вместо XML.

Конфигурирование контейнеров с помощью кода

Возможно, самый легкий способ компоновки приложения – написать код, который выполняет компоновку. Может показаться, что такой подход идет в разрез со всей сущностью механизма внедрения зависимостей, потому что он жестко кодирует то, какие конкретные реализации должны использоваться для всех абстракций. Тем не менее, если это выполняется в Composition Root, то оно изменяет только одно из преимуществ, перечисленных в таблице 1-1.

Преимущество позднего связывания теряется, если зависимости жестко закодированы, но, как я уже упоминал в главе "Дегустационное меню" механизма внедрения зависимостей", это может и не иметь отношения ко всем типам приложений. Если ваше приложение разворачивается в ограниченное количество экземпляров в контролируемой среде, было бы проще заново скомпилировать и развернуть приложение, когда вам нужно заменить модули.

Я часто думаю, что люди чересчур ревностно относятся к определению конфигурационных файлов. Часто язык программирования является прямолинейным и мощным механизмом конфигурирования.

Мартин Фаулер

При использовании кода в качестве конфигурации мы явно устанавливаем такие же самые дискретные преобразования, которые мы устанавливали при использовании XML конфигурации – только мы используем код вместо XML.

За единственным исключением в виде Spring.NET, все DI-контейнеры полностью поддерживают технологию использования кода в качестве конфигурации как альтернативы XML конфигурации – в действительности, в большинстве из них эта технология присутствует в виде используемого по умолчанию механизма, а XML конфигурация используется в качестве необязательной возможности.

API, открытый для поддержки технологии использования кода в качестве конфигурации, отличается для каждого DI-контейнера, но всеобщей целью все еще остается определение дискретных преобразований между абстракциями и конкретными типами.

Подсказка

Используйте код в качестве конфигурации вместо XML конфигурации, за исключением тех моментов, когда вам нужно позднее связывание. Компилятор может быть полезным, а Visual Studio, которая формирует систему, будет автоматически копировать все необходимые зависимости в выходную папку.

Многие конфигурационные API используют дженерики и Fluent Builders для регистрации компонентов; StructureMap – не исключение.

Пример: Конфигурирование шаблонного коммерческого приложения с помощью кода

В разделе "Конфигурирование контейнеров при помощи XML" вы увидели, как сконфигурировать шаблонное коммерческое приложение с помощью XML, используя Unity. Я также мог бы продемонстрировать использование кода в качестве конфигурации на примере Unity, но в этом примере я буду использовать StructureMap; так как он обладает более выразительным API, он лучше подходит для страниц этой книги.

Используя конфигурационное API StructureMap, вы можете выразить конфигурацию из листинга 3-1 более компактно, как это продемонстрировано в следующем листинге.

Листинг 3-2: Конфигурирование StructureMap с помощью кода

```
c.For<IBasketService>()
    .Use<BasketService>();
c.For<BasketDiscountPolicy>()
    .Use<RepositoryBasketDiscountPolicy>();
string connectionString = ConfigurationManager
    .ConnectionStrings["CommerceObjectContext"].ConnectionString;
c.For<BasketRepository>().Use<SqlBasketRepository>()
    .Ctor<string>().Is(connectionString);
c.For<DiscountRepository>().Use<SqlDiscountRepository>()
    .Ctor<string>().Is(connectionString);
c.For<ProductRepository>().Use<SqlProductRepository>()
    .Ctor<string>().Is(connectionString);
c.For<CurrencyProvider>().Use<SqlCurrencyProvider>()
    .Ctor<string>().Is(connectionString);
```

Сравните этот код с кодом из листинга 3-1 и заметьте, насколько он более компактен – несмотря на то, что выполняет он тоже самое. Такое простое преобразование, как преобразование IBasketService в BasketService, выражается с помощью видовых методов For и Use. Переменная с фактически является так называемым ConfigurationExpression, но воспринимайте ее как контейнер.

Для того чтобы поддерживать те классы, для которых нужна строка соединения, вы продолжаете последовательность For/Use путем вызова метода Ctor и передачи строки соединения. Метод Ctor выполняет поиск строкового параметра в конструкторе конкретного класса и использует переданное значение для этого параметра.

Остальная часть кода повторяет эти два паттерна.

Использование кода в качестве конфигурации не только компактнее XML конфигурации, но также поддерживается компилятором. Типы аргументов, используемые в листинге 3-2, представляют собой реальные типы, которые проверяет компилятор. Переменное API StructureMap поставляется даже с некоторыми видовыми ограничителями, которые сообщают компилятору о проверке того, совпадает ли тип, определяемый методом Use с абстракциями, обозначенными с помощью метода For. Если преобразование невозможно, то код не компилируется.

Несмотря на то, что технология использования кода в качестве конфигурации безопасна и проста в применении, ее нужно больше сопровождать, нежели вам того хотелось. Каждый

раз при добавлении в приложение нового типа вы также должны помнить и о его регистрации – а многие регистрации похожи друг на друга. Автоматическая регистрация относится к этой теме.

Конфигурирование контейнеров с помощью соглашений

Обратили ли вы в листинге 3-2 внимание на то, насколько схожи большинство регистраций? В частности все компоненты доступа к данным, базирующиеся на SQL Server, руководствуются универсальным паттерном в тех случаях, когда вы конфигурируете компонент подходящей строкой соединения.

Неоднократное написание такого кода регистрации нарушает принцип DRY (Don't Repeat Yourself – Не повторяйся). К тому же он выглядит как непродуктивный фрагмент кода инфраструктуры, который не добавляет особую значимость в приложение. Вы можете сэкономить время и допустить меньшее количество ошибок, если сможете автоматизировать процесс регистрации компонентов.

Все более популярной становится архитектурная модель – "соглашения по конфигурации" (Convention over Configuration). Вместо того чтобы писать и поддерживать большие объемы конфигурационного кода, вы можете принять соглашения, которые влияют на базу кода.

Способ, при помощи которого ASP.NET MVC находит контроллеры по их именам, – это отличный пример простого соглашения.

1. Поступает запрос контроллера с именем `Home`.
2. Используемая по умолчанию фабрика контроллеров ищет в списке известных пространств имен класс с названием `HomeController`. Если она находит такой класс и этот класс реализует `Controller`, то это как раз то, что нужно.
3. Используемая по умолчанию фабрика контроллеров использует конструктор по умолчанию найденного класса для того, чтобы создать экземпляр контроллера.

Здесь используются, по крайней мере, два соглашения: контроллер должен иметь название `[Имя контроллера]Controller` и должен обладать конструктором по умолчанию.

Вы можете отступить от этих соглашений посредством реализации своей собственной `ControllerFactory`, а это как раз то, что я и делал до настоящего времени, чтобы поддержать технологию внедрения через конструктор – более подробно я буду рассуждать об этом в главе "Построение объектов".

Было бы здорово, если бы вам удалось использовать несколько соглашений для того, чтобы избавиться от всех этих приводящих к ошибкам и трудоемким конфигурациям контейнеров. Для `DefaultControllerFactory` добавлять новые контроллеры также просто, как и добавлять соответствующе названный класс в корректное пространство имен. Нам бы хотелось сохранить это соглашение даже при использовании технологии внедрения через конструктор.

Многие DI-контейнеры предоставляют возможности автоматической регистрации, которые позволяют нам вводить свои собственные соглашения.

Определение

Автоматическая регистрация – это возможность автоматически регистрировать компоненты в контейнере путем поиска реализаций необходимых абстракций в одной или более одной сборке.

Соглашения можно применять не только к контроллерам ASP.NET MVC. Чем больше соглашений вы добавляете, тем больше вы сможете автоматизировать различные составляющие конфигурации контейнеров.

Подсказка

Принцип "Соглашения по конфигурации" имеет больше преимуществ, нежели просто поддержка DI конфигурации. Он делает ваш код более последовательным, поскольку код будет автоматически работать, пока вы будете следовать вашим соглашениям.

В действительности вам может понадобиться комбинировать автоматическую регистрацию с технологией использования кода в качестве конфигурации или XML конфигурацией, поскольку у вас может не получиться приспособить каждый единичный компонент к значимому соглашению. Тем не менее чем больше вы сможете приспособить вашу базу кода к соглашениям, тем более поддерживаемым он будет.

Пример: Конфигурирование шаблонного коммерческого приложения с помощью механизма автоматической регистрации

StructureMap поддерживает автоматическую регистрацию, но думаю, было бы более интересное использовать все-таки другой DI-контейнер для конфигурирования шаблонного коммерческого приложения при помощи соглашений. Я выбрал Autofac, поскольку он обладает достаточно читабельным API автоматической регистрации.

Если вы рассмотрите листинги 3-1 и 3-2, то надеюсь, что вы согласитесь с тем, что регистрирование различных компонентов доступа к данным – это наиболее повторяющиеся части кода. Можем ли мы выразить для них соглашение некоторого рода?

Все четыре конкретных типа обладают следующими общими характеристиками:

- Все они определены в одной и той же сборке.
- Каждый из них является конкретным классом, унаследованным от абстрактного базового класса.
- Имя каждого из них начинается с `Sql`.
- Каждый из них обладает единственным открытым конструктором, который принимает строковый параметр с названием `connString`.

Кажется, будто соответствующее соглашение будет выражать эти сходства путем просмотра соответствующей сборки и регистрирования всех классов, которые соответствуют соглашению. Для Autofac это выглядело бы следующим образом:

```
string connectionString = ConfigurationManager
    .ConnectionStrings["CommerceObjectContext"]
    .ConnectionString;
var a = typeof(SqlProductRepository).Assembly;
builder.RegisterAssemblyTypes(a)
    .Where(t => t.Name.StartsWith("Sql"))
    .As(t => t.BaseType)
    .WithParameter("connString", connectionString);
```

Это отдельное соглашение должно просматривать сборку, которая содержит компоненты доступа к данным. Существует несколько способов получения ссылки на эту сборку, но самый простой способ – подобрать характерный тип, например `SqlProductRepository`, и получить сборку из него. Вы могли бы выбрать и другой класс или найти сборку по имени.

Теперь, когда у вас есть сборка, вы можете сообщить контейнеру о том, что вы хотите просмотреть ее. Метод `RegisterAssemblyTypes` указывает на намерение регистрировать все типы сборки, которые соответствуют критерию, согласно которому имя класса должно начинаться `Sql`. Переменная `builder` – это экземпляр класса `ContainerBuilder`, но вы можете считать, что он является контейнером.

Каждый из классов, которые выполняют это с помощью фильтра `where`, должны быть зарегистрированы относительно их базового класса. Например, базовым классом `SqlProductRepository` является `ProductRepository`, его можно получить путем преобразования `ProductRepository` в `SqlProductRepository`.

В конце концов, вы утверждаете, что каждый конструктор обладает параметром `connString` и что его значение необходимо определить по строке соединения, считанной из конфигурационного файла.

Сравнения этого соглашения с четырьмя регистрациями листинга 3-2 может быть недостаточно, поскольку мы также поддерживаем два других DI-контейнера. Вы все еще можете думать, что это преимущество выглядит незначительным. Тем не менее, соглашение масштабирует намного лучше.

Поскольку в текущем примере присутствует только четыре компонента доступа к данным, вы сохраняете всего несколько операторов с помощью соглашений. Несмотря на это, как только соглашение записано, оно управляет сотнями компонентов без особых усилий.

Вы также можете обратиться к другим преобразованиям из листингов 3-1 и 3-2 посредством соглашений, но на данный момент от этого не будет никакой пользы. В качестве примера вы можете зарегистрировать все сервисы с помощью следующего соглашения:

```
builder.RegisterAssemblyTypes(typeof(BasketService).Assembly)
    .Where(t => t.Name.EndsWith("Service"))
    .AsImplementedInterfaces();
```

Это соглашение просматривает определенную сборку на факт наличия типов, названия которых заканчиваются на `Service`, и регистрирует каждый тип относительно интерфейса, который он реализует. Фактически это соглашение регистрирует `BasketService` относительно интерфейса `IBasketService`, но поскольку в настоящее время у вас нет никаких других совпадений с этим соглашением, вы ничего из этого не получаете. Однако возможно имеет смысл сформулировать соглашение заранее для того, чтобы вдохновить разработчиков на следование этому соглашению.

Автоматическая регистрация – это мощная технология, которая имеет все необходимое, чтобы сделать DI-контейнер невидимым. После размещения соглашений вам, возможно, понадобится изменять конфигурацию контейнера только в редких случаях.

До настоящего момента вы видели три разных подхода к конфигурированию DI-контейнера:

- XML
- Использование кода в качестве конфигурации
- Автоматическая регистрация

Ни один из них не является взаимно исключаящим. Вы можете выбрать вариант смешивания автоматической регистрации с определенными преобразованиями абстракций в конкретные типы, и даже смешать все три подхода, чтобы немножко выполнять автоматическую регистрацию, немножко использовать код в качестве конфигурации, а также иметь немного конфигурации в XML для целей позднего связывания.

В качестве эмпирического правила вам следует использовать автоматическую регистрацию как стартовую точку, дополненную технологией использования кода в качестве конфигурации для управления более особыми случаями. Вам следует сохранить XML для тех случаев, когда вам нужно уметь изменять реализацию без повторной компиляции приложения (что намного реже, нежели вы думаете).

Теперь, когда мы узнали, как сконфигурировать DI-контейнер и как с помощью него преобразовывать диаграммы объектов, вы должны представлять, как это использовать. Использование DI-контейнера – это одно, а вот корректное его использование – это другое.

3.3. Паттерны DI-контейнеров

DI-контейнеры – это мощные инструменты, но, как и для всех инструментов, существуют корректные и некорректные способы их использования. Так же, как и повара знают, что нужно бережно относиться к своим ножам, так и вы должны знать, что следует правильно обращаться с вашим DI-контейнером – он не может отрубить вам пальцы, но вы можете не получить те преимущества, которыми он обладает.

Самое важное, что вам нужно понять – в каком месте архитектуры приложения следует использовать DI-контейнер. После того, как вы это поймете, вам еще нужно будет знать, *каким* образом его использовать. Следующие два мини-паттерна предоставляют ответы на эти вопросы.

Composition Root

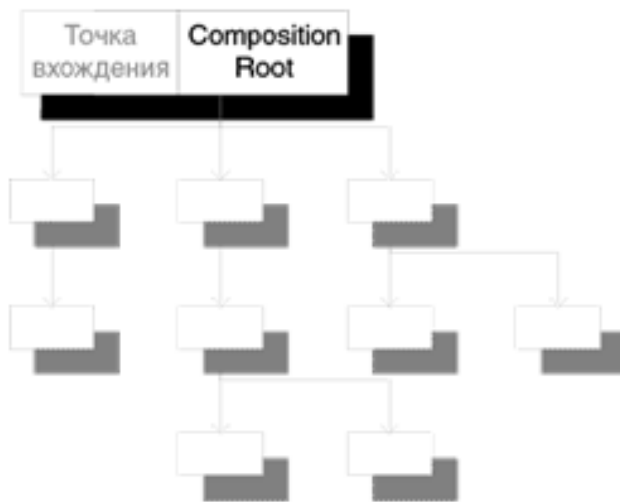
В каком месте нам следует формировать диаграммы объектов?

Как можно ближе к точке входа в приложение.

DI-контейнер – это библиотека, которую вы можете использовать в тех местах, в которых только пожелаете – но это не означает, что вы *должны* его использовать. Несмотря на то, что вы можете расширить использование контейнера таким образом, чтобы он распространился на огромное количество ваших классов, вместо этого сосредоточьте его в единственной области своего приложения. Это место называется Composition Root, и вам следует использовать DI-контейнер только из этого места.

Сущность Composition Root особо не связана с DI-контейнерами. Composition Root также применяется в тех случаях, когда вы используете Poor man's DI, но я думаю, что важно обсудить Composition Root именно в этом контексте, поскольку понимание этого паттерна позволяет вам использовать ваш DI-контейнер корректно и эффективно. Перед тем, как я приступлю к обсуждению роли Composition Root при использовании DI-контейнеров, я кратко, в общих чертах, рассмотрю его.

Рисунок 3-6: При формировании приложения из множества слабо связанных классов композиция должна осуществляться как можно ближе к точке входа в приложение. Composition Root формирует диаграмму объектов, которая затем выполняет фактическую работу приложения.



Composition Root как основная сущность

При написании слабо связанного кода вы создаете множество классов, которые вы должны скомпоновать для того, чтобы создать приложение. Компоновать незначительное количество этих классов в единицу времени может быть довольно заманчивой идеей для создания небольших подсистем, но такой подход ограничивает вашу способность перехватывать эти системы с целью модификации их поведения. Вместо этого вы должны компоновать все классы одновременно.

Определение

Composition Root – это (предпочтительно) уникальное местоположение в приложении, где модули соединяются друг с другом.

Подсказка

Composition Root может разворачиваться в рамках составных классов, пока они располагаются в единственном модуле.

При изолированном рассмотрении Constructor Injection вас может заинтересовать такой вопрос: "не перемещает ли он решение о выборе зависимости всего лишь в другое место?". Да, это именно так, и это хорошо; это означает, что вы получаете центральное место, в котором можете соединять классы. Composition Root играет роль стороннего компонента, который связывает покупателей с их сервисами. В действительности, Нэт Прайс предпочитает термин "Стороннее соединение" вместо "механизм внедрения зависимостей" именно по этой причине.

Чем дольше вы будете откладывать принятие решения о том, как соединять классы, тем дольше вы будете сохранять ваши опции открытыми. Таким образом, Composition Root необходимо размещать как можно ближе к точке входа в приложение.

Примечание

Мне нравится думать о Composition Root как об архитектурном эквиваленте понятия *Бережливой разработки программного обеспечения* (Lean Software Development): Последний ответственный момент. Смысл в том, чтобы откладывать все решения настолько долго, насколько это позволяют правила приличия (но не дольше), потому что

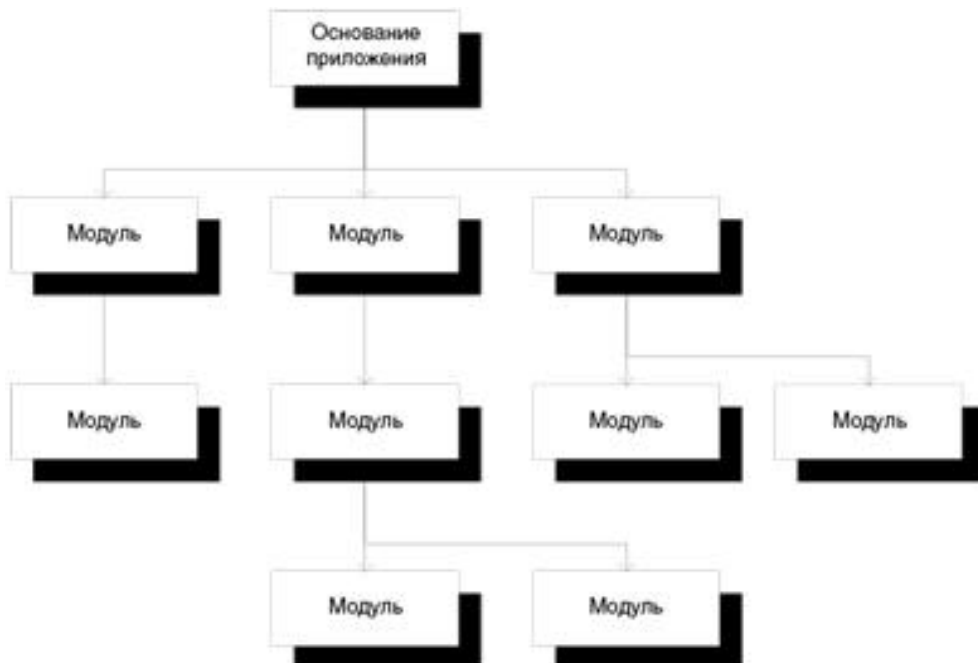
нам хотелось бы сохранять наши опции открытыми и основывать наши решения на как можно большем объеме информации. Когда дело доходит до компоновки приложений, мы можем таким же образом отложить принятие решения о передаче зависимостей в основание приложения.

Даже у модульного приложения, которое для своего формирования использует слабое связывание и позднее связывание, есть основание, которое содержит точку входа в приложение. Ниже приведены примеры:

- Консольное приложение – исполняемый файл (.exe) с методом Main.
- ASP.NET веб-приложение – это библиотека (.dll) с обработчиком событий Application_Start в файле Global.asax.
- WPF приложение – исполняемый файл (.exe) с файлом App.xaml.
- WCF сервис – это библиотека (.dll) с классом, который наследуется от интерфейса сервиса, несмотря на то, что вы можете заполучить более низкоуровневую точку входа путем создания пользовательской ServiceHostFactory.

Существует множество других технологий, но общим для всех этих технологий является то, что один модуль содержит точку входа в приложение: это *основание* приложения, что проиллюстрировано на рисунке 3-7. Composition Root приложения должно располагаться в основании приложения таким образом, чтобы оно могло правильно формировать приложение.

Рисунок 3-7: Точка входа в приложение – это основание модульного приложения. Либо напрямую, либо не напрямую основание использует остальные модули. Composition Root должно размещаться в основании приложения – как можно ближе к точке входа



Вы не должны пытаться компоновать классы в любом модуле, потому что такой подход ограничивает ваши возможности. Все классы модулей приложения должны использовать Constructor Injection (или, в редких случаях, один из других паттернов, описанный в главе "DI-паттерны") и оставлять задачу формирования диаграммы объектов приложения за

Composition Root. Любой используемый DI-контейнер должен быть ограничен Composition Root.

Использование DI-контейнера в Composition Root

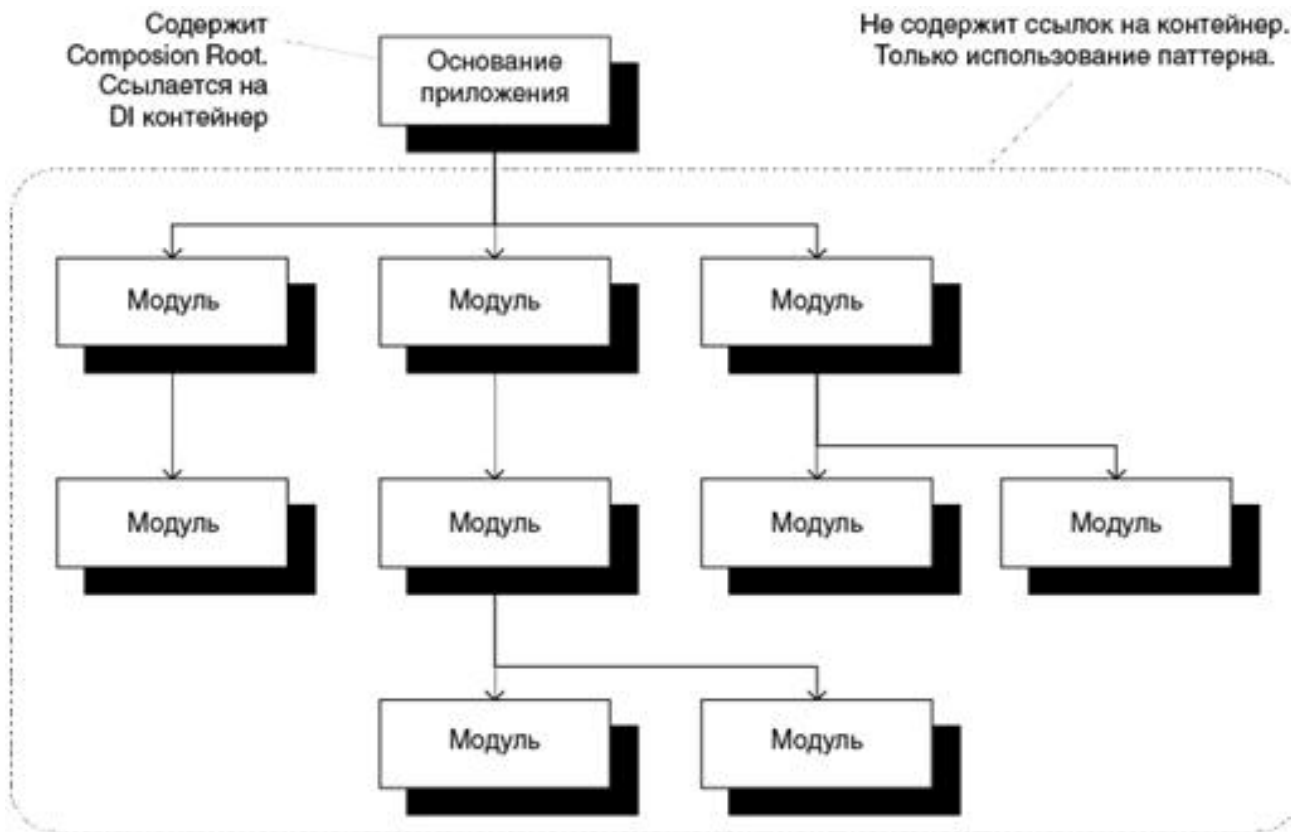
DI-контейнер может ошибочно использоваться в качестве Service Locator, но его можно использовать только как движок, который формирует диаграммы объектов. При рассмотрении DI-контейнера с этой точки зрения имеет смысл ограничить его Composition Root. Такой подход также имеет большое преимущество, заключающееся в удалении любого связывания между DI-контейнером и остальной частью базы кода.

Подсказка

К DI-контейнеру следует обращаться из Composition Root. Все остальные модули не должны ссылаться на контейнер.

На рисунке 3-8 вы можете увидеть, что только Composition Root ссылается на DI-контейнер. Остальная часть приложения не имеет ссылок на контейнер и вместо него полагается на паттерны, описанные в главе "DI-паттерны". DI-контейнеры понимают эти паттерны и используют их для того, чтобы формировать диаграмму объектов приложения.

Рисунок 3-8: Только Composition Root, содержащееся в основании приложения, должно иметь ссылку на DI-контейнер. Все остальные модули приложения должны целиком полагаться на DI-контейнеры и не ссылаться на контейнер



Composition Root может быть реализован с DI-контейнером. Это означает, что вы используете контейнер для формирования полноценной диаграммы объектов приложения в вызове единственного метода `Resolve`. Всякий раз, когда я говорю разработчикам о том,

чтобы они поступали именно так, я всегда могу сказать, что это доставляет им неудобства, поскольку они боятся, что это ужасно неэффективно и неудобно для выполнения. Вам не придется об этом беспокоиться, потому что это почти никогда не случается, и в тех немногих ситуациях, когда это происходит, существуют способы решения этой проблемы.

Подсказка

Не беспокойтесь об издержках выполнения, возникающих в ситуациях, когда DI-контейнер используется для формирования большой диаграммы объектов. Это почти никогда не является проблемой.

Когда дело касается приложений, в основе которых лежат запросы, например веб-сайтов или сервисов, вы можете сконфигурировать контейнер всего один раз, но преобразовывать диаграмму объектов для каждого входящего запроса. Шаблонное коммерческое приложение является примером такой ситуации.

Пример: Реализация Composition Root

Шаблонное коммерческое приложение из раздела "Расширение шаблонного приложения" должно обладать Composition Root для того, чтобы формировать диаграммы объектов для входящих HTTP-запросов. Что касается всех остальных .NET веб-приложений, то для них точкой входа является метод `Application_Start` файла `Global.asax`.

В этом примере я использую DI-контейнер Castle Windsor, но код может быть таким же и для любого другого контейнера. Для Castle Windsor метод `Application_Start` мог бы выглядеть следующим образом:

```
protected void Application_Start()
{
    MvcApplication.RegisterRoutes(RouteTable.Routes);

    var container = new WindsorContainer();
    container.Install(new CommerceWindsorInstaller());

    var controllerFactory = new WindsorControllerFactory(container);

    ControllerBuilder.Current.SetControllerFactory(controllerFactory);
}
```

Перед тем как вы сможете сконфигурировать контейнер, вы должны создать новый экземпляр. Поскольку полноценная установка приложения инкапсулирована в классе под названием `CommerceWindsorInstaller`, вы устанавливаете его в контейнер для того, чтобы настроить этот контейнер. Код `CommerceWindsorInstaller`, очевидно, реализован с помощью API Castle Windsor, но концептуально он идентичен примеру из раздела "Конфигурирование DI-контейнеров".

Для того чтобы позволить контейнеру подключать контроллеры в приложение, вы должны применить соответствующий шов ASP.NET MVC, который называется `IControllerFactory` (подробно рассматривается в разделе "Построение ASP.NET MVC приложений"). На данный момент достаточно понимать, что для интеграции с ASP.NET MVC вы должны создать адаптер в рамках контейнера и сообщить об этом фреймворку.

Поскольку метод `Application_Start` выполняется всего лишь раз, контейнер является единственным экземпляром, который инициализируется всего один раз. При получении

запросов этот экземпляр контейнера должен обрабатывать все запросы одновременно – но поскольку все контейнеры реализованы с помощью методов `Resolve`, корректно исполняемых в многопоточной среде, это не является проблемой.

Поскольку вы устанавливаете ASP.NET MVC с пользовательской `WindsorControllerFactory`, он будет вызывать ее метод `GetControllerInstance` для каждого входящего HTTP-запроса (подробнее об этом вы можете прочитать в разделе "Построение ASP.NET MVC приложений"). Реализация делегирует работу контейнеру:

```
protected override IController GetControllerInstance(  
    RequestContext requestContext, Type controllerType)  
{  
    return (IController)this.container.Resolve(controllerType);  
}
```

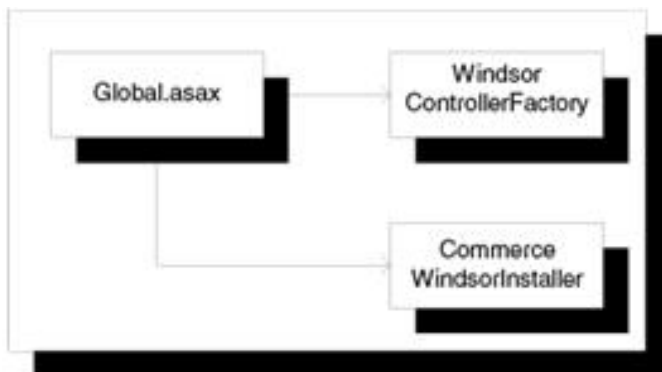
Обратите внимание на то, что вы более или менее вернулись к вводным примерам из раздела "Контейнер "Hello"". Метод `Resolve` формирует окончательную диаграмму, которая должна использоваться для этого конкретного запроса, и возвращает эту диаграмму. Это единственное место в приложении, где вы вызываете метод `Resolve`.

Подсказка

База кода приложения должна содержать единственный вызов метода `Resolve`.

В этом примере `Composition Root` развернут в рамках нескольких классов, что продемонстрировано на рисунке 3-9. Это предсказуемо – важно, что все классы содержатся в одном и том же модуле, которым в данном случае является основание приложения.

Рисунок 3-9: `Composition Root` развернут в рамках трех классов, но все они определены в пределах одного и того же модуля



Самое важное, на что здесь нужно обратить внимание, – это то, что эти три класса являются единственными классами во всем шаблонном приложении, которые ссылаются на DI-контейнер. Вся остальная часть кода приложения использует только паттерн `Construction Injection`; вернитесь назад и перечитайте главу "Комплексный пример", если вы мне не верите.

Подсказка

Мне хотелось бы обобщить все правила, содержащиеся в данном разделе, перефразировав голливудский принцип: не вызывайте контейнер; он сам вызовет вас.

Common Service Locator

Существует проект с открытым исходным кодом под названием Common Service Locator (<http://commonservicelocator.codeplex.com/>), целью которого является отсоединение кода приложения от конкретного DI-контейнера путем скрытия каждого контейнера за универсальным интерфейсом IServiceLocator.

Надеюсь, что такое объяснение того, как Composition Root эффективно отделяет остальную часть кода приложения от DI-контейнеров, теперь позволит вам понять, почему вам не нужен Common Service Locator. Как я буду объяснять в разделе "Service Locator", в связи с тем, что Service Locator является анти-паттерном, будет лучше не использовать его – тем более с Composition Root он вам и не нужен.

Более подробно о том, как реализовывать Composition Roots в различных фреймворках (включая ASP.NET MVC), вы можете прочитать в главе "Построение объектов". В данном контексте то, как вы это делаете, более важно, чем то, где вы это делаете. Как и подразумевает его название, Composition Root – это составляющая основания приложения, в которой вы компонуете все слабо связанные классы. Это справедливо и при использовании DI-контейнера, и при использовании Poor Man's DI. Тем не менее, при использовании DI-контейнера вам следует руководствоваться паттерном Register Resolve Release.

Register Resolve Release

Как нам следует использовать DI-контейнер?

Следуя строгой последовательности вызовов методов Register Resolve Release

Паттерн Composition Root описывает то, где вам следует использовать DI-контейнер. Тем не менее, он не сообщает о том, как его использовать. Паттерн Register Resolve Release отвечает на этот вопрос.

DI-контейнер следует использовать в трех, следующих друг за другом фазах называемых Register, Resolve и Release. В таблице 3-3 более подробно описана каждая из этих фаз

Определение

Паттерн Register Resolve Release указывает на то, что методы DI-контейнера должны вызываться в этой строгой последовательности: Регистрация (*Register*), Решение (*Resolve*) и Освобождение (*Release*) (см. рисунок 3-10).

Рисунок 3-10: Методы DI-контейнера должны вызываться в продемонстрированной строгой последовательности: сначала метод Register, затем метод Resolve и в завершении метод Release



Таблица 3-3: Фазы контейнера

Фаза	Что происходит на данной фазе?	Что прочитать дальше
Register	Регистрация компонентов контейнера. Вы конфигурируете контейнер, сообщая ему о том, какие классы он может использовать, как ему следует преобразовывать абстракции в конкретные типы и о том (необязательная информация), как определенные классы должны соединяться вместе.	В разделе "Конфигурирование DI-контейнеров" я уже обсуждал то, как сконфигурировать DI-контейнер. В части "DI-контейнеры" я буду подробно обсуждать конфигурацию шести индивидуальных DI-контейнеров.
Resolve	Преобразует компоненты основания. Запрос одного типа преобразуется в одну диаграмму объектов.	В разделе "Знакомство с DI-контейнерами" мы увидели, как преобразовывать диаграммы объектов с помощью DI-контейнера. В части "DI-контейнеры" вы можете более подробно почитать об API конкретных контейнеров.
Release	Освобождает компоненты из контейнера. Все диаграммы объектов, образованные на предыдущей фазе, должны быть освобождены в тот момент, когда в них уже нет нужды. Это сигнализирует о том, что контейнер может очистить диаграмму объектов, что определенно важно в тех случаях, когда некоторые компоненты являются лишними.	В главе "Жизненный цикл объектов" я обсуждаю вопрос управления жизненным циклом объектов, включая важность процесса очистки объекта. Кроме того в части "DI-контейнеры" я рассматриваю API управления жизненным циклом индивидуальных DI-контейнеров.

Вы должны использовать эти три фазы в правильном порядке и вам не разрешается волей-неволей двигаться назад или вперед. К примеру, вам не следует возвращаться и заново конфигурировать контейнер, как только вы приступили к преобразованию диаграммы объектов. Иногда люди спрашивают о том, как добавить в контейнер дополнительные компоненты уже после того, как они приступили к преобразованию компонентов. Не делайте этого – это принесет вам одни огорчения.

Примечание

Некоторые DI-контейнеры не поддерживают явный Release диаграмм объектов и вместо этого полагаются на .NET сборщика мусора (garbage collector). При использовании таких контейнеров вы должны применять модифицированный паттерн Register Resolve и обращаться к потенциальным утечкам ресурсов в ваших реализациях объекта. Более подробно читайте об этом в главе "Жизненный цикл объектов".

В следующем разделе я буду говорить о методах Register, Resolve и Release, а также о фазах. Castle Windsor действительно обладает тремя методами с точно такими же

названиями, а фазы названы в честь этих методов. Другие DI-контейнеры могут использовать другие названия, но их основополагающая сущность идентична. Я использую названия только Castle Windsor, поскольку они обеспечивают логичную терминологию – а также приятную аллитерацию.

Статическая структура

В своем истинном представлении паттерн Register Resolve Release утверждает, что вам следует выполнять вызов только одного метода в каждой фазе. Кшиштоф Козмиц называет это паттерном трех вызовов (Three Calls Pattern) – вам разрешено сделать только три вызова методов в контейнере.

Методы Resolve и Release упрощают это. В разделе "Composition Root" я уже утверждал о том, что в приложении должен содержаться единственный вызов метода Resolve. Как следствие вы должны всегда выпускать то, что вы преобразовываете.

Подсказка

Любая диаграмма объектов, сформированная с помощью метода Resolve, должна быть списана при помощи метода Release.

Конфигурирование DI-контейнера в единственном вызове метода требует более подробных разъяснений. Причиной того, что регистрация компонентов должна происходить в рамках единственного вызова метода, является то, что вы должны расценивать конфигурацию DI-контейнера как единичное, элементарное действие. После завершения конфигурации контейнер должен расцениваться как доступный только для чтения.

Autofac даже делает это представление явным при помощи выделения конфигурации контейнера в отдельный `ContainerBuilder`: вы регистрируете компоненты с помощью `ContainerBuilder`, и когда вы делаете это, вы просите `ContainerBuilder` создать экземпляр контейнера из конфигурации. В Autofac вы не конфигурируете контейнер напрямую.

При рассмотрении конфигурации в виде элементарного действия становится проще управлять кодом конфигурации, поскольку становится очевидным, в каком месте она должна выполняться. Многие DI-контейнеры также используют эту концепцию для того, чтобы замораживать конфигурацию, как только вы приступаете к преобразованию диаграмм объектов из него. Такой подход заставляет их выполняться лучшим образом.

Если вы вспомните листинг 3-2, то можете заявить, что он содержит более одного вызова метода. Регистрация всегда включает в себя множество операторов, но большинство DI-контейнеров обладают механизмом пакетирования, который позволяет инкапсулировать все эти конфигурационные операторы в единственном классе (возможно состоящим из других классов). Autofac называет их модулями (Modules), StructureMap называет их реестрами (Registries), а Castle Windsor – инсталляторами (Installers). Общим для всех них является тот факт, что все они могут использоваться для конфигурации контейнера с помощью единственного вызова метода. В разделе "Composition Root" вы уже видели, как Castle Windsor использует инсталлятор:

```
container.Install(new CommerceWindsorInstaller());
```

Для DI-контейнеров, не обладающих механизмом пакетирования, вы всегда можете создать пользовательский класс, который инкапсулирует конфигурацию в единичном методе.

К совету, заключающемуся в том, что для Resolve и Release необходимо использовать только по одной единственной строке кода, следует относиться серьезно – но для фазы Register данный факт должен восприниматься более концептуально. Важным вопросом является тот факт, что регистрация должна быть завершена до вызова метода Resolve. Рисунок 3-11 иллюстрирует то, как выглядит последовательность, включая инкапсуляцию множества вызовов метода Register.

Рисунок 3-11: На фазе Register может иметь место любое количество вызовов метода Register, но вы все еще должны расценивать его как элементарное действие. На фазах Resolve и Release вам, буквально, нужно только по одному вызову каждого метода



Универсальным источником заблуждений является то, что паттерн трех вызовов создает твердое убеждение о том, как часто каждый метод должен появляться в базе кода, но он ничего не говорит о том, сколько раз он должен вызываться.

Динамическое взаимодействие

Название паттерна Three Calls Pattern может привести вас к мысли о том, что каждый метод должен вызываться всего лишь раз. Источник этого заблуждения находится в самом названии, и это одна из нескольких причин того, почему я предпочитаю название Register Resolve Release.

Паттерн Three Calls утверждает, что для вызова каждого метода должна существовать единственная строка кода. Тем не менее, в зависимости от обстоятельств, некоторые методы могут вызываться более одного раза.

В однопоточном приложении таком, как настольное приложение, утилита командной строки или пакетное задание, каждый метод обычно вызывается только один раз, что проиллюстрировано на рисунке 3-12.

Рисунок 3-12: В однопоточном приложении каждый метод будет вызываться только один раз. За конфигурацией контейнера непосредственно следует формирование диаграммы объектов приложения, которая выполняет реальную работу. После окончания работы перед выходом из приложения вызывается метод Release.



В приложениях, основанных на запросах, таких, как веб-сайт, веб-сервис или получатель асинхронного сообщения, Composition Root формирует диаграмму объектов для каждого входящего запроса. В этом типе приложения, как это проиллюстрировано на рисунке 3-13, метод `Register` вызывается только один раз, тогда как методы `Resolve` и `Release` вызываются попарно для каждого запроса – потенциально большее число раз.

Рисунок 3-13: В приложениях, основанных на запросах, метод `Register` вызывается только один раз, тогда как методы `Resolve` и `Release` вызываются много раз – по одному на каждый запрос



Важно отметить, что вам нужно сконфигурировать контейнер только один раз. Контейнер – это совместно используемый экземпляр, применяемый для преобразования составных запросов, но конфигурация должна оставаться стабильной и завершенной.

Динамическая картина `Register Resolve Release` – это почти инверсия статического представления – сравните рисунок 3-11 и рисунок 3-13. В статическом представлении мы допускаем составные строки кода, которые вызывают метод `Register`, а в динамическом представлении этот блок кода должен вызываться строго один раз. С другой стороны, статическим правилом является то, что вы должны иметь только одну строку кода, которая вызывает `Resolve` и `Release`, но во время выполнения они могут вызываться много раз.

Это может казаться сложным и трудным, но как демонстрирует следующий пример, этого всего лишь вызовы трех методов.

Пример: использование `Register Resolve Release`

В данном примере вы будете реализовывать Composition Root шаблонного приложения из раздела "Расширение шаблонного приложения" с помощью DI-контейнера Castle Windsor. Это тот же самый контейнер, который вы использовали в примере раздела "Composition Root", поэтому этот пример нужно читать как продолжение предыдущего.

Точкой входа в приложение является метод `Application_Start`, и поскольку это приложение является веб-сайтом, фаза `Register` изолирована от фаз `Resolve` и `Release`, потому что вы должны сконфигурировать контейнер всего лишь раз. Код остается таким же, как и в предыдущем примере, но я хочу немного сместить фокус:

```
protected void Application_Start()
{
    MvcApplication.RegisterRoutes(RouteTable.Routes);

    var container = new WindsorContainer();
    container.Install(new CommerceWindsorInstaller());

    var controllerFactory = new WindsorControllerFactory(container);

    ControllerBuilder.Current.SetControllerFactory(controllerFactory);
}
```

Согласно паттерну Register Resolve Release вызов первого метода, который вы совершаете для экземпляра container, должен быть вызовом элементарного Register. В данном случае метод называется Install и CommerceWindsorInstaller инкапсулирует индивидуальные регистрации в единственном классе. Следующий листинг демонстрирует реализацию CommerceWindsorInstaller.

Листинг 3-3: Инкапсуляция составных регистраций

```
1.  public class CommerceWindsorInstaller : IWindsorInstaller
2.  {
3.      public void Install(IWindsorContainer container,
4.          IConfigurationStore store)
5.      {
6.          container.Register(AllTypes
7.              .FromAssemblyContaining<HomeController>()
8.              .BasedOn<IController>()
9.              .Configure(r => r.LifeStyle.PerWebRequest));
10.
11.         container.Register(AllTypes
12.             .FromAssemblyContaining<BasketService>()
13.             .Where(t => t.Name.EndsWith("Service"))
14.             .WithService
15.             .FirstInterface());
16.
17.         container.Register(AllTypes
18.             .FromAssemblyContaining<BasketDiscountPolicy>()
19.             .Where(t => t.Name.EndsWith("Policy"))
20.             .WithService
21.             .Select((t, b) => new[] { t.BaseType }));
22.
23.         string connectionString = ConfigurationManager
24.             .ConnectionStrings["CommerceObjectContext"]
25.             .ConnectionString;
26.
27.         container.Register(AllTypes
28.             .FromAssemblyContaining<SqlProductRepository>()
29.             .Where(t => t.Name.StartsWith("Sql"))
30.             .WithService
31.             .Select((t, b) => new[] { t.BaseType })
32.             .Configure(r => r.LifeStyle.PerWebRequest
33.                 .DependsOn((new
34.                     {
35.                         connString = connectionString
36.                     }))) );
37. }
```

Строка 5, 10, 16, 26: Вызовы Register

CommerceWindsorInstaller кажется сложным, но важно отметить, что он инкапсулирует четыре вызова метода Register и что это единственный способ, с помощью которого он взаимодействует с контейнером. Остальная часть кода сейчас не важна. В нем используются соглашения для конфигурации контейнера. Более подробно об API механизма автоматической регистрации Castle Windsor вы можете прочитать в разделе "Конфигурирование контейнера" (глава "Castle Windsor").

Поскольку шаблонное приложение является веб-сайтом, Resolve и Release должны быть реализованы в единой паре. Для каждого HTTP-запроса вы должны преобразовать диаграмму объектов, которая будет обрабатывать этот запрос, а когда это закончится, вы должны выпустить его снова. Вы делаете это из класса под названием WindsorControllerFactory, который наследуется от DefaultControllerFactory, относящегося к ASP.NET MVC – более подробно о шве ASP.NET MVC вы можете прочитать в разделе "Построение ASP.NET MVC приложений".

ASP.NET MVC фреймворк вызывает метод GetControllerInstance для того, чтобы преобразовать IControllers и метод ReleaseController при обработке запроса. Ниже приведены подходящие для нас методы вызова методов Resolve и Release:

```
protected override IController GetControllerInstance(  
    RequestContext requestContext, Type controllerType)  
{  
    var controller = this.container.Resolve(controllerType);  
    return (IController)controller;  
}  
  
public override void ReleaseController(IController controller)  
{  
    this.container.Release(controller);  
}
```

В методе GetControllerInstance вы передаете аргумент controllerType в метод Resolve и возвращаете результирующую диаграмму объектов. При обработке запроса ASP.NET MVC фреймворк вызывает метод ReleaseController с экземпляром IController, созданным до этого с помощью метода GetControllerInstance, и вы можете передать этот экземпляр controller в метод Release.

Обратите внимание на то, что это единственное появление методов Resolve и Release во всей базе кода приложения.

Этот пример погружается несколько глубже предыдущего примера, который демонстрировал паттерн Composition Root, но в сущности это тот же самый код. Паттерн Composition Root указывает, где вам следует формировать диаграммы объектов, тогда как Register Resolve Release имеет дело с тем, как использовать DI-контейнер в пределах Composition Root.

В следующей главе я сделаю обзор большего количества DI-паттернов, но перед тем, как это сделать, я хочу сделать небольшой крюк и обсудить то, как DI-контейнеры приспособляются к всеобщей .NET экосистеме.

3.4. Перспектива DI-контейнеров

Теперь, когда я описал, что такое DI-контейнер и как применить его в Composition Root, я хочу слегка сменить темп и предоставить вам обзор текущего состояния DI-контейнеров в .NET экосистеме. Это такие самые устойчивые аспекты DI-контейнеров, как историческое происхождение, и то, почему доступно так много контейнеров с открытым исходным кодом.

Поскольку для выбора предлагается большое изобилие DI-контейнеров, я также хочу предоставить вам некоторое руководство по тому, как выбрать контейнер.

Выбор DI-контейнера

Решение использовать механизм внедрения зависимостей в качестве технологии не должно зависеть от выбора конкретного DI-контейнера. Механизм внедрения зависимостей в первую очередь является технологией, и я буду использовать Poog man's DI на протяжении почти всех частей "Каталог DI" и "DI самостоятельно" для того, чтобы подчеркнуть этот вопрос.

Однако DI-контейнер упростит вашу жизнь, поэтому используйте его там, где только можете. Если использовать контейнеры согласно выделенным в этой книге паттернам, то можно столкнуться с несколькими недостатками, но нужно рассмотреть еще кое-что.

Процесс принятия решения

DI-контейнер – это стабильная зависимость, поэтому с точки зрения DI использование DI-контейнера не является проблемой, но существуют другие, второстепенные проблемы, которые нужно рассмотреть:

- Добавление другой библиотеки всегда добавляет некоторую сложность в приложение – не в терминах удобства сопровождения, а в терминах кривой обучения. Новым разработчикам нужно будет не только понять код приложения, но также понять API выбранного DI-контейнера. Надеюсь, что в этой главе я сумел дать вам понять, что с помощью изолированного использования контейнера в Composition Root вы сможете защитить контейнер от начинающих разработчиков. Если вы используете механизм автоматической регистрации, то контейнер может даже взять на себя заботу об инфраструктуре, не привлекая при этом к себе внимание.
- За исключением платформы Managed Extensibility Framework (MEF), вам нужно разворачивать сборки DI-контейнера вместе со своим приложением. Это могло бы потенциально иметь легальные последствия, хотя это и не правдоподобно. Все универсальные DI-контейнеры с открытым исходным кодом имеют разрешающие лицензии, но я не судья, поэтому не делайте ставку на мои слова: проконсультируйтесь со своим собственным консультантом.
- И еще раз за исключением MEF все остальные DI-контейнеры являются библиотеками с открытым исходным кодом. Для каждого из этих контейнеров вам приходится оценивать, насколько вы доверяете людям или организации, создавшим эти контейнеры.
- Существуют технические различия между различными DI-контейнерами. Во введении к части "DI-контейнеры" я предоставил вам таблицу, в которой перечисляются преимущества и недостатки каждого контейнера, рассматриваемого

в этой книге. Вы можете использовать эту таблицу как стартовую точку, а потом чтение глав о каждом из контейнеров станет для вас интересным.

Выбор DI-контейнера не должен быть сложным. Возьмите один для раскрутки, а потом посмотрите, подходит ли он вашим нуждам – если не подходит, то замените его другим. Когда вы заключите DI-контейнер в Composition Root, вы сможете заменять контейнеры относительно свободно.

Выбранные DI-контейнеры

Я не хочу рассказывать вам о том, какой DI-контейнер выбрать. Выбор DI-контейнера включает в себя больше чем просто техническую оценку. Вы также должны оценить, подходит ли вам лицензионная модель, доверяете ли вы людям и организации, которая разрабатывает и поддерживает в работоспособном состоянии DI-контейнер, насколько он подходит IT-стратегии вашей организации и т.д.

Большинство .NET DI-контейнеров являются проектами с открытым исходным кодом – это также следует иметь в виду, поскольку для них может и не быть официальной поддержки, а документация часто ограничена.

В таблице 3-4 перечисляются DI-контейнеры, рассматриваемые в части "DI-контейнеры". Выбор основывается на таких критериях, как релевантность, доля на рынке, характерные черты, но этот список является субъективным и неокончательным. Несколько популярных контейнеров (например, Ninject) не включены в этот список, главным образом, из-за ограничений во времени и пространстве.

Таблица 3-4: Выбранные DI-контейнеры. Доступно больше контейнеров, но эти выбраны либо потому что они широко используются, либо они представляют собой интересный аспект DI или потому что они станут важными в будущем

Название	Организация	Комментарии
Castle Windsor	Open Source	Состоявшийся и широко используемый
StructureMap	Open Source	Состоявшийся и широко используемый
Spring.NET	SpringSource	Состоявшийся и широко используемый порт Java Spring DI-контейнера
Autofac	Open Source	Наиболее современный DI-контейнер, созданный на основании характеристик языка C# 3.0
Unity	Microsoft patterns&practices	Первая роль компании Microsoft в пространстве DI, но не продукт по существу
Managed Extensibility Framework (MEF)	Microsoft	Поставляется вместе с .NET 4, но в действительности не является DI-контейнером

Часть "DI-контейнеры" посвящена этим DI-контейнерам, причем для каждого из них выделена отдельная глава.

Обратите внимание на то, как в этой области доминируют проекты с открытым исходным кодом и другие некоммерческие проекты, причем компания Microsoft играет незначительную роль в этом деле.

Microsoft и DI

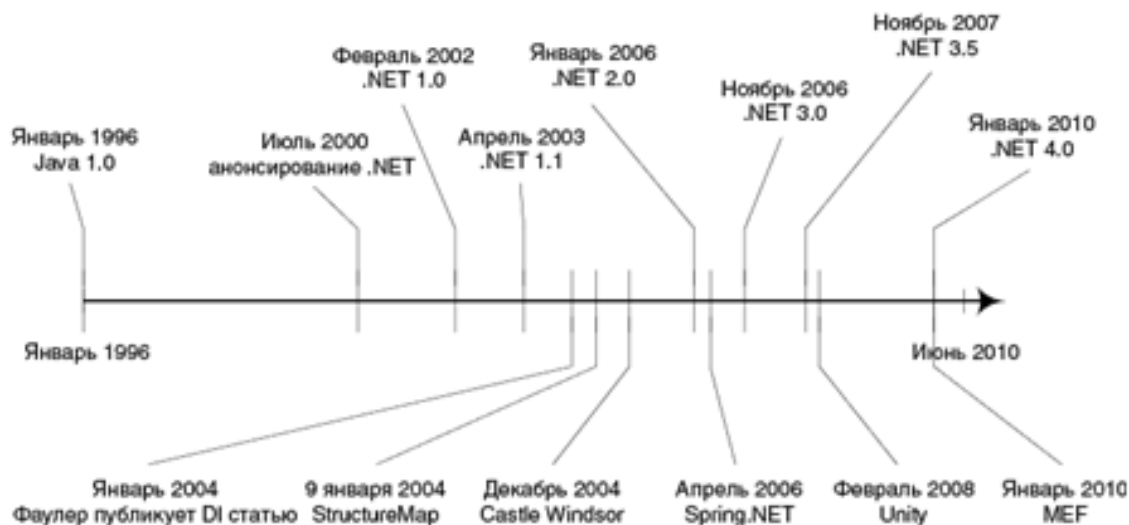
Несмотря на то, что платформа .NET является продуктом компании Microsoft, другие организации (часто отдельные лица) становятся более известными, когда дело доходит до механизма внедрения зависимостей в .NET. Вкратце это может относиться к тому факту, что компания Microsoft не предлагает ни одного DI-контейнера в стандартной библиотеке классов (BCL). Хотя как предложение единственным DI-контейнером компании Microsoft является появившийся относительно недавно Unity.

Думаю, справедливо будет сказать, что за множество первых лет жизни .NET Framework компания Microsoft блаженно игнорировала любую сущность DI. Не просто точно объяснить, почему так происходило, и я сомневаюсь, что это когда-либо было явной стратегией.

Краткая история DI в .NET

Ниже приведена моя субъективная попытка сделать обзор истории DI в .NET для того, чтобы объяснить, почему компания Microsoft столь долго игнорировала механизм внедрения зависимостей (см. рисунок 3-14). Насколько я осведомлен, на этот вопрос нет авторитетного ответа.

Рисунок 3-14: Временные рамки релизов выбранных платформ и DI-контейнеров. Обратите внимание на то, насколько состоявшейся оказалась Java в 2004 году по сравнению с .NET.



Кажется, что до того, как попасть в .NET, DI выросла из сообщества Java с открытым исходным кодом. Мартин Фаулер опубликовал свою статью о DI в начале 2004 года в качестве реакции на непрерывную работу. В это время текущей версией .NET была версия 1.1, и компания Microsoft работала над версией .NET 2.0, в то время как Java быстрыми темпами достигала своего десятилетия. Я верю, что компания Microsoft в то время просто направила свои усилия в другом направлении. Даже если бы они в то время были хорошо осведомлены насчет DI, думаю, для них все равно были бы приоритетны другие возможности, например, дженерики.

В июне 2004 года был выпущен DI-контейнер StructureMap, обогнав Castle Windsor всего на полгода.

В конце 2005 года компания Microsoft выпустила .NET 2.0, главной новой возможностью которой стали дженерики, а затем Microsoft решила сфокусироваться на WCF, WPF, а позднее LINQ для своего следующего крупного релиза (Visual Studio 2008).

Тем временем DI медленно приобретал популярность. В 2006 году появился Spring.NET.

В начале 2008 Microsoft patterns&practices выпустили Unity, которая для большинства последователей школы Microsoft казалась приближением к DI. Этот четырехгодовой промежуток времени дал опытным личностям хороший старт, а популярность DI-контейнеров, подобных StructureMap и Castle Windsor, возросла.

О снова DI

Интересным наблюдением из рисунка 3-14 является то, как быстро члены сообщества .NET разработчиков подхватили идею DI. Согласно домашней странице Castle Windsor понятия зарождались даже до статьи Фаулера:

Castle вырос из проекта Apache Avalon в середине 2003 года в виде попытки создания самой простой инверсии управления контейнером.

Веб-страничка Castle от 01.08.2008

На протяжении длительного времени для .NET DI-контейнеры оставались источником движения, а многие ведущие члены симпатизировали разработке Agile. В действительности даже если архитектура модульного приложения имеет множество различных преимуществ, все равно кажется, что в первую очередь именно вопрос тестируемости мотивировал людей на разработку и использование DI-контейнеров (что также является справедливым и в моем случае).

В это время официальной методологией разработки в компании Microsoft была Microsoft Solutions Framework (MSF) 3.0 – каскадный процесс, который оставляет небольшое пространство для таких приемов разработки Agile, как разработка через тестирование (TDD). Короче говоря, это было совершенно другим образом мышления.

С течением времени разработка Agile, TDD и DI доказали свою эффективность и приобрели популярность, а Microsoft, казалось, также медленно продвигалась к поддержке этого стиля. В противоположность ситуации 2004 года команды разработки теперь открыто обсуждают DI, TDD и другие вопросы, относящиеся к разработке Agile.

DI-контейнеры компании Microsoft

С течением времени команда patterns&practices компании Microsoft разработала множество корректур для различных областей, относящихся к .NET. Большая часть опыта, приобретенного из этих проектов, использовалась для определения рамок дальнейшей разработки самого .NET Framework. В качестве примера Updater Application Block предоставил богатый опыт, который в дальнейшем использовался при разработке ClickOnce.

В начале 2008 года команда patterns&practices выпустила свою первую Community Technical Preview (СТР) (небольшая ознакомительная версия) версию Unity, их новый DI-контейнер, а в апреле 2008 года вышел релиз 1.0. Unity – это развитый DI-контейнер, который поддерживает композицию объектов, механизм управления жизненным циклом и

механизм перехвата. Это не продукт компании Microsoft, но скорее проект с открытым исходным кодом, который возможно разработан Microsoft.

Object Builder

Кажется, существует некоторая путаница в вопросе о том, когда точно patterns&practices представили миру DI-контейнер. При введении в начале 2006 года Enterprise Library для .NET 2.0 команды patterns&practices она содержала модуль с названием Object Builder, который использовался для создания сложных объектов из составных элементов.

Она была атрибутно-управляемой и работала только для классов, которые были сильно интегрированы с самим Object Builder. Его никогда не представляли как DI-контейнер, хотя было общепризнано, что на его основе можно было бы создать DI-контейнер.

Многие люди ошибочно верят, что Object Builder был первым DI-контейнером компании Microsoft, но это не правда: этот титул получил Unity.

Вместе с .NET 4.0 компания Microsoft предоставила Managed Extensibility Framework (MEF), который отмечает, что к первым DI сущностям явно обращались в рамках самой .NET. При первом выпуске MEF не был полноценным контейнером, который поддерживал все три аспекта DI, но скорее был движком, который фокусировался на композиции объектов.

Команда MEF хорошо осведомлена о таких аспектах, как управление жизненным циклом и механизм перехвата, поэтому не маловероятно, что мы будем свидетелями того, что MEF будет включен в полноценный DI-контейнер в следующие несколько лет (как я пишу здесь, технические обзоры указывают на то, что это будет именно так).

3.5. Резюме

DI-контейнер может стать чрезвычайно полезным инструментом, если вы правильно его используете. Самое важное – понять, что использование механизма внедрения зависимостей никоим образом не зависит от использования DI-контейнера. Приложение может состоять из множества слабо связанных классов и модулей, и ни один из этих модулей не должен ничего знать о контейнере. Наиболее эффективный способ убедиться в том, что код приложения не осведомлен ни об одном DI-контейнере, – жестко реализовать паттерн Register Resolve Release в Composition Root. Это эффективно предостережет вас от случайного применения анти-паттерна Service Locator, потому что он заключает контейнер в небольшую изолированную область кода.

Если использовать DI-контейнер таким образом, то он станет движком, который берет на себя заботу о большей части инфраструктуры приложения. Он формирует диаграмму объектов на основании его конфигурации. Это может быть особенно существенно, если вы применяете конфигурацию, базирующуюся на соглашениях – при соответствующей реализации он может заботиться о формировании диаграммы объектов, и вы можете сконцентрировать ваши усилия на добавлении новых классов, реализующих новые возможности. Контейнер будет автоматически обнаруживать новые классы, которые соответствуют установленным соглашениям, и делать их доступными для пользователей.

В некоторых случаях вам нужно больше явного контроля над конфигурацией контейнера. Наиболее эффективно использовать код в качестве конфигурации, но если вам нужно поддерживать позднее связывание, вы также можете использовать XML для конфигурации DI-контейнеров.

Эта глава подытоживает первую часть книги. Целью первой части было знакомство с DI. Предыдущие главы в общих чертах знакомили с механизмом внедрения зависимостей, тогда как данная глава объясняет то, как DI-контейнеры относятся к DI, а также в общих чертах объясняет процесс проектирования приложения. Я подумал, что исторический обзор DI-контейнеров в .NET экосистеме как раз подходит для завершения главы, чтобы действительно ввести в игру различные контейнеры.

Глава представляет Composition Root и Register Resolve Release как два мини-паттерна, которые относятся к DI-контейнерам. В следующей главе мы сосредоточимся на паттернах проектирования.

Каталог DI

В части 1 представлен обзор DI, с обсуждением целей и преимуществ DI и объяснением, как DI *контейнеры* вписываются в общую картину. Хотя в главе 2 содержится развернутый пример, я уверен, что после первых глав у вас остались еще некоторые нерешенные вопросы. В части 2 мы копнем немного глубже, чтобы ответить на некоторые из этих вопросов.

Как следует из названия части 2, это каталог паттернов, анти-паттернов и рефакторинга. Некоторые люди не любят паттерны проектирования, потому что считают их сухими или слишком абстрактными. Лично я люблю паттерны, потому что они обеспечивают нас языком высокого уровня, который делает нас более эффективными и краткими, когда мы обсуждаем программное обеспечение. Это мое намерение – использовать данный каталог, чтобы показать язык паттернов для DI. Хотя описание паттерна должно содержать некоторые обобщения, я попытался сделать каждый шаблон конкретным, с использованием примеров.

Вы можете прочитать все три главы в последовательности, но каждый элемент в каталог также описан так, что вы можете прочитать о нем отдельно.

Глава 4 содержит мини-каталог паттернов проектирования DI. В некотором смысле, эти паттерны представляют собой нормативное руководство о том, как реализовать DI, но вы должны знать, что я не считаю, что все они имеют одинаковое значение. *Constructor Injection* на сегодняшний день является наиболее важным паттерном проектирования, тогда как все остальные паттерны должны рассматриваться как крайние случаи, которые могут применяться в определенных условиях. Паттерн *Ambient Context*, в частности, должен быть использован настолько редко, что я серьезно задумывался не включать его в книгу (я оставил его только потому, что те, кто читают книги перед публикацией, попросили меня оставить его).

В то время как глава 4 дает вам набор обобщенных решений, глава 5 содержит каталог ситуаций, которых необходимо избегать. Эти анти-паттерны описывают общие, но неправильные пути решения типичных проблем DI. В каждом случае анти-паттерн рассказывает, как выявить проблему и как ее решить. Важно знать и понимать эти анти-паттерны, чтобы избегать ловушек, которые они расставляют, и поскольку в главе 4 представлен наиболее важный паттерн, наиболее важным анти-паттерном является *Service Locator*.

Когда вы начнете применять DI в реальных задачах программирования, вы столкнетесь с некоторыми проблемами. Я думаю, у нас всех были моменты сомнений, когда мы чувствовали, что мы понимаем инструмент или технику, и все же мы думали: «*В теории это может сработать, но мой случай особенный ...*». Всякий раз, когда я ловлю себя на такой мысли, я понимаю, что мне многому еще предстоит научиться.

Во время моей карьеры я видел определенный набор проблем, которые появляются вновь и вновь. Каждая из этих проблем имеет общее решение, которое можно применить для перемещения вашего кода ко одному из паттернов DI из главы 4. В духе *рефакторинга паттернов* я решил назвать эту главу *DI рефакторингом*, потому что она содержит каталог проблем и соответствующих решений.

Часть 2 представляет полный каталог паттернов, анти-антипаттернов и рефакторинга. Я думаю, что это будет самая полезная часть книги, потому что она наиболее «устойчивая». Будем надеяться, что вы вернетесь к этим главам через месяцы или даже годы после первого прочтения.

[4. DI паттерны](#)

[5. DI анти-паттерны](#)

[6. DI рефакторинг](#)

4. DI паттерны

Меню

- Constructor Injection
- Property Injection
- Method Injection
- Ambient Context

Как и все профессионалы, повара имеют свой жаргон, который позволяет им общаться о сложных процессах приготовления пищи на своем языке, что часто звучит очень непонятно для остальных из нас. Не помогает даже то, что по большей части они используют термины, основанные на французском языке (даже если вы и говорите по-французски, такая проблема есть).

Соусы являются отличным примером того, как повара используют свою профессиональную терминологию. В главе 1 мы кратко обсудили *sauce béarnaise*, но я не уточнил таксономию, которая его окружает (см. рисунок 4-1).

Рисунок 4-1: Некоторые соусы основаны на *sauce hollandaise*. В *sauce béarnaise* лимон заменяется уменьшенным количеством уксуса и определенными травами, в то время как отличительной чертой *sauce mousseline* является то, что в него *вымешивают* взбитые сливки: эта технология также используется для изготовления *mousse au chocolat*.



sauce béarnaise – это в действительности *sauce hollandaise*, где лимонный сок заменяется уменьшенным количеством уксуса, луком-шалотом, кервелем и эстрагоном. Другие соусы основаны на *sauce hollandaise*, в том числе мой любимый, *sauce mousseline*, который сделан с *вымешиванием* взбитых сливок в *hollandaise*.

Вы заметили жаргон? Вместо того чтобы сказать "тщательно вбивая сливки в соус, старайтесь не свернуть его," я использовал термин *вымешайте*. Когда вы знаете, что это значит, это намного легче сказать и понять.

Термин *вымешать* не ограничивается соусами: это общий способ совместить что-то взбитое с другими ингредиентами. При создании классического шоколадного мусса (*mousse au chocolat*), например, я *вымешиваю* взбитые яичные белки в смесь взбитых яичных желтков и растопленного шоколада.

При разработке программного обеспечения у нас самих есть сложный и непроходимый жаргон. Хотя вы можете и не знать, что относится к поварскому термину *bain-marie*, я

уверен, что большинство поваров будут совершенно потеряны, если вы скажете им, что "строки являются неизменными классами, представляющими последовательности Unicode символов".

Когда дело доходит до разговоров о том, как структурировать код для решения конкретного типа задач, у нас есть *паттерны проектирования*, которые дают имена общим решениям. Таким же образом, как термины *sauce hollandaise* и *вымешивать* помогают нам поговорить о том, как сделать *sauce mouseline*, паттерны помогают нам говорить о том, как структурирован код. Система событий в .NET основана на паттерне проектирования Наблюдатель (*Observer*), а циклы *foreach* на Итератор (*Iterator*).

В этой главе я расскажу о четырех основных DI паттернах, перечисленных на рисунке 4-2. Поскольку глава построена так, чтобы показать каталог паттернов, каждый паттерн описан так, чтобы его можно было прочитать отдельно. Тем не менее, Внедрение в конструктор (*Constructor Injection*) на сегодняшний день является наиболее важным из четырех паттернов.

Рисунок 4-2: Структура этой главы принимает форму каталога паттернов. Каждый паттерн описан так, что вы можете прочитать о нем отдельно от других паттернов.



Не беспокойтесь, если вы обладаете ограниченными знаниями паттернов проектирования в целом. Основная цель паттерна проектирования заключается в создании подробного и автономного описания конкретного способа достижения цели – рецепта, если угодно.

Для каждого шаблона я приведу краткое описание, пример кода, преимущества и недостатки и так далее. Вы можете прочитать обо всех четырех паттернах или ограничиться только теми, которые вас интересуют. Наиболее важным паттерном является Внедрение в конструктор (*Constructor Injection*), который вы должны использовать в большинстве ситуаций; вы увидите другие все более и более специализированные паттерны по мере продвижения по главе.

[4.1. Внедрение в конструктор \(Constructor Injection\)](#)

[4.2. Внедрение в свойство \(Property Injection\)](#)

[4.3. Внедрение в метод \(Method Injection\)](#)

[4.4. Окружающий контекст \(Ambient Context\)](#)

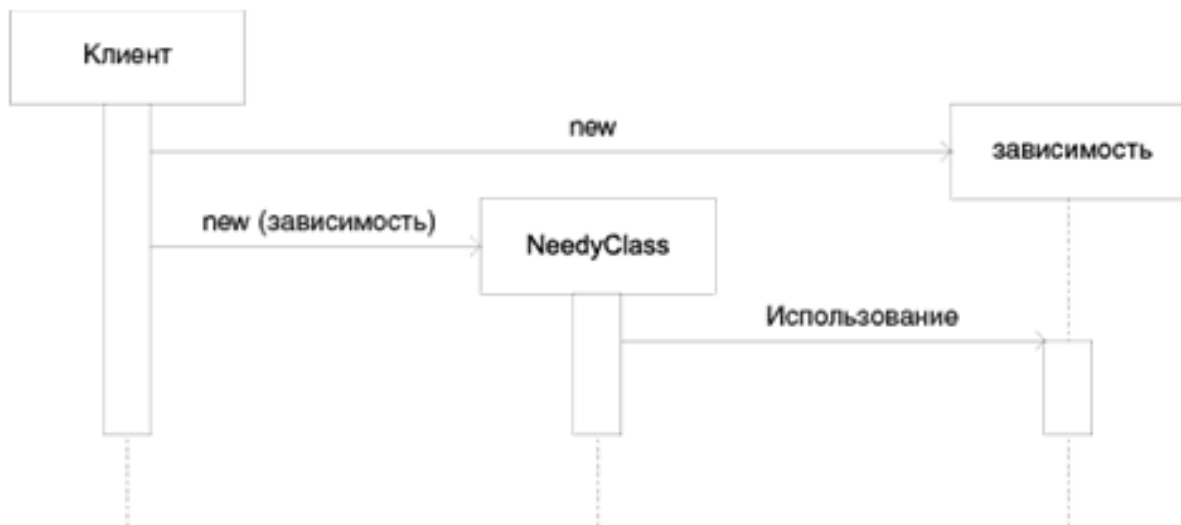
[4.5. Резюме](#)

4.1. Внедрение в конструктор (Constructor Injection)

Как мы можем гарантировать, что необходимая зависимость всегда доступна для класса, который мы в настоящее время разрабатываем?

Запрашивая все вызывающие элементы, чтобы доставить зависимость в качестве параметра конструктору класса.

Рисунок 4-3: `NeedyClass` нуждается в экземпляре зависимости, чтобы работать, поэтому он требует любого клиента, чтобы предоставить экземпляр через свой конструктор. Это гарантирует, что экземпляр доступен `NeedyClass` всякий раз, когда он необходим.



Когда класс требует экземпляр зависимости, чтобы работать вообще, мы можем предоставить эту зависимость через конструктор класса, включая ее, чтобы сохранить ссылку для будущего (или немедленного) использования.

Как это работает

Класс, которому нужна зависимость, должен предоставить открытый конструктор, который принимает экземпляр необходимой зависимости в качестве аргумента конструктора. В большинстве случаев, это должен быть только доступный конструктор. Если необходима более чем одна зависимость, могут быть использованы дополнительные аргументы конструктора.

Листинг 4-1: Внедрение в конструктор

```
1. private readonly DiscountRepository repository;
2. public RepositoryBasketDiscountPolicy(
3.     DiscountRepository repository)
4. {
5.     if (repository == null)
6.     {
7.         throw new ArgumentNullException("repository");
8.     }
9.     this.repository = repository;
10. }
```

Строки 2-3: Внедрить зависимость как аргумент конструктора

Строки 5-8: Ограждающее условие (Guard Clause)

Строка 9: Сохранение зависимости для дальнейшего использования

Строка 1: Поле зависимости только для чтения

Зависимость (в предыдущем листинге это абстрактный класс `DiscountRepository`) является требуемым аргументом конструктора. Любой клиентский код, который не предоставляет экземпляра зависимости, не может скомпилироваться. Однако так как интерфейсы и абстрактные классы являются ссылочными типами, вызывающий элемент может передать `null` в качестве аргумента, чтобы вызываемый код мог быть скомпилирован, и мы должны защитить класс против таких злоупотреблений при помощи ограждающего условия (*Guard Clause*).

Поскольку объединенные усилия компилятора и ограждающего условия гарантируют, что аргумент конструктора является действительным, если не выбрасывается исключение, тогда конструктор может сохранить зависимость для будущего использования, ничего не зная о реальной реализации.

Хорошей практикой является то, чтобы отметить поле, содержащее зависимость, как `readonly`: это гарантирует, что как только инициализационная логика конструктора будет выполнена, поле не может быть изменено. Это не является обязательным с точки зрения DI, но это защитит вас от случайного изменения поля (например, установка его на `null`) где-то в другом месте зависимого от класса кода.

Примечание

Не загружайте конструктор любой другой логикой. Принцип единственной обязанности (*Single Responsibility Principle*) предполагает, что члены должны делать только одну вещь, и теперь, когда мы используем конструктор для внедрения зависимостей, мы предпочли бы держать его свободным от других вещей.

Примечание

Подумайте о внедрении в конструктор как о *статическом объявлении зависимости класса*. Сигнатура конструктора компилируется при помощи типа и доступна для всеобщего обозрения. Она четко говорит, что класс требует зависимости, которые он запрашивает через свой конструктор.

После возвращения конструктора новый экземпляр класса находится в согласованном состоянии с соответствующим экземпляром зависимости, внедренном в него. Поскольку он содержит ссылку на эту зависимость, он может использовать ее так часто, как необходимо. Тут не нужна проверка на `null`, поскольку гарантировано, что экземпляр будет иметься в наличии.

Когда это использовать

Внедрение в конструктор должно быть вашим выбором по умолчанию для DI. Оно решает наиболее распространенную задачу, где класс *требует* одну или несколько зависимостей и не имеется никаких приемлемых *Local Default*.

Внедрение в конструктор хорошо решает эту задачу, потому что оно гарантирует, что зависимость присутствует. Если класс абсолютно не может функционировать без зависимости, эта гарантия является ценной.

Совет

Если это вообще возможно, ограничивайтесь одним конструктором. Перегруженные конструкторы приводят к неоднозначности: какой конструктор должен использовать DI контейнер?

В случаях, когда локальная библиотека может предоставить хорошую реализацию умолчанию, внедрение в свойство (Property Injection) может подойти лучше, но это нечастный случай. В предыдущих главах я показал много примеров хранилищ в качестве зависимостей. Это хорошие примеры зависимостей, когда локальная библиотека не может предоставить подходящую реализацию по умолчанию, так как надлежащая реализация принадлежит специализированным библиотекам доступа к данным (Data Access).

Таблица 4-1: Преимущества и недостатки внедрения в конструктор

Преимущества	Недостатки
Внедрение точно есть	Некоторые фреймворки усложняют использование внедрения в конструктор
Легко реализовать	

Помимо гарантированного внедрения, о чем уже говорилось, этот паттерн также легко реализовать с помощью четырех шагов, показанных в листинге 4-1.

Основным недостатком внедрения в конструктор является то, что вам нужно изменить текущую платформу приложения для его поддержки. Большинство фреймворков предполагают, что у ваших классов будет конструктор по умолчанию, и что им может потребоваться специальная помощь для создания экземпляров, когда конструктор по умолчанию отсутствует. В главе 7 я объясню, как включить внедрение в конструктор для общих фреймворков.

Очевидным недостатком внедрения в конструктор является то, что он требует, чтобы весь граф зависимости инициализировался немедленно, часто при запуске приложения. Однако, хотя это звучит не очень приятно, это редко является проблемой. В конце концов, даже для сложного графа объекта мы обычно говорим о создании десятков новых экземпляров объектов, а создание экземпляра объекта это то, что .NET Framework делает очень быстро. Любые узкие места, которые может иметь ваше приложение, появятся и в других местах, так что не беспокойтесь об этом.

В очень редких случаях это может все же быть реальной проблемой, но в главе 8 я опишу опцию *Delayed*, которая является одним из возможных средств решения этой проблемы

На данный момент я просто скажу, что могут (в крайнем случае) быть потенциальные проблемы с начальной загрузкой, и давайте двигаться дальше.

Использование

Хотя внедрение в конструктор, как правило, повсеместно в приложениях, использующих DI, оно не очень часто присутствует в .NET Base Class Library (BCL). Главным образом это потому, что BCL представляет собой набор библиотек, а не полноценное приложение.

Два взаимосвязанных примера, когда мы видим своего рода внедрение конструктора в BCL, это `System.IO.StreamReader` и `System.IO.StreamWriter`. Оба принимают в свои конструкторы экземпляр `System.IO.Stream`. У них также есть много перегруженных конструкторов, которые принимают путь к файлу, а не экземпляр `Stream`, но есть методы, которые внутренне создают `FileStream` на основе указанного пути к файлу: далее показаны все конструкторы `StreamWriter`, но конструкторы `StreamReader` схожи:

```
public StreamWriter(Stream stream);
public StreamWriter(string path);
public StreamWriter(Stream stream, Encoding encoding);
public StreamWriter(string path, bool append);
public StreamWriter(Stream stream, Encoding encoding, int bufferSize);
public StreamWriter(string path, bool append, Encoding encoding);
public StreamWriter(string path, bool append, Encoding encoding, int
bufferSize);
```

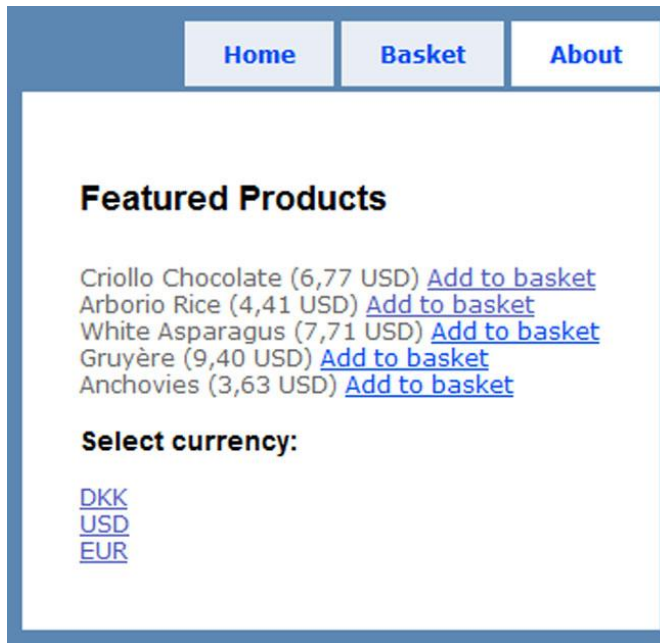
Класс `Stream` является абстрактным классом, который служит в качестве абстракции, над которой работают `StreamWriter` и `StreamReader` для выполнения своих обязанностей. Вы можете указать любой реализацию `Stream` в их конструкторах, и они будут ее использовать, но они выбросят `ArgumentNullException`, если вы попытаетесь присвоить `Stream null`.

Хотя BCL может предоставить нам примеры, когда мы видим, как используется внедрение конструктора, но всегда более поучительно увидеть пример. Следующий раздел проведет вас через полный пример реализации.

Пример: Добавление выбора валюты для корзины покупок

Я хотел бы добавить новую функцию к примеру коммерческого приложения, который я представил в главе 2, а именно, способность выполнять конвертацию валют. Я расширю пример в данной главе, чтобы продемонстрировать различные DI паттерны в деле, и когда я все сделаю главная страница должно быть как на рисунке 4-4.

Рисунок 4-4: Пример коммерческого приложения с реализованной конвертацией валюты. Теперь пользователь может выбрать три разные валюты, и цена товара и общая стоимость (на странице корзины) будут отображаться в этой валюте.



Одна из первых вещей, которая вам нужна, это `CurrencyProvider` – зависимость, предоставляющая вам валюты, которые вы запрашиваете. Вы определяете это следующим образом:

```
public abstract class CurrencyProvider
{
    public abstract Currency GetCurrency(string currencyCode);
}
```

Класс `Currency` является другим абстрактным классом, который обеспечивает конверсию между собой и другими валютами:

```
public abstract class Currency
{
    public abstract string Code { get; }
    public abstract decimal GetExchangeRateFor(string currencyCode);
}
```

Вы хотите, чтобы функция конвертации валюты была на всех страницах, где отображаются цены, так что вам нужна она в обоих `HomeController` и `BasketController`. Поскольку обе реализации очень похожи, я покажу только `BasketController`.

`CurrencyProvider`, скорее всего, представляют собой ресурс «вне процесса», например, веб-сервис или базу данных, которые предоставляют курсы конверсии. Это означает, что наиболее подходящим была бы реализация конкретного `CurrencyProvider` в отдельном проекте (например, в библиотеке `Data Access`). Следовательно, здесь нет никакой подходящей *Local Default*. В то же время, классу `BasketController` понадобится наличие `CurrencyProvider`; внедрение в конструктор – это то, что надо. Следующий листинг показывает, как зависимость `CurrencyProvider` внедряется в `BasketController`.

Листинг 4-2: Внедрение CurrencyProvider в BasketController

```
1. private readonly IBasketService basketService;
2. private readonly CurrencyProvider currencyProvider;
3. public BasketController(IBasketService basketService,
4. CurrencyProvider currencyProvider)
5. {
6.     if (basketService == null)
7.     {
8.         throw new
9.             ArgumentNullException("basketService");
10.    }
11.    if (currencyProvider == null)
12.    {
13.        throw new
14.            ArgumentNullException("currencyProvider");
15.    }
16.    this.basketService = basketService;
17.    this.currencyProvider = currencyProvider;
18. }
```

Строки 3-4: Внедрить зависимость как аргумент конструктора

Строки 6-15: Ограждающее условие (Guard Clause)

Строки 16-17: Сохранение зависимости для дальнейшего использования

Строки 1-2: Поле зависимости только для чтения

Поскольку класс `BasketController` уже имеет зависимость в `IBasketService`, вы добавляете новую зависимость `CurrencyProvider` в качестве второго аргумента конструктора, а затем следуете той же последовательности, что изложена в листинге 4-1: ограждающее условие гарантирует, что зависимости не являются `null`, что означает, что хранить их для последующего использования в `readonly` полях безопасно.

Теперь, когда `CurrencyProvider` гарантированно присутствует в `BasketController`, он может быть использован в любом месте, например, в методе `Index`:

```
public ActionResult Index()
{
    var currencyCode = this.CurrencyProfileService.GetCurrencyCode();
    var currency = this.currencyProvider.GetCurrency(currencyCode);
    // ...
}
```

Я еще не обсуждал `CurrencyProfileService`, так что теперь знайте, что он предоставляет предпочтительный код валюты текущего пользователя. Далее мы будем обсуждать `CurrencyProfileService` более подробно.

С имеющимся кодом валюты может быть вызван `CurrencyProvider`, чтобы предоставить `Currency`, который представляет этот код. Заметьте, что вы можете использовать поле `currencyProvider` без необходимости проверять его заранее, потому что оно гарантированно присутствует.

Теперь, когда у вас есть `Currency`, вы можете продолжить, чтобы выполнить остальную работу в методе `Index`; обратите внимание, что я еще не показал эту реализацию. По мере

продвижения по этой главе я буду дорабатывать этот метод и попутно добавлю больше функционала по конвертации валюты.

Связанные паттерны

Внедрение в конструктор является наиболее применимым DI паттерном, а также его легче всего реализовать правильно. Он применяется, когда зависимость является *обязательной*.

Если нам нужно сделать зависимость опциональной, мы можем перейти ко *внедрению в свойство* (Property Injection), если у нас есть подходящая *Local Default*.

Когда зависимость представляет Cross-Cutting Concern (CCC), который должен быть потенциально доступен для любого модуля в приложении, мы можем использовать *Ambient Context*.

Следующим паттерном в этой главе является внедрение в свойство, которое тесно связано с внедрением в конструктор; единственный решающий параметр заключается в том, является ли зависимость *обязательной* или нет.

4.2. Внедрение в свойство (Property Injection)

Как мы можем включить DI в качестве опции в классе, когда у нас есть хорошая Local Default?

Раскрывая записываемое свойство, которое позволяет вызывающим элементам предоставить зависимость, если они хотят переопределить поведение по умолчанию.

Рисунок 4-5: SomeClass имеет опциональную зависимость для ISomeInterface; вместо того, чтобы требовать от вызывающих элементов предоставить экземпляр, он дает вызывающим элементам возможность определить его через свойство.



Когда класс имеет хорошую *Local Default*, но мы все еще хотим оставить его открытой для расширения, мы можем раскрыть доступное для записи свойство, что позволяет клиенту указать другую реализацию зависимости класса, чем по умолчанию.

Примечание

Внедрение в свойство также известно как внедрение в сеттер.

В соответствии с рисунком 4-5, клиенты, желающие воспользоваться SomeClass как есть, могут создать новый экземпляр класса и использовать его, в то время как клиенты, желающие изменить поведение класса могут это сделать путем установки свойства Dependency для другой реализации ISomeInterface.

Как это работает

Класс, который использует зависимость, должен предоставить открытое, доступное для записи свойство типа зависимости. В каркасной реализации это может быть таким же простым, как следующий листинг.

Листинг 4-3: Внедрение в свойство

```
public partial class SomeClass
{
    public ISomeInterface Dependency { get; set; }
}
```

SomeClass зависит от ISomeInterface. Клиенты могут поставлять реализации ISomeInterface, устанавливая свойство Dependency. Обратите внимание, что в отличие от внедрения в конструктор, вы не можете отметить поле свойства Dependency как readonly, потому что вы разрешаете вызывающим элементам менять это свойство в любой момент жизненного цикла SomeClass.

Другие члены класса могут использовать внедренную зависимость, чтобы выполнять свои обязанности, например:

```
public string DoSomething(string message)
{
    return this.Dependency.DoStuff(message);
}
```

Однако такая реализация является хрупкой, потому что нет гарантии, что свойство `Dependency` возвращает экземпляр `ISomeInterface`. Код как этот, выбросит `NullReferenceException`, так как значение свойства `Dependency` равно `null`:

```
var mc = new SomeClass();
mc.DoSomething("Ploeh");
```

Эта проблема может быть решена, если позволить конструктору установить экземпляр по умолчанию для свойства в сочетании с ограждающим условием в сеттере свойства.

Другая проблема возникает, если вы позволяете клиентам переключать зависимость в середине времени жизни класса. Эту проблему можно решить путем введения внутреннего флага, который позволяет клиенту установить зависимость один раз.

Пример далее показывает, как можно справиться с этими сложностями, но прежде, чем я доберусь до этого, я хотел бы объяснить, когда уместно пользоваться внедрением в свойство.

Когда это использовать

Внедрение в свойство следует использовать только тогда, когда класс, который вы разрабатываете, имеет хорошую *Local Default*, и вы все еще хотите разрешить вызывающим элементам предоставлять различные реализации зависимости класса.

Внедрение в свойство лучше использовать, когда зависимость *не является обязательной*.

Примечание

Существует полемика вокруг вопроса о том, показывает ли внедрение в свойство необязательную зависимость. В качестве общего принципа API дизайна я считаю, что свойства необязательны, потому что вы можете легко забыть назначить их, и компилятор не жалуется. Если вы согласны с этим принципом в общем, вы также должны принять его в частном случае с DI.

Local Default

Когда вы разрабатываете класс, у которого есть зависимость, вы, возможно, думаете о конкретной реализации этой зависимости. Если вы пишете доменный сервис, который обращается к хранилищу, вы скорее всего, планируете разработать реализацию этого хранилища, которая использует реляционную базу данных.

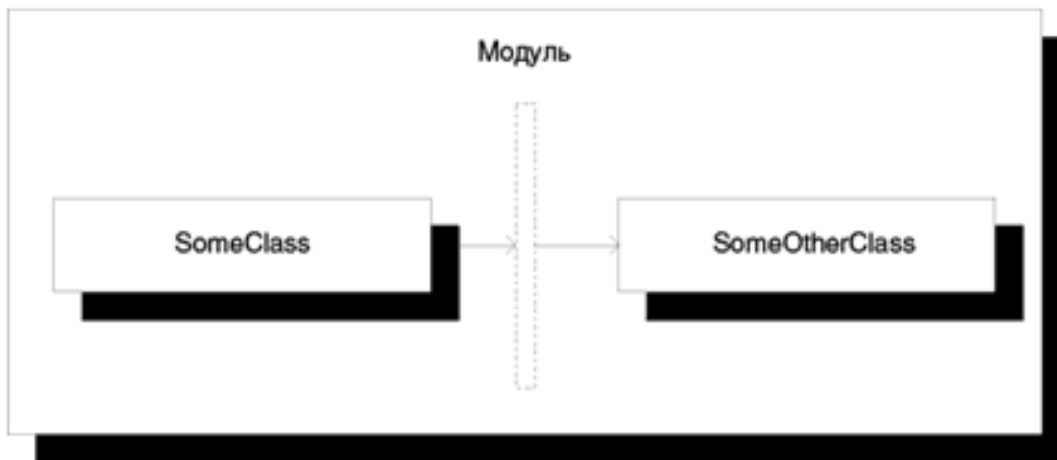
Заманчиво было бы сделать так, чтобы эта реализация использовалась по умолчанию классом на стадии разработки. Однако когда такой предполагаемый элемент по умолчанию реализован в другой сборке, использование его как дефолтного означало бы

создание жесткой ссылки на эту другую сборку, что нарушает многие преимущества слабой связанности, описанной в главе 1.

И наоборот, если предполагаемая реализация по умолчанию определяется в той же библиотеке как класс, у вас не будет этой проблемы. К сожалению, это не случай с хранилищами, но такие *Local Default* чаще всего встречаются как стратегии.

Пример в этом разделе содержит *Local Default*.

Рисунок 4-6: Даже в пределах одного модуля мы можем ввести абстракции (представлены вертикальным прямоугольником), которые помогают снизить связанность классов в этом модуле. Основным мотивом для этого является повышение поддержки модуля, что достигается тогда, когда классы варьируют независимо друг от друга.



В главе 1 мы обсуждали много веских причин для написания кода со слабой связанностью, когда модули изолированы друг от друга. Тем не менее, слабая связанность может с большим успехом также применяться к классам в одном модуле. Часто это делается путем введения абстракции в пределах одного модуля, когда классы «общаются» через абстракции, вместо того чтобы быть тесно связанными друг с другом.

Рисунок 4-6 иллюстрирует, что абстракции могут быть определены, реализованы и использованы внутри одного модуля с основной целью открытия классов для расширения.

Примечание

Концепция открытия класса для расширения охватывается принципом открытости/закрытости (*Open/Closed Principle*), который, если вкратце, утверждает, что класс должен быть открытым для расширения, но закрытым для изменений.

Когда мы реализуем классы, следуя принципу открытости/закрытости, мы можем иметь в виду *Local Default*, но мы по-прежнему даем клиентам способ расширить класс, заменив зависимость чем-то еще.

Примечание

Внедрение в свойство является лишь одним из многих способов применения принципа открытости/закрытости.

Совет

Иногда вы только хотите дать точку расширения, но оставить *Local Default* как пустую операцию. В таких случаях вы можете использовать паттерн *Null Object* для реализации *Local Default*.

Совет

Иногда вы хотите оставить *Local Default* на месте, но иметь возможность добавить больше реализаций. Вы можете добиться этого путем моделирования зависимости вокруг паттернов Наблюдатель (*Observer*) или Компонировщик (*Composite*).

До сих пор я не показал вам ни одного примера внедрения в свойство, потому что применимость этого паттерна является более ограниченной.

Таблица 4-2: Преимущества и недостатки внедрения в свойство

Преимущества	Недостатки
Легко понять	Ограниченная применимость
	Не совсем просто реализовать надежно

Основным преимуществом внедрения в свойство является то, что его легко понять. Я часто видел, как этот паттерн используется в качестве первой попытки, когда люди решают применять DI.

Представление может быть обманчивым, и внедрение в свойство сопряжено с трудностями. Его сложно реализовать надежным образом. Клиенты могут забыть (или не захотеть) предоставить зависимость, или по ошибке присвоить `null` в качестве значения. Кроме того: что должно произойти, если клиент попытается *изменить* зависимость в середине жизненного цикла класса? Это может привести к противоречивому или неожиданному поведению, поэтому вы можете захотеть защитить себя от этого события.

С внедрением в конструктор вы можете защитить класс против таких инцидентов, применяя ключевое слово `readonly` к полю, но это невозможно, когда вы раскрываете зависимость как записываемое свойство. Во многих случаях внедрение в конструктор гораздо проще и более надежно, но бывают ситуации, когда внедрение в свойство является правильным выбором. Это в том случае, когда предоставление зависимости является обязательным, потому что у вас есть хорошая *Local Default*.

Существование хорошей *Local Default* частично зависит от степени детализации модулей. .NET Base Class Library (BCL) поставляется как довольно большой пакет; до тех пор, пока *default* остается в пределах BCL, можно утверждать, что она также и *local*. В следующем разделе я кратко остановлюсь на этой теме.

Использование

В .NET BCL, внедрение в свойство является немного более используемым, чем внедрение в конструктор, вероятно, потому что много хороших *Local Default* определяются в разных местах.

`System.ComponentModel.IComponent` имеет доступное для записи свойство `Site`, которое позволяет определить экземпляр `ISite`. Это главным образом используется в разработке сценариев (например, Visual Studio), чтобы изменить или усилить компонент, когда он находится в дизайнера.

Другой пример, который сильнее отражает то, как мы привыкли думать о DI, можно найти в Windows Workflow Foundation. Класс `WorkflowRuntime` дает вам возможность добавлять, получать и удалять сервисы. Это не совсем внедрение в свойство, потому что API позволяет добавлять ноль или несколько нетипизированных сервисов посредством одного API общего назначения:

```
public void AddService(object service)
public T GetService<T>()
public object GetService(Type serviceType)
public void RemoveService(object service)
```

Хотя `AddService` выбросит `ArgumentNullException` если значение сервиса является `null`, нет никакой гарантии, что вы можете получить сервис заданного типа, потому что он, возможно, никогда не будет добавлен к текущему экземпляру `WorkflowRuntime` (на самом деле, это потому что метод `GetService` является *Service Locator*).

С другой стороны, `WorkflowRuntime` поставляется с большим количеством *Local Default* для каждого из требуемых сервисов, которые ему нужны, и они даже имеют префикс *Default*, например `DefaultWorkflowSchedulerService` и `DefaultWorkflowLoaderService`. Если, например, не добавлен альтернативный `WorkflowSchedulerService` либо с помощью метода `AddService`, либо конфигурационного файла приложения, используется класс `DefaultWorkflowSchedulerService`.

После этих BCL примеров давайте перейдем к более существенным примерам использования и реализации внедрения в свойство.

Пример: Определение сервиса профиля валюты для `BasketController`

В разделе 4.1.4 я начал добавлять функционал по конверсии валюты в пример коммерческого приложения и вкратце показал вам некоторую реализацию метода `Index` в `BasketController`, но умолчал о появлении `CurrencyProfileService`. Дело вот в чем:

Приложению нужно знать, какую валюту пользователь желает видеть. Если обратиться к рисунку 4-4, вы заметите некоторые ссылки на валюту в нижней части экрана. Когда пользователь нажимает одну из этих ссылок, вы должны сохранить где-то выбранную валюту и связать этот выбор с пользователем. `CurrencyProfileService` облегчает хранение и загрузку выбранной пользователем валюты:

```
public abstract class CurrencyProfileService
{
    public abstract string GetCurrencyCode();
    public abstract void UpdateCurrencyCode(string currencyCode);
}
```

Это абстракция, которая кодирует действия применения и извлечения текущего пользовательского выбора валюты.

В ASP.NET MVC (и ASP.NET в целом) у вас есть известная часть инфраструктуры, которая занимается таким сценарием: сервис Profile. Отличная реализация *Local Default* для CurrencyProfileService это то, что оборачивает ASP.NET сервис Profile и обеспечивает необходимую функциональность, определенную методами GetCurrencyCode и UpdateCurrencyCode. BasketController будет использовать этот DefaultCurrencyProfileService по умолчанию, когда раскрывает свойство, которое позволит вызывающему элементу заменить его чем-то другим.

Листинг 4-4: Раскрытие свойства CurrencyProfileService

```
1. private CurrencyProfileService currencyProfileService;
2. public CurrencyProfileService CurrencyProfileService
3. {
4.     get
5.     {
6.         if (this.currencyProfileService == null)
7.         {
8.             this.CurrencyProfileService =
9.                 new DefaultCurrencyProfileService(
10.                    this.HttpContext);
11.         }
12.         return this.currencyProfileService;
13.     }
14.     set
15.     {
16.         if (value == null)
17.         {
18.             throw new ArgumentNullException("value");
19.         }
20.         if (this.currencyProfileService != null)
21.         {
22.             throw new InvalidOperationException();
23.         }
24.         this.currencyProfileService = value;
25.     }
26. }
```

Строки 6-12: Отложенная инициализация *Local Default*

Строки 20-23: Зависимость определяется только один раз

DefaultCurrencyProfileService сам использует внедрение в конструктор, потому что ему нужен доступ к HttpContext и потому что HttpContext не доступен для BasketController во время создания; он должен отложить создание DefaultCurrencyProfileService, пока свойство не будет запрошено впервые. В этом случае требуется отложенная инициализация, но в других случаях *Local Default* может быть назначена в конструкторе. Обратите внимание, что *Local Default* назначается через открытый сеттер, который гарантирует, что все ограничивающие условия были оценены

Первое ограждающее условие гарантирует, что зависимость не имеет значение null. Следующее ограждающее условие гарантирует, что зависимость может быть назначена только один раз. В данном случае я предпочитаю, чтобы CurrencyProfileService не мог быть изменен после того, как был назначен, поскольку в противном случае это может привести к противоречивому поведению, где выбор валюты сначала сохраняется при помощи одного CurrencyProfileService, а затем извлекается из другого места, что, скорее всего, дает другое значение.

Вы можете также заметить, что поскольку вы используете сеттер для отложенной инициализации, зависимость будет также заблокирована, как только свойство будет прочтено. Еще раз, это является защитой клиентов от случая, когда зависимость впоследствии меняется без уведомления.

Если вы прошли через все ограждающие условия, вы можете сохранить экземпляр для дальнейшего использования.

По сравнению с внедрением в конструктор, это гораздо более сложно. Внедрение в свойство может выглядеть простым в сыром виде, как показано в листинге 4-3, но при правильном применении это, как правило, гораздо более сложно: и в этом примере я даже проигнорировал проблему безопасности потока.

Когда `CurrencyProfileService` на месте, метод `Index` в `BasketController` теперь может использовать его для получения предпочтительной валюты пользователя:

```
public ActionResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.CurrencyProvider.GetCurrency(currencyCode);
    // ...
}
```

Это тот же фрагмент кода, что показан в разделе 4.1.4. `CurrencyProfileService` используется для получения выбранной пользователем валюты, а `CurrencyProvider` в дальнейшем используется для извлечения этой валюты.

В разделе 4.3.4, я вернусь к методу `Index`, чтобы показать, что произойдет дальше.

Связанные паттерны

Вы используете *внедрение в свойство*, когда зависимость не является обязательной, потому что у вас есть хорошая *Local Default*. Если у вас нет *Local Default*, вы должны изменить реализацию на *внедрение в конструктор*.

Когда зависимость представляет CROSS-CUTTING CONCERN, который должен быть доступен для всех модулей в приложении, вы можете реализовать его как *Ambient Context*.

Но прежде чем мы перейдем к этому, *внедрение в метод*, описанное в следующем разделе, требует несколько иного подхода, поскольку его чаще применяют в ситуациях, когда у нас уже есть зависимость, которую мы хотим передать отдельным операциям.

4.3. Внедрение в метод (Method Injection)

Как мы можем внедрить зависимость в класс, когда она различается для каждой операции?

Передавая ее как параметр метода.

Рисунок 4-7: Клиент создает экземпляр `SomeClass`, но сначала внедряет экземпляр зависимости `ISomeInterface` с каждым вызовом метода.



Когда зависимость может меняться с каждым вызовом метода, вы можете передать ее через параметр метода

Как это работает

Вызывающий элемент внедряет зависимость в качестве параметра метода в каждый вызов метода. Это так же просто, как данная сигнатура метода:

```
public void DoStuff(ISomeInterface dependency)
```

Часто зависимость будет представлять некоторый контекст для операции, который поставляется вместе с "правильным" значением:

```
public string DoStuff(SomeValue value, ISomeContext context)
```

В данном случае параметр `value` представляет собой значение, над которым должен работать метод, тогда как `context` содержит информацию о текущем контексте операции. Вызывающий элемент внедряет зависимость в метод, а метод использует или игнорирует зависимость, в зависимости от того, нужно это или нет.

Если сервис использует зависимость, он должен проверить сначала ссылки на `null`, как показано в следующем листинге.

Листинг 4-5: Проверка параметров метода на `null` перед использованием

```
public string DoStuff(SomeValue value, ISomeContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException("context");
    }
    return context.Name;
}
```

Ограждающее условие гарантирует, что контекст доступен для остальной части тела метода. В данном примере метод использует имя контекста для возвращения значения, поэтому важно убедиться, что контекст доступен.

Если метод не использует внедренную зависимость, ему не нужно содержать ограждающее условие. Это звучит странно, ведь если параметр не используется, то зачем он вообще нужен? Тем не менее, вам может потребоваться сохранить его, если метод является частью реализации интерфейса.

Когда это использовать

Внедрение в метод лучше использовать тогда, когда зависимость может меняться с каждым вызовом метода. Это может быть в том случае, когда зависимость сама по себе представляет значение, но когда вызывающий элемент желает предоставить потребителю информацию о контексте, в котором вызывается операция.

Это часто бывает в сценариях надстройки, где надстройка предоставлена с информацией о контексте времени выполнения через параметр метода. В таких случаях требуется, чтобы надстройка реализовывала интерфейс, который определяет метода (методы) с внедрением.

Представьте интерфейс надстройки с такой структурой:

```
public interface IAddIn
{
    string DoStuff(SomeValue value, ISomeContext context);
}
```

Любой класс, реализующий этот интерфейс, может быть использован в качестве надстройки. Некоторые классы могут вообще не заботиться о контексте, в то время как другие реализации должны. Клиент может использовать список надстроек, вызывая каждую со значением и контекстом, возвращая суммарный результат. Это показано в следующем листинге.

Листинг 4-6: Пример клиента надстройки

```
1. public SomeValue DoStuff(SomeValue value)
2. {
3.     if (value == null)
4.     {
5.         throw new ArgumentNullException("value");
6.     }
7.     var returnValue = new SomeValue();
8.     returnValue.Message = value.Message;
9.     foreach (var addIn in this.addIns)
10.    {
11.        returnValue.Message =
12.            addIn.DoStuff(returnValue, this.context);
13.    }
14.    return returnValue;
15. }
```

Строки 11-12: Передана контекста надстройке

Закрытое поле `AddIns` является списком экземпляров `IAddIn`, что позволяет клиенту пройти циклом по списку для вызова каждого метода надстройки `DoStuff`. Каждый раз,

когда метод `DoStuff` вызывается для надстройки, контекст операции, представленный полем `context`, передается в качестве параметра метода

Примечание

Внедрение в метод тесно связано с использованием *фабрик абстракций*, описанных в разделе 6.1. Любая фабрика абстракций, которая принимает абстракцию в качестве входных данных, может рассматриваться как вариант внедрения в метод.

Время от времени, значение и контекст операции инкапсулируются в одной абстракции, которая работает как комбинация обоих.

Таблица 4-3: Преимущества и недостатки внедрения в метод

Преимущества	Недостатки
Позволяет вызывающему элементу предоставить конкретный для операции контекст	Ограниченная применимость

Внедрение в метод отличается от других DI паттернов, которое мы видели до сих пор, тем, что внедрение не происходит в *Composition Root*, а скорее, динамически во время вызова. Это позволяет вызывающему элементу предоставить конкретный для операции контекст, который является общим механизмом расширения, используемым в .NET BCL.

Использование

.NET BCL дает много примеров внедрения в метод, особенно в пространстве имен `System.ComponentModel`.

`System.ComponentModel.Design.IDesigner` используется для реализации пользовательского функционала времени разработки для компонентов. Он имеет метод `Initialize`, который принимает экземпляр `IComponent`, поэтому он знает, какой компонент он в настоящее время помогает разрабатывать. Дизайнеры создаются реализациями `IDesignerHost`, которые также принимают экземпляры `IComponent` в качестве параметров для создания дизайнеров:

```
IDesigner GetDesigner(IComponent component);
```

Это хороший пример сценария, когда параметр сам несет в себе информацию: компонент может нести информацию о том, какой `IDesigner` создать, но в то же время, это также компонент, над которым должен впоследствии работать дизайнер.

Другой пример в пространстве имен `System.ComponentModel` обеспечивается классом `TypeConverter`. Некоторые из его методов принимают экземпляр `ITypeDescriptorContext`, который, как следует из названия, передает информацию о контексте текущей операции. Поскольку таких методов много, я не хочу перечислять их все, но вот характерный пример:

```
public virtual object ConvertTo(ITypeDescriptorContext context,  
    CultureInfo culture, object value, Type destinationType)
```

В этом методе контекст операции передается явно параметром `context`, в то время как значение для преобразования и конечный тип передаются в виде отдельных параметров.

Исполнители могут использовать или игнорировать параметр `context`, как они посчитают нужным.

ASP.NET MVC также содержит несколько примеров внедрения в метод. Интерфейс `IMoделBinder` может быть использован для преобразования HTTP GET или POST данных в строго типизированные объекты. Вот метод:

```
object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext);
```

В методе `BindModel` параметр `controllerContext` содержит информацию о контексте операции (между прочим, `HttpContext`), тогда как `BindingContext` несет в себе больше явной информации о значениях, полученных от браузера.

Когда я говорю, что внедрение в конструктор должно быть вашим предпочтительным DI паттерном, я предполагаю, что вы создаете приложения на основе фреймворка. С другой стороны, если вы строите фреймворк, внедрение в метод часто может быть полезным, поскольку оно позволяет передавать информацию о контексте для настройки фреймворка. Это одна из причин, почему внедрение в метод так плодотворно используется в BCL.

Пример: конвертация валюты в корзине

В предыдущих примерах мы видели, как `BasketController` в примере коммерческого приложения извлекает предпочтительную валюту пользователя. Я дополню пример конвертации валюты путем конвертирования `Basket` в валюту пользователя.

`Currency` — это абстракция, которая моделирует валюту.

Листинг 4-7: `Currency`

```
public abstract class Currency
{
    public abstract string Code { get; }
    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

Свойство `Code` возвращает код валюты для экземпляра `Currency`. Ожидается, что коды `Currency` — это международные коды валют. Например, код валюты для датской кроны — это DKK, в то время как USD — это доллары США.

Метод `GetExchangeRateFor` возвращает обменный курс между экземпляром `Currency` и другой валютой. Заметим, что это абстрактный метод и обозначает, что я не делаю никаких предположений о том, как обменный курс будет найден исполнителем.

В следующем разделе мы рассмотрим, как экземпляры `Currency` используются для преобразования цен, и как эта абстракция может быть реализована и проведена так, чтобы вы могли конвертировать некоторые цены в такие валюты как доллары США или евро.

Внедрение Currency

Вы будете использовать абстракцию `Currency` как несущую информацию зависимость, чтобы выполнить конверсию валют для корзины, так что вы добавите метод `ConvertTo` к классу `Basket`:

```
public Basket ConvertTo(Currency currency)
```

Он пройдет циклом по всем элементам в корзине и сконвертирует их подсчитанные цены в приведенную валюту, возвращая новый экземпляр `Basket` со сконвертированными элементами. Через серию делегированных вызовов метода, реализация, наконец, предоставляется классом `Money`, как показано в следующем листинге.

Листинг 4-8: Конвертация Money в другую валюту

```
1. public Money ConvertTo(Currency currency)
2. {
3.     if (currency == null)
4.     {
5.         throw new ArgumentNullException("currency");
6.     }
7.     var exchangeRate =
8.         currency.GetExchangeRateFor(this.CurrencyCode);
9.     return new Money(this.Amount * exchangeRate,
10.        currency.Code);
11. }
```

Строка 1: Внедрить Currency в качестве параметра метода

`Currency` внедряется в метод `ConvertTo` через параметр `currency` и проверяется вездесущим огибающим условием, которое гарантирует, что экземпляр `currency` доступен для остальной части тела метода.

Обменный курс к текущей валюте (представленный `this.CurrencyCode`) извлекается из предоставленной `currency`, используется для расчета и возвращает новый экземпляр `Money`.

С методами `ConvertTo` вы можете, наконец, реализовать метод `Index` для `BasketController`, как показано в следующем листинге.

Листинг 4-9: Конвертация валюты в Basket

```
1. public ActionResult Index()
2. {
3.     var currencyCode =
4.         this.CurrencyProfileService.GetCurrencyCode();
5.     var currency =
6.         this.currencyProvider.GetCurrency(currencyCode);
7.     var basket = this.basketService
8.         .GetBasketFor(this.User)
9.         .ConvertTo(currency);
10.    if (basket.Contents.Count == 0)
11.    {
12.        return this.View("Empty");
13.    }
14.    var vm = new BasketViewModel(basket);
15.    return this.View(vm);
16. }
```

Строки 7-9: Конвертация пользовательской корзины в выбранную валюту

`BasketController` использует экземпляр `IBasketService` для получения пользовательской корзины. Вы можете вспомнить из главы 2, что зависимость `IBasketService` предоставляется `BasketController` через внедрение в конструктор. Как только у вас есть экземпляр `Basket`, вы можете конвертировать его в нужную валюту, используя метод `ConvertTo`, переданный экземпляру `currency`.

В данном случае вы используете внедрение в метод, потому что абстракция `Currency` несет информацию, но будет варьировать по контексту (в зависимости от выбора пользователя). Вы могли бы реализовать тип `Currency` в качестве конкретного класса, но это ограничило бы вашу способность определять, как извлекаются валютные курсы.

Теперь, когда мы увидели, как используется класс `Currency`, пришло время изменить нашу точку зрения и посмотреть, как он может быть реализован.

Реализация `Currency`

Я еще не говорил о том, как реализован класс `Currency`, потому что это не столь важно с точки зрения внедрения в метод. Как вы помните из раздела 4.1.4 и как вы можете видеть в листинге 4-9, экземпляр `Currency` обрабатывается экземпляром `CurrencyProvider`, который мы внедрили в класс `BasketController` путем внедрения в конструктор.

Чтобы упростить пример, я показал, что произойдет, если вы решили реализовать `CurrencyProvider` и `Currency` при помощи базы данных SQL Server и LINQ to Entities. Это предполагает, что в базе данных имеется таблица с курсами валют, которая была заполнена заранее каким-то внешним механизмом. Вы также могли бы использовать веб-сервис, чтобы запросить обменные курсы из внешнего источника.

Реализация `CurrencyProvider` передает строку подключения для реализации `Currency`, которая использует эту информацию для создания `ObjectContext`. Суть дела заключается в реализации метода `GetExchangeRateFor`, показанного в следующем листинге.

Листинг 4-10: Реализация `Currency`, поддерживаемая SQL Server

```
public override decimal GetExchangeRateFor(string currencyCode)
{
    var rates = (from r in this.context.ExchangeRates
        where r.CurrencyCode == currencyCode
        || r.CurrencyCode == this.code
        select r)
        .ToDictionary(r => r.CurrencyCode);
    return rates[currencyCode].Rate / rates[this.code].Rate;
}
```

Первое, что нужно сделать, это получить курсы из базы данных. В таблице приведены цены, определенные против одной, единой валюты (DKK), так что вам нужны оба курса, чтобы иметь возможность выполнить надлежащую конверсию между любыми двумя валютами. Вы будете индексировать извлекаемые валюты кодом валюты, так что вы легко их найдете на заключительном этапе расчета.

Эта реализация потенциально выполняет много коммуникаций «вне процесса» с базой данных. Метод `ConvertTo` в `Basket` в конечном итоге вызывает этот метод в цикле, и

обращение к базе данных при каждом вызове, скорее всего, будет иметь пагубные последствия для производительности. Я вернусь к этой проблеме в следующем разделе.

Связанные паттерны

В отличие от других DI паттернов из этой главы, в основном мы используем внедрение в метод тогда, когда в нас уже есть экземпляр зависимости, который мы хотим передать для разных операций, но мы не знаем конкретные типы операций во время разработки (например, в случае с надстройками).

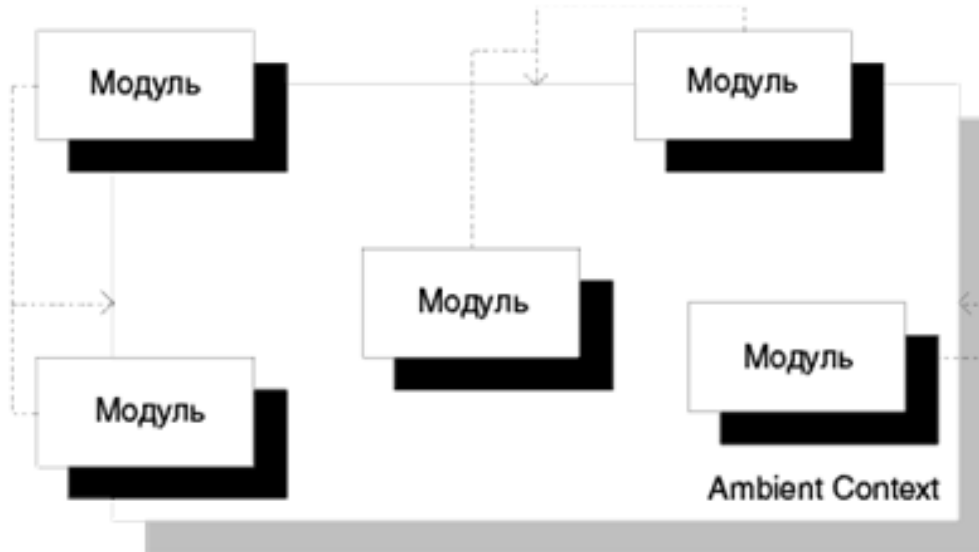
С внедрением в метод мы находимся по другую сторону дороги по сравнению с другими DI паттернами: мы не потребляем зависимости, а даем ее. У типов, которым мы поставляем зависимость, нет выбора в том, как моделировать DI или нуждаются ли они в зависимости вообще. Они могут потреблять ее или игнорировать, как они посчитают нужным.

4.4. Окружающий контекст (Ambient Context)

Как мы можем сделать зависимость доступной для каждого модуля, не загрязняя каждый API Cross-Cutting Concerns?

Делая ее пригодной для использования через статический доступ.

Рисунок 4-8: Каждый модуль при надобности может получить доступ к *Ambient Context*



По-настоящему универсальный *Cross-Cutting Concern* потенциально может загрязнить большую часть API для приложения, если вам нужно передать экземпляр каждому элементу. В качестве альтернативы можно определить контекст, доступный всем, кто в нем нуждается, и который может быть проигнорирован всеми остальными.

Как это работает

Ambient Context (окружающий контекст) доступен любому потребителю через статическое свойство или метод. Потребляющий класс может использовать его так:

```
public string GetMessage ()
{
    return SomeContext.Current.SomeValue;
}
```

В данном случае контекст имеет статическое свойство `Current`, к которому потребитель может получить доступ. Это свойство может быть по-настоящему статическим или может быть связано с выполняемым в данный момент потоком.

Чтобы быть полезным в DI сценариях, контекст сам по себе должен быть абстракцией и должна иметься возможность менять контекст извне: для предыдущего примера это обозначает, что свойство `Current` должно быть доступно для записи. Контекст сам по себе может быть реализован так, как показано в следующем листинге.

Листинг 4-11: Ambient Context

```
1. public abstract class SomeContext
2. {
3.     public static SomeContext Current
4.     {
5.         get
6.         {
7.             var ctx =
8.                 Thread.GetData (
9.                     Thread.GetNamedDataSlot ("SomeContext"))
10.                as SomeContext;
11.             if (ctx == null)
12.             {
13.                 ctx = SomeContext.Default;
14.                 Thread.SetData (
15.                     Thread.GetNamedDataSlot ("SomeContext"),
16.                     ctx);
17.             }
18.             return ctx;
19.         }
20.         set
21.         {
22.             Thread.SetData (
23.                 Thread.GetNamedDataSlot ("SomeContext"),
24.                 value);
25.         }
26.     }
27.     public static SomeContext Default =
28.         new DefaultContext();
29.     public abstract string SomeValue { get; }
30. }
```

Строки 7-10: Получить текущий контекст из TLS

Строки 22-24: Сохранить текущий контекст в TLS

Строка 29: Значение, переносимое контекстом

Контекст является абстрактным классом, который позволяет заменить один контекст другой реализацией во время выполнения.

В данном примере свойство `Current` сохраняет текущий контекст в Локальном Хранилище Потока (*Thread Local Storage (TLS)*), что обозначает, что каждый поток имеет свой собственный контекст, который независим от контекста любого другого потока. В случаях, когда для TLS не был присвоен контекст, возвращается реализация по умолчанию. Важно иметь возможность гарантировать, что ни один потребитель не получит `NullReferenceException`, когда он попытается получить доступ к свойству `Current`, поэтому нужно иметь хорошую *Local Default*. Отметим, что в этом случае свойство `Default` распределяется по всем потокам. Это работает, потому что в данном примере `DefaultContext` (класс, который наследуется от `SomeContext`) является неизменным. Если контекст по умолчанию изменяемый, вам нужно будет назначить отдельный экземпляр для каждого потока, чтобы предотвратить перекрестное загрязнение потоков.

Внешние клиенты могут назначить новый контекст для TLS. Обратите внимание, что возможно присвоить `null`, но если это произойдет, то следующая операция чтения автоматически переназначит контекст по умолчанию.

Весь смысл использования *Ambient Context* заключается во взаимодействии с ним. В данном примере это взаимодействие представлено одиночным абстрактным строковым свойством, но контекстный класс может быть и простым, и сложным, когда это необходимо.

Внимание

Для простоты я слегка пропустил безопасность потоков в коде в листинге 4-11. Если вы решили реализовать основанный на TLS *Ambient Context*, убедитесь, что вы знаете, что делаете.

Совет

Пример в листинге 4-11 использует TLS, но вы также можете использовать `CallContext` для получения подобного результата.

Примечание

Ambient Context не обязательно должен быть связан с потоком или вызываемым контекстом. Иногда имеет больше смысла сделать так, чтобы он применялся ко всему *AppDomain*, указав его как `static`.

Если вы хотите заменить контекст по умолчанию пользовательским контекстом, вы можете создать пользовательскую реализацию, которая наследуется от контекста, и назначить ее в нужное время:

```
SomeContext.Current = new MyContext();
```

Для контекста на основе TLS вы должны присвоить пользовательский экземпляр, когда вы создадите новый поток в то время как по-настоящему универсальный контекст можно назначить в *Composition Root*.

Когда это использовать

Ambient Context должен быть использован только в редчайших случаях. Для большинства случаев больше подходят внедрение в конструктор или внедрение в свойство, но у вас может быть реальный Cross-Cutting Concern, который загрязняет каждый API в вашем приложении, если вам нужно передать его всем сервисам.

Внимание

Ambient Context сходный по структуре с анти-паттерном *Service Locator*, который я опишу в главе 5. Разница состоит в том, что *Ambient Context* предоставляет экземпляр только одной, строго типизированной зависимости, в то время как *Service Locator* предположительно должен обеспечить экземпляры для каждой зависимости, которую вы можете запросить. Различия являются тонкими, так что убедитесь, что вы полностью понимаете, когда следует применять *Ambient Context*, прежде чем сделать это. Если вы сомневаетесь, выберите другой DI паттерн.

В разделе 4.4.4, что я реализую *TimeProvider*, который может быть использован, чтобы получить текущее время, и я также объясню, почему я предпочитаю его статическим членам *DateTime*. Текущее время является настоящим Cross-Cutting Concern, потому что

вы не можете предсказать, каким классам в каких слоях оно может понадобиться. Большинство классов, вероятно, могут использовать текущее время, но лишь небольшая часть из них собираются это сделать.

Потенциально это может заставить вас писать много кода с дополнительным параметром `TimeProvider`, потому что вы не знаете, когда он сможет вам понадобиться:

```
public string GetSomething(SomeService service,
    TimeProvider timeProvider)
{
    return service.GetStuff("Foo", timeProvider);
}
```

Предыдущий метод передает параметр `TimeProvider` для этого сервиса. Это может выглядеть безобидно, но когда мы затем просмотрим метод `GetStuff`, мы обнаружим, что это никогда не используется:

```
public string GetStuff(string s, TimeProvider timeProvider)
{
    return this.Stuff(s);
}
```

В данном случае параметр `TimeProvider` передается в качестве дополнительного багажа только потому, что он может однажды понадобиться. Это загрязняет API ненужными CCC, и код становится дурнопахнущим.

Ambient Context может быть решением этой проблемы, если встретятся условия, описанные в таблице 4-4.

Таблица 4-4: Условия для реализации Ambient Context

Условие	Описание
Вам нужно, чтобы контекст был запрашиваемым.	Если вам нужно только записать некоторые данные (все методы для контекста возвращают <i>void</i>), <i>перехват</i> (Interception) является лучшим решением. Может показаться, что это редкий случай, но он довольно частый: создать лог, что что-то случилось, записать метрики производительности, доказать, что контекст безопасности не подвержен риску – все подобные действия являются чистыми <i>утверждениями</i> (Assertion), которые лучше моделируются при помощи перехвата. Вы должны только рассмотреть возможность использования Ambient Context, если необходимо запросить его для некоторого значения (например, текущего времени).
Существует хорошая <i>Local Default</i> .	Существование Ambient Context является неявным (подробнее об этом далее), поэтому очень важно, чтобы контекст <i>просто работал</i> , даже в тех случаях, когда он не назначен явно.
Он должен быть гарантированно доступен.	Даже при надлежащей Local Default, важно сделать так, чтобы не было возможности присвоить null, что делает контекст недоступным и все клиенты выбросят <code>NullReferenceExceptions</code> . Листинг 4-11 показывает некоторые из шагов, которые можно предпринять, чтобы обеспечить это.

В большинстве случаев, преимущества Ambient Context не оправдывают недостатки, поэтому убедитесь, что вы можете удовлетворить всем этим условиям, а если не можете, рассмотрите другие альтернативы.

Таблица 4-5: Преимущества и недостатки Ambient Context

Преимущества	Недостатки
Не засоряет API	Неявный
Всегда доступен	Тяжело корректно реализовать
	Может неправильно работать в конкретных рантаймах

Фактически самым большим недостатком Ambient Context является его имплицитность, но, как видно из листинга 4-11, может также быть трудно реализовать его правильно и могут возникнуть проблемы с некоторыми средами исполнения (ASP.NET).

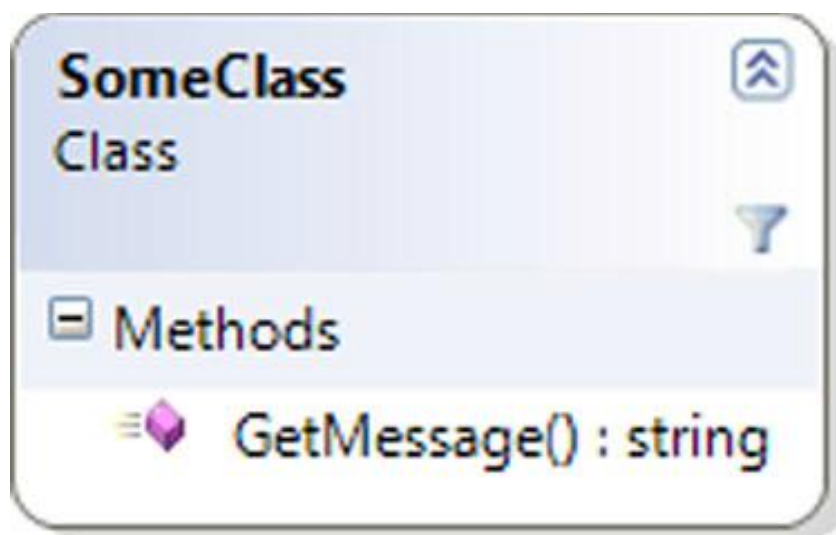
В следующих разделах мы более детально рассмотрим каждый из недостатков, которые описаны в таблице 4-5.

Неявность

Рассмотрим класс, показанный на рисунке 4-9: он не проявляет никаких внешних признаков использования Ambient Context, а метод GetMessage реализуется следующим образом:

```
public string GetMessage()
{
    return SomeContext.Current.SomeValue;
}
```

Рисунок 4-9: Класс и его метод GetMessage не проявляют внешних признаков использования Ambient Context, но это вполне может быть так.



Когда Ambient Context правильно реализован, вы можете, по крайней мере, ожидать, что не будет выброшено никаких исключений, но в этом примере контекст влияет на поведение метода, поскольку он определяет возвращаемое значение. Если контекст изменится, может поведение измениться, и вы сразу и не поймете, почему это произошло.

Примечание

В *Domain-Driven Design*, Эрик Эванс обсуждает *Intention-Revealing Interfaces* (Evans, Domain-Driven Design, 246), где речь идет о том, что API должен коммуницировать, что он делает только при помощи своего открытого интерфейса. Когда класс использует Ambient Context, он делает с точностью до наоборот: ваш шанс узнать, что имеет место быть тот самый случай, заключается только в чтении документации или просмотре кода.

Помимо потенциальных тонких ошибок это неявность также затрудняет обнаружение точки расширяемости класса. Ambient Context позволяет внедрить пользовательское поведение в любой класс, который использует его, но не очевидно, что это может быть так. Вы можете обнаружить это, читая документацию или понимая реализацию гораздо более подробно, чем вы могли бы хотеть.

Запутанная реализация

Правильная реализация Ambient Context может оказаться непростой задачей. По крайней мере, вы должны гарантировать, что контекст находится всегда в последовательном состоянии, то есть он не должен выбрасывать любые `NullReferenceExceptions` только потому, что одна реализация контекста была удалена без замены на другую.

Чтобы убедиться в этом, вам необходимо иметь подходящую *Local Default*, которая может быть использована, если никакая другая реализация не была явно определена. В листинге 4-11 я использовал отложенную инициализацию свойства `Current`, потому что C# не включает потоко-статические инициализаторы.

Когда Ambient Context представляет собой поистине универсальную концепцию, вы можете получить это при помощи простого записываемого Одиночка (*Singleton*): один экземпляр, который распространяется по всему `AppDomain`. Я покажу вам пример этого далее.

Ambient Context может также представлять контекст, который варьируется в зависимости от контекста в стеке вызовов, например от того, кто инициировал запрос. Мы видим это часто в веб-приложениях и веб-сервисах, где тот же самый код выполняется в контексте нескольких пользователей – и каждый на своем собственном потоке. В этом случае Ambient Context может иметь сходство с выполняемым в данный момент потоком и храниться в TLS, как мы видели в листинге 4-11, но это приводит к другим вопросам, в частности для ASP.NET.

Проблемы с ASP.NET

Когда Ambient Context использует TLS, могут возникать проблемы с ASP.NET, потому что он может менять потоки в определенные моменты жизненного цикла страницы, и нет никакой гарантии, что все, что хранится в TLS, будет скопировано из старого потока в новый.

Если такое случается, то вместо TLS вы должны использовать текущий `HttpContext` для хранения специфичных для запроса данных.

Это поведение по переключению потоков не является проблемой, если Ambient Context – это универсально распространяющийся экземпляр, потому что Singleton является общим для всех потоков в `AppDomain`.

Использование

.NET BCL содержит несколько реализаций Ambient Context.

Безопасность решается при помощи интерфейса

`System.Security.Principal.IPrincipal`, который связан с каждым потоком. Вы можете получить или установить текущий принципал для потока при помощи аксессуара `Thread.CurrentPrincipal`.

Другой Ambient Context на основе TLS моделирует текущую культуру потока.

`Thread.CurrentCulture` и `Thread.CurrentUICulture` и позволяют получить доступ и изменить язык и региональные параметры текущей операции. Многие форматирующие API, такие как парсинг и преобразование типов значений, неявно используют текущие региональные параметры и язык, если иное не предоставлено явно.

Трассировка является примером универсального Ambient Context. Класс `Trace` не связан с конкретным потоком, но действительно является общим для всего `AppDomain`. Вы можете написать сообщение трассировки отовсюду при помощи метода `Trace.Write`, и оно будет написано для любого количества `TraceListeners`, которые конфигурируются свойством `Trace.Listeners`.

Пример: кеширование Currency

Абстракция `Currency` в примере коммерческого приложения из предыдущих разделов примерно такая же «говорящая», как и интерфейс. Каждый раз, когда вы хотите конвертировать валюту, вы вызываете метод `GetExchangeRateFor`, который потенциально ищет обменный курс в какой-то внешней системе. Это гибкий API дизайн, потому что вы можете посмотреть курс фактически в режиме реального времени, если вам это нужно, но в большинстве случаев в этом не будет необходимости, и это, скорее всего, может стать узким местом.

Реализация на основе SQL Server, которую я представил в листинге 4-10, конечно, выполняет запрос к базе данных каждый раз, когда вы запрашиваете обменный курс. Когда приложение отображает покупательскую корзину, каждый элемент в корзине конвертируется, так что это приводит к запросам к базе данных для каждого элемента в корзине, даже если курс не меняется от первого до последнего элемента. Было бы лучше кэшировать обменный курс на некоторое время, чтобы приложению не нужно было стучаться к базе данных по поводу одного и того же обменного курса несколько раз в пределах одной доли секунды.

В зависимости от того, насколько это важно — иметь текущие валюты, время кэша может быть коротким или длинным: кэш для одной секунды или для нескольких часов. Тайм-аут должен быть настраиваемым.

Чтобы определить, когда истекает кэш валюты, вам нужно знать, сколько времени прошло с того момента, когда валюта была закеширована, так что вы должны иметь доступ к текущему времени. `DateTime.UtcNow` кажется встроенным Ambient Context, но это не так, потому что вы не можете назначить время, только запросить его.

Неспособность переопределить текущее время редко вызывает проблемы в реальном приложении, но может быть проблемой при модульном тестировании.

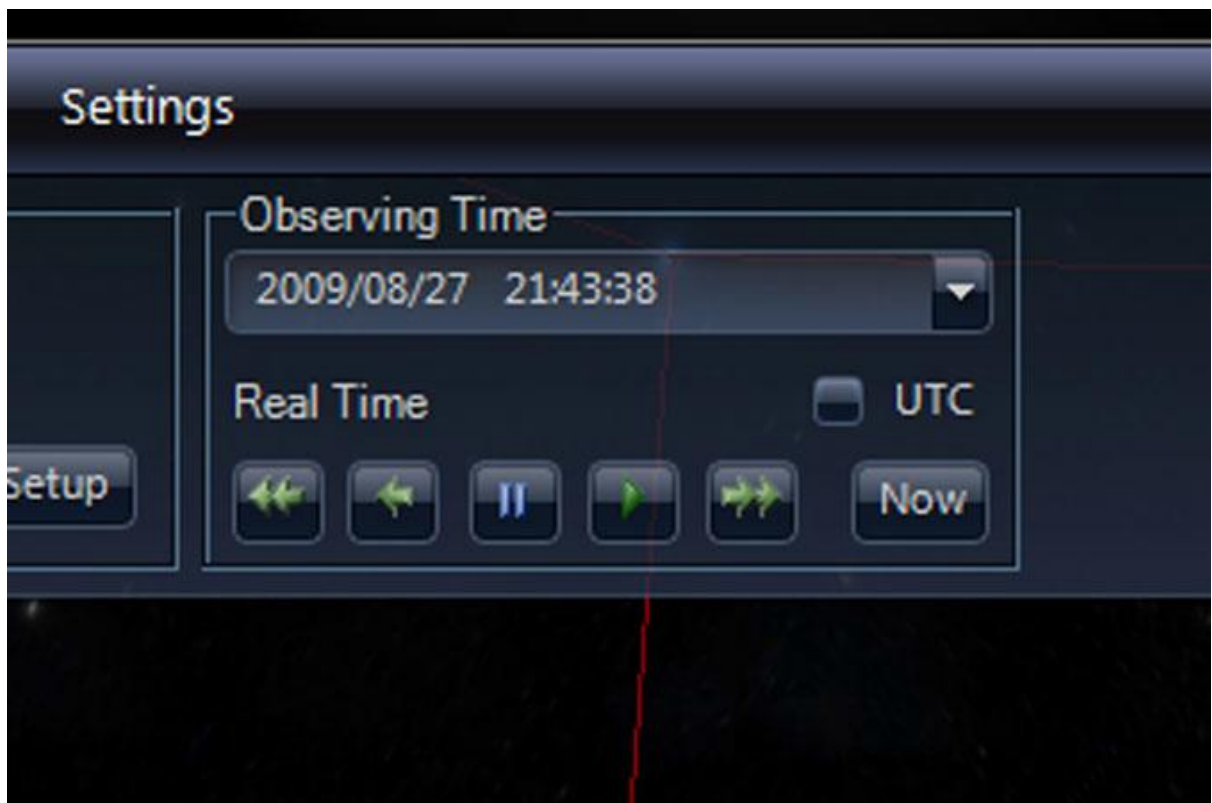
Моделирование времени

В то время как обычному веб-приложению вряд ли нужна возможность изменять текущее время, другой тип приложений может извлечь большую пользу от этой способности.

Однажды я написал довольно сложный движок моделирования, который зависел от текущего времени. Поскольку я всегда использую Test-Driven Development (TDD), я уже использовал абстракцию текущего времени, так что я мог внедрить экземпляры `DateTime`, которые отличались от реального машинного времени. Это оказалось огромным преимуществом, когда мне позже понадобилось ускорить время в симуляции на несколько порядков. Все, что я должен был сделать, это зарегистрировать провайдер времени, который ускорял время, и вся симуляция немедленно ускорялась.

Если вы хотите увидеть аналогичный функционал, вы можете посмотреть на клиентское приложение Всемирного телескопа (WorldWide Telescope, <http://www.worldwidetelescope.org>), которое позволяет моделировать ночное небо в ускоренном времени. На рисунке ниже показан скриншот элемента управления, который позволяет запускать время вперед и назад с различной скоростью. Я понятия не имею, реализовали ли разработчики эту возможность при помощи провайдера времени с `Ambient Context`, но это то, что сделал бы я.

Всемирный телескоп позволяет поставить время на паузу или промотать его назад и вперед с различной скоростью. Это симулирует вид ночного неба в разное время.



В случае с примером коммерческого приложения, я хочу иметь возможность контролировать время, когда я пишу юнит тесты, чтобы я мог убедиться, что кэш валют истекает правильно.

TimeProvider

Время является довольно универсальной концепцией (даже если время движется с разной скоростью в разных частях Вселенной), поэтому я могу моделировать его как всеобщее распространенный ресурс. Поскольку нет никаких причин иметь отдельные провайдеры времени для каждого потока, Ambient Context TimeProvider является записываемым Singleton, как показано в следующем листинге.

Листинг 4-12: TimeProvider Ambient Context

```
1. public abstract class TimeProvider
2. {
3.     private static TimeProvider current;
4.     static TimeProvider()
5.     {
6.         TimeProvider.current =
7.             new DefaultTimeProvider();
8.     }
9.     public static TimeProvider Current
10.    {
11.        get { return TimeProvider.current; }
12.        set
13.        {
14.            if (value == null)
15.            {
16.                throw new ArgumentNullException("value");
17.            }
18.            TimeProvider.current = value;
19.        }
20.    }
21.    public abstract DateTime UtcNow { get; }
22.    public static void ResetToDefault()
23.    {
24.        TimeProvider.current =
25.            new DefaultTimeProvider();
26.    }
27. }
```

Строки 6-7: Инициализация TimeProvider по умолчанию

Строки 14-17: Ограждающее условие

Строка 21: Важная часть

Цель класса TimeProvider состоит в том, чтобы вы могли контролировать, как время доводится до клиентов. Как описано в таблице 4-4, *Local Default* важна, поэтому вы статически инициализируете класс, чтобы использовать класс DefaultTimeProvider (я покажу вам это в ближайшее время).

Еще одно условие из таблицы 4-4 заключается в том, что вы должны гарантировать, что TimeProvider никогда не будет в нестабильном состоянии. Поле current никогда не должно быть null, поэтому ограждающее условие гарантирует, что этого никогда не будет.

Все это основа, чтобы сделать TimeProvider доступным отовсюду. Смыслом его существования является способность обрабатывать экземпляры DateTime, представляющие текущее время. Я целенаправленно смоделировал имя и сигнатуру

абстрактного свойства после `DateTime.UtcNow`. При необходимости я могу также добавили такие абстрактные свойства как `Now` и `Today`, но я не нуждаюсь в них для этого примера.

Наличие надлежащей и значимой *Local Default* является важным, и к счастью, для этого примера это не трудно сделать, потому что она должна просто вернуть текущее время. Это означает, что пока вы явно не войдете и не назначите другой `TimeProvider`, любой клиент, использующий `TimeProvider.Current.UtcNow`, получит реальное текущее время.

Реализация `DefaultTimeProvider` показана в следующем листинге.

Листинг 4-13: Провайдер времени по умолчанию

```
public class DefaultTimeProvider : TimeProvider
{
    public override DateTime UtcNow
    {
        get { return DateTime.UtcNow; }
    }
}
```

Класс `DefaultTimeProvider` наследуется от `TimeProvider`, чтобы предоставить реальное время каждый раз, когда клиент читает свойство `UtcNow`.

Когда `CachingCurrency` использует `Ambient Context TimeProvider` для получения текущего времени, он получит реальное текущее время, пока вы напрямую не назначите приложению другой `TimeProvider`; и я планирую сделать это только в моих модульных тестах.

Кэширование валют

Для реализации кэширования валют, нужно реализовать *Decorator* (*Decorator*), который меняет "правильную" реализацию `Currency`.

Примечание

Паттерн проектирования *Decorator* является важной частью перехвата, я буду обсуждать это более подробно в главе 9.

Вместо изменения существующей, поддерживаемой `SQL Server` реализации `Currency`, показанной в листинге 4-10, вы просто обернете кэш вокруг нее и только вызовете реальную реализацию, если кэш истек или не содержит записи.

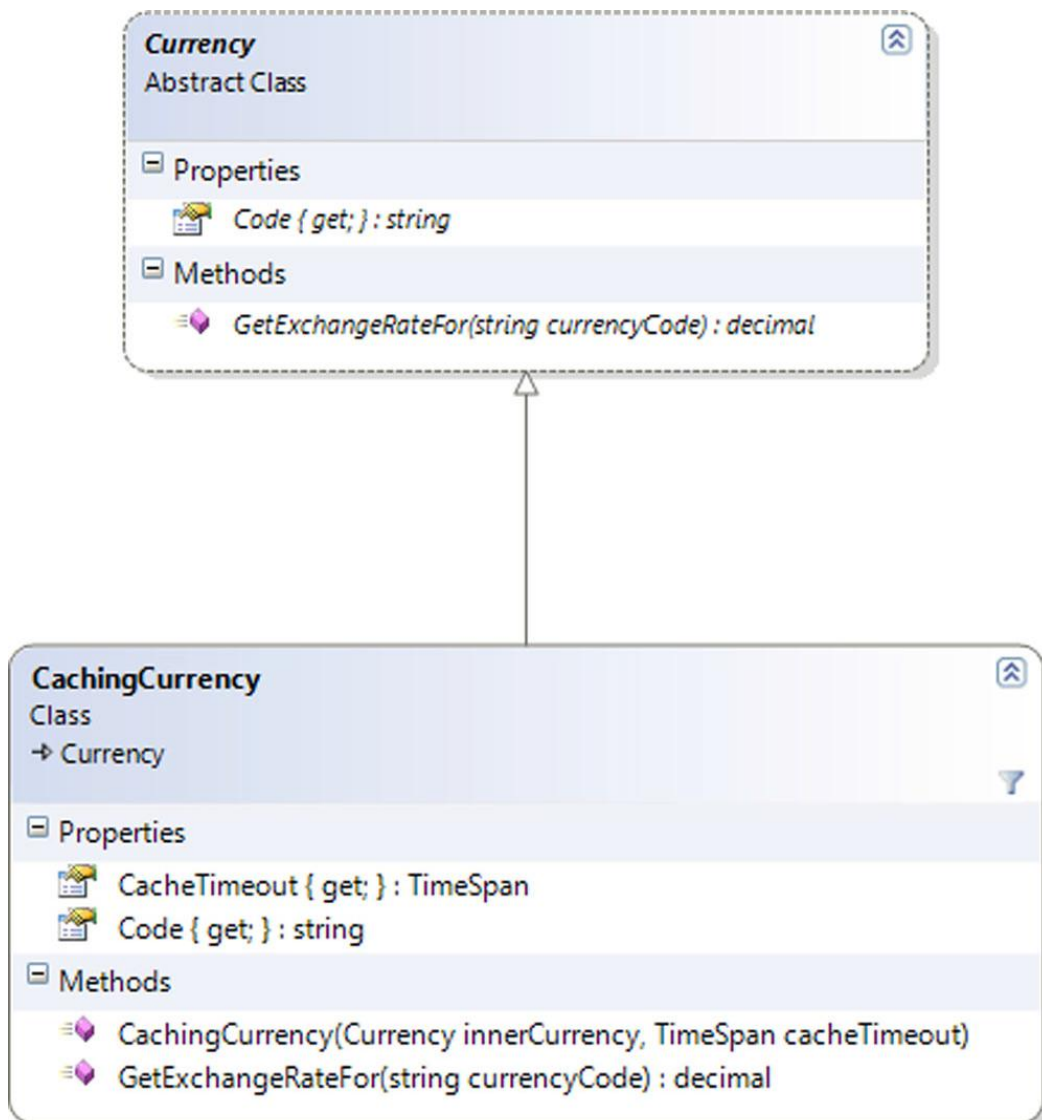
Как вы помните из раздела 4.1.4, `CurrencyProvider` — это абстрактный класс, который возвращает экземпляры `Currency`. `CachingCurrencyProvider` реализует тот же базовый класс и оборачивает функционал содержащегося `CurrencyProvider`. Всякий раз, когда он запрашивает `Currency`, он возвращает `Currency`, созданный содержащимся `CurrencyProvider`, но обернутый в `CachingCurrency` (см. рисунок 4-10).

Рисунок 4-10: `CachingCurrencyProvider` оборачивает "реальный" `CurrencyProvider` и возвращает экземпляры `CachingCurrency`, которые оборачивают "реальные" экземпляры `Currency`.



Такой паттерн позволяет мне кэшировать любую реализацию валюты, а не только реализацию на основе SQL Server, которая есть у меня в настоящее время. Рисунок 4-11 показывает план класса `CachingCurrency`.

Рисунок 4-11: `CachingCurrency` принимает в свой конструктор внутреннюю валюту (`innerCurrency`) и время действия кэша (`cacheTimeout`) и оборачивает функционал внутренней валюты



Совет

Паттерн Декоратор является одним из лучших способов реализации *разделения ответственности* (Separation of Concerns, SoC).

CachingCurrency использует внедрение в конструктор, чтобы получить "реальный" экземпляр, чьи курсы валют он должен хранить в кэше. Например, CachingCurrency делегирует свое свойство Code свойству Code внутреннего Currency.

Интересной частью реализации CachingCurrency является его метод GetExchangeRateFor, показанный в следующем листинге.

Листинг 4-14: Кэширование обменного курса

```
1. private readonly Dictionary<string, CurrencyCacheEntry> cache;
2. public override decimal GetExchangeRateFor(string currencyCode)
3. {
4.     CurrencyCacheEntry cacheEntry;
5.     if ((this.cache.TryGetValue(currencyCode,
6.         out cacheEntry))
7.         && (!cacheEntry.IsExpired))
8.     {
9.         return cacheEntry.ExchangeRate;
10.    }
11.    var exchangeRate =
12.        this.innerCurrency
13.            .GetExchangeRateFor(currencyCode);
14.    var expiration =
15.        TimeProvider.Current.UtcNow + this.CacheTimeout;
16.    this.cache[currencyCode] =
17.        new CurrencyCacheEntry(exchangeRate, expiration);
18.    return exchangeRate;
19. }
```

Строки 4-10: Вернуть закэшированный обменный курс, если он подходит

Строки 16-17: Сохранить в кэше обменный курс

Когда клиент запрашивает обменный курс, вы сначала перехватываете вызов, чтобы найти код валюты в кэше. Если для запрошенного кода валюты есть действительная запись в кэше, вы возвращаете закэшированный обменный курс, и остальная часть метода пропускается. Я вернусь к оценке того, истекла ли запись чуть позже.

Только если действующего закэшированного обменного курса нет, вы вызываете внутренний Currency, чтобы получить обменный курс. Прежде чем вернуть его, необходимо его кэшировать. Первый шаг состоит в вычислении срока истечения, и тут вы используете TimeProvider Ambient Context, вместо более традиционного DateTime.Now. С вычисленным сроком истечения вы можете кэшировать запись перед возвратом результата.

Вычисление того, истек ли срок действия кэша, также делается при помощи TimeProvider Ambient Context.

```
return TimeProvider.Current.UtcNow >= this.expiration;
```

Класс `CachingCurrency` использует `TimeProvider` `Ambient Context` во всех местах, где ему нужно текущее время, так что можно написать модульный тест, который точно контролирует время.

Модификация времени

При модульном тестировании класса `CachingCurrency`, вы можете точно контролировать время совершенно независимо от часов реальной системы. Это позволяет писать детерминистические модульные тесты, хотя тестируемая система (`System Under Test`, `SUT`) зависит от концепции текущего времени. Следующий листинг показывает тест, который проверяет, что хотя `SUT` запрашивает обменный курс четыре раза, внутренняя валюта вызывается только дважды: при первом запросе и снова, когда истекает время действия кэша.

Листинг 4-15: Юнит тест на предмет того, что валюта корректно кэшируется и что срок действия корректно заканчивается

```
1. [Fact]
2. public void InnerCurrencyIsInvokedAgainWhenCacheExpires()
3. {
4.     // Fixture setup
5.     var currencyCode = "CHF";
6.     var cacheTimeout = TimeSpan.FromHours(1);
7.     var startTime = new DateTime(2009, 8, 29);
8.     var timeProviderStub = new Mock<TimeProvider>();
9.     timeProviderStub
10.        .SetupGet(tp => tp.UtcNow)
11.        .Returns(startTime);
12.     TimeProvider.Current = timeProviderStub.Object;
13.     var innerCurrencyMock = new Mock<Currency>();
14.     innerCurrencyMock
15.        .Setup(c => c.GetExchangeRateFor(currencyCode))
16.        .Returns(4.911m)
17.        .Verifiable();
18.     var sut =
19.         new CachingCurrency(innerCurrencyMock.Object,
20.             cacheTimeout);
21.     sut.GetExchangeRateFor(currencyCode);
22.     sut.GetExchangeRateFor(currencyCode);
23.     sut.GetExchangeRateFor(currencyCode);
24.     timeProviderStub
25.        .SetupGet(tp => tp.UtcNow)
26.        .Returns(startTime + cacheTimeout);
27.     // Exercise system
28.     sut.GetExchangeRateFor(currencyCode);
29.     // Verify outcome
30.     innerCurrencyMock.Verify(
31.         c => c.GetExchangeRateFor(currencyCode),
32.         Times.Exactly(2));
33.     // Teardown (implicit)
34. }
```

Строка 12: Установка `TimeProvider` `Ambient Context`

Строка 21: Должна быть вызвана внутренняя валюта

Строки 22-23: Должна быть закэширована

Строки **24-26**: Время истечения срока действия

Строка **28**: Должна быть вызвана внутренняя валюта

Строки **30-32**: Проверка на то, что внутренняя валюта была вызвана правильно

Внимание, жаргон

Следующий текст содержит некоторую терминологию модульного тестирования: я выделил ее курсивом, а поскольку эта книга не о модульном тестировании, я отправляю вас к книге *xUnit Test Patterns* (Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code* (New York: Addison-Wesley, 2007), которая является источником всех этих имен паттернов.

Одна из первых вещей, которую нужно сделать в этом тесте, заключается в создании «дублера для тестирования» (Test Double) для `TimeProvider`, который будет возвращать экземпляры `DateTime`, как они определены, вместо того чтобы основываться на часах системы. В этом тесте я использую динамический *mock* фреймворк, называемый `Moq` (<http://code.google.com/p/moq/>), чтобы определить, что свойство `UtcNow` вернет тот же `DateTime`, пока не указано иное. Когда все определено, эта *заглушка* (Stub) внедряется в `Ambient Context`.

Первый вызов `GetExchangeRateFor` должен вызвать внутренний `CachingCurrency`, потому что еще ничего не находится в кэше, а два следующих вызова должны вернуть кэшированное значение, потому что срок еще не истек, в соответствии с *заглушкой* `TimeProvider`.

С парой кэшированных вызовов настала пора времени продвинуться вперед, вы меняете *заглушку* `TimeProvider`, чтобы вернуть экземпляр `DateTime`, для которого как раз истек срок действия кэша, и снова вызываете метод `GetExchangeRateFor`, ожидая, что он вызовет внутренний `Currency` во второй раз, потому что действительные записи кэша теперь уже истекли.

Поскольку вы ожидаете, что внутренний `Currency` вызывается дважды, вы, наконец, убеждаетесь в этом, говоря внутреннему `Currency Mock`, что метод `GetExchangeRateFor` должен быть вызван ровно два раза.

Одна из многих опасностей `Ambient Context` заключается в том, что как только он назначен, он остается одинаковым, пока не будут изменены снова, но в связи с его неявной природой, об этом можно легко забыть. В модульном тесте, например, поведение, определенное с помощью теста в листинге 4-15, остается одинаковым, если не будет явного сброса (что я делаю в *Fixture Teardown*). Это может привести к серьезным ошибкам (на этот раз в моем тестовом коде), потому что это распространится и загрязнит тесты, которые выполняются после этого теста.

`Ambient Context` выглядит обманчиво простым в реализации и использовании и может привести ко многим трудным для поиска ошибкам. Его можно использовать, но делайте это только тогда, когда не существует лучшей альтернативы. Это как хрен: отлично подходит для определенных вещей, но определенно не универсален.

Связанные паттерны

Ambient Context может быть использован для моделирования Cross-Cutting Concern, хотя и требует, чтобы у нас была подходящая *Local Default*.

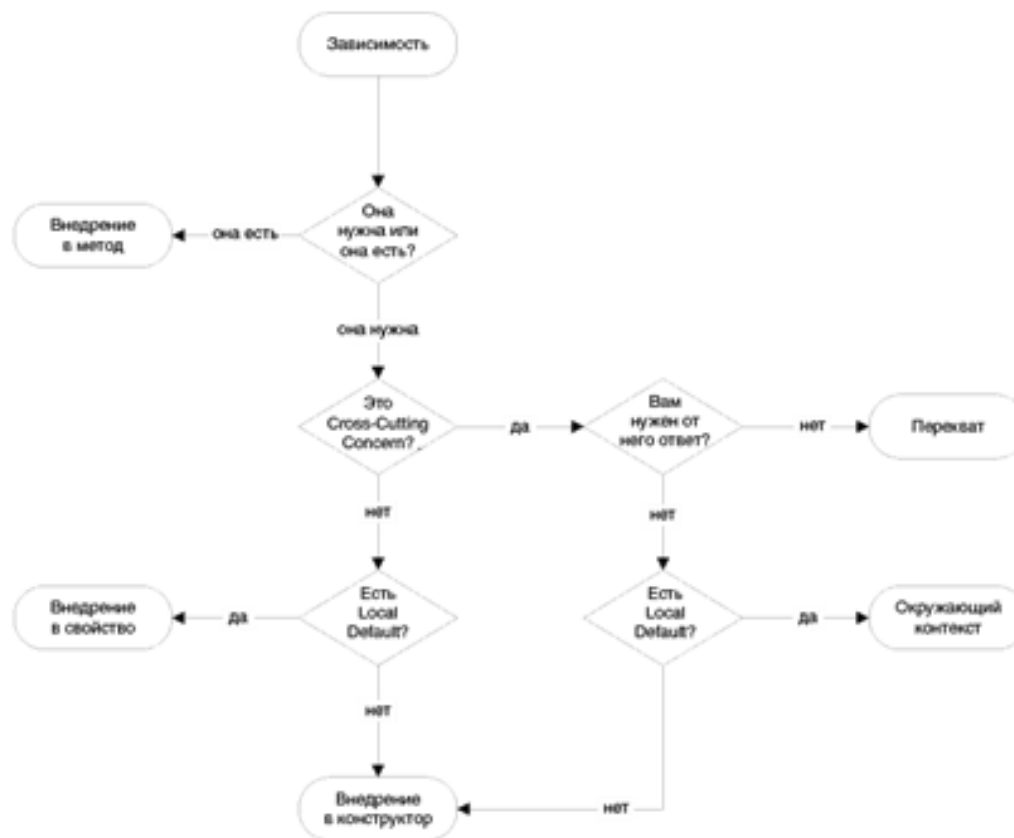
Если окажется, что зависимость вообще не является Cross-Cutting Concern, вы должны изменить DI стратегию. Если у вас есть *Local Default*, вы можете переключиться на внедрение в свойство, а в противном случае вы должны использовать внедрение в конструктор.

4.5. Резюме

Паттерны, представленные в этой главе, являются центральной частью DI. Вооружившись Composition Root и соответствующим набором DI паттернов, вы можете реализовать основы DI. При применении DI есть много нюансов и мелких деталей, которые нужно выучить, но паттерны покрывают основную механику, которая отвечает на вопрос: *как я могу внедрить зависимости?*

Эти паттерны не являются взаимозаменяемыми. В большинстве случаев вашим первым выбором должно быть внедрение в конструктор, но бывают ситуации, когда один из других паттернов является лучшей альтернативой. Рисунок 4-12 показывает процесс принятия решений, который может помочь вам принять решение о выборе нужного паттерна, но если есть сомнения, выберите внедрение в конструктор: вы никогда сильно не ошибетесь в этом выборе.

Рисунок 4-12: В большинстве случаев вы должны выбирать внедрение в конструктор, но бывают ситуации, когда один из других DI паттернов подходит лучше.



Первая вещь, которую нужно рассмотреть, заключается в понимании того, является ли зависимость тем, что вам нужно, или тем, что у вас уже есть, но вы хотите, чтобы она коммуницировала с другими сопредельными операциями. В большинстве случаев вам, вероятно, нужна зависимость, но в сценариях надстройки вы можете захотеть передать надстройке текущий контекст. Каждый раз, когда зависимость может отличаться от операции к операции, внедрение в метод является хорошим кандидатом для реализации.

Когда зависимость представляет Cross-Cutting Concern, выбор подходящего паттерна зависит от направления коммуникации. Если вам нужно только что-нибудь записать

(например, время, которое заняла операция, или то, какие значения передаются), лучшей альтернативой является перехват (о котором я расскажу в главе 9). Он также хорошо работает, если ответ, который вам нужен от него, уже включен в определение интерфейса. Кэширование является прекрасным примером этого использования перехвата.

Если вам необходимо запросить Cross-Cutting зависимость для ответа, который *не* включен в оригинальный интерфейс, вы можете использовать Ambient Context, только если у вас есть надлежащая *Local Default*, которая позволяет оформить сам контекст разумным поведением по умолчанию, которое работает для всех клиентов без явной настройки.

Когда зависимость не представляет Cross-Cutting Concern, *Local Default* все еще является решающим фактором, поскольку она может сделать явное внедрение зависимости необязательным – используется реализация по умолчанию, если не указана переопределяющая реализация. Этот сценарий может быть эффективно использован при помощи внедрения в свойство.

В любых других случаях применяется паттерн внедрения в конструктор. Исходя из рисунка 4-12, кажется, что внедрение в конструктор является последним оплотом, который входит в игру только тогда, когда все остальное не дает результатов. Это верно лишь отчасти, поскольку в большинстве случаев специализированные паттерны не применяются, и по умолчанию на поле остается внедрение в конструктор. Его легче понять и проще надежно реализовать, чем любой из других DI паттернов. Вы можете построить целые приложения только с одним внедрением в конструктор, но понимание других паттернов может помочь вам в разумном выборе нужного в некоторых случаях, когда он не подходит.

В этой главе содержится систематический каталог, который объясняет, как вы должны внедрять зависимости в ваши классы. В следующей главе DI рассматривается с противоположной стороны, и в ней я расскажу, как не нужно обращаться с DI.

5. DI анти-паттерны

В гастрономическом плане Дания была одной из развивающихся стран в 1970х: я был там, но я никогда не страдал, потому что не знал ничего лучшего. В основном использовались мясо и картофель, но постепенно приходили и зарубежные идеи. Я думаю, что одна из причин заключалась в том, что это было на заре эры массового туризма.

Датчане массово отправлялись на юг, в другие части Европы, и самые смелые пробовали местную еду. После возвращения домой более молодое поколение все больше готовило пасту, но ни один итальянец не распознал бы датскую версию соуса болоньезе.

Вот, что я думаю, произошло. Какому-то предприимчивому датскому туристу понравилась тальятелла алла болоньезе (*tagliatelle alla bolognese*) так сильно, что она решила попытаться приготовить ее, когда вернулась домой. (Я предполагаю, что это была женщина, потому что мужчины в то время много не готовили). Она сделала все, чтобы вспомнить, что входит в соус, но это было не так просто из-за долгой поездки на автобусе обратно в Данию.

Что касается ингредиентов, грудинка и красное вино были забыты, прежде чем она покинула Италию, мясной бульон и куриная печень были потеряны из памяти где-то в Австрии или Швейцарии, а овощи один за другим выпадали из головы во время долгого пути через (Западную) Германию. Когда она пересекла датскую границу, все, что осталось от оригинального рецепта, – это нарезанный лук и мясной фарш, и их подают только с одним типом пасты, которая легко доступна в Дании в то время – спагетти.

Мы ели блюдо, которое получилось в результате, в течение многих лет и любили его. В какой-то момент в 1980-х годах, в рецепт были добавлены томатная паста и орегано, чтобы сделать его более похожим на исходный. Это был рецепт, который я использовал более чем десять лет, пока кто-то не подсказал мне, что для улучшения вкусовых качеств можно добавить еще морковь, сельдерей, куриную печень, красное вино и так далее.

Смысл истории заключается в том, что я думал, что делаю рагу алла болоньезе (*ragù alla bolognese*), тогда как в действительности я не был даже близок к этому. Мне никогда не приходило в голову ставить под сомнение подлинность рецепта, потому что я вырос с ним. Хотя подлинность не является конечным критерием, блюдо, приготовленное по подлинному рецепту, намного вкуснее, и я не собираюсь возвращаться к моему старому рецепту.

В предыдущей главе я кратко сравнил паттерны проектирования с рецептами. Паттерн обеспечивает общий язык, который мы можем использовать, чтобы сжато обсудить сложное понятие, и рагу алла болоньезе – это такое понятие, потому что мы можем обсудить, как оно согласуется с тальятелле или лазаньей. С другой стороны, когда понятие (или, скорее, реализация) искривлено, мы получаем анти-паттерн.

Примечание

Анти-паттерн представляет собой описание часто встречающихся решений проблемы, которые несомненно приводят к негативным последствиям (William J. Brown et al., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (New York: Wiley Computer Publishing), 1998, 7).

Анти-паттерны часто бывают вызваны незнанием (как с моим соусом болоньезе), и их нужно избегать, а знание этих общих ловушек может помочь вам избежать их. Они являются более или менее формализованным способом описания распространенных ошибок, которые люди совершают снова и снова, независимо друг от друга.

В этой главе я опишу некоторые общие анти-паттерны, связанные с DI. За время моей карьеры я видел, как все они используются в той или иной форме, и, каюсь, я сам применял более чем один из них. Во многих случаях они казались искренними попытками применить DI в приложении, но без полного понимания основ DI, реализация приносила больше вреда, чем пользы.

Изучение этих анти-паттернов должно дать вам представление о том, каких ловушек нужно избегать в ваших первых DI проектах. Ваши ошибки не будут выглядеть точно так же, как мои или примеры, представленные здесь, но эта глава покажет вам, где кроются опасные признаки.

Анти-паттерны могут быть исправлены путем рефакторинга кода к одному из DI паттернов, приведенных в главе 4. Сложность исправления в каждом отдельном случае зависит от деталей реализации, но для каждого анти-паттерна я предоставлю некоторые обобщенные рекомендации, как провести рефакторинг по направлению к паттерну.

Совет

В этой главе я не уделяю слишком много места рефакторингу от DI анти-паттерна к DI паттерну, потому что это не главная тема этой книги. Если вы заинтересованы в получении дополнительной информации о том, как вы можете переместить существующее приложение в направлении DI, есть целая книга, где обсуждается рефакторинг таких приложений: *Working Effectively with Legacy Code* (Michael Feathers, *Working Effectively with Legacy Code* (New York: Prentice Hall, 2004)). Хотя в ней не рассматривается исключительно DI, она охватывает многие из тех же понятий, что и я в данной книге.

Анти-паттерны, описанные в этой главе, приведены в таблице 5-1. Рисунок 5-1 иллюстрирует структуру главы.

Таблица 5-1: DI анти-паттерны

Анти-паттерн	Описание
Control Freak	Зависимости управляются напрямую, в отличие от <i>инверсии управления</i> (Inversion of Control, IoC).
Bastard Injection	<i>Foreign Default</i> используются как значения по умолчанию для зависимостей.
Constrained Construction	Предполагается, что конструкторы имеют особую, индивидуальную сигнатуру.
Service Locator	Неявный сервис может подавать зависимости потребителю, но нет гарантии, что он это сделает.

Рисунок 5-1: Структура этой главы принимает форму каталога анти-паттернов. Каждый анти-паттерн описан так, что вы можете прочитать о нем отдельно от других анти-паттернов.



Внимание

Эта глава отличается от других глав, потому что большая часть кода, который я покажу вам, дает примеры того, как *не* реализовывать DI. Не пытайтесь повторить это дома!

Так же, как внедрение в конструктор является наиболее важным DI паттерном, Control Freak является самым доминирующим анти-паттерном. Он всеми силами предотвращает вас от применения любого надлежащего и нужного внедрения зависимостей, так что вам придется сосредоточить свою энергию на этом анти-паттерне, прежде чем заняться другими. С другой стороны, наиболее опасным является Service Locator, потому что создается впечатление, что он на самом деле решает проблему.

Остальная часть этой главы описывает каждый анти-паттерн более подробно. Вы можете прочитать ее от начала до конца или только о том, в чем вы заинтересованы: каждый анти-паттерн занимает отдельный раздел. Тем не менее, если вы решили прочитать только об одном анти-паттерне, вы должны сосредоточить свое внимание на Control Freak.

[5.1. Control Freak](#)

[5.2. Bastard Injection](#)

[5.3. Constrained Construction](#)

[5.4. Service Locator](#)

[5.5. Резюме](#)

5.1. Control Freak

Что является противоположностью инверсии управления? Первоначально термин *инверсия управления* был придуман, чтобы определить противоположность нормальному положению дел, но мы не можем на самом деле говорить об анти-паттерне «Business as Usual». Вместо этого, после долгих размышлений, я назвал его Control Freak, чтобы описать класс, который никак не желает терять контроль над своими зависимостями.

Это происходит каждый раз, когда мы создаем новый экземпляр типа с помощью ключевого слова `new`. Когда мы делаем так, мы явно заявляем, что собираемся контролировать жизненный цикл экземпляра и что никто не получит шанс на перехват этого конкретного объекта.

Совет

Частота появления в коде ключевого слова `new` является очень грубым показателем того, насколько код сильно связан.

Анти-паттерн Control Freak появляется всякий раз, когда мы получаем экземпляр зависимости, прямо или косвенно используя ключевое слово `new` в любом месте, кроме Composition Root.

Примечание

Хотя ключевое слово `new` является запахом кода, когда дело доходит до *меняющихся зависимостей*, вам не нужно беспокоиться об использовании его для *стабильных зависимостей*. В целом ключевое слово `new` не является "незаконными", но вы должны воздерживаться от его использования, чтобы получить экземпляры *меняющихся зависимостей*.

Наиболее вопиющим примером Control Freak является то, когда мы не делаем никаких усилий, чтобы ввести абстракции в наш код. Вы видели несколько примеров в главе 2, когда Мэри реализовала свое коммерческое приложение (раздел 2.1.1). В таком подходе нет попытки ввести DI, но даже там, где разработчики слышали о DI и компоновке, анти-паттерн Control Freak часто может быть найден в различных вариациях.

В следующих разделах я покажу вам некоторые примеры, которые напоминают код, что я видел в производственном использовании. В любом случае, у разработчиков были лучшие намерения программирования, используя интерфейсы (*programming to interfaces*), но они никогда не понимали движущие силы и мотивацию.

Пример: обновление зависимостей

Многие разработчики слышали о принципе *программировании, используя интерфейсы*, но не понимают глубокий смысл всего этого. В попытке сделать правильную вещь или следовать советам и рекомендациям, они пишут код, который не имеет особого смысла.

В главе 2, вы видели пример `ProductService`, который использует экземпляр абстрактного класса `ProductRepository` (листинг 2-6), чтобы получить список

рекомендуемых товаров. В качестве напоминания, вот соответствующий метод по своей природе:

```
public IEnumerable<Product> GetFeaturedProducts(IPrincipal user)
{
    return from p in this.repository.GetFeaturedProducts()
    select p.ApplyDiscountFor(user);
}
```

По сравнению с листингом 2-6 я опустил ограждающее условие, но характерный момент заключается в том, что переменная `repository` представляет абстрактный класс. В главе 2 вы заметили, как поле `repository` может быть заполнено через внедрение в конструктор, но я видел и другие, более наивные попытки, например, вот такую.

Листинг 5-1: Обновление `ProductRepository`

```
1. private readonly ProductRepository repository;
2. public ProductService()
3. {
4.     string connectionString =
5.         ConfigurationManager.ConnectionStrings
6.         ["CommerceObjectContext"].ConnectionString;
7.     this.repository = new SqlProductRepository(connectionString);
8. }
```

Строки 7: Напрямую создается новый экземпляр

Поле `repository` объявлено как абстрактный класс `ProductRepository`, поэтому любой член в классе `ProductService` (например, `GetFeaturedProducts`) будут разрабатываться при помощи интерфейса. Хотя это звучит правильно, но это не принесет особой пользы, потому что во время выполнения тип всегда будет `SqlProductRepository`. Нет никакой возможности перехватить или изменить переменную `repository`, пока вы не поменяете код и перекомпилируете его.

Вы не получите многого, определив переменную как интерфейс или абстрактный класс, если вы ее жестко закодировали на то, чтобы она всегда имела конкретный тип. Единственное небольшое преимущество заключается в том, что конкретный тип определяется только в одном или нескольких местах в программном коде, поэтому замена одной конкретной реализации другой не потребует крупного рефакторинга. В этом примере вы должны только довести другую реализацию `ProductRepository` в конструктор, в то время как остальная часть `ProductService` будет работать без изменений.

Обновление зависимостей напрямую при помощи `new` является лишь одним примером анти-паттерна `Control Freak`. Прежде чем я перейду к анализу и возможным путям исправления проблемы, созданной `Control Freak`, давайте посмотрим на несколько примеров, которые дадут вам более полное представление о контексте и общих неудачных попытках решения некоторых из полученных проблем.

В этом конкретном примере очевидно, что решение не является оптимальным. Большинство разработчиков попытается усовершенствовать свой подход, как вы увидите в следующем примере.

Пример: фабрика

Наиболее распространенная (ошибочная) попытка исправить явные проблемы с обновлением зависимости при помощи `new` включает в себя какую-то фабрику. Есть несколько вариантов, где речь идет о фабриках, и я вкратце опишу каждый из них:

- Конкретная фабрика (Concrete Factory)
- Абстрактная фабрика (Abstract Factory)
- Статическая фабрика (Static Factory)

Если сказать Мэри Роуэн (из главы 2), что она может иметь дело только с абстрактным классом `ProductRepository`, она введет `ProductRepositoryFactory`, которая будет создавать экземпляры, нужные для того, чтобы работа была сделана. Давайте послушаем, как она обсуждает такой подход со своей коллегой Йенсом: я думаю, что их обсуждение будет охватывать варианты фабрик, которые я перечислил:

Мэри: Нам нужен экземпляр `ProductRepository` в классе `ProductService`. Тем не менее, `ProductRepository` является абстрактным, поэтому мы не можем просто создать его новые экземпляры, и наш консультант говорит, что мы не должны также создавать новые экземпляры `SqlProductRepository`.

Йенс: Как насчет фабрики?

Мэри: Да, я подумала о том же, но я не знаю, что делать дальше. Я не понимаю, как она решит наши проблемы. Смотри ...

Мэри начинает писать код, чтобы показать проблему.

Конкретная фабрика

Это код, который пишет Мэри:

```
public class ProductRepositoryFactory
{
    public ProductRepository Create()
    {
        string connectionString =
            ConfigurationManager.ConnectionStrings
                ["CommerceObjectContext"].ConnectionString;
        return new SqlProductRepository(connectionString);
    }
}
```

Мэри: Эта `ProductRepositoryFactory` инкапсулирует знания о том, как создать экземпляры `ProductRepository`, но это не решает проблемы, потому что нам нужно использовать это в `ProductService` вот так:

```
var factory = new ProductRepositoryFactory();
this.repository = factory.Create();
```

Мэри: Видишь? Теперь мы просто должны создать новый экземпляр класса `ProductRepositoryFactory` в `ProductService`, но это все же жестко кодирует

использование `SqlProductRepository`. Единственное, чего мы добились, это переместили проблему в другой класс.

Йенс: Да, я вижу... А мы не можем решить проблему при помощи абстрактной фабрики?

Давайте поставим на паузу обсуждения Мэри и Йенса, чтобы оценить то, что произошло. Мэри совершенно права в том, что класс конкретной фабрики не решает проблему `Control Freak`, а только перемещает ее дальше. Это делает код более сложным. `ProductService` теперь непосредственно контролирует жизненный цикл фабрики, а фабрика непосредственно управляет жизненным циклом `ProductRepository`, поэтому мы до сих пор не можем перехватить или заменить экземпляр `repository` во время выполнения.

Примечание

Не делайте вывод по этому разделу, что я вообще против использования классов конкретной фабрики. Конкретная фабрика может решить другие проблемы, такие как повторение кода, путем инкапсуляции сложной логики создания. Она просто не имеет никакой ценности для DI. Используйте ее, когда это имеет смысл.

Довольно очевидно, что конкретная фабрика не решит никаких проблем DI, и я думаю, что я никогда видел, чтобы она использовалась таким образом. Комментарий Йенса об абстрактной фабрике звучит более перспективно.

Абстрактная фабрика

Давайте продолжим дискуссию Мэри и Йенса и посмотрим, что Йенс может сказать об абстрактной фабрике.

Йенс: А что если мы сделаем фабрику абстрактной? Например, вот так:

```
public abstract class ProductRepositoryFactory
{
    public abstract ProductRepository Create();
}
```

Йенс: Это означает, что мы жестко не кодируем любые ссылки на `SqlProductRepository`, и мы можем использовать фабрику в `ProductService`, чтобы получить экземпляры `ProductRepository`.

Мэри: Но теперь, когда фабрика абстрактная, как мы получим ее новые экземпляры?

Йенс: мы создадим ее реализацию, которая возвращает экземпляры `SqlProductService`.

Мэри: Да, но как мы создадим экземпляр этого?

Йенс: мы просто используем ключевое слово `new` в `ProductService`... Ой, подожди...

Мэри: Это просто вернет нас назад, от куда мы начали.

Мэри и Йенс быстро поняли, что абстрактная фабрика не меняет ситуацию. Суть заключается в том, что им нужен был экземпляр абстрактного класса `ProductRepository`, а теперь вместо этого нужен экземпляр абстрактной `ProductRepositoryFactory`.

Абстрактная фабрика

Абстрактная фабрика является одним из шаблонов проектирования из оригинальной книги *Design Patterns*. Она полезна для DI, потому что она может инкапсулировать сложную логику, которая создает другие зависимости.

Она предлагает хорошую альтернативу полной передачи управления, которая участвует в инверсии управления, потому что она частично позволяет потребителю контролировать жизненный цикл зависимостей, созданных фабрикой; фабрика по-прежнему управляет тем, что создается и как это создается.

Паттерн *Abstract Factory* встречается гораздо чаще, нежели вы можете ожидать: имена включенных классов часто скрывают этот факт. Класс `CurrencyProvider` представленный в разделе 4.1.4, на самом деле является абстрактной фабрикой с другим именем: это абстрактный класс, который создает экземпляры другого абстрактного класса (`Currency`).

В разделе 6.1 мы вернемся к паттерну *Abstract Factory*, чтобы увидеть, как он может помочь в решении одного типа проблем, которые часто происходят с DI.

Мэри и Йенс сделали только такую реализацию фабрики, которая не была бы вредна для них. С другой стороны, это также не решит их проблемы, а поскольку предполагается, что логика создания для экземпляров `ProductRepository` не будет сложной, такое использование не имеет никакой ценности.

Теперь, когда Мэри и Йенс отвергли единственную безопасную реализацию фабрики, только один вариант остается открытым.

Статическая фабрика

Мэри и Йенс собираются прийти к выводу. Давайте послушаем, как они принимают решения о подходе, который, как они думают, будет работать:

Мэри: Давай сделаем статическую фабрику. Я тебе покажу:

```
public static class ProductRepositoryFactory
{
    public static ProductRepository Create()
    {
        string connectionString =
            ConfigurationManager.ConnectionStrings
            ["CommerceObjectContext"].ConnectionString;
        return new SqlProductRepository(connectionString);
    }
}
```

Мэри: Теперь, когда класс статический, нам не нужно думать, как его создать.

Йенс: Но у нас по-прежнему остается жесткая закодированность в том, что мы возвращаем экземпляры `SqlProductRepository`, поэтому разве это нам как-то поможет?

Мэри: Мы могли бы справиться с этим через настройку конфигурации, которая определяет, какой тип `ProductRepository` создать. Примерно вот так:


```

public static ProductRepository Create()
{
    var repositoryType =
        ConfigurationManager.AppSettings["productRepository"];
    switch (repositoryType)
    {
        case "sql":
            return ProductRepositoryFactory.CreateSql();
        case "azure":
            return ProductRepositoryFactory.CreateAzure();
        default:
            throw new InvalidOperationException("...");
    }
}

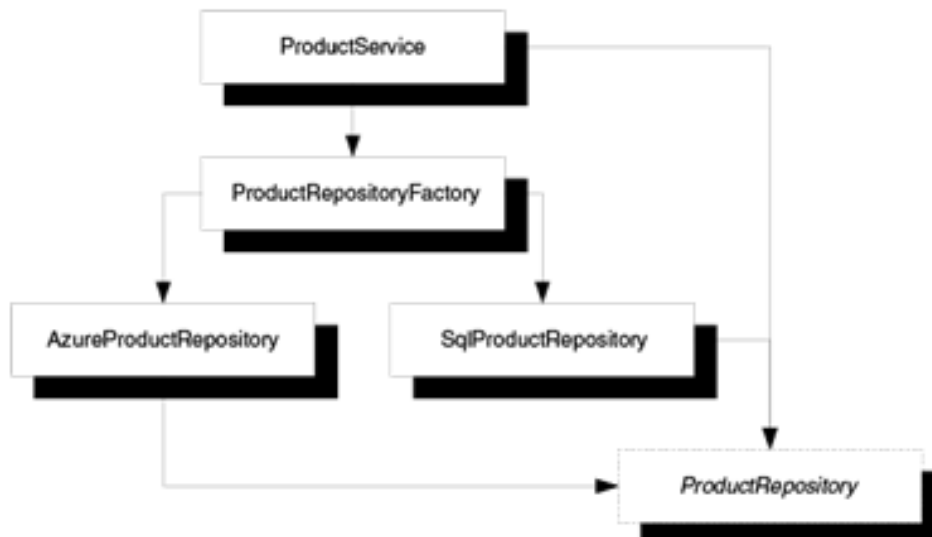
```

Мэри: Видишь? Таким образом, мы можем определить, должны ли мы использовать реализацию на основе SQL Server или реализацию на основе Windows Azure, и нам даже не нужно перекомпилировать приложение, чтобы переходить от одной к другой.

Йенс: Супер! Это то, что мы сделаем. Консультант должен быть счастлив.

Есть несколько причин, почему такая статическая фабрика не дает удовлетворительного решения первоначальной цели программирования при помощи интерфейсов. Давайте посмотрим на граф зависимостей на рисунке 5-2.

Рисунок 5-2: Граф зависимостей для предполагаемого решения: статический
 ProductRepositoryFactory используется для создания экземпляров ProductRepository.



Я не приукрашиваю

Если бы я был консультантом в этом примере, я не был бы в восторге. На самом деле, такое "решение" было предложено на проекте, с которым я был связан, и я написал 14-страничный документ, описывающий, почему это не будет работать и что делать вместо этого.

Это был довольно большой проект, который был нацелен на центральную деловую область компании Fortune 500, поэтому надлежащая модульность была важна в связи со сложностью применения. К сожалению, я стал участвовать в проекте слишком поздно, и

мои предложения были отклонены, потому что они несли драматические изменения в уже разработанный код.

Я переключился на другие проекты, но позже я узнал, что, хотя команде удалось сделать достаточно для выполнения контракта, проект был признан неудачным, и полетели головы.

Было бы неразумным утверждать, что проект не удался только потому, что не было использовано DI, но принятый подход был симптомом отсутствия надлежащего проектирования. Я не могу сказать, что я был удивлен, узнав, что проект так и не удался.

Все классы должны ссылаться на абстрактный класс `ProductRepository`:

- `ProductService`, потому что потребляет экземпляры `ProductRepository`
- `ProductRepositoryFactory`, потому что раскрывает экземпляры `ProductRepository`
- `AzureProductRepository` и `SqlProductRepository`, потому что они реализуют `ProductRepository`

`ProductRepositoryFactory` зависит от обоих классов `AzureProductRepository` и `SqlProductRepository`. Поскольку `ProductService` напрямую зависит от `ProductRepositoryFactory`, он также зависит от обеих реализаций `ProductRepository`.

Крах зависимости

Плохая ситуация возникает, когда абстракция `ProductRepository` и потребляющий `ProductService` определены в одной сборке (как это имеет место с реализациями, которые я создавал в книге до сих пор). Давайте предположим, что это сборка доменной модели. В этом случае, `ProductRepositoryFactory` также должна быть в этой же сборке, иначе у нас была бы циклическая ссылка, которая не представляется возможной.

Тем не менее, фабрика имеет ссылки на обе реализации, а у них есть ссылка на сборку доменной модели, поскольку они реализуют класс `ProductRepository`. Опять же, единственный способ избежать циклической ссылки заключается в размещении конкретных реализаций в той же сборке.

Когда `AzureProductRepository` и `SqlProductRepository` реализованы в сборке доменной модели, это полностью идет вразрез с принципом разделения понятий (*Separation of Concerns*). Мы, по существу, останемся с монолитным приложением.

Единственный выход из этой проблемы заключается в определении абстракции `ProductRepository` в отдельной сборке. Это может быть хорошей идеей по многим другим причинам, но этого не достаточно, чтобы сделать статическую фабрику жизнеспособным решением для DI.

Вместо слабо связанных реализаций `ProductRepository`, Мэри и Йенс в итоге получили тесно связанные модули. Хуже того, фабрика всегда тянет за собой все реализации, даже те, которые не нужны.

Если Мэри и Йенсу когда-нибудь понадобится третий тип `ProductRepository`, им придется изменить фабрику и перекомпилировать решение. Хотя их решение может быть настраиваемым, оно не является расширяемым.

Также невозможно заменить конкретные реализации `ProductRepository` тестовыми реализациями, такими как динамические *mock*, потому что это потребует определить экземпляры `ProductRepository` во время выполнения, а не статически в файле конфигурации во время разработки.

Примечание

Динамические *mock* выходят за рамки этой книги, но я кратко коснулся данной темы когда я описывал тестируемость в главе 1 (раздел 1.2.2).

Короче говоря, кажется, что статическая фабрика решает проблему, но в действительности лишь усугубляет ее. Даже в лучшем случае это заставит вас ссылаться на непостоянные зависимости.

Теперь, когда вы увидели много примеров Control Freak, я надеюсь, у вас есть довольно хорошее представление о том, что искать: вхождения ключевого слова `new` рядом с зависимостями. Это может позволить вам избежать наиболее очевидных ловушек, но если вам нужно оградить себя от возникновения этого анти-паттерна, в следующем разделе обсуждается, как бороться с такой проблемой.

Анализ

Control Freak является антитезой инверсии управления. Когда мы напрямую управляем созданием изменчивых зависимостей, мы в конечном итоге получаем тесно связанный код, теряя многие (если не все) преимущества слабой связанности, описанной в главе 1.

Влияние

С тесно связанным кодом, который является результатом Control Freak, теряются многие преимущества модульной конструкции:

- Хотя мы можем настроить приложение для работы с одной из нескольких предварительно сконфигурированных зависимостей, мы не можем заменить их, когда захотим. Не возможно предоставить реализацию, которая была создана после того, как было скомпилировано приложение, и, конечно же, не возможно предоставить конкретные экземпляры реализации.
- Все сложнее становится повторно использовать модуль, поскольку он приносит с собой зависимости, которые могут быть нежелательны в новом контексте.
- Это усложняет параллельную разработку, потому что потребляющее приложение тесно связано со всеми реализациями его зависимостей.
- Страдает тестируемость, потому что динамические *mock* не могут быть использованы в качестве заменителей зависимостей.

При тщательном проектировании мы можем все еще быть в состоянии реализовывать тесно связанные приложения с четко определенными обязанностями, так что поддерживаемость не страдает, но даже при этом цена слишком высока. Нам нужно отойти от Control Freak к надлежащим DI.

Рефакторинг к DI

Чтобы избавиться от Control Freak, мы должны сделать рефакторинг нашего кода к одному из паттернов проектирования DI, представленным в главе 4. В качестве первого

шага, мы должны использовать руководство, чтобы определить, какой паттерн выбрать. В большинстве случаев это будет внедрение в конструктор. Шаги рефакторинга заключаются в следующем:

1. Убедитесь, что вы *программируете, используя интерфейсы*. В примерах, которые я только что представил, это уже имело место, но в других ситуациях, возможно, потребуется сначала извлечь интерфейс и изменить объявления переменных.
2. Если вы создаете конкретную реализацию зависимостей в нескольких местах, переместите их все в один метода создания. Убедитесь, что возвращаемое значение этого метода выражается в виде абстракции, а не конкретного типа.
3. Теперь, когда у вас есть только одно место, где вы создаете экземпляр, переместите это создание из потребляющего класса путем реализации одного из DI паттернов, такого как внедрение в конструктор.

В случае с примерами `ProductService` из предыдущих разделов, внедрение в конструктор является отличным решением:

```
private readonly ProductRepository repository;
public ProductService(ProductRepository repository)
{
    if (repository == null)
    {
        throw new ArgumentNullException("repository");
    }
    this.repository = repository;
}
```

В некоторых случаях исходный код использует сложную логику для определения того, как создавать экземпляры зависимости. В таком случае эта сложная логика может быть реализована в фабрике, а затем можно извлечь интерфейс такой фабрики для создания абстрактной фабрики. В сущности, это означает, что зависимость меняется, так что новая абстрактная фабрика становится зависимостью вместо первоначальной абстракции, и мы можем применить ту же логику рефакторинга к фабрике. В большинстве случаев мы в конечном итоге внедряем фабрику в потребляющий класс через его конструктор.

Control Freak является наиболее распространенным DI анти-паттерном. Он представляет собой способ по умолчанию создания экземпляров в большинстве языков программирования, так что его можно видеть даже в тех приложениях, где разработчики никогда не использовали DI. Это такой естественный и глубоко укорененный способ создания новых объектов, что многим разработчикам трудно от этого отказаться. Даже тогда, когда разработчики начинают думать о DI, многие все же полагают, что они должны как-то контролировать, когда и где создаются экземпляры. Отказаться от такого контроля может быть трудным психическим прыжком, но даже если вы сделаете это, есть и другие, хотя и более мелкие, ловушки, которых нужно избегать.

Control Freak на сегодняшний день является самым разрушительным анти-паттерном, но даже если у вас все под контролем, могут возникнуть более тонкие проблемы. В следующих разделах рассматриваются эти анти-паттерны. Хотя они менее проблематичны, чем Control Freak, их также, как правило, легче решить, так что внимательно смотрите и исправляйте их, когда вы их обнаружите.

5.2. Bastard Injection

Перегруженные конструкторы являются довольно распространенными во многих базах кода .NET (включая BCL). Часто многие перегруженные варианты обеспечивают разумные значения по умолчанию для одного или двух полномасштабных конструкторов, которые принимают все соответствующие параметры в качестве входных данных.

Время от времени мы видим другое использование, когда речь идет о DI. Довольно общий анти-паттерн определяет предназначенный для тестирования перегруженный вариант конструктора, который позволяет нам явно определить зависимость, в то время как конструктор по умолчанию используется в производственном коде.

Это может быть вредным, когда реализация зависимости по умолчанию представляет *Foreign Default*, а не *Local Default*.

Когда мы полностью охватываем DI, такие перегруженные конструкторы становятся лишними в лучшем случае. Учитывая негативные последствия, лучше избегать их.

ProductService с Foreign Default

Когда Мэри первоначально реализовала класс `ProductService` (в главе 2), она имела в виду только одну зависимость: реализацию на основе `SQL Server`. Класс `SqlProductRepository` изначально был задуман как единственная реализация `ProductRepository`, поэтому казалось очевидным использовать его по умолчанию.

Foreign Default

Foreign Default является противоположностью *Local Default*. Это реализация зависимости, которая используется по умолчанию, даже если она определена в другом модуле, чем ее потребитель.

В качестве примера давайте рассмотрим реализации `Repository`, которые мы видели в примере коммерческого приложения в предыдущих главах. Сервис, такой как `ProductService`, требует для работы экземпляр `ProductRepository`. Во многих случаях, когда мы разрабатываем такие приложения, мы имеем в виду разумную реализацию: ту, которая реализует нужный функционал путем чтения и записи данных в и из реляционной базы данных. Было бы заманчиво использовать такую реализацию по умолчанию.

Проблема заключается в том, что реализация по умолчанию, которую мы имеем в виду (`SqlProductRepository`), определена в другом модуле, нежели `ProductService`. Это заставляет нас принять нежелательную зависимость для модуля `CommerceSqlDataAccess`, как показано здесь.

Когда `ProductService` использует `SqlProductRepository` в качестве реализации по умолчанию, это заставляет нас делать жесткую ссылку на модуль `CommerceSqlDataAccess`, а мы этого не хотим.



Использование нежелательных модулей отнимает у нас многие преимущества слабой связанности, которые обсуждались в главе 1. Все сложнее становится повторное использование модуля `CommerceDomain`, потому что он потянет за собой модуль `CommerceSqlDataAccess`, а мы, возможно, не захотим использовать это в другом контексте. Это также усложняет параллельную разработку, потому что класс `ProductService` теперь напрямую зависит от класса `SqlProductRepository`.

Таковы основные причины того, что вы должны избегать *Foreign Default*, если это вообще возможно.

Мэри еще не чувствует себя комфортно с идеей внедрения в конструктор, потому что у нее есть проблема в выяснении, где будет размещена структура объекта. Она до сих пор не обращала внимания на концепцию *Composition Root*.

Приглашенный консультант сказал ей использовать внедрение в конструктор для `ProductService`, но она по-прежнему считает, что она должна создать его новый экземпляр следующим образом:

```
var productService = new ProductService();
```

Для этого она добавляет следующий код в класс `ProductService`.

Листинг 5-2: `ProductService` с Bastard Injection

```
private readonly ProductRepository repository;
public ProductService()
    : this(ProductService.CreateDefaultRepository())
{
}
public ProductService(ProductRepository repository)
{
    if (repository == null)
    {
        throw new ArgumentNullException("repository");
    }
    this.repository = repository;
}
private static ProductRepository CreateDefaultRepository()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
            ["CommerceObjectContext"].ConnectionString;
    return new SqlProductRepository(connectionString);
}
```

Строки 2-5: Конструктор по умолчанию

Строки 6-13: Внедрение в конструктор

Класс `ProductService` теперь имеет конструктор по умолчанию, который вызывает его другой конструктор, используя *Foreign Default*.

Другой конструктор правильно реализует паттерн внедрение в конструктор, имея ограждающее условие, а затем сохраняя внедренный `ProductRepository` в поле только для чтения. Конструктор по умолчанию вызывает этот конструктор с *Foreign Default*, созданной в закрытом методе `CreateDefaultRepository`. Класс `SqlProductRepository` является *Foreign Default*, поскольку он определен в другой сборке, чем класс `ProductService`. Это приводит к тому, что сборка, содержащая класс `ProductService`, тесно связана со сборкой, содержащей класс `SqlProductRepository`.

Хотя `ProductService` можно использовать с различными типами `ProductRepository`, передавая их с помощью самого гибкого перегруженного варианта конструктора, Мэри не сможет перехватить экземпляр `ProductRepository` в приложении, если она настаивает на использовании конструктора по умолчанию.

Анализ

Bastard Injection наиболее часто встречается, когда разработчики пытаются сделать свои классы тестируемыми без полного понимания DI. При написании модульных тестов для класса очень важно, чтобы мы могли заменить изменчивую зависимость дублирующим тестом, так чтобы мы могли правильно изолировать тестируемую систему (SUT) от ее зависимостей, и внедрение в конструктор позволяет сделать именно это.

Хотя Bastard Injection включает тестируемость, он имеет некоторые нежелательные последствия.

Конкретный пример: ASP.NET MVC

Когда вы создаете новый ASP.NET MVC проект, автоматически создается несколько стандартных классов контроллеров. Одним из них является класс `AccountController`, который использует Bastard Injection. В исходном коде это объясняется даже в комментариях:

```
// Этот конструктор используется MVC фреймворком,
// чтобы создать экземпляр контроллера при помощи
// форм аутентификации по умолчанию и провайдеров членства.
public AccountController()
    : this(null, null)
{
}
// Этот конструктор не используется MVC фреймворком,
// но он используется для упрощения юнит тестирования этого типа.
// Посмотрите комментарии в конце этого файла
// для более полной информации.
public AccountController(IFormsAuthentication formsAuth,
    IMembershipService service)
{
    this.FormsAuth = formsAuth ?? new FormsAuthenticationService();
    this.MembershipService = service ?? new AccountMembershipService();
}
```

Как я могу сказать, что Bastard Injection – это плохо, когда кажется, что Microsoft использует и одобряет его? В данном случае, мотивация, кажется, исключительно связана

с тестируемостью, и Bastard Injection вполне соответствует этой цели – он просто не соответствует другим целям модульности, таким как способность к замещению и повторному использованию модулей, а также параллельной разработке.

Другие придерживаются того же мнения. Айенде Райен отметил следующее в своем блоге, в котором обсуждалось ASP.NET MVC приложение:

Я имею в виду, если вы хотите сделать слабый IoC – вперед. Но, пожалуйста, не создавайте этого бастарда.

Эта фраза вдохновила меня назвать анти-паттерн так, как я это сделал.

Влияние

Основная проблема с Bastard Injection заключается в использовании *Foreign Default*. Хотя тестируемость включена, мы больше не можем свободно повторно использовать класс, потому что это потянет за собой зависимость, которую мы не хотим. Кроме того, параллельная разработка усложняется, потому что класс сильно зависит от своей зависимости.

В дополнение к последствиям Bastard Injection для модульности приложения, существование множества конструкторов также представляет собой другой тип проблемы. Когда есть только один конструктор, DI контейнер может автоматически проводить все зависимости, потому что никогда не встает вопрос, какой конструктор использовать.

Когда существует более чем один конструктор, выбор между конструкторами становится неоднозначным. DI контейнеру приходится прибегать к своего рода эвристике, чтобы сделать выбор между различными конструкторами, или он может от этого отказаться. Как ни странно, это один из немногих сценариев, где бедное DI менее подвержено влиянию, потому что мы можем решать в каждом конкретном случае индивидуально, когда мы вручную проводим зависимости.

Среди различных анти-паттернов DI, Bastard Injection не так вреден, как Control Freak, но и от него также гораздо легче избавиться.

Рефакторинг по направлению к DI

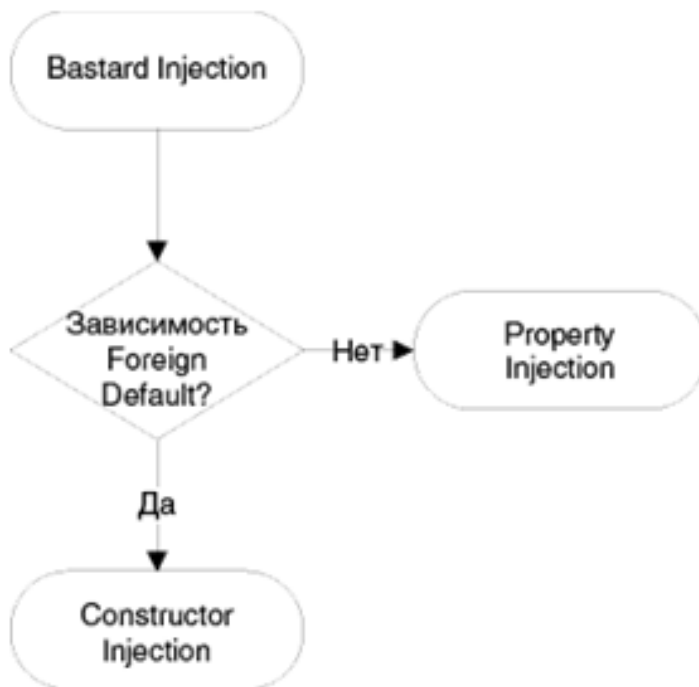
Bastard Injection часто является результатом ошибочной попытки реализовать DI. Хорошо, что есть такие основы, как программирование к интерфейсам, так что провести рефакторинг к надлежащему DI паттерну легко.

Совет

Даже если вы думаете, что воздействие Bastard Injection вас не касается, вы все равно должны провести рефакторинг к надлежащему DI паттерну. Ведь это так просто сделать, что не стоит даже сомневаться в нужности этого.

Первый шаг заключается в выборе того, какой DI паттерн соответствует цели. Рисунок 5-3 иллюстрирует простой процесс принятия решений. Если значение по умолчанию, которые было использовано до сих пор, это *Foreign Default*, лучшим выбором является внедрение в конструктор. В другом случае, хорошей альтернативой считается внедрение в свойство.

Рисунок 5-3: При рефакторинге от Bastard Injection решающий фактор заключается в том, является ли зависимость *Foreign Default* или *Local Default*.



Во многих случаях, значение по умолчанию, используемое конструктором по умолчанию, представляет *Foreign Default*. В этих случаях внедрение в конструктор является лучшим ответом, потому что он так прост в реализации и хорошо справляется с любыми видами зависимостей. Конструктор, который принимает зависимость в качестве параметра, уже существует, так что единственное изменение, которое мы должны сделать в используемом классе, заключается в удалении конструктора по умолчанию.

Это, несомненно, приведет к некоторым ошибкам компилятора, но на данный момент мы можем опереться на компилятор и переместить весь код, который создает рассматриваемый класс, в Composition Root.

Когда значение по умолчанию представляет *Local Default*, ситуация очень напоминает основной сценарий для внедрения в свойство. Хотя механика и разная, структура такая же: в обоих случаях у нас есть подходящая *Local Default*, но мы все еще хотим открыть наш потребляющий класс для расширяемости.

Это вырождающийся случай Bastard Injection, где воздействие гораздо менее серьезное. Поскольку значение по умолчанию является *Local Default*, нет никакого влияния на степень компоновки класса; единственным негативным последствием является то, что двусмысленность конструктора делает автовнедрение более сложным.

В данном случае мы могли бы реализовать внедрение в свойство, удалив конструктор, который принимает зависимость в качестве параметра, и заменив его свойством для записи. Если это изменение приводит к ошибкам компилятора, мы снова можем опереться на компилятор и переместить создание кода в Composition Root.

В Composition Root есть много различных способов компоновать зависимости, в том числе такие, которые далеки от идеальных, как покажет следующий анти-паттерн.

5.3. Constrained Construction

Самая большая проблема должным образом реализовать DI заключается в том, чтобы все классы с зависимостями были перемещены в Composition Root. Пока мы достигнем этого, мы пройдем долгий путь.

И даже тогда все еще существуют некоторые ловушки, которые нужно отслеживать. Распространенной ошибкой является требование того, чтобы у всех зависимостей был конструктор с конкретной сигнатурой. Как правило, это происходит от желания достичь поздней связанности, так чтобы зависимости могли быть определены во внешнем файле конфигурации и тем самым изменены без повторной компиляции приложения.

Примечание

Так называемый паттерн *Провайдер* (Provider), используемый в ASP.NET, является примером Constrained Construction, потому что *Провайдеры* должны иметь конструкторы по умолчанию. Это, как правило, усугубляется тем, что конструктор *Провайдера* пытается читать файл конфигурации приложения. Часто конструктор генерирует исключение, если необходимый раздел файла конфигурации недоступен.

Примечание

Этот раздел относится только к тем сценариям, где нужно позднее связывание. В сценариях, где мы напрямую ссылаемся на зависимости из корня приложения, у нас нет этой проблемы – но опять же, у нас также нет возможности для замены зависимостей без повторной компиляции.

В главе 3 мы кратко коснулись этого вопроса. В этом разделе мы рассмотрим его более тщательно.

Пример: отложенная связанность ProductRepository

В примере коммерческого приложения некоторые классы зависят от абстрактного класса ProductRepository. Это означает, что для создания этих классов в первую очередь необходимо создать экземпляр ProductRepository. На данный момент вы узнали, что Composition Root – это нужное место, чтобы сделать это. В приложении ASP.NET для этого есть Global.asax; следующий листинг показывает соответствующую часть, где создается экземпляр ProductRepository.

Листинг 5-3: Неявное ограничение конструктора ProductRepository

```
1. string connectionString =
2.     ConfigurationManager.ConnectionStrings
3.     ["CommerceObjectContext"].ConnectionString;
4. string productRepositoryTypeName =
5.     ConfigurationManager.AppSettings
6.     ["ProductRepositoryType"];
7. var productRepositoryType =
8.     Type.GetType(productRepositoryTypeName, true);
9. var repository =
10.    (ProductRepository)Activator.CreateInstance(
11.        productRepositoryType, connectionString);
```

Строки 9-11: Создание экземпляра конкретного типа

Первое, что должно вызвать подозрение – это то, что строка соединения считывается из файла `web.config`. Зачем вам нужна строка соединения, если вы планируете обрабатывать `ProductRepository` как абстракцию? Хотя, возможно, это и маловероятно, но вы можете захотеть реализовать `ProductRepository` с базой данных в памяти или XML файлом. REST-сервис хранения данных, такой как Windows Azure Table Storage Service предлагает более реалистичную альтернативу, но в очередной раз самым популярным выбором, кажется, остаются реляционные базы данных. Повсеместное распространение баз данных ведет к тому, что слишком легко забыть, что строка соединения неявно представляет выбор реализации.

Чтобы сделать позднюю привязку `ProductRepository`, вы должны определить, какой тип был выбран в качестве реализации. Это можно сделать, прочитав имя типа, определенное сборкой, из `web.config` и создав экземпляр типа с таким именем. Это само по себе не является проблемой – трудность возникает только тогда, когда вам нужно создать экземпляр этого типа.

С наличием `Type` вы можете создать экземпляр с помощью класса `Activator`. Метод `CreateInstance` вызывает конструктор типа, поэтому вы должны передать верные параметры конструктору, чтобы предотвратить исключение. В этом случае нужно указать строку соединения.

Если бы вы ничего не знали о приложении, только видели код в листинге 5-3, то вы бы удивились, почему строка соединения передается в качестве аргумента конструктора неизвестному типу. Это не имело бы больше смысла, если бы реализация была основана на REST веб-сервисе или XML файле.

На самом деле, это не имеет смысла, так как это представляет собой случайное ограничение для конструктора зависимости. В данном случае у вас есть неявное требование, чтобы любая реализация `ProductRepository` имела конструктор, который принимает одну строку в качестве входных данных. Это дополнение к явному ограничению, что класс должен наследоваться от `ProductRepository`.

Примечание

Неявное ограничение, что конструктор должен принять одну строку, по-прежнему оставляет нам большую степень гибкости, потому что мы можем кодировать много разной информации в строках для дальнейшего декодирования. Представьте себе, что вместо этого ограничения был бы конструктор, который принимает `TimeSpan` и число, и вы можете себе представить, насколько ограничивающим это было бы.

Можно утверждать, что `ProductRepository` на основе XML файла также потребует строку в качестве параметра конструктора, хотя этой строкой будет имя файла, а не строка соединения. Тем не менее, концептуально это все равно будет странно, потому что вам все равно нужно было бы определить это имя файла в элементе `connectionStrings` в `web.config` (и в любом случае, я думаю, что такой гипотетический `XmlProductRepository` должен принять `XmlReader` в качестве аргумента конструктора, а не имя файла).

Моделирование конструкции зависимости исключительно на явных ограничениях (интерфейса или базового класса) является намного более хорошим и более гибким вариантом.

Анализ

В предыдущем примере неявное ограничение требует от реализующих элементов иметь конструктор с одним строковым параметром. Более распространенным ограничением является то, что все реализации должны иметь конструктор по умолчанию, так чтобы работала простейшая форма `Activator.CreateInstance`:

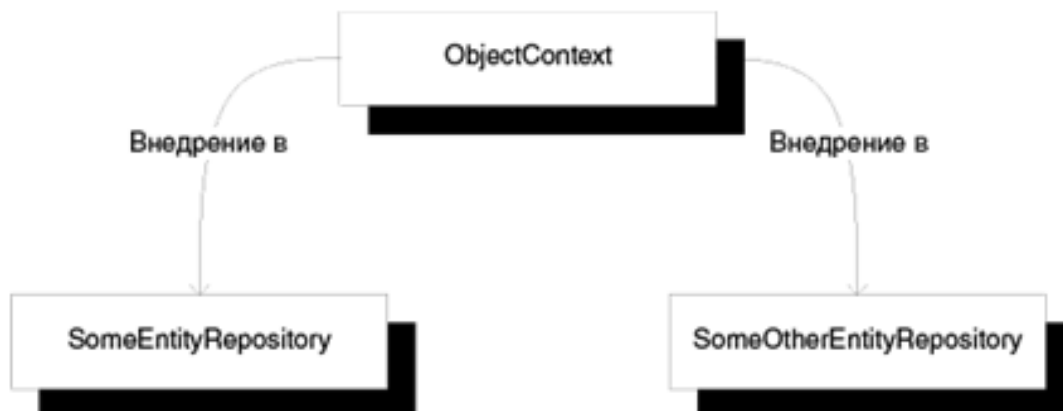
```
var dep = (ISomeDependency) Activator.CreateInstance (type) ;
```

Хотя это можно назвать наименьшим общим знаменателем, цена гибкости слишком высока.

Влияние

Независимо от того, как мы ограничиваем строение объекта, мы теряем гибкость. Может возникнуть соблазн заявить, что все реализации зависимостей должны иметь конструктор по умолчанию – в конце концов, они могли бы выполнять свою инициализацию внутренне, например, чтением конфигурационных данных, таких как конфигурационные строки, непосредственно из файла `.config`. Однако это ограничило бы нас по-другому, потому что мы, возможно, захотели бы иметь возможность компоновать приложение слоями экземпляров, которые включают другие экземпляры. В некоторых случаях, например, мы могли бы захотеть распределить экземпляры между различными потребителями, как показано на рисунке 5-4.

Рисунок 5-4: В этом примере мы хотим создать единственный экземпляр класса `ObjectContext` и внедрить этот же экземпляр в оба репозитория. Это возможно только в том случае, если мы можем внедрить экземпляр извне.



Когда у нас есть более чем один класс, требующий одну и ту же зависимость, мы, возможно, захотим поделиться одним экземпляром со всеми этими классами. Это возможно только тогда, когда мы можем внедрить этот экземпляр извне. Хотя мы могли бы написать код внутри каждого из этих классов, чтобы прочитать информацию о типах из конфигурационного файла, и использовать `Activator.CreateInstance` для создания правильного типа экземпляра, мы никогда не смогли бы поделиться одним экземпляром таким способом – вместо этого у нас было бы несколько экземпляров одного класса, которые занимали бы больше памяти.

Примечание

Только потому, что DI позволяет нам делиться одним экземпляром среди многих потребителей, не означает, что мы всегда должны это сделать. Совместное использование экземпляра экономит память, но может создать проблемы взаимодействия, например, проблем многопоточности. Решение о том, хотим мы поделиться экземпляром или нет, тесно связано с концепцией жизненного цикла объекта, которая обсуждается в главе 8.

Вместо введения неявных ограничений о том, как объекты должны быть построены, мы должны реализовать наш Composition Root, так чтобы он мог работать с любым конструктором или методом фабрики, которые мы можем ему дать.

Рефакторинг по направлению к DI

Что делать, если у нас нет ограничений для компонентов конструктора, а нам нужно позднее связывание? Может возникнуть соблазн ввести абстрактную фабрику, которую можно использовать для создания экземпляров необходимой абстракции, а затем потребовать, чтобы реализации такой абстрактной фабрики имели конструкторы по умолчанию, но это приведет к тому, что мы будем лишь ходить вокруг да около проблемы, а не решать ее.

Внимание

Хотя мы можем использовать абстрактные фабрики, чтобы успешно реализовать позднее связывание, это требует дисциплины. В общем, нам будет лучше с надлежащим DI контейнером, но я, тем не менее, покажу, как это сделать.

Давайте кратко рассмотрим такой подход. Представьте себе, что у вас есть абстракция сервиса, образно называемая `ISomeService`. Схема абстрактной фабрики подсказывает, что вам также нужен интерфейс `ISomeServiceFactory`. Рисунок 5-5 иллюстрирует эту структуру.

Рисунок 5-5: `ISomeService` представляет реальную зависимость. Однако чтобы сохранить реализующие элементы свободными от неявных ограничений, вы пытаетесь разрешить вопрос поздней связанности путем введения `ISomeServiceFactory`, которая будет использоваться для создания экземпляров `ISomeService`. И вам потребуется любая фабрика, поскольку у нее есть конструктор по умолчанию.



Теперь давайте предположим, что вы хотите использовать реализацию `ISomeService`, которому требуется рабочий экземпляр `ISomeRepository`, как показано в следующем листинге.

Листинг 5-4: SomeService который требует ISomeRepository

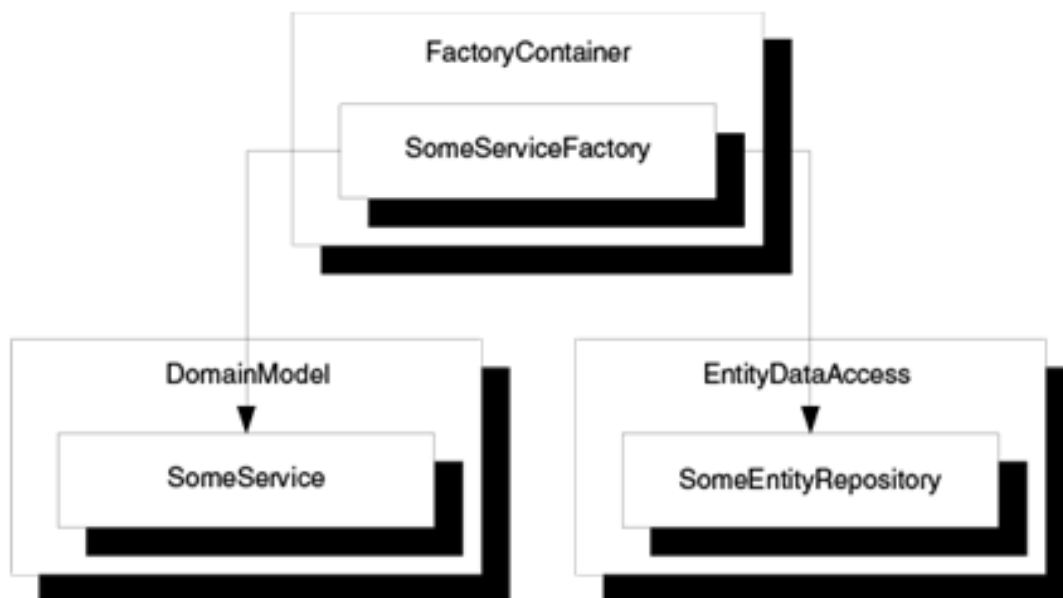
```
public class SomeService : ISomeService
{
    public SomeService(ISomeRepository repository)
    {
    }
}
```

Класс SomeService реализует интерфейс ISomeService, но требует экземпляр ISomeRepository. Поскольку единственный конструктор не является конструктором по умолчанию, пригодится ISomeServiceFactory.

Теперь вы хотите использовать реализацию ISomeRepository, основанную на Entity Framework. Вы называете эту реализацию SomeEntityRepository, и она определена в другой сборке, чем SomeService.

Поскольку вы не хотите перетаскать ссылку в библиотеку EntityDataAccess наряду с SomeService, единственным решением является реализация SomeServiceFactory в другой сборке, чем SomeService, как показано на рисунке 5-6.

Рисунок 5-6: Класс SomeServiceFactory должен быть реализован в отдельной сборке, нежели SomeService, чтобы предотвратить связанность библиотеки DomainModel и библиотеки EntityDataAccess.



Хотя ISomeService и ISomeServiceFactory похожи на сплоченную пару, важно реализовать их в двух различных сборках, так как фабрика должна иметь ссылки на все зависимости, чтобы иметь возможность их правильно связывать.

По соглашению реализация ISomeServiceFactory имеет конструктор по умолчанию, так что вы можете написать имя типа, определенной сборкой, в файле .config и использовать Activator.CreateInstance для создания экземпляра. Каждый раз, когда вам нужно связывать вместе новую комбинацию зависимостей, необходимо реализовать новую ISomeServiceFactory, чтобы провести именно эту комбинацию, а затем настроить приложение для использования этой фабрики вместо предыдущей. Это означает, что вы

не можете определить произвольные комбинации зависимостей без написания и компиляции кода, но вы можете сделать это без перекompиляции самого приложения.

По сути, такая абстрактная фабрика становится абстрактным Composition Root, который определен в сборке отдельно от основного приложения. Хотя это, безусловно, является жизнеспособным подходом, как правило, гораздо легче использовать DI контейнер общего назначения, который может сделать все это для нас сам на основе файлов конфигурации.

Анти-паттерн Constrained Construction применяется действительно только тогда, когда мы используем позднее связывание, потому что когда мы используем раннее связывание, компилятор гарантирует, что мы никогда не введем неявные ограничения в том, как строятся компоненты.

Последний анти-паттерн применяется более часто – некоторые люди даже считают его настоящим паттерном, а не анти-паттерном.

5.4. Service Locator

Это может быть трудным – отказаться от идеи прямого контроля зависимостей, поэтому многие разработчики выводят статические фабрики (как описано в разделе 5.1.2) на новый уровень. Это приводит к анти-паттерну Service Locator.

Внимание

Это довольно спорно – называть Service Locator анти-паттерном. Некоторые люди считают его настоящим паттерном проектирования, тогда как другие (и я в том числе) считают анти-паттерном. В этой книге я решил описать его как анти-паттерн, потому что я думаю, что у него больше недостатков, чем преимуществ, но не удивляйтесь, если вы увидите, что он одобрен в других местах. Важно понять преимущества и недостатки, чтобы иметь возможность принять для себя обоснованное решение.

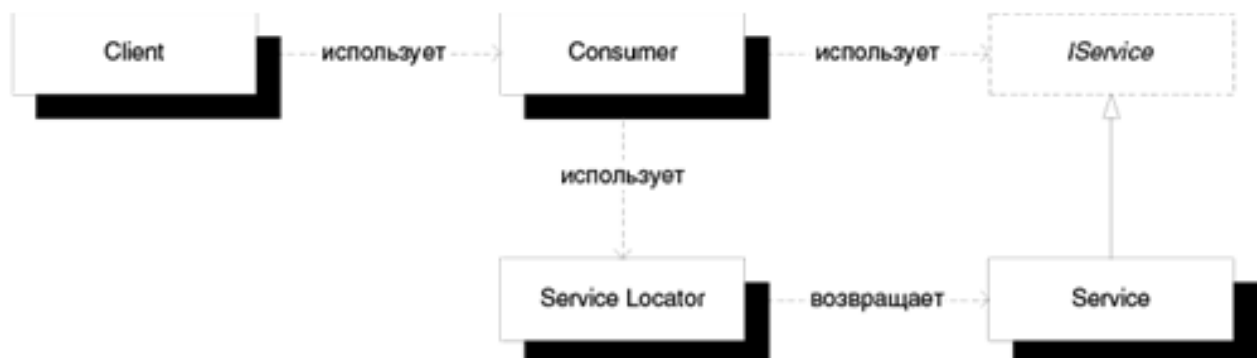
Service Locator был введен в качестве паттерна проектирования Мартином Фаулером в 2004, поэтому обозначение его как анти-паттерн является большим шагом. Короче говоря, он вводит статическую фабрику с дополнительной деталью, что становится возможно внедрить сервисы в эту фабрику.

Примечание

Термин *сервис* в данном контексте приблизительно эквивалентен термину *зависимость*.

Как это чаще всего реализуется, Service Locator является статической фабрикой, которая может быть сконфигурирована с конкретными сервисами, пока первый потребитель не начинает ее использовать (см. рисунок 5-7). Это, вероятно, может произойти в Composition Root. В зависимости от конкретной реализации, Service Locator может быть настроен с кодом, когда читаются файлы конфигурации или используется их комбинация.

Рисунок 5-7: Основная ответственность Service Locator заключается в том, чтобы обрабатывать экземпляры сервисов, когда потребители их запрашивают. Consumer использует интерфейс IService и запрашивает экземпляр от Service Locator, который затем возвращает экземпляр той конкретной реализации, которую он должен вернуть



Основная обязанность Service Locator заключается в том, чтобы обрабатывать экземпляры сервисов, когда потребители их запрашивают. Потребитель использует интерфейс IService и запрашивает экземпляр у Service Locator, который затем возвращает экземпляр любой конкретной реализации, которую он сконфигурирован вернуть.

Моя личная история с Service Locator

У меня с Service Locator были интенсивные отношения в течение нескольких лет, прежде чем мы расстались. Хотя я точно не помню, когда я впервые наткнулся на статью Фаулера, мне показалось, что он предоставил мне потенциальное решение проблемы, которую я обдумывал в течение некоторого времени: как внедрить зависимости.

Как было описано, паттерн Service Locator казался ответом на все мои вопросы, и я быстро начал разрабатывать Service Locator для первой версии *Microsoft patterns & practices' Enterprise Library*. Это было размещено на ныне не существующем сайте GotDotNet. Хотя у меня все еще есть исходный код, я потерял историю релизов, когда GotDotNet закрыли, поэтому я не могу сказать наверняка, но я, кажется, опубликовал первую версию в середине 2005 года.

В 2007 году я выпустил полностью переписанный релиз, ориентированный на Enterprise Library 2. Он по-прежнему доступен на CodePlex, но я давно уже отказался от него, потому что я вскоре пришел к выводу, что это был действительно анти-паттерн.

Как вы можете видеть, у меня ушло несколько лет интенсивного использования, чтобы понять недостатки Service Locator и что существуют лучшие альтернативы. По этой причине я с легкостью понимаю, почему так много разработчиков находят его привлекательным, несмотря на его недостатки. Паттерны, описанные в главе 4, предлагают более совершенные варианты, но вы не поймете недостатки Service Locator, пока не изучите их.

Внимание

Если вы посмотрите только на статическую структуру классов, DI контейнер выглядит как Service Locator. Разница мизерная и заключается не в механике реализации, а в том, как вы ее используете. В сущности, просьба к контейнеру или локатору разрешить полный граф зависимости из Composition Root является правильным использованием. Просьба о "зернистых" сервисах из любого другого места подразумевает анти-паттерн Service Locator.

Давайте рассмотрим пример, где он сконфигурирован с кодом.

Пример: ProductService, использующий Service Locator

Для примера давайте вернемся к нашему проверенному и истинному ProductService, который требует экземпляр абстрактного класса ProductRepository. В данном случае ProductService может использовать статический метод GetService, чтобы получить необходимый экземпляр:

```
this.repository = Locator.GetService<ProductRepository>();
```

В данном примере я реализую методы, используя параметры дженерик типа, чтобы указать тип запрашиваемого сервиса, но я также мог бы использовать экземпляр Type, чтобы указать тип.

Как показано в следующем листинге, эта реализация класса Locator является минималистичной, насколько это возможно. Я мог бы добавить ограждающее условие и обработку ошибок, но я хотел выделить основное поведение. Код может также включать в

себя возможность, которая позволяет загружать его конфигурацию из файла `.config`, но я оставлю это в качестве упражнения для вас.

Листинг 5-5: Минимальная реализация Service Locator

```
1. public static class Locator
2. {
3.     private readonly static Dictionary<Type, object> services
4.         = new Dictionary<Type, object>();
5.     public static T GetService<T>()
6.     {
7.         return (T)Locator.services[typeof(T)];
8.     }
9.     public static void Register<T>(T service)
10.    {
11.        Locator.services[typeof(T)] = service;
12.    }
13.    public static void Reset()
14.    {
15.        Locator.services.Clear();
16.    }
17. }
```

Строки 5-8: Получить сервис

`Locator` — это класс с только статическими членами, так что вы могли бы также явно отметить его как статический класс. Он содержит все настроенные сервисы во внутреннем словаре, который связывает абстрактные типы с конкретными экземплярами.

Клиенты, такие как `ProductService`, могут использовать метод `GetService`, чтобы запросить экземпляр абстрактного типа `T`. Поскольку в этом примере кода не содержится ограждающее условие или обработка ошибок, данный метод сгенерирует довольно туманное `KeyNotFoundException`, если запрашиваемый тип не имеет записи в словаре, но вы можете представить, как добавить код, чтобы выбросить более осмысленное исключение.

Метод `GetService` может возвращать только экземпляр запрошенного типа, если он ранее был вставлен во внутренний словарь. Это можно сделать с помощью метода `Register`. Опять же, этот пример кода не содержит ограждающего условия, так что можно было бы зарегистрировать `null`, но более надежная реализация не должна этого допустить.

В некоторых случаях (особенно при модульном тестировании), важно иметь возможность сбросить `Service Locator`. Эта функциональность обеспечивается методом `Reset`, который очищает внутренний словарь.

Такие классы, как `ProductService` полагаются на сервис, чтобы быть доступными в `Service Locator`, поэтому очень важно, что он был ранее настроен. В модульных тестах это может быть сделано при помощи тестового дублера (поддельного объекта, `TestDouble`), реализованного динамической `mock` библиотекой, такой как `Moq`, как в этом примере:

```
var stub = new Mock<ProductRepository>().Object;
Locator.Register<ProductRepository>(stub);
```

Сначала мы создаем заглушку абстрактного класса `ProductRepository`, а затем с помощью статического метода `Register` настраиваем `Service Locator` с этим экземпляром.

Если это сделано, прежде чем `ProductService` используется в первый раз, `ProductService` будет использовать настроенный `Stub`, чтобы работать с `ProductRepository`. В производственном приложении `Service Locator` будет настроен с правильной реализацией `ProductRepository` в `Composition Root`.

Этот способ обнаружения зависимостей из класса `ProductService` определенно работает, если наш единственный критерий успеха заключается в том, что зависимость может быть использована и заменена по желанию, но это имеет некоторые другие серьезные недостатки.

Анализ

`Service Locator` является опасным паттерном, потому что он *почти* работает. Мы можем обнаружить зависимости из потребляющих классов, и мы можем заменить эти зависимости различными реализациями – даже поддельными объектами из юнит тестов.

Если мы применим модель анализа, изложенную в главе 1, чтобы оценить, соответствует ли `Service Locator` преимуществам модульной конструкции приложения, мы увидим, что в основном он соответствует:

- У нас есть поддержка для *позднего (отложенного) связывания* при смене регистрации.
- Возможна *параллельная разработка*, потому что мы программируем, используя интерфейсы, и можем по желанию заменить модули.
- Мы можем достичь хорошего разделения понятий, поэтому нас ничего не останавливает от написания *поддерживаемого* кода. Но это все сложнее сделать.
- Мы можем заменить зависимости дублирующими элементами, поэтому тестируемость возможна.

Существует только одна область, где `Service Locator` не дотягивает.

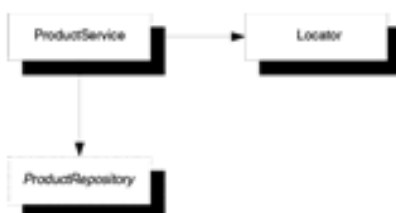
Влияние

Основная проблема с `Service Locator` заключается в том, что он влияет на повторное использование классов, которые его потребляют. Это проявляется в двух направлениях:

- Модуль потянет за собой избыточную зависимость.
- Это не очевидно, что используется DI.

Давайте сначала посмотрим на граф зависимостей для `ProductService` из примера в разделе 5.4.1, который показан на рисунке 5-8. В дополнение к ожидаемой ссылке на абстрактный класс `ProductRepository`, `ProductService` также зависит от класса `Locator`.

Рисунок 5-8: Граф зависимости для реализации `ProductService`, которая использует `Service Locator`, чтобы обработать экземпляры абстрактного класса `ProductRepository`.



Это означает, что для повторного использования класса `ProductService`, мы должны перераспределить не только его релевантную зависимость `ProductRepository`, а также зависимость `Locator`, которая существует только по механическим причинам. Если класс `Locator` определен в другом модуле, чем `ProductService` и `ProductRepository`, новые приложения, которые желают повторно использовать `ProductService`, должны также принять и этот модуль.

Чтобы понять, в чем дело, представьте, что новое приложение, повторно использующее `ProductService`, уже использует другую DI стратегию, сосредотачиваясь на внедрении в конструктор. `ProductService` не вписывается в эту стратегию, но вместо этого навязывает свою собственную стратегию, которая сильно загрязняет DI архитектуру нового приложения. Чтобы это использовать, разработчики должны принять существование `Service Locator`, и после того как он введен, он может быть случайно использован начинающим разработчиком для других целей, где существуют лучшие альтернативы.

Возможно, мы могли даже просто перетерпеть эту дополнительную зависимость для `Locator`, если бы это было действительно необходимо для работы DI – мы бы рассматривали это в качестве налога, подлежащего уплате, чтобы получить другие преимущества. Тем не менее, есть лучшие варианты (например, внедрение в конструктор), так что эта зависимость является избыточной.

Чтобы добавить соли на рану, ни эта избыточная зависимость, ни ее соответствующий дубликат, `ProductRepository`, явно не видны разработчикам, желающим использовать класс `ProductService`. Рисунок 5-9 показывает, что Visual Studio не может предложить никаких рекомендаций по использованию этого класса.

Рисунок 5-9: Единственная вещь, которую нам может сказать IntelliSense о классе `ProductService`, это то, что у него есть конструктор по умолчанию. Его зависимости невидимы.



```
var ps = new ProductService()
```

`ProductService.ProductService ()`

Когда мы хотим создать новый экземпляр класса `ProductService`, Visual Studio может сказать нам только, что класс имеет конструктор по умолчанию. Однако если мы впоследствии попытаемся запустить код, который мы только что написали, мы получим сообщение об ошибке времени выполнения, если мы забыли зарегистрировать экземпляр `ProductRepository` при помощи класса `Locator`. Это может произойти, если мы хорошо не знаем класс `ProductService`.

Совет

Представьте себе, что код, который мы пишем, попадает в незадокументированный, туманный `.dll`. Как легко его может использовать кто-то еще? Это возможно – разработать API, которые близки к самодокументированию, и хотя для этого требуется практика, это достойная цель.

Примечание

Проблема с Service Locator заключается в том, что любой клиент, использующий его, не знает о его уровне сложности. Он выглядит простым, но оказывается сложным — и мы этого не узнаем, пока не получим исключение времени выполнения.

Проблема с классом `ProductService` заключается в том, что он далек от самодокументируемости: мы не можем сказать, какая зависимость должна присутствовать, прежде чем он будет работать. На самом деле, разработчики `ProductService` могут даже решить добавить несколько зависимостей в будущие версии, поэтому код, который работает в текущем варианте, может не сработать в будущей версии, и мы даже не получим ошибку компиляции, которая предупредит нас. С Service Locator легко можно случайно ввести критические изменения.

Внимание

Использование дженериков может ввести вас в заблуждение, что Service Locator строго типизирован. Однако даже API, как в листинге 5-5, слабо типизированный, потому что мы можем запросить любой тип. Возможность компилировать код при помощи вызова метода `GetService<T>` не дает нам никакой гарантии, что он не будет выбрасывать исключения налево и направо во время выполнения.

Примечание

При модульном тестировании у нас есть дополнительная проблема, что тестирующий дублер, зарегистрированный в одном тесте, вызовет взаимозависимые тесты (Interdependent Tests), потому что он останется в памяти, когда будет выполнен следующий тест. Поэтому необходимо использовать методы тестовых фикстур (Fixture Teardown) после каждого теста, вызывая `Locator.Reset()`, и мы должны помнить все время, что это делается вручную, а это легко забыть.

Это все действительно плохо. Service Locator может показаться безобидным, но это может привести ко всяким неприятным ошибкам выполнения. Как избежать этих проблем?

Рефакторинг по направлению к DI

Когда мы решаем избавиться от Service Locator, мы должны найти способ сделать это. Как всегда, главной альтернативой по умолчанию должно быть внедрение в конструктор, если ни один из других DI паттернов из главы 4 не подходит лучше.

Внимание

Когда мы смотрим на структуру Service Locator, она близка к Ambient Context. Оба неявно используют Одиночки (Singletons) но разница заключается в наличии *Local Default*. Ambient Context гарантирует, что он всегда может предоставить соответствующий экземпляр запрошенного сервиса (как правило, имеется только один). А Service Locator не может дать такую гарантию, потому что он, в сущности, является слабо типизированным контейнером сервисов, о которых он не имеет встроенных знаний.

Во многих случаях класс, который потребляет Service Locator, может вызываться по всему коду. В таких случаях он действует в качестве замены для ключевого слова `new`. Если это

так, первый шаг рефакторинга заключается в закреплении создания каждой зависимости в одном методе.

Если мы не будем иметь поле для хранения экземпляра зависимостей, мы можем ввести такое поле и убедитесь, что остальная часть кода использует это поле, когда потребляет зависимость. Отметьте поле как `readonly`, чтобы оно не могло быть изменено за пределами конструктора. Это заставляет нас присваивать значения полю из конструктора при помощи `Service Locator`. Теперь мы можем ввести параметр конструктора, который присваивает значение полю, вместо `Service Locator`, который затем может быть удален. Представляя параметр зависимости конструктору, можно нарушить работу существующих потребителей, поэтому мы также должны справиться с этим и переместить продвижение всех зависимостей в `Composition Root`.

Рефакторинг класса, который использует `Service Locator`, похож на рефакторинг класса, который использует `Control Freak`, потому что `Service Locator` – это просто окольный вариант `Control Freak`. Раздел 5.1.3 содержит дополнительные заметки о рефакторинге реализаций `Control Freak` к использованию `DI`.

На первый взгляд `Service Locator` может выглядеть как настоящий `DI` паттерн, но не обманывайте себя: он может решить вопрос слабой связанности, но он создает другие проблемы на этом пути. `DI` паттерны, представленные в главе 4, предлагают лучшие альтернативы с меньшим количеством недостатков. Это верно как для анти-паттерна `Service Locator`, так и для других анти-паттернов, представленных в этой главе. Даже если они разные, все они имеют общую черту – решить создаваемые ими проблемы можно с помощью одного из `DI` паттернов из главы 4.

5.5. Резюме

Поскольку DI представляет собой набор паттернов и технических приемов, ни один инструмент не может механически проверить, правильно ли мы их применили. В главе 4 мы рассмотрели паттерны, которые описывают, как DI можно использовать должным образом, но это только одна сторона медали. Важно также понимать, где можно наделать ошибок, даже если имеешь лучшие намерения. Есть важные уроки, которые можно извлечь из неудач, но мы не всегда должны учиться на собственных ошибках, иногда мы можем учиться на ошибках других людей.

В этой главе я описал наиболее распространенные DI ошибки в виде анти-паттернов. Я видел все эти ошибки в реальной жизни более чем один раз, и я признаю себя виновным по всем этим пунктам:

- Меня зовут Марк Симан, и я использовал Control Freak.
- Меня зовут Марк Симан, и я использовал Bastard Injection.
- Меня зовут Марк Симан, и я использовал Constrained Construction.
- Меня зовут Марк Симан, и я использовал Service Locator.

К счастью, я давно избавился от этих привычек. Я чист уже много лет.

Первая и самая опасная привычка, от которой стоит избавиться, это мнимая необходимости иметь прямой контроль над зависимостями. Это легко – обнаружить Control Freak: каждое место, когда вы используете ключевое слово `new` (в C#, по крайней мере), чтобы создать экземпляр изменчивой зависимости, вы являетесь *Control Freak*, и не имеет значения, сколько слоев фабрик вы используете, чтобы скрыть этот факт. Единственное место, где можно использовать ключевое слово `new` для зависимостей, это Composition Root.

Избавиться от Control Freak – это наиболее важная задача на сегодняшний день. Только тогда, когда вам удалось отсеять экземпляры Control Freak, вам стоит обратить внимание на другие анти-паттерны: они гораздо менее разрушительные.

Совет

Control Freak мешает вам использовать слабую связанность, другие же DI анти-паттерны просто делают использование неудобным, поэтому я сразу акцентирую ваше внимание на Control Freak.

Bastard Injection разрешает DI, но потом портит партию, увлекая за собой избыточные зависимости. К счастью, реализацию Bastard Injection легко изменить при помощи внедрения в конструктор, поэтому никакой необходимости существовать с Bastard Injection нет. Мы получаем больше, чем теряем, при переходе к правильному решению: на самом деле, мы только теряем время, необходимое для выполнения рефакторинга.

Constrained Construction накладывает искусственные ограничения на типы, которые мы используем для реализации абстракций. В большинстве случаев это принимает форму ограничения всех реализаций, чтобы иметь конструктор по умолчанию, но в некоторых случаях конструкторы должны принять определенные параметры для инициализации компонента.

Вы должны снять эти ограничения и использовать DI контейнер или вручную внедрить в объекты необходимые зависимости, где бы они не были. Если у вас есть сценарий, где необходимо инициализировать некоторые компоненты при помощи информации о текущем контексте, нужно применять внедрение в метод.

Service Locator может выглядеть привлекательно, но я считаю его анти-паттерном, хотя это несколько спорное мнение. Несмотря на то, что он решает некоторые проблемы DI, он вводит другие проблемы, которые перевешивают его преимущества. Нет никаких причин принимать его недостатки, поскольку DI паттерны, представленные в главе 4, предлагают лучшие альтернативы. Это общая тема для всех анти-паттернов, описанных в этой главе: DI паттерны из главы 4 предлагают решения проблем, вызванных анти-паттернами.

Сейчас уже вы должны знать, чего следует избегать и что вы в идеале должны делать вместо этого, но все же есть проблемы, которые выглядят так, как будто их трудно решить. В следующей главе обсуждаются такие проблемы и их решения.

6. DI рефакторинг

Меню

- Преобразование динамических значений в абстракции
- Работа с недолговечными зависимостями
- Разрешение циклических зависимостей
- Работа с Constructor Over-injection
- Мониторинг связывания

Вы могли заметить, что мне нравится *sauce béarnaise* или *sauce hollandaise* в целом. Одной из причин является то, что они очень хороши на вкус, а другая заключается в том, что их довольно сложно сделать. В дополнение к проблеме приготовления *sauce hollandaise* дает нам еще одно испытание: он должен быть подан немедленно (или мне так кажется).

Раньше это было далеко от идеала, когда у меня были гости. Вместо того, чтобы приветствовать своих гостей и заставить их чувствовать себя желанными и расслабленными, я отчаянно взбивал соус на кухне, оставив их развлекать самих себя.

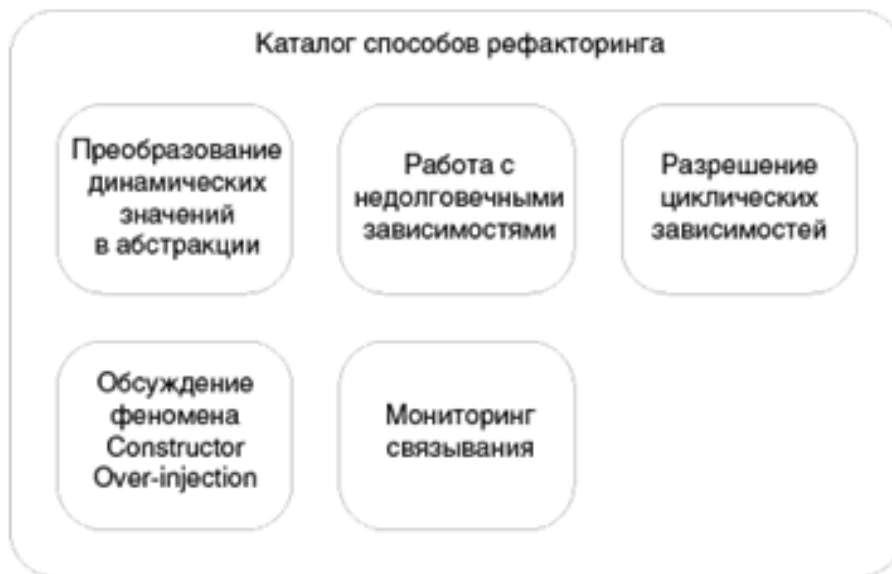
После нескольких повторных выступлений, моя очень общительная жена решила взять дело в свои руки. Мы живем через дорогу от ресторана, так что в один прекрасный день она разговорила поваров, чтобы выяснить секрет, который позволил бы мне приготовить подлинный голландский соус заблаговременно. Оказывается, что такой секрет есть, так что теперь я могу сделать вкусный соус для моих гостей, не подвергая их в атмосферу стресса и безумия.

Каждый имеет свои хитрости. Это также верно для разработки программного обеспечения в целом и, в частности, DI. Есть проблемы, которые просто продолжают появляться, и во многих случаях существуют хорошо известные способы борьбы с ними.

На протяжении многих лет я видел, как люди прилагают все усилия для изучения DI, и мне пришлось в голову, что многие из вопросов похожи по своей структуре. В этой главе мы рассмотрим наиболее распространенные проблемы, которые появляются, когда мы применяем DI к коду, и как мы можем решить их. Когда мы закончим, вы должны быть в состоянии лучше распознавать и решать такие проблемы, когда они появляются.

Как и две предыдущие главы этой части книги, эта глава организована в виде каталога — на этот раз проблем и решений (или, если угодно, рефакторинга). Рисунок 6-1 показывает структуру главы.

Рисунок 6-1: Структура этой главы представляет собой каталог способов рефакторинга и решений обычных DI проблем. Каждый раздел является независимым.



В каждом разделе я представляю общую проблему и ее решение, в том числе на примере. Вы можете прочитать каждый раздел самостоятельно или в последовательности, как вам удобнее. Цель каждого раздела заключается в ознакомлении с решением часто встречающихся проблем, в том, что вы будете лучше оснащены для борьбы с ними, если вы с ними столкнетесь.

[6.1. Преобразование динамических значений в абстракции](#)

[6.2. Работа с недолговечными зависимостями](#)

[6.3. Разрешение циклических зависимостей](#)

[6.4. Обсуждение феномена Constructor Over-injection](#)

[6.5. Мониторинг связывания](#)

[6.6. Резюме](#)

6.1. Преобразование динамических значений в абстракции

Когда вы начинаете применять DI, одна из первых трудностей, с которой вы, вероятно, столкнетесь, заключается в том, что абстракции зависят от значений времени выполнения. Например, сайт с онлайн картой может предложить рассчитать маршрут между двумя точками. Это может дать вам выбор по расчету маршрута: вы хотите кратчайший путь? Самый быстрый маршрут на основе имеющихся видов транспорта? Самый живописный маршрут?

Каждый вариант представляет собой отдельный алгоритм, и приложение может обрабатывать каждый алгоритм как абстракцию, поэтому оно обрабатывает алгоритмы одинаково. Для расчета маршрута приложению нужен алгоритм маршрута, и это не важно, какой именно. Мы должны сказать ему, какой алгоритм он должен использовать, но мы этого не узнаем до времени выполнения, потому что он основан на выборе пользователя.

В этом разделе рассказывается, как мы можем решить такого рода проблему. Прежде чем перейти к примеру, мы кратко поговорим об общей проблеме. Когда мы закончим, вашей рефлекторной реакцией на этот вызов должен быть ввод абстрактной фабрики.

Абстракции с зависимостями времени выполнения

Когда мы используем внедрение в конструктор, мы неявно утверждаем — мы ожидаем, что зависимость должна быть *однозначной* во время выполнения. Рассмотрим сигнатуру конструктора, как эта:

Рисунок 6-1a:

```
public RepositoryBasketDiscountPolicy(DiscountRepository repository)
```

Двусмысленность не допустима!

Это никогда не будет работать, если *во время выполнения* не ясно, какая реализация `DiscountRepository` должна быть использована. Во время разработки мы можем рассматривать зависимость как абстракцию и следовать Принципу подстановки Барбары Лисков (Liskov Substitution Principle), но во время выполнения решение о том, какой `DiscountRepository` использовать, *должно* быть принято до того, как будет создан `RepositoryBasketDiscountPolicy`. Поскольку зависимость запрашивается через конструктор, мы не можем принять решения после этого момента.

Это лишь означает, что как только класс `RepositoryBasketDiscountPolicy` идет в работу, не может быть никакой двусмысленности относительно `DiscountRepository`. Прочие потребители также могут запрашивать экземпляры `DiscountRepository`, и будут ли они все получать одинаковые или разные экземпляры, имеет меньшее значение. Такие зависимости часто представляют сервисы, а не доменные объекты. Концептуально, есть только один экземпляр данного сервиса.

Примечание

Как вы увидите в главе 9, может быть несколько реализаций одной и той же абстракции в работе в одно и то же время. Однако, с точки зрения потребителя, есть только одна.

Сервисы принадлежат к общей группе зависимостей, но иногда зависимость представляет надлежащий доменный объект. Это особенно верно, когда речь идет о меняющемся поведении абстракциях, такие как Стратегии (Strategies). Предыдущий алгоритм расчета маршрута является одним из таких примеров. Другим может быть коллекция графических редакторов для растровых эффектов: каждый эффект выполняет преобразование растрового изображения, но все они могут быть раскрыты в приложении как абстракции – это также архитектура, позволяющая поддерживать надстройки.

В таких случаях мы не можем запросить зависимость через конструктор, потому что *Composer* не будет знать, какую реализацию выбрать. В игре может быть ноль, один или много экземпляров в разное время жизненного цикла приложения. Зависимость неоднозначна во время разработки.

Как всегда, в разработке программного обеспечения, решением является окольный путь: на этот раз, абстрактная фабрика.

Абстрактная фабрика

Паттерн проектирования *абстрактная фабрика* (Abstract Factory) решает проблему, когда мы по желанию должны быть в состоянии запросить экземпляр абстракции. Он предлагает мост между абстракций и конкретными значениями времени выполнения, что позволяет нам переводить значение времени выполнения в зависимость.

На следующем рисунке показано, как это работает, то есть, вводится новая абстракция, которая создает экземпляры первоначально требуемой абстракции.

Рисунок: Если мы должны быть в состоянии создать экземпляры `IFoo` по запросу, нам нужен способ сделать это. Абстрактная фабрика – это другая абстракция, которую мы можем использовать для создания таких экземпляров по мере необходимости.



Абстрактная фабрика – сама абстракция, единственной целью которой является создание экземпляров первоначально требуемых абстракции. Если мы должны быть в состоянии создать экземпляры `IFoo` из конкретных экземпляров `Bar`, соответствующая абстрактная фабрика может выглядеть следующим образом:

```
public interface IFooFactory
{
    IFoo Create(Bar bar);
}
```

В ухудшенном варианте абстрактная фабрика может не принимать никаких входных параметров:

```
public interface IFooFactory
{
    IFoo Create();
}
```

В таком случае абстрактная фабрика становится чистой фабрикой, в то время как аспект преобразования исчезает.

Абстрактная фабрика является одним из наиболее полезных паттернов проектирования. Имейте это в виду, потому что она может быть использована для решения многих вопросов, связанных с DI.

Совет

Когда один или несколько параметров, переданных абстрактной фабрике, сами по себе являются абстракциями, эта техника также становится примером внедрения в метод.

Абстрактная фабрика является универсальным решением, когда мы должны создать зависимости из значения времени выполнения.

Проектные требования

Насколько полезной может быть абстрактная фабрика, настолько осторожно мы должны применять ее. Зависимости, созданные абстрактной фабрикой, должны *концептуально* требовать значению времени выполнения. Переход от значения времени выполнения в абстракцию должен иметь смысл на концептуальном уровне. Если вы чувствуете желание ввести абстрактную фабрику, чтобы иметь возможность создавать экземпляры конкретной реализации, стоит воспользоваться протекающей абстракцией (Leaky Abstraction).

Протекающие абстракции

Также, как разработка через тестирование (Test-Driven Development, TDD) обеспечивает тестируемость, безопаснее сначала определить интерфейсы, а затем дальнейшую программу для них. Тем не менее, бывают случаи, когда у нас уже есть конкретный тип и теперь мы хотим извлечь интерфейс.

Когда мы это делаем, мы должны позаботиться о том, чтобы лежащая в основе реализация не протекала. Один из моментов, когда это может случиться, если мы только извлекаем интерфейс из данного конкретного типа, но все параметры и возвращаемые типы по-прежнему являются конкретными типами, определенными в той же библиотеке.

Если нам нужно извлечь интерфейс, мы должны делать это рекурсивным образом, гарантируя, что все типы, извлекаемые корневым интерфейсом, сами являются интерфейсами. Я называю это глубоким извлечением (Deep Extraction), а результат – *глубокими интерфейсами* (Deep Interfaces).

В ASP.NET MVC есть некоторые примеры извлечения глубоких интерфейсов. Например, у `HttpContextBase` есть свойство `Request` типа `HttpRequestBase`, и так далее. Эта абстракция была рекурсивно извлечена из `System.Web.HttpContext`.

Всегда учитывайте, имеет ли данная абстракция смысл для других реализаций, нежели той, которую вы имеете в виду. Если это не так, вы должны пересмотреть вашу структуру.

Абстрактные фабрики бывают разных форм и обликов, и не всегда может быть очевидным, что они у вас есть.

Примечание

Любая абстракция, которая создает экземпляры других абстракций, является фабрикой абстракций (абстрактной фабрикой). И для этого не нужно иметь имя, которое заканчивается на *Factory*.

Давайте посмотрим на пару примеров: сначала простой, идиоматический пример, а впоследствии более сложный пример, где абстрактная фабрика скрыта под другим именем.

Пример: выбор алгоритма маршрута

Во введении к этому разделу кратко обсуждался сайт с онлайн картой, где пользователь может выбрать различные алгоритмы расчета маршрута. В этом разделе мы рассмотрим то, как применить абстрактную фабрику, чтобы удовлетворить этому требованию

В веб-приложениях вы можете только передать примитивные типы от браузера на сервер, поэтому когда пользователь выбирает алгоритм маршрута из выпадающего списка, вы должны представить это как число или строку. Перечисление – это на самом деле просто число, так что на сервере можно представить выбор с помощью `RouteType`:

```
public enum RouteType
{
    Shortest = 0,
    Fastest,
    Scenic
}
```

Однако то, что вам нужно, это экземпляр `IRouteAlgorithm`, который может рассчитать маршрут для вас. Для перехода от значения времени выполнения `RouteType` к `IRouteAlgorithm`, вы можете определить абстрактную фабрику:

```
public interface IRouteAlgorithmFactory
{
    IRouteAlgorithm CreateAlgorithm(RouteType routeType);
}
```

Это позволяет реализовать метод `GetRoute` для `RouteController` путем внедрения `IRouteAlgorithmFactory` и использовать это для перевода значения времени выполнения в нужную зависимость: `IRouteAlgorithm`. Следующий листинг демонстрирует взаимодействие.

Листинг 6-1: Использование `IRouteAlgorithmFactory`

```
1. public class RouteController
2. {
3.     private readonly IRouteAlgorithmFactory factory;
4.     public RouteController(IRouteAlgorithmFactory factory)
5.     {
6.         if (factory == null)
7.         {
8.             throw new ArgumentNullException("factory");
```

```

9.         }
10.        this.factory = factory;
11.    }
12.    public IRoute GetRoute(RouteSpecification spec,
13.        RouteType routeType)
14.    {
15.        IRouteAlgorithm algorithm =
16.            this.factory.CreateAlgorithm(routeType);
17.        return algorithm.CalculateRoute(spec);
18.    }
19. }

```

Строки 15-16: Преобразование значения времени выполнения

Строка 17: Использовать преобразованный алгоритм

Ответственность класса `RouteController` заключается в обработке веб запросов. Метод `GetRoute` получает спецификацию пользователя о пунктах отправления и назначения, а также выбранный тип маршрута (через `RouteType`). Вам нужна абстрактная фабрика для преобразования значения времени выполнения `RouteType` в экземпляр `IRouteAlgorithm`, поэтому вы запрашиваете экземпляр `IRouteAlgorithmFactory`, используя стандартное внедрение в конструктор.

В методе `GetRoute` вы можете использовать `factory` для преобразования переменной `routeType` в `IRouteAlgorithm`. Когда это будет сделано, вы можете использовать это для расчета маршрута и возвращения результата.

Примечание

Для краткости я опустил ограждающее условие в методе `GetRoute`. Тем не менее, предоставляемый `RouteSpecification` может быть `null`, поэтому в более совершенной реализации нужно сделать на это проверку.

Наиболее очевидная реализация `IRouteAlgorithmFactory` будет включать простой оператор `switch` и возвращать три различные реализации `IRouteAlgorithm` на основе входных данных. Тем не менее, я оставляю это как упражнение для читателя.

Этот пример продемонстрировал переход значения времени выполнения к зависимости с использованием абстрактной фабрики в чистом виде. Следующий пример показывает более сложный вариант, в котором с первого взгляда вы можете даже и не понять, что используется абстрактная фабрика.

Пример: использование `CurrencyProvider`

В большей части главы 4 вы видели, как реализовать конвертацию валюты в контроллере ASP.NET MVC. Тип `Currency` является абстрактным классом, воспроизведенным здесь, чтобы вам не пришлось возвращаться в раздел 4.1.4:

```

public abstract partial class Currency
{
    public abstract string Code { get; }
    public abstract decimal GetExchangeRateFor(string currencyCode);
}

```

На первый взгляд кажется немного странным, обрабатывать такое понятие, как валюта, в качестве абстракции, потому что оно звучит скорее как *Объект-значение* (Value Object). Тем не менее, обратите внимание, что метод `GetExchangeRateFor` позволяет нам запросить его для практически неограниченного множества конверсий. Предположим, есть 100 курсов обмена, каждый экземпляр `Currency` будет потреблять больше, чем 2 КБ памяти. Это вроде бы и не так много, но, возможно, потребуется оптимизация, например, использование паттерна проектирования *Приспособленец* (Flyweight).

Другой вопрос, который сразу возникает при конвертации валюты, касается денег (sic!) валюты другими словами, ее актуальности. Такие приложения, как трейдерское программное обеспечение для монетарных рынков, требуют того, чтобы курсы валют обновлялись несколько раз в секунду, в то время как международные коммерческие сайты, скорее всего, обойдутся несколькими обновлениями для стабильной валюты. Такие приложения могут также включать разметку или стратегии округления, добавляя потенциальную сложность в реализацию типа `Currency`. В свете этого, абстрактный класс `Currency` кажется вполне разумным.

Когда потребителю, например, ASP.NET MVC контроллеру, необходимо преобразовать цены, он затребует `Currency` как зависимость для выполнения конвертации. В примере коммерческого приложения, используемого в этой книге, у класса `Money`, который используется для представления цен, есть этот метод конверсии:

```
public Money ConvertTo(Currency currency)
```

Потребители, такие как контроллер, могут предоставить экземпляр `Currency` всем ценам, чтобы конвертировать их, но теперь возникает вопрос, какой экземпляр `Currency`?

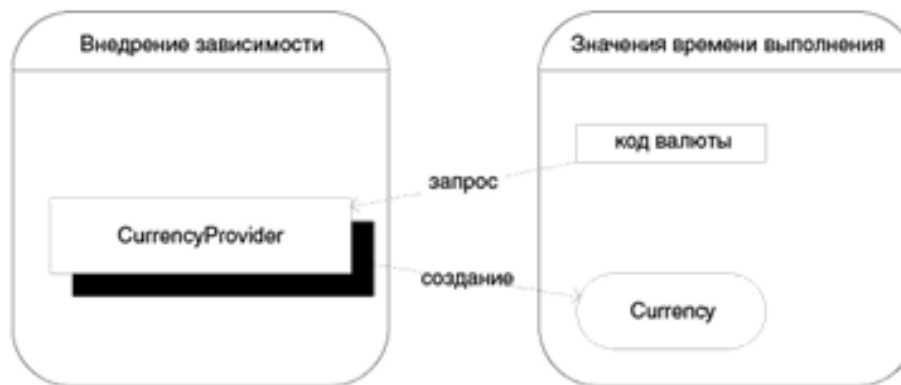
Выбор целевого `Currency` зависит от значения времени выполнения: выбранной пользователем валюты. Это означает, что мы не можем запросить один объект `Currency` через внедрение в конструктор, потому что *Composer* не сможет узнать, какую валюту использовать.

Как вы видели в разделе 4.1.1, решением является внедрение `CurrencyProvider` вместо одного `Currency`:

```
public abstract class CurrencyProvider
{
    public abstract Currency GetCurrency(string currencyCode);
}
```

Рисунок 6-2 показывает, как контроллер обычно извлекает код предпочтительной валюты пользователя из профиля и использует внедренный `CurrencyProvider` для создания соответствующего экземпляра `Currency`.

Рисунок 6-2: Внедренный `CurrencyProvider` используется для отображения простого значения времени выполнения (строку кода валюты) в зависимости времени выполнения (экземпляр `Currency`).



Хотя он имеет другое имя, `CurrencyProvider` является абстрактной фабрикой, которая помогает нам преодолеть разрыв между значением времени выполнения и зависимостью времени выполнения. `Currency` концептуально зависит от кода валюты, поэтому мы можем быть уверены, что мы не внедрили протекающую абстракцию путем введения `CurrencyProvider`.

Еще один пример из главы 4 показывает дегенеративный случай, где нет никаких первоначальных входных параметров. В разделе 4.2.4 вы видели, как у абстрактного `CurrencyProfileService` есть метод `GetCurrencyCode`, который будет возвращать текущий валютный код пользователя:

```
public abstract string GetCurrencyCode();
```

Хотя метод `GetCurrencyCode` возвращает строку, а не абстракцию, вы можете рассматривать `CurrencyProfileService` как вариант абстрактной фабрики.

В `HomeController` вы объединяете оба варианта, чтобы выяснить предпочтительную валюту пользователя:

```
var currencyCode = this.CurrencyProfileService.GetCurrencyCode();
var currency = this.currencyProvider.GetCurrency(currencyCode);
```

И в `CurrencyProfileService`, и в `currencyProvider` внедряются абстрактные фабрики, которые доступны для любого члена класса `HomeController`. В разделах 4.1.4 и 4.2.4 показано, как они внедряются.

Всякий раз, когда мы должны подготовить значение времени выполнения, и мы хотим иметь возможность варьировать средствами, с помощью которых мы создаем это значение независимо от потребителя, мы можем внедрить абстрактную фабрику. Как правило, это *stateless* (не сохраняющий состояние) сервис, поэтому он подходит лучше к тому способу, как мы обычно обрабатываем зависимости, и мы можем использовать внедрение в конструктор или внедрение в свойство, чтобы предоставить потребителю фабрику.

Существует еще один тип сценариев, при котором абстрактная фабрика также обеспечивает хорошее решение. Это происходит, если мы должны работать с недолговечными зависимостями.

6.2. Работа с недолговечными зависимостями

Некоторые зависимости кажутся концептуально недолговечными. Они обычно представляют соединения с внешними ресурсами, такими как базы данных или веб-сервисы. Такие связи должны быть закрытыми или произойдут утечки ресурсов. В этом разделе мы рассмотрим лучший способ решения таких проблем.

Как и в предыдущем разделе, мы начнем с изучения общего случая, а затем приступим к примеру. Когда мы закончим, вы должны понимать две вещи:

- Вы можете моделировать такие взаимодействия при помощи абстрактной фабрики, которая создает одноразовые экземпляры.
- Вы должны стремиться скрыть этот паттерн за *stateless* абстракцией.

Прежде чем перейти к примеру, давайте посмотрим, что заставило меня сказать это.

Закрывание соединений через абстракции

Смысл слабой связанности и принцип подстановки Барбары Лисков состоит в том, что зависимость может быть реализована любым количеством способов. Даже тогда, когда вы имеете в виду конкретную реализацию, потенциально в будущем на ум может прийти радикально отличная реализация.

Тем не менее, некоторые зависимости представляют доступ к внешним ресурсам, а эти, как правило, переходят на вопросы, связанные с использованием ресурсов. Я, конечно, говорю о соединениях в различных видах и формах.

Большинство .NET разработчиков знают, что они должны открыть соединение ADO.NET прямо перед его использованием и закрыть его снова, как только работа будет закончена. Современные API интерфейсы, как LINQ to SQL или LINQ to Entities, автоматически сделают это для нас так, поэтому мы не должны работать с этим напрямую.

Хотя любой программист должен знать о правильно используемом паттерне, касающемся ADO.NET соединений, гораздо менее известно, что это же самое верно для WCF клиентов. Они должны быть закрыты, как только мы закончим с определенным набором операций или сервисов, потому что в противном случае они могут оставить «мертвые» ресурсы на стороне сервера.

WCF сервисы и состояние

Фундаментальное правило сервисной ориентации заключается в том, что сервисы не должны сохранять состояние (должны быть *stateless*). Если мы будем следовать этому правилу, тогда, безусловно, WCF клиент не оставит «живые» ресурсы на стороне сервера, так?

Удивительно, но это не может быть не так. Даже если мы построим сервис полностью *stateless*, WCF может таким не быть. Это зависит от связывания.

Один из многих примеров относится к безопасности. Основанная на сообщениях безопасность, как правило, влияет на производительность. Это верно, потому что асимметричные ключи требуют большого объема вычислений, но это еще более верно для *Federated security*, потому что несколько обменов сообщениями участвуют в создании контекста безопасности. Поведением по умолчанию для WCF является создание защищенного диалога на основе обмена асимметричными ключами. Сервис и клиент используют «рукопожатие» асимметричной безопасности для обмена специальным симметричным ключом, который используется для обеспечения безопасности всех последующих сообщений, которые являются частью этой сессии.

Однако такое поведение требует, чтобы обе стороны сохранили общий секрет в памяти. Клиент должен распрощаться с сервисом, когда он закончит сессию, или это будет «мертвый» симметричный ключ на сервере. Это, в конечном счете, может быть очищено после тайм-аута, но до тех пор это все занимает память. Чтобы сохранить ресурсы на сервере, клиент должен явно закрыть "соединение", когда оно будет завершено.

Хотя это верно не для всех WCF связываний, но таких много, так что мы должны гарантировать, что наши клиенты WCF являются «хорошими ребятами».

Как мы можем совместить необходимость закрыть WCF соединение с желанием избежать протекающей абстракции? Этот вопрос может быть рассмотрен на двух уровнях:

- Скрытие всей логики управления соединениями за абстракцией
- Подражание открытию и закрытию соединений на более детальном уровне

Первый вариант предпочтительнее, но иногда также требуется и второй. Оба варианта могут быть объединены, чтобы получить лучшее от обоих.

Скрытие управления соединениями за абстракцией

DI не является оправданием для написания приложений с утечками памяти, так что мы должны иметь возможность явно закрывать соединения как можно скорее. С другой стороны, любая зависимость может представлять или не представлять связь «вне процесса», так что у нас была бы протекающая абстракция, если бы мы должны были смоделировать абстракцию, чтобы она включала метод `Close`.

Некоторые люди прибегают к тому, что разрешают зависимостям быть наследованными от `IDisposable`. Тем не менее, метод `Dispose` – это просто метод `Close` с другим именем, так что такой подход не решит основной проблемы.

К счастью, технологии доступа к базам данных, такие как LINQ to SQL и LINQ to Entities демонстрируют правильный подход. В обоих случаях мы имеем доступ к данным через контекст, который содержит соединение. Всякий раз, когда мы общаемся с базой данных, контекст автоматически открывает и закрывает соединение по мере необходимости, полностью освобождая нас от бремени борьбы с этим.

Наша первая реакция должна заключаться в том, чтобы сделать то же самое. Рисунок 6-3 показывает, как определить абстракцию на уровне, который достаточно «крупнозернист», чтобы реализация могла открывать и закрывать соединения по мере необходимости.

Рисунок 6-3: Мы можем разработать интерфейс, который достаточно «крупнозернист», чтобы каждый метод включал в себя все взаимодействия с внешним ресурсом в одном пакете. *Consumer* вызывает метод для интерфейса *IResource*. Реализация этого метода может открыть соединение и вызывать несколько методов по отношению ко внешним ресурсам до закрытия соединения и возвращения результата потребителю.



Потребитель никогда не знает, что некоторые реализации могут открывать и закрывать соединения от его имени.

Когда это возможно, мы должны стремиться к разработке пользовательской зависимости, так чтобы нам никогда явно не нужно было иметь дело с жизненным циклом зависимости на этом уровне. Есть, однако, случаи, когда мы не можем этого сделать.

Открытие и закрытие зависимостей

Проблема с «крупнозернистыми» API заключается в том, что они не могут быть достаточно гибкими. Иногда нужна просто абстракция, которая позволяет нам явно моделировать жизненный цикл зависимости, которая в противном случае приведет к утечке памяти.

Внимание

Остановка одной утечки приводит к другой. Мы меняем утечки памяти на "дырявые" абстракции (Leaky Abstractions).

Наиболее распространенный жизненный цикл, который нам нужно смоделировать, показан на фигуре 6-4.

Рисунок 6-4: Наиболее распространенный жизненный цикл соединения заключается в том, что мы создаем, используем и закрываем его, когда заканчиваем работу с ним. Это жизненный цикл, который мы должны смоделировать, если мы должны моделировать такие вещи.



В разделе 6-1 было показано, как использовать абстрактную фабрику для создания зависимости по желанию, так что мы должны найти идиому кодирования, которая подходит к закрытию соединения. Как показано на рисунке 6-4, мы можем использовать паттерн `IDisposable`, чтобы работать с зависимостями, использующими соединения.

Внимание

С одноразовыми зависимостями код «пахнет». Используйте их только тогда, когда нет другого выбора. Подробнее об этом в разделе 8.2.

Другими словами, мы можем смоделировать почти любое взаимодействие, которое соответствует жизненному циклу из рисунка 6-4, при помощи абстрактной фабрики, которая создает одноразовые зависимости (см. рисунок 6-5).

Рисунок 6-5: Мы можем смоделировать управление соединением и аналогичные жизненные циклы, принимая зависимость от абстрактной фабрики, такой как `IFooFactory`, показанной здесь. Каждый раз, когда потребителю нужен экземпляр `IFoo`, он создается `IFooFactory`, но потребитель должен помнить, что должен избавиться от него соответствующим образом.



Паттерн, показанный на рисунке 6-5, часто лучше всего реализовать с помощью ключевого слова `C# using` (или аналогичной конструкции в других языках).

Как покажет следующий пример, имеет смысл объединять оба подхода, которые мы только что обсуждали. Доступ к ресурсам моделируется как «крупнозернистая» абстракция, которая защищает потребителя от непосредственной работы с управлением жизненным циклом, в то время как реализация использует описанное сочетание абстрактной фабрики и одноразовых зависимостей. Давайте посмотрим, как это работает.

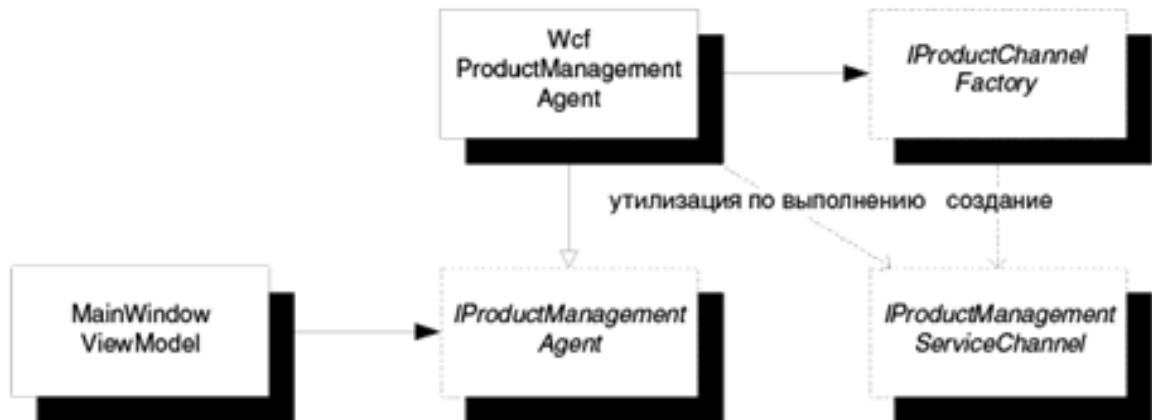
Пример: вызов сервиса управления продуктом

Представьте себе Windows Presentation Foundation (WPF), который обеспечивает богатый пользовательский интерфейс для управления каталогом продукции. Такое приложение может общаться с бэкэндом (сервером) через WCF сервис, который предоставляет необходимые операции по управлению каталогом продукции.

На рисунке 6-6 показано, как реализация сочетает в себе обе техники из предыдущего раздела.

Рисунок 6-6: Класс MainWindowViewModel потребляет интерфейс

`IProductManagementAgent`. Это крупнозернистый интерфейс, который предоставляет соответствующие методы для потребителя. С точки зрения `MainWindowViewModel`, нет никакого управления соединением. Когда приложение запущено, класс `WcfProductManagementAgent` обеспечивает реализацию крупнозернистого интерфейса. Он делает это, потребляя абстрактную фабрику `IProductChannelFactory`, которая создает одноразовые экземпляры. Интерфейс `IProductManagementServiceChannel` наследуется от `IDisposable`, что позволяет `WcfProductManagementAgent` избавиться от WCF клиента, когда операции были успешно введены.



Примечание

Мы вернемся к этому WPF приложению в разделах 6.3.2 и 7.4.2.

Потребитель защищен от управления соединением, которое является частью реализации `WcfProductManagementAgent`.

Всякий раз, когда класс `MainWindowViewModel` хочет вызвать сервисную операцию, он вызывает зависимость `IProductManagementAgent`. Это совершенно нормальная зависимость, внедренная через конструктор. Это, например, показывает, как удалить продукт:

```
this.agent.DeleteProduct(productId);
```

В этом случае `this.agent` является внедренной зависимостью `IProductManagementAgent`. Как видите, здесь нет никакого явного управления соединением, но если вы посмотрите на реализацию в `WcfProductManagementAgent`, вы увидите, как абстрактная фабрика используется в комбинации с одноразовой зависимостью:

```
using (var channel = this.factory.CreateChannel())
{
    channel.DeleteProduct(productId);
}
```

У вас нет внедренного WCF клиента, который можно использовать для вызова сервисной операции, потому что вы должны закрыть клиент, как только вы с ним закончите, и это не возможно – повторно использовать WCF каналы. Вместо этого у вас есть внедренная абстрактная фабрика, которую вы используете, чтобы создать новый канал. Поскольку операция заключается в `using`, выход из зоны видимости отключает канал.

Зависимость `factory` является экземпляром интерфейса `IProductChannelFactory`. Это пользовательский интерфейс, созданный по данному случаю:

```
public interface IProductChannelFactory
{
    IProductManagementServiceChannel CreateChannel();
}
```

Тем не менее, интерфейс `IProductManagementServiceChannel` является автоматически сгенерированным интерфейсом, созданным вместе со всеми другими типами WCF прокси. Каждый раз, когда мы создаем ссылку на сервис в Visual Studio или используем `svcutil.exe`, такой интерфейс создается вместе с другими типами. Привлекательной особенностью этого автоматически сгенерированного интерфейса является то, что он реализует `IDisposable` вместе со всеми сервисными операциями.

WCF понимает этот тип, что делает реализацию `IProductChannelFactory` тривиальной, поскольку мы можем использовать `System.ServiceModel.ChannelFactory<TChannel>` для создания экземпляров.

Как доминирующий принцип, я предпочитаю не сохраняющие состояние и крупнозернистые интерфейсы, как `IProductManagementAgent`, чтобы оградить пользователей от деталей реализации. Хотя мы должны рассматривать одноразовые зависимости как протекающие абстракции, утечка может содержаться в конкретной реализации; и, делая это, мы получаем тестируемость без ущерба для общей структуры.

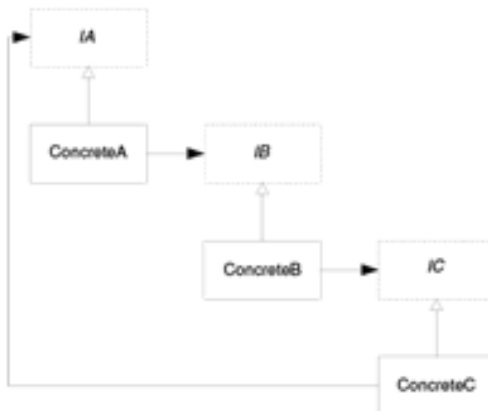
Абстрактная фабрика является чрезвычайно полезным шаблоном проектирования. Он помогает решать проблемы с зависимостями времени выполнения и недолговечными зависимостями. Мы также можем включить его в попытку решить проблемы с циклическими зависимостями, но это не играет центральную роль в данном контексте.

6.3. Разрешение циклических зависимостей

Иногда реализации зависимости оказываются циклическими. Одна реализация требует еще одну зависимость, реализация которой требует первую абстракцию. Такой граф зависимостей не может быть удовлетворительным.

Важно понимать, что абстракции сами по себе могут быть совершенно нециклическими, а конкретная реализация может ввести цикл. На рисунке 6-7 показано, как это может произойти.

Рисунок 6-7: Циклы в графе зависимостей могут произойти даже тогда, когда абстракции не имеют отношения друг к другу. В этом примере каждая реализация реализует отдельный интерфейс, но также требует зависимость. Поскольку ConcreteC требует IA, но единственной реализацией IA является ConcreteA со своей зависимостью для IB и так далее, то есть у нас есть цикл, который не может быть разрешен, как есть.



Пока цикл остается, мы не можем удовлетворить все зависимости, и наши приложения не будут иметь возможность запускаться. Ясно, что надо что-то делать, но что?

В этом разделе мы рассмотрим вопрос о циклических зависимостях, в том числе на примере. Когда мы закончили, ваша первая реакция должна состоять в том, чтобы попытаться переделать свои зависимости. Если это невозможно, вы можете разорвать порочный круг путем рефакторинга от внедрения в конструктор ко внедрению в свойство. Это представляет собой ослабление инвариантов класса, так что это нужно делать очень обдуманно.

Разрешение проблем с циклами зависимостей

Всякий раз, когда я сталкиваюсь с циклом зависимости, вот мой первый вопрос: "Где я ошибся?"

Совет

Цикл зависимостей указывает на «плохо пахнущий» код. Если такое появится, вы должны серьезно пересмотреть структуру и код.

Цикл зависимостей должен немедленно вызвать тщательную оценку причин цикла. Они часто основаны либо на неправильных предположениях, либо на серьезных нарушениях правил однонаправленных зависимостей. В слоевом приложении классы должны говорить с другими классами только в своем слое или близлежащем нижнем слое.

Если цикл проходит более одного слоя, мы знаем, что что-то в корне неверно. Как показано на рисунке 6-8, это обозначает, что некоторые ссылки идут не в ту сторону.

Рисунок 6-8: Когда цикл пересекает одну или более границ слоя, по крайней мере, одна ссылка архитектурно незаконна. В данном случае, ссылка от D до A является незаконной. Если такая ситуация возникает, решать ее нужно немедленно.



Если у нас есть цикл в пределах одного слоя, то немного непонятно, что происходит. Это даже может быть результатом хорошей идеи, которая просто закончилась неудачной реализацией.

Нам необходимо сломать цикл любым способом. Пока цикл существует, приложение не будет работать.

Любой цикл является плохо пахнущим кодом, поэтому наша первая реакция должна заключаться в том, чтобы переделать вовлеченные части и предотвратить цикл. Таблица 6-1 показывает некоторые общие направления, в которых мы можем двигаться.

Таблица 6-1: Некоторые стратегии разработки, чтобы сломать циклические зависимости

Стратегия	Описание
События	Вы можете часто сломать цикл, изменив одну из абстракций так, чтобы она вызывала события вместо явного вызова зависимости, сообщающие зависимости, что что-то произошло. События особенно уместны, если только одна сторона вызывает <code>void</code> методы для своей зависимости. .NET события являются применением шаблона проектирования <i>Наблюдатель</i> (Observer), и вы можете иногда рассматривать вопрос о явной реализации. Это особенно верно, если вы решите использовать <i>доменные события</i> (Domain Events), чтобы разорвать порочный круг. Тут есть потенциал обеспечить возможность истинной асинхронной односторонней передачи сообщений.
Внедрение в свойство	Если ничего не помогает, мы можем разорвать порочный круг путем рефакторинга одного класса от внедрения в конструктор к внедрению в свойство. Это самый крайний вариант, потому что он только лечит симптомы.

Я не намерен тщательно исследовать первый вариант, потому что существующая литература уже предоставляет подробную информацию.

Совет

Попытайтесь решить проблему циклов с помощью событий. Если это не поможет, попробуйте *Наблюдатель*. И только если у вас все еще не получается разорвать цикл, используйте внедрение в свойство.

Не ошибитесь: цикл зависимостей является плохо пахнущим кодом. Наш первый приоритет заключается в том, чтобы проанализировать код и понять, почему появился цикл. Когда мы поймем, почему, мы должны изменить структуру и код.

Тем не менее, иногда мы не можем изменить дизайн. Даже если мы понимаем причину возникновения цикла, API-«нарушитель» может быть вне нашего контроля.

Прерывание цикла при помощи внедрения в свойство

В некоторых случаях ошибка проектирования находится вне нашего контроля, но нам все еще нужно разорвать порочный круг. В таких случаях мы можем сломать цикл, используя внедрение в свойство.

Внимание

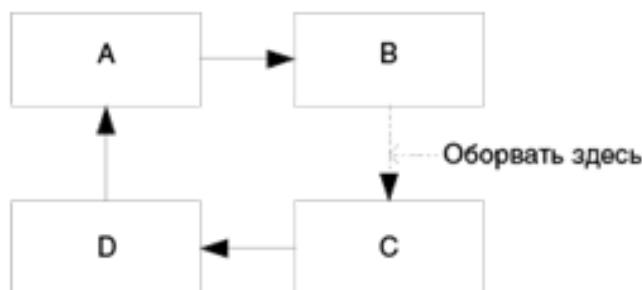
Вы должны прибегать к решению проблемы с циклами при помощи внедрения в свойство только в крайнем случае. Оно лечит симптомы, а не лечит болезнь.

Чтобы разорвать порочный круг, мы должны проанализировать его, чтобы выяснить, где мы можем сделать разрыв. Поскольку использование внедрения в свойство предлагает факультативную, а не необходимую зависимость, очень важно тщательно проверить все зависимости, чтобы определить, где разрыв принесет наименьший вред.

На рисунке 6-9, в требует экземпляр `IC` (интерфейс, который реализует `C`). Мы можем разорвать цикл, изменив зависимость для `B` от внедрения в конструктор во внедрение в свойство. Это означает, что мы можем сначала создать `B` и внедрить его в `A`, а затем впоследствии присвоить `C` `B`:

```
var b = new B();  
var a = new A(b);  
b.C = new C(new D(a));
```

Рисунок 6-9: Учитывая цикл, мы должны сначала решить, где его оборвать. В данном случае мы решили сломать цикл между `B` и `C`.



Использование внедрения в свойство таким способом добавляет дополнительную сложность в В, потому что теперь он должен быть в состоянии работать тогда, когда его зависимость пока не доступна.

Совет

Классы никогда не должны выполнять работу, вовлекая зависимости в свои конструкторы, потому что внедренная зависимость может быть еще не полностью инициализирована.

Если мы не хотим ослабить любой оригинальный класс таким образом, мы можем ввести виртуальную прокси (Virtual Proxy), которая оставляет в нетронутым:

```
var lb = new LazyB();
var a = new A(lb);
lb.B = new B(new C(new D(a)));
```

LazyB реализует IB, как это делает В. Тем не менее, он принимает свою зависимость IB через внедрение в свойство вместо внедрения в конструктор, что позволяет нам разорвать цикл, не нарушая инвариантов любого из оригинальных классов.

Хотя классы с образными именами A-D иллюстрируют структуру решения, более реалистичный пример является более оправданным.

Пример: создание окна

Одна из наиболее распространенных ситуаций, когда мы не можем перестроить наш выход из цикла зависимостей, это когда мы имеем дело с внешними API. Одним из таких примеров является WPF.

В WPF мы можем использовать MVVM паттерн, чтобы реализовать разделение понятий путем деления кода на представления и лежащие в основе модели. Модели присваиваются представлению через свойство DataContext. Это, по существу, внедрение в свойство в действии.

Совет

Вы можете прочитать больше о создании WPF приложений при помощи MVVM в разделе 7.4.

DataContext служит как зависимость для Window, но модель играет большую роль в управлении тем, какие представления и где активируются. Одним из действий, которое модель должна быть в состоянии выполнить, — это сделать всплывающим диалоговое окно. И один из способов реализации этого заключается во внедрении абстракции, как эта, в модель:

```
public interface IWindow
{
    void Close();
    IWindow CreateChild(object viewModel);
    void Show();
    bool? ShowDialog();
}
```

С внедренным `IWindow` любая модель может создать новые `Window` и отобразить их в виде модальных или немодальных окон. Однако чтобы реализовать этот интерфейс, мы должны иметь ссылку на реальный `Window`, чтобы правильно установить свойство `Owner`. В следующем листинге показана реализация метода `CreateChild`.

Листинг 6-2: Создание дочернего окна

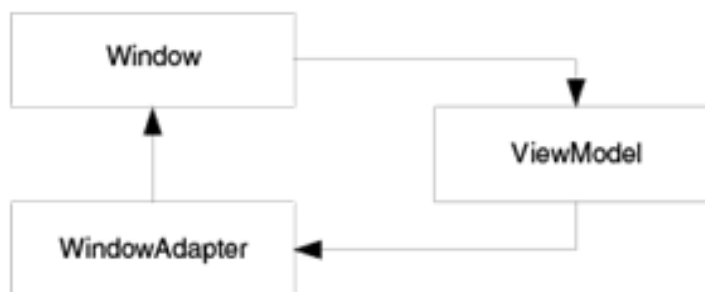
```
public virtual IWindow CreateChild(object viewModel)
{
    var cw = new ContentWindow();
    cw.Owner = this.wpfWindow;
    cw.DataContext = viewModel;
    WindowAdapter.ConfigureBehavior(cw);
    return new WindowAdapter(cw);
}
```

`ContentWindow` — это WPF окно, которое вы можете использовать, чтобы показать новое окно. Важно установить владельца `Window`, прежде чем показывать его, потому что иначе могут произойти странные ошибки, когда фокусированное или модальное окно скрыто за другими окнами. Чтобы предотвратить такие ошибки, вы устанавливаете свойство `Owner` для текущего `Window`. Поле `wpfWindow` является другим экземпляром `System.Windows.Window`.

Вы также присваиваете `viewModel` новому `Window DataContext`, прежде чем обернуть его в новую реализацию `IWindow` и вернуть его.

Вопрос в том, что с этой реализацией у вас есть `ViewModel`, которые требуют `IWindow`, реализация `IWindow`, которая требует `WPFWindow`, и `WPFWindow`, которые через их `DataContext` требуют, чтобы работала `ViewModel`. Рисунок 6-10 показывает этот цикл.

Рисунок 6-10: Цикл WPF MVVM. В MVVM `Window` зависит от `ViewModel`, которая, в свою очередь, зависит от экземпляра `IWindow`. Надлежащей реализацией `IWindow` является `WindowAdapter`, который зависит от `WPF Window`, чтобы иметь возможность установить владельца каждого `Window` и избежать ошибок фокусировки.



Мы ничего не можем тут изменить, чтобы выйти из циклической зависимости. Связь между `Window` и `ViewModel` зафиксирована, потому что `System.Windows.Window` является внешним API (определенным в BCL). Кроме того, `WindowAdapter` зависит от `Window`, чтобы избежать ошибок фокусировки, так что это отношение дано также и извне.

Единственное отношение, которое можно изменить, это только между `ViewModel` и его `IWindow`. Технически вы можете перепроектировать все это, чтобы использовать события, но это приведет к довольно нелогичному API. Для отображения диалогового окна вам нужно было бы вызвать событие и надеяться, что кто-то подпишется, показывая модальное окно. Кроме того, вам пришлось бы возвращать результат диалогового окна по

ссылке через аргументы исходного события. Вызов события был бы блокирующим вызовом. Это было бы технически возможным, но странным, так что мы исключим это.

Кажется, мы не можем переделать наш выход из цикла, так как же мы можем разорвать его?

Прерывание цикла

Нам нужно найти отношение, где мы сможем прервать цикл и ввести внедрение в свойство. В данном случае это легко, потому что отношение между `WPFWindow` и `ViewModel` уже использует внедрение в свойство. Это и будет место прерывания.

Самым простым решением будет подключить что-нибудь еще и установить свойство `DataContext` для `MainWindow` в последнюю очередь перед показом. Это возможно, но не особенно дружелюбно по отношению к DI контейнерам, потому что это потребовало бы от нас явно присвоить зависимость, после того как построение было выполнено.

В качестве альтернативы мы можем инкапсулировать отложенное присвоение в адаптер отложенной загрузки. Это позволяет соединить все должным образом с DI контейнером.

Примечание

В следующем примере показан этот же проект, который также описан в разделе 7.4.2. Вы можете увидеть весь код в примерах для книги.

Давайте посмотрим, как инкапсулировать создание реализации `IWindow`, которая правильно загружает `MainWindowViewModel` и присваивает ее экземпляру `WPFMainWindow`. Чтобы помочь сделать это, вы вводите эту абстрактную фабрику:

```
public interface IMainWindowViewModelFactory
{
    MainWindowViewModel Create(IWindow window);
}
```

Класс `MainWindowViewModel` имеет более чем одну зависимость, но все зависимости, кроме `IWindow`, могут быть удовлетворены сразу, так что вам не нужно передавать их в качестве параметра методу `Create`. Вместо этого вы можете внедрить их в конкретную реализацию `IMainWindowViewModelFactory`.

Вы используете `IMainWindowViewModelFactory` как зависимость в реализации `IWindow`, унаследованной от `WindowAdapter`, что представлен в листинге 6-2. Это позволяет отложить инициализацию реализации `IWindow`, пока не будет вызван первый метод. Здесь вы видите, как переписывается метод `CreateChild` из листинга 6-2:

```
public override IWindow CreateChild(object viewModel)
{
    this.EnsureInitialized();
    return base.CreateChild(viewModel);
}
```

Перед выполнением реальной работы вы должны убедиться, что все зависимости полностью инициализированы. Когда они инициализированы, вы можете безопасно вызвать базовую реализацию.

Следующий листинг показывает, как реализуется метод `EnsureInitialized` при помощи внедренной `IMainWindowViewModelFactory`.

Листинг 6-3: Отложенная инициализация зависимостей

```
1. private void EnsureInitialized()
2. {
3.     if (this.initialized)
4.     {
5.         return;
6.     }
7.     var vm = this.vmFactory.Create(this);
8.     this.WpfWindow.DataContext = vm;
9.     this.DeclareKeyBindings(vm);
10.    this.initialized = true;
11. }
```

Строка 7: Создать `ViewModel`

Строка 8: Внедрить `ViewModel` в `Window`

При инициализации `MainWindowAdapter` вы в первый раз вызываете внедренную абстрактную фабрику для создания желаемой `ViewModel`. Это возможно на данный момент, потому что экземпляр `MainWindowAdapter` уже создан, и поскольку он реализует `IWindow`, вы можете передать экземпляр методу `Create`.

Когда у вас есть `ViewModel`, вы можете присвоить ее `DataContext` инкапсулированного `WPF Window`. С небольшой дальнейшей настройкой `Window` теперь полностью инициализирован и готов к использованию.

В `Composition Root` приложения вы можете подключить все это вот так:

```
IMainWindowViewModelFactory vmFactory =
    new MainWindowViewModelFactory(agent);
Window mainWindow = new MainWindow();
IWindow w =
    new MainWindowAdapter(mainWindow, vmFactory);
```

Переменная `MainWindow` становится свойством `WpfWindow` в листинге 6-3, а `vmFactory` соответствует полю с одноименным названием. При вызове методов `Show` или `ShowDialog` для результирующего `IWindow` вызывается метод `EnsureInitialize` и все зависимости удовлетворены.

Эта комбинация отложенной инициализации и помощи абстрактной фабрики может быть хорошим дополнительным штрихом, но на первом месте – это наличие внедрения в свойство, которое позволяет разорвать порочный круг. В данном случае вам «повезло», потому что `WPF Window` уже использует внедрение в свойство через свое свойство `DataContext`.

Всегда имейте в виду, что лучший способ решить проблему цикла – это пересмотреть API, так чтобы цикл исчез. Тем не менее, в тех редких случаях, когда это невозможно или крайне нежелательно, мы должны разорвать цикл при помощи внедрения в свойство, по крайней мере, в одном месте. Это позволяет составить остальную часть графа объекта отдельно от зависимости, связанной со свойством. Когда остальная часть графа объекта полностью заполнена, то можно внедрить соответствующий экземпляр с помощью

свойства. В качестве дополнительного шага мы можем инкапсулировать эту логику назначения свойства в классе и использовать абстрактную фабрику, чтобы присвоить значение свойства в самый последний момент.

Внедрение в свойство сигнализирует о том, что зависимость не является обязательной, поэтому это изменение не должно нам даваться легко. Внедрение в конструктор намного предпочтительнее в большинстве случаев, но оно может быть сложным для некоторых людей. Давайте посмотрим, почему.

6.4. Обсуждение феномена Constructor Over-injection

Если у вас нет специальных требований, внедрение в конструктор должно быть вашим предпочтительным паттерном внедрения. Тем не менее, некоторым людям неудобно, если число зависимостей растет. Они не любят конструкторы со слишком большим числом параметров.

В этом разделе мы рассмотрим очевидную проблему растущего числа параметров конструктора и то, почему это хорошо, а не плохо. Как вы увидите, это не значит, что мы должны принять длинные списки параметров в конструкторах, поэтому мы также рассмотрим, что мы можем сделать со слишком большим числом аргументов конструктора. В данном разделе есть пример.

Распознавание и решение проблемы Constructor Over-injection

Когда список параметров конструктора становится слишком большим, мы называем это явление *Constructor Over-injection* и считают его плохо пахнущим кодом. Этот плохо пахнущий код не появляется, но усугубляется в результате использования DI. Хотя нашей начальной реакцией может быть то, что нам не нравится внедрение в конструктор из-за Constructor Over-injection, мы должны быть благодарны, что общая проблема проектирования открывается нам.

В этом разделе мы сначала найдем время, чтобы оценить, как Constructor Over-injection немного облегчает нашу жизнь, а затем рассмотрим соответствующие последствия.

Constructor Over-injection как сигнал

Хотя внедрение в конструктор легко реализовать и использовать, оно доставляет неудобство, когда конструкторы начинают выглядеть так:

```
public MyClass(IUnitOfWorkFactory uowFactory,
    CurrencyProvider currencyProvider,
    IFooPolicy fooPolicy,
    IBarService barService,
    ICoffeeMaker coffeeMaker,
    IKitchenSink kitchenSink)
```

Я не могу сказать, что я кого-то обвиняю, если он не любит такой конструктор, но я не виню внедрение в конструктор. Я могу согласиться, что конструктор с шестью параметрами сигнализирует о плохо пахнущем коде, но это свидетельствует о нарушении *Принципа единственной обязанности*, а не о проблемах, связанных с DI.

Совет

Внедрение в конструктор позволяет легко определить нарушения принципа единственной обязанности.

Вместо того чтобы чувствовать неловкость из-за Constructor Over-injection, мы должны принять его как удачный побочный эффект внедрения в конструктор. Это сигнал, который

предупреждает нас всякий раз, когда класс берет на себя слишком большую ответственность.

Моим личным порогом являются четыре аргумента конструктора. Всякий раз, когда я добавляю третий аргумент, я начинаю рассматривать вопрос, могу ли я спроектировать вещи по-другому, но я могу жить с четырьмя аргументами для нескольких классов. Ваш лимит может быть другим, но когда вы его пересекаете, приходит время рефакторинга.

То, как мы проводим рефакторинг определенного класса, который слишком сильно вырос, зависит от конкретных обстоятельств: на месте ли объектная модель, домен, бизнес-логика и так далее. Разделение *Божественного класса* (God Class) на более мелкие, более сфокусированные классы в соответствии с известными паттернами проектирования – это всегда хороший ход.

Тем не менее, бывают случаи, когда бизнес требования обязывают нас делать много разных вещей в одно и то же время. Это часто случается в пограничной области приложения. Подумайте о крупнозернистой операции веб сервиса, которая запускает много бизнес событий. Один из способов моделирования таких операций заключается в сокрытии множества зависимостей за *Фасадными сервисами* (Facade Services).

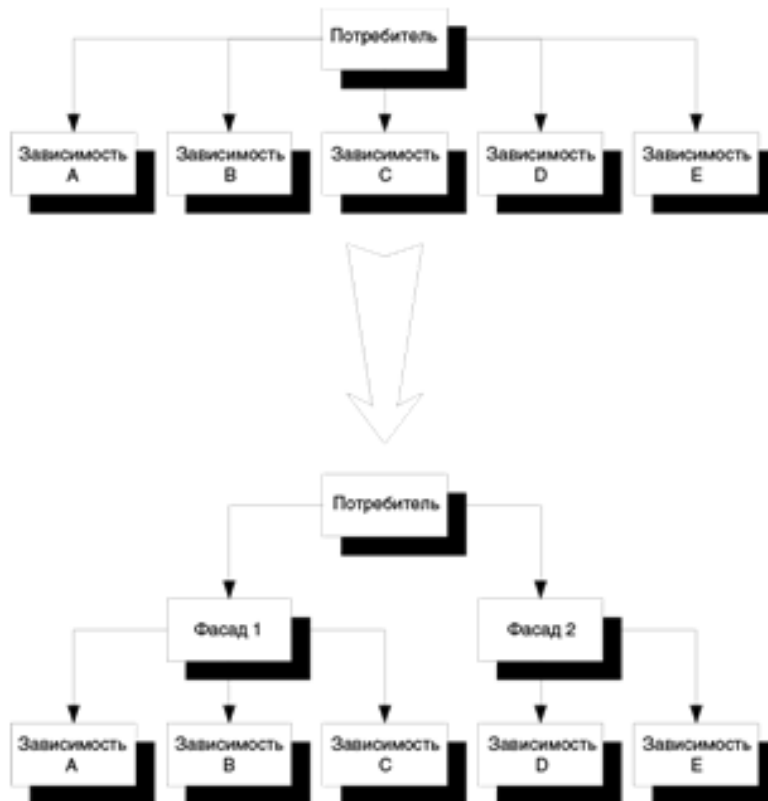
Рефакторинг по направлению к Фасадным сервисам

Есть много способов, как мы можем разработать и осуществить необходимые операции, так чтобы они не нарушали принцип единственной обязанности. В главе 9 мы обсудим, как паттерн проектирования *Декоратор* (Decorator) может помочь нам со стеком Cross-Cutting Concerns, вместо внедрения их в потребляющие элементы в виде сервисов. Это может устранить лишнее число аргументов конструктора.

Тем не менее, в некоторых случаях единая точка входа должна организовать много зависимостей. Одним из примеров является операция веб сервиса, которая запускает сложное взаимодействие различных сервисов. Точка входа пакета информации может столкнуться с той же проблемой.

На рисунке 6-11 показано, как мы можем провести рефакторинг ключевых отношений по направлению к фасадным сервисам.

Рисунок 6-11: На самой верхней диаграмме видно, что у потребителя есть пять зависимостей, которые являются сильным признаком того, что нарушен принцип единственной обязанности. Тем не менее, если роль потребителя заключается в том, чтобы организовать эти пять зависимостей, мы не можем какую-либо выбросить. Вместо этого мы можем ввести фасадные сервисы, которые организуют части отношений. В нижней части диаграммы у потребителя есть только две зависимости, а у фасадов две и три зависимости.



Рефакторинг к фасадным сервисам – это просто трюк, чтобы избавиться от слишком большого числа зависимостей. Ключом является определение естественных кластеров взаимодействия. На рисунке 6-11 показано, что зависимости А–С формируют естественный кластер взаимодействия, и также делают D и E.

Благотворный сторонний эффект заключается в том, что открытие этих естественных кластеров раскрывает ранее неизвестные отношения и доменные понятия. В ходе этого процесса мы обращаем неявные понятия в явные понятия. Каждый фасад становится сервисом, который фиксирует это взаимодействие на более высоком уровне, и единственная обязанность потребителя заключается в том, чтобы организовать эти высокоуровневые сервисы.

Примечание

Фасадные сервисы являются абстрактными фасадами, как следует из названия.

Фасадные сервисы связаны с *Parameter Objects*, но вместо объединения и раскрытия компонентов, фасадный сервис раскрывает только инкапсулированное поведение, скрывая компоненты.

Очевидно, что мы можем повторить этот рефакторинг, если у нас есть такая сложное приложение, что у потребителя появляется слишком много зависимостей в фасадных сервисах. Создание фасадов для фасадных сервисов является вполне разумным шагом.

В пограничных сферах нашего приложения (например, пользовательский интерфейс или веб сервис) мы можем работать с набором крупнозернистых абстракций. Рассматривая реализации зависимостей, мы видим, что за крупнозернистыми сервисами стоят тонкозернистые (детальные) сервисы, которые являются комбинациями еще более детальных сервисов. Это позволяет нам быстро все рассмотреть на первичном уровне, обеспечивая при этом то, что каждая окончательная реализация придерживается принципа единственной обязанности.

Давайте рассмотрим пример.

Пример: рефакторинг приема заказов

Примеру коммерческого приложения, к которому мы обращаемся на время от времени, нужно иметь возможность получать заказы. Это часто лучше всего делать в отдельном приложении или подсистеме, потому что в этот момент меняется семантика транзакции.

Когда вы работаете с покупательской корзиной, вы можете динамически вычислить удельные цены, курсы валют, а также скидки, но когда клиент делает заказ, все эти значения должны быть получены и заморожены так, как они были предоставлены, когда покупатель одобрил заказ. В таблице 6-2 представлен процесс приема заказа.

Таблица 6-2: Когда подсистема заказов получает новый заказ, она должна выполнить различные операции

Действие	Требуемые зависимости
Сохранить заказ	OrderRepository
Отправить имейл о заказе покупателю	IMessageService
Сообщить системе учета сумму счета	IBillingSystem
Выбрать лучшие склады, чтобы подобрать и отправить заказ на основе товаров в заказе и близости к адресу доставки	ILocationService, IInventoryManagement
Запросить выбранные склады подобрать и отправить весь заказ или часть его	IInventoryManagement

Пять различных зависимостей требуется для того, чтобы просто получить заказ. Представьте себе, сколько других зависимостей вам нужно будет обработать по другим операциям, связанным с заказами!

Давайте сначала рассмотрим, как это будет выглядеть, если потребляющий класс `OrderService` непосредственно импортировал все эти зависимости, а в дальнейшем вы увидите, как можно провести рефакторинг функционала с помощью фасадных сервисов.

Слишком много детальных зависимостей

Если вы позволите `OrderService` непосредственно потреблять все пять зависимостей, структура будет такой, как показано на рисунке 6-12.

Рисунок 6-12: У `OrderService` есть пять зависимостей, и это сигнализирует о том, что нарушен принцип единственной обязанности.



Если вы используете внедрение в конструктор для класса `OrderService`, у вас будет конструктор с пятью параметрами. Это слишком много, и указывает на то, что `OrderService` имеет слишком много обязанностей. С другой стороны, все эти зависимости необходимы, поскольку класс `OrderService` должен реализовать все необходимые функции, когда он получает новый заказ.

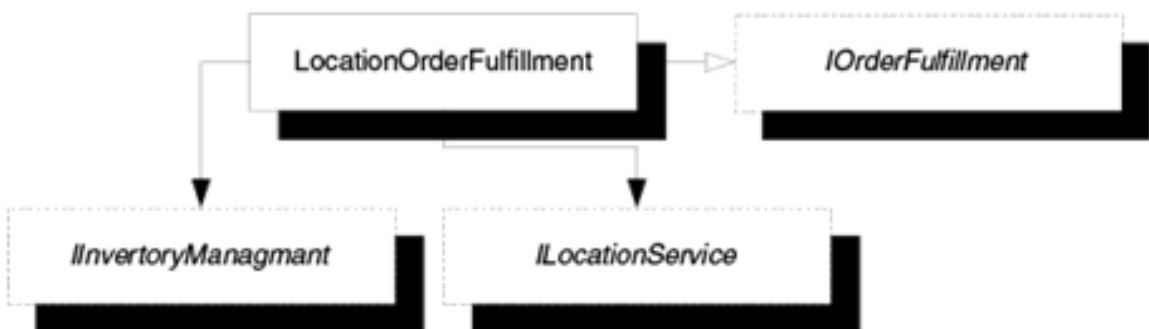
Вы можете решить эту проблему, переделав `OrderService`.

Рефакторинг по направлению к фасадным сервисам

Первое, что вам нужно сделать, это посмотреть на естественные кластеры взаимодействия с целью выявления потенциальных фасадных сервисов. Взаимодействие между `ILocationService` и `IInventoryManagement` должны немедленно привлечь ваше внимание, потому что вы используете их, чтобы найти ближайшие склады, которые могут выполнить заказ. Потенциально это может быть более сложный алгоритм, но после того как вы выбрали склады, вам нужно уведомить их о заказе.

Если вы подумаете об этом чуть больше, то оказывается, что `ILocationService` является деталью реализации уведомления соответствующих складов о заказе. Все взаимодействие может быть скрыто за интерфейсом `IOrderFulfillment`, как показано на рисунке 6-13. Интересно, что *выполнение заказов* звучит очень похоже на концепцию; есть вероятность, что вы только что обнаружили неявную доменную концепцию и сделали ее явной.

Рисунок 6-13: Взаимодействие между `IInventoryManagement` и `ILocationService` осуществлено в классе `LocationOrderFulfillment`, который реализует интерфейс `IOrderFulfillment`. Потребители интерфейса `IOrderFulfillment` понятия не имеют, что у реализации есть две зависимости.



Реализация по умолчанию `IOrderFulfillment` потребляет две исходные зависимости, поэтому имеет конструктор с двумя параметрами, а это хорошо. Как дополнительное преимущество, вы инкапсулировали алгоритм для нахождения лучших складов по данному заказу в многократно используемый компонент.

Этот рефакторинг объединяет две зависимости в одну, но оставляет вас с четырьмя зависимостями класса `OrderService`. Вам нужно искать другие возможности для объединения зависимостей в фасад.

Следующая вещь, которую вы могли заметить, заключается в том, что все требования включают уведомления других систем о заказе. Это означает, что вы можете определить общую абстракцию, которая моделирует уведомления, возможно, что-то вроде этого фрагмента кода:

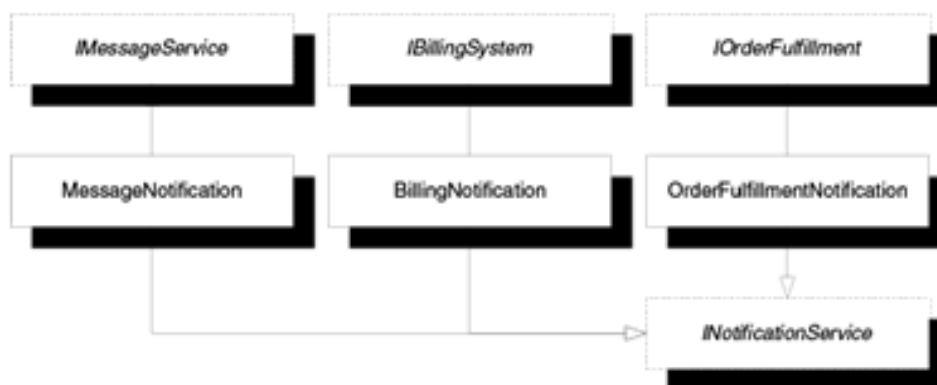
```
public interface INotificationService
{
    void OrderAdded(Order order);
}
```

Совет

Паттерн проектирования *Domain Event* (доменное событие) может быть хорошей альтернативой для данного сценария.

Каждое уведомление внешней системы может быть реализовано с помощью этого интерфейса. Вы можете даже рассмотреть оберывание `OrderRepository` в `INotificationService`, но вполне вероятно, что классу `OrderService` будет необходим доступ к другим методам `OrderRepository` для реализации других функций. Рисунок 6-14 показывает, как вы реализуете другие уведомления при помощи `INotificationService`.

Рисунок 6-14: Каждое уведомление внешней системы может быть спрятано за `INotificationService`: даже новый интерфейс `IOrderFulfillment`, который вы только ввели.

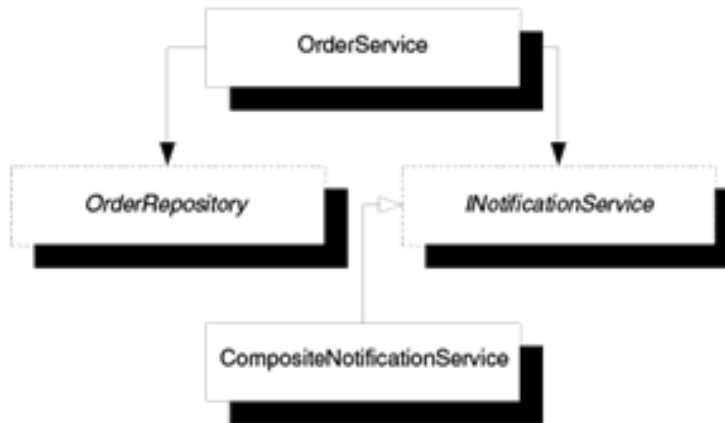


Вы можете удивиться, чем же это помогает, потому что вы обернули каждую зависимость в новый интерфейс. Количество зависимостей не уменьшилось, и это мне поможет?

Да. Поскольку все три уведомления реализуют один интерфейс, вы можете обернуть их в *Компоновщик* (Composite). Это еще одна реализация `INotificationService`, которая обрабатывает коллекцию экземпляров `INotificationService` и вызывает метод `OrderAdded` для них всех.

С концептуальной точки зрения это также имеет смысл, поскольку с высокоуровневого представления вы не заботитесь о деталях того, как OrderService уведомляет другие системы. Тем не менее, вас волнует, что он делает. Рисунок 6-15 показывает конечные зависимости OrderService.

Рисунок 6-15: Окончательный OrderService с зависимостями после рефакторинга. Вы оставляете OrderRepository как отдельную зависимость, потому что вам нужны его дополнительные методы для реализации других функций OrderService. Все другие уведомления скрыты за интерфейсом INotificationService. Во время выполнения вы используете CompositeNotificationService, который содержит оставшиеся три уведомления.



Это уменьшает OrderService до двух зависимостей, что является намного более разумным числом. Функциональность не изменилась, что делает это истинным рефакторингом. С другой стороны, изменился концептуальный уровень OrderService. В его обязанности сейчас входит: получить заказ, сохранить его и уведомить другие системы. Детали того, какие системы извещены и как это реализовано, были вытеснены на более детальный уровень.

Даже если вы везде последовательно использовали внедрение в конструктор, ни один конструктор класса не должен требовать более двух параметров (CompositeNotificationService принимает IEnumerable<INotificationService> как один аргумент).

Constructor Over-injection – это не проблема, связанная с DI в целом или, в частности, с внедрением в конструктор. Скорее, это сигнал о том, что класс имеет слишком много обязанностей. В этом классе плохо пахнет код, а не внедрение в конструктор; и как всегда, мы должны внимательно относиться к плохо пахнущему коду и улучшить его.

Есть много способов, которыми мы можем провести рефакторинг к паттернам, но одним из вариантов является введение фасадных сервисов, которые моделируют концепции на более высоком уровне абстракции. Это решает проблему нарушения принципа единственной обязанности и часто в процессе раскрывает ранее неизвестные доменные понятия.

Это один из многих способов, когда DI помогает нам лучше написать код. Поскольку слабая связанность является настолько ценной, мы хотим убедиться, что слабосвязанный код остается слабосвязанным. В следующем разделе рассматривается, как это сделать.

6.5. Мониторинг связывания

Слабая связанность является ценной, но это удивительно легко – ввести сильную (тесную) связанность. Все, что нужно, это начинающий разработчик и малейшая неосторожность, и будут введены жесткие ссылки. В Visual Studio можно легко добавить новые ссылки на существующий проект, но часто мы хотим того избежать. Необходимо соблюдать дисциплину, чтобы гарантировать, что каждый модуль фокусируется на своей зоне ответственности.

В этом разделе мы рассмотрим некоторые техники, которые могут быть полезны, когда мы хотим убедиться, что слабосвязанный код остается слабосвязанным. Возможно, мы хотим защитить код от своих ошибок, или, возможно, джуниор разработчикам в команде нужно немного помощи.

Ничто не сравнится с человеческим взаимодействием, когда речь заходит о передаче знаний. Парное программирование является идеальным, но это все же хорошая идея, просматривать то, что написано вручную, при помощи автоматизированных инструментов. В следующих разделах мы рассмотрим, насколько полезным может быть автоматизированное тестирование, а также изучим специализированный инструмент, который называется NDepend.

Связывание при юнит тестировании

Если у нас есть набор юнит тестов, с которым мы регулярно работаем, мы можем быстро добавить несколько модульных тестов, посвященных изучению зависимости, и они провалятся, если будет обнаружена необоснованная зависимость. Используя систему типов .NET, мы легко можем написать модульный тест, который пройдет циклом по всем ссылкам сборки и провалится, если найдет что-то там, чего там быть не должно.

В примере коммерческого приложения уже есть юнит тесты, так что вы легко можете добавить еще один. Следующий листинг показывает модульный тест, который защищает модуль логики представления (Presentation Logic module) от прямого обращения к модулю доступа к данным на основе SQL Server (SQL Server-based Data Access module).

Листинг 6-4: Обеспечение слабой связанности при помощи юнит тестирования

```
[Fact]
public void SutShouldNotReferenceSqlDataAccess()
{
    // Fixture setup
    Type sutRepresentative = typeof(HomeController);
    var unwanted = "Ploeh.Samples.Commerce.Data.Sql";
    // Exercise system
    var references =
        sutRepresentative.Assembly
            .GetReferencedAssemblies();
    // Verify outcome
    Assert.False(
        references.Any(a => a.Name == unwanted),
        string.Format(
            "{0} should not be referenced by SUT",
            unwanted));
    // Teardown
}
```

Этот тест ищет зависимости модуля презентационной логики. Чтобы получить список ссылок, вам нужно запросить сборку. Вы можете получить сборку из любого типа, содержащегося в этой сборке, так что вы можете выбрать один. Часто лучше всего выбирать тип, который останется в сборке на долгое время, потому что иначе вам придется переписывать тест, если выбранный тип будет удален. В данном тесте выбран `HomeController`, потому что у сайта всегда будет главная страница.

Кроме того, необходимо определить сборку, на которую не должны ссылаться. Вы можете использовать ту же технику и выбрать репрезентативный тип из этой сборки, но это будет означать, что вы должны сослаться на эту сборку из модульного теста. Это не так плохо, как ссылки на нежелательную сборку из рабочего кода, но это все равно создаст искусственную связь между этими двумя библиотеками – можно сказать, они становятся «уличенными в связи». Хотя безопасность типа желательна, слабая связанность имеет козыри в данном случае, поэтому вы определяете нежелательную сборку при помощи строки (но см. следующие обсуждения относительно других возможностей).

Получить сборку, на которую ссылаются, от репрезентативного типа так же просто, как один раз вызвать метод. Теперь вы можете использовать простой LINQ запрос, чтобы убедиться, что ни одна из этих сборок, на которые ссылаются, не имеет нежелательного имени. В утверждении (прим. переводчика: в принципе AAA при юнит тестировании последнее A – это Assert, в данном случае «утверждение») вы также выводите сообщение, которое отображается, если утверждение не выполняется.

Совет

Это утверждение использует простой LINQ запрос, но вы можете заменить его циклом `foreach`, если вы разрабатываете на .NET 3.0 или более ранних версиях.

Совет

Вы также можете изменить логику на обратную и написать тест, так чтобы только конкретные ссылки из заранее подготовленного списка были разрешены, а все другие ссылки считались незаконными.

Тестирование связанности при помощи Red/Green/Refactor

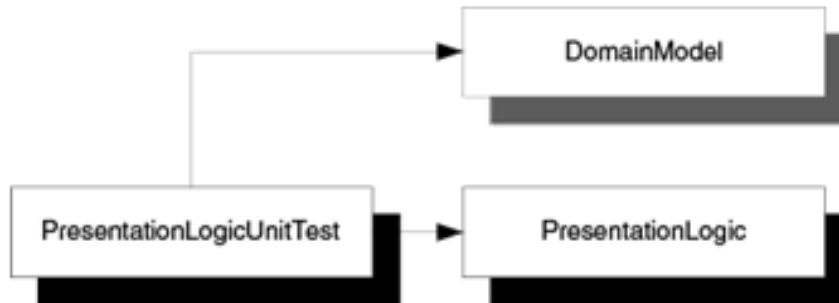
Если вы используете TDD для реализации вашего кода, вы привыкли к так называемому циклу разработки *Red/Green/Refactor*, где вы сначала пишете неудачный тест, затем он проходит успешно, и, наконец, вы меняете код, чтобы сделать его более легким в поддержке.

Оказывается, сделать так, чтобы тест, предотвращающий тесную связанность, не сработал, сложнее, чем вы думаете. Даже если целевой проект Visual Studio имеет ссылку на нежелательную зависимость, компилятор включит только ссылку, если она используется.

Таким образом, чтобы сделать тест, который не работает, мы должны сначала добавлять нежелательную ссылку, а затем написать фиктивную строку кода (dummy code), которая использует тип из нежелательной зависимости. Как только мы увидели, что тест не прошел, мы можем затем изменить процесс на противоположный, чтобы тест прошел. Очевидно, что это не проблема, если тестируемая библиотека уже нарушает ограничение связанности.

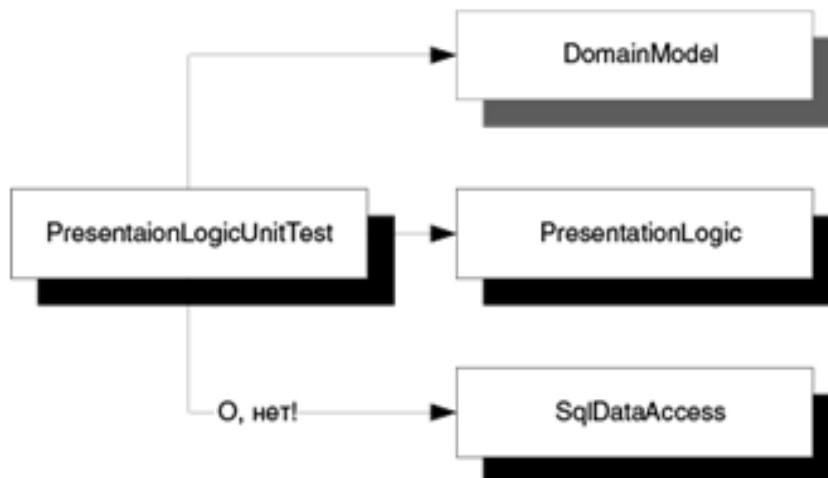
В предыдущем примере юнит тест добавлен к существующему набору юнит тестов, предназначенному для модуля логики представления. Рисунок 6-16 иллюстрирует ссылки в действии.

Рисунок 6-16: Библиотека `PresentationLogicUnitTest` – это набор тестов, нацеленный на библиотеку `PresentationLogic`. Чтобы сделать это, данная библиотека должна содержать ссылку к своей цели, а также общие абстракции, которые определены в доменной модели. Поскольку `PresentationLogicUnitTest` не нацелен на доменную модель, модуль `DomainModel` показан серым цветом.



Листинг 6-4 определяет нежелательную сборку при помощи простой строки, но было бы более безопасно для типа определить его при помощи репрезентативного типа. Однако вам может потребоваться добавить в юнит тест ссылку к модулю доступа к данным на основе SQL Server (SQL Server-based Data Access module), как показано на рисунке 6-17.

Рисунок 6-17: Если мы хотим безопасности типов, добавив репрезентативный тип из библиотеки `SqlDataAccess` в `PresentationLogicUnitTest`, мы вводим новую зависимость в набор юнит тестов по той единственной причине, что мы хотим убедиться, что это никогда не будет случайно добавлено в библиотеку `PresentationLogic`. Иронично, не так ли?



Вы можете подумать, что добавление дополнительной ссылки к проекту модульного тестирования – это не так плохо, но это имеет больше недостатков, которые не сразу становятся очевидными.

Косвенные зависимости

Подробное рассмотрение того, почему проект юнит тестов должен ссылаться только на тот проект, на который он направлен, выходит за рамки этой книги, но в целом проблема

состоит в том, что юнит тесты создают косвенную зависимость между `PresentationModel` и `SqlDataAccess`. Хотя оба эти проекта могут существовать и быть скомпилированы отдельно, проект модульного тестирования связывает их вместе.

Эта косвенная зависимость может быть сломана только тогда, когда будет исключен модульный тест, который изначально ее создал. Тем не менее, юнит тесты написаны для того, чтобы выполняться, так что это далеко от желаемого.

Если мы хотим оставить такие юнит тесты, предотвращающие тесную связанность, в рамках существующего проекта модульного тестирования, цена добавления жесткой ссылки на все нежелательные сборки слишком велика. Лучше всего определять нежелательные зависимости при помощи строк, как показано в листинге 6-4.

Недостатком является то, что если мы изменим название запрещенной сборки, тест становится бесполезным или, возможно, еще хуже – мы будем думать, что мы защищены, а на самом деле это будет не так.

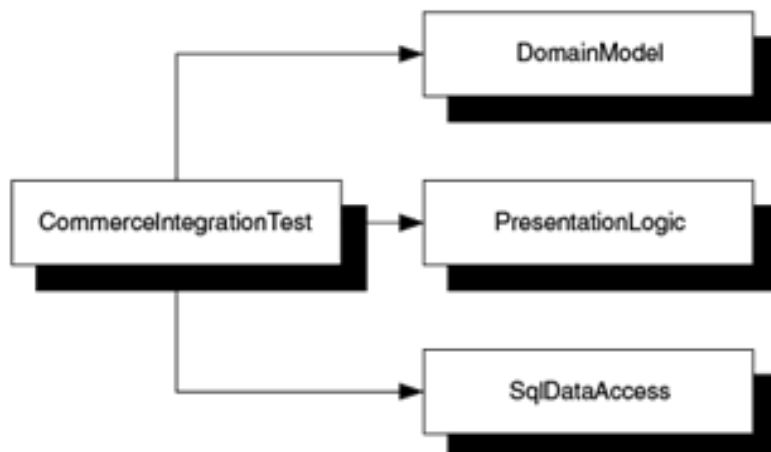
Это не проблема, если у нас есть основания полагать, что имена сборок стабильны. Если же это не так, нам нужна другая стратегия.

Связывание при интеграционном тестировании

Существуют веские причины того, почему проекты юнит тестов должны ссылаться только на свои цели. Оставаясь надежными в условиях изменения имен сборок, нам все же иногда могут понадобиться безопасные для типов ссылки на все нежелательные зависимости. Это звучит как противоречивое требование, но мы можем решить эту головоломку, введя новый проект по интеграционному тестированию.

Вы можете добавить новый тестовый проект в решение *Commerce* (Commerce solution) и добавить все нужные ссылки. Рисунок 6-18 показывает это решение, и хотя он очень похож на рисунок 6-17, разница состоит в том, что для интеграционного теста все ссылки легальны и одинаково действительны.

Рисунок 6-18: Проект `CommerceIntegrationTest` содержит автоматизированные тесты, которые проверяют, что отношения между модулями являются правильными. В отличие от модульных тестов, интеграционные тесты могут содержать столько ссылок, сколько необходимо, для проведения данного теста.



Интеграционные тесты

Интеграционное тестирование – это еще один тип автоматизированного тестирования на уровне API. Разница между модульным тестом и интеграционным тестом заключается в том, что модульное тестирование имеет дело с модулями (юнитами) в изоляции, в то время как интеграционные тесты сосредоточены на проверке того, что несколько юнитов (часто в разных библиотеках) интегрируются друг с другом, как нужно.

Согласно определению, проект по интеграционному тестированию может ссылаться на все зависимости, которые ему нужны, чтобы делать свою работу, так что это хорошо подходит для тестов, соблюдающих архитектурные ограничения.

Набор интеграционных тестов тесно связан с конкретной совокупностью модулей, так что он менее многообразный. Он должен содержать только тесты, которые абсолютно могут быть определены как интеграционные тесты, и тесты, которые защищают от нежелательной связанности, принадлежат к этой категории. Листинг 6-5 показывает безопасный для типов тест, эквивалентный тесту из листинга 6-4. Он следует тому же плану, но меняется, когда дело доходит до выявления нежелательной зависимости.

Листинг 6-5: Обеспечение слабой связанности при помощи интеграционного теста

```
1. [Fact]
2. public void PresentationModuleShouldNotReferenceSqlDataAccess()
3. {
4.     // Fixture setup
5.     Type presentationRepresentative =
6.         typeof(HomeController);
7.     Type sqlRepresentative =
8.         typeof(SqlProductRepository);
9.     // Exercise system
10.    var references =
11.        presentationRepresentative.Assembly
12.            .GetReferencedAssemblies();
13.    // Verify outcome
14.    AssemblyName sqlAssemblyName =
15.        sqlRepresentative.Assembly.GetName();
16.    AssemblyName presentationAssemblyName =
17.        presentationRepresentative.Assembly.GetName();
18.    Assert.False(references.Any(a =>
19.        AssemblyName.ReferenceMatchesDefinition(
20.            sqlAssemblyName, a)),
21.        string.Format(
22.            "{0} should not be referenced by {1}",
23.            sqlAssemblyName,
24.            presentationAssemblyName));
25.    // Teardown
26. }
```

Строки 10-12: Получить сборки, на которые ссылаются

Строки 14-17: Получить имена сборок

Строки 18-20: Искать нежелательную зависимость

Теперь, когда у вас есть ссылки на все необходимые зависимости, вы можете выбрать тип из каждого модуля, который можно использовать, чтобы представить их сборки. В отличие от предыдущего примера, это можно определить безопасным для типа способом.

Также как и раньше, вы получите список всех сборок, на которые ссылается библиотека `PresentationLogic`. Используя `AssemblyName` каждой сборки, вы проверяете, чтобы ссылки не содержали сборку на основе SQL Server. Встроенный статический метод `ReferenceMatchesDefinition` сравнивает имена сборок.

Возможно, вы заметили, что тесты в листингах 6-4 и 6-5 аналогичны. Вы могли бы написать новые тесты, как тот, что в листинге 6-5, изменив два репрезентативных типа и оставив все остальное, как есть.

Следующим логическим шагом было бы выделение общей части теста в параметризованный тест (Parameterized Test). Это позволит вам написать простой список почти декларативных тестов, которые определяют, что разрешено или не разрешено в этой конкретной совокупности модулей.

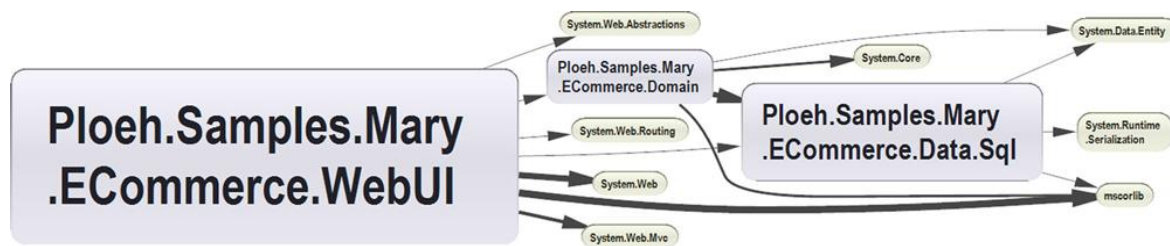
Модульные тесты и интеграционные тесты являются отличными вариантами, если вы уже используете автоматизированные тесты на API уровня. Если же нет, вы должны начать делать это как можно скорее, но есть и другие альтернативы.

Использование NDepend для мониторинга связанности

Если по каким-то непостижимым причинам вы не хотите использовать юнит тесты, вы можете использовать инструмент под названием NDepend (<http://ndepend.com>), который предупредит вас, если вы или члены вашей команды ввели нежелательную связанность.

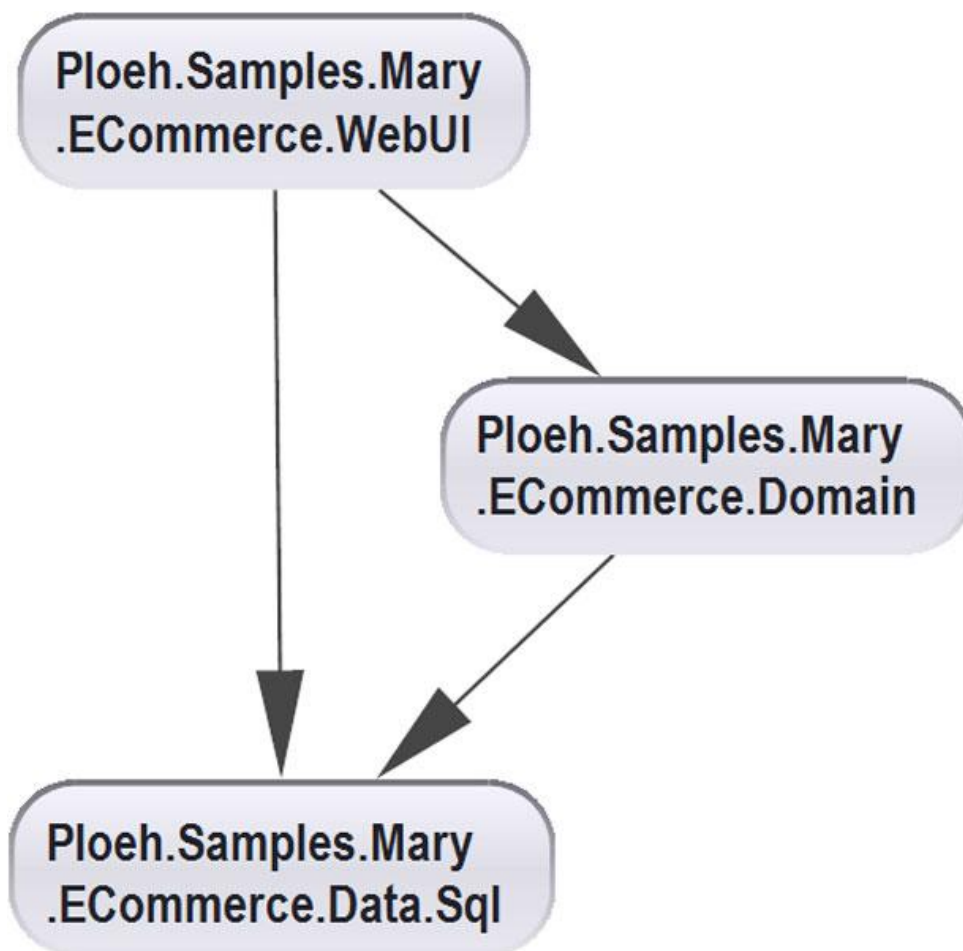
NDepend является коммерческим инструментом программного обеспечения, который анализирует проекты или решения и дает много статистики о коде. В качестве примера, он может генерировать графы зависимостей, которые мало чем отличаются от тех, что вы видели в этой книге. Если проанализировать коммерческое решение Мэри из главы 2, мы получим граф, показанный на рисунке 6-19.

Рисунок 6-19: Граф зависимостей, сгенерированный NDepend, для коммерческого решения Мэри. По умолчанию NDepend включает все зависимости, в том числе модули из BCL. Размер блоков отображает число строк кода в каждом модуле, а толщина стрелки отражает число элементов, используемых по всем ссылкам.



Это кажется сложным, но мы можем скрыть BCL модули, и мы получим рисунок 6-20.

Рисунок 6-20: Измененный график NDepend для коммерческого решения Мэри. На этом графе я вручную удалил все BCL модули и сделал блоки и стрелки одного размера.



Выглядит ли рисунок 6-20 знакомыми? Если вы обладаете эйдетической памятью, вы можете вспомнить рисунок 2-10, в ином же случае просто вернитесь к нему. Обратите внимание, что они представляют одни и те же структуры и иллюстрируют одни и те же отношения.

NDepend может сделать гораздо больше, чем нарисовать симпатичные графики. Одна из его наиболее мощных возможностей – это Code Query Language (CQL), который позволяет нам запрашивать наш код по широкому спектру информации и имеет синтаксис, напоминающий синтаксис SQL.

Если бы Мэри написала CQL проверку, прежде чем разрабатывать решение, она была бы предупреждена до того, как был нанесен большой ущерб. Вот запрос, который мог бы избавить ее от многих неприятностей:

```
WARN IF Count > 0 IN SELECT ASSEMBLIES WHERE  
IsDirectlyUsing "ASSEMBLY:Ploeh.Samples.Mary.ECommerce.Data.Sql" AND  
NameIs "Ploeh.Samples.Mary.ECommerce.Domain"
```

После выполнения этот CQL запрос выдает предупреждение, если доменный модуль напрямую обращается к модулю SQL Server Data Access. В решении Мэри этот запрос действительно выдаст предупреждение.

Мы можем написать столько CQL запросов для решения, сколько захотим, и запустить их либо с помощью редактора или автоматизировать процесс с помощью командной строки. В обоих случаях генерируются XML файлы с результатами анализа, поэтому мы можем написать свои собственные инструменты автоматизации, которые принимают соответствующие меры, если мы хотим включить такой шаг в автоматизированный процесс сборки.

Примечание

Я лишь поверхностно рассмотрел возможности NDepend. Он может делать много других вещей, но я хотел сосредоточиться на его способности следить за связанностью.

NDepend и автоматизированные тесты – это два способа автоматического контроля кода, чтобы незаконные зависимости случайно не прокрались. Мы можем использовать один или оба из этих вариантов в рамках *тестирования сборки* (Build Verification Test, BVT) или *непрерывной интеграции* (Continuous Integration, CI).

Для больших баз кода, которые поддерживаются большими командами, это может защитить нас от значительных проблем. Хотя мы не можем следить за всем, что происходит, и выполняем проверки кода вручную на каждом этапе, автоматизированные инструменты могут предупредить нас, если произойдет что-то странное.

Внимание

Некоторые инструменты могут вам «наврать», так что не стоит слепо верить им, когда они говорят вам, что есть проблемы. Всегда используйте ваш опыт и знания для оценки предупреждений. Откажись от них, если вы не согласны.

Обращайте внимание на каждый случай, и если есть реальная проблема, решайте ее.

Рассмотрите возможность использования автоматизированных инструментов для мониторинга связанности в больших базах кода. Это может предотвратить случайную тесную связанность и «порчу» вашего кода, а сами сосредоточьтесь на других проблемах, описанных в этой главе.

6.6. Резюме

DI не представляет собой особой сложности, если вы понимаете несколько основных принципов, но как вы поняли, вы почти наверняка столкнетесь с проблемами, которые могут поставить в тупик на некоторое время. В этой главе мы сделали попытку рассмотреть некоторые из наиболее распространенных проблем.

Один из самых универсальных и полезных шаблонов проектирования, связанных с DI, является абстрактная фабрика. Мы можем использовать ее для перевода простых значений времени выполнения, таких как строки или числа, введенных пользователями, в экземпляры сложных абстракций. Мы также можем использовать абстрактные фабрики в сочетании с интерфейсом `IDisposable`, чтобы имитировать краткосрочные зависимости, такие как соединения с внешними ресурсами.

Совет

Переводите значения времени выполнения в зависимости при помощи абстрактных фабрик.

Совет

Имитируйте соединения при помощи абстрактных фабрик, которые создают одноразовые зависимости.

Проблема, которая иногда возникает, — это циклы зависимостей. Они, как правило, появляются из-за того, что API-интерфейсы являются слишком строгими. Чем больше API строится вокруг парадигмы запросов, тем больше шансов получить циклы. Мы можем избежать циклов, соблюдая *Принцип Голливуда* (Hollywood Principle). Методы `void` сигнатурами могут быть преобразованы в события, которые часто могут быть использованы для прерывания циклов. Если перепроектирование невозможно, мы можем разорвать цикл, изменив одно внедрение в конструктор во внедрение в свойство. Тем не менее, это надр делать аккуратно, потому что это меняет семантику потребителя.

Совет

Ломайте циклы при помощи внедрения в свойство.

Внедрение в конструктор должно быть вашим предпочтительным DI паттерном; дополнительным преимуществом является то, что всегда становится очевидно, когда вы нарушаете принцип единственной обязанности. Когда один класс имеет слишком много зависимостей, это сигнал, что мы должны его перестроить. Возможно, мы можем разбить его на несколько меньших классов, но иногда мы должны сохранить все функциональные возможности в рамках одного класса.

Совет

Решайте проблему Constructor Over-injection путем рефакторинга к фасадным сервисам.

В этих случаях мы можем поднять уровень абстракции, вставив слой фасадных сервисов между потребителем и зависимостями. Выполнение такого рефакторинга часто приводит к положительному побочному эффекту, что некоторые из этих фасадных сервисов

оказываются ранее неизвестными, неявными доменными концепциями. Выявление неявных концепций и вывод их в явные концепции является усовершенствованием доменной модели.

В то время как мы выполняем этот мелкий рефакторинг, мы не должны упускать из виду общую картину. Автоматизированные тесты или инструменты могут помочь нам уследить за тем, если тесная связанность появляется в части кода.

Если мы напишем много юнит тестов (в частности, если мы используем Test-Driven Development), тесная связанность быстро проявит себя в сложном и хрупком тестовом коде. Или, вероятно, потому что это невозможно – провести модульное тестирование больших частей приложения.

Совет

Пишите автоматизированные тесты для обеспечения слабой связанности.

Если мы не будем писать юнит тесты, тесную связанность можно не заметить, но мы увидим многие из ее симптомов: когда код усложняется, его все сложнее и сложнее поддерживать. Хороший и чистый дизайн, к которому мы изначально стремились, медленно превращается в спагетти-код. Добавление новой функциональной возможности требует от нас того, чтобы мы залезли даже в несвязанные области.

В этой главе описаны решения для часто встречающихся проблем с DI. Вместе с двумя предыдущими главами она образует каталог паттернов, анти-паттернов и рефакторинга. Данный каталог представляет собой часть 2 книги. В части 3 мы повернемся лицом к трем измерениям DI: *Композиции объектов* (Object Composition), *Управлению жизненным циклом* (Lifetime Management) и *Перехвату* (Interception).

DI самостоятельно

В главе 1 "Дегустационное меню" механизма внедрения зависимостей" я предоставил краткий обзор трех аспектов механизма внедрения зависимостей: композиция объектов, управление жизненным циклом и механизм перехвата. В этой части книги я раскрою эту точку зрения в трех отдельных главах. Многие DI-контейнеры обладают чертами, которые напрямую относятся к этим аспектам. Некоторые DI-контейнеры предоставляют черты всех трех аспектов, тогда как остальные поддерживают только некоторые из них.

Тем не менее, поскольку DI-контейнер является необязательным инструментом, я считаю, что важнее объяснить основные принципы и приемы, которые контейнеры обычно используют для реализации этих характеристик. В части 2 "DI самостоятельно" исследуется то, как вы можете сделать это самостоятельно, а не с помощью DI-контейнера. Теоретически вы можете использовать эту информацию для создания своего собственного DI-контейнера (но, пожалуйста, не делайте это – миру не нужен еще один DI-контейнер), или применять же механизм внедрения зависимостей вообще без использования контейнера – это то, что мы называем *Poor man's DI*. Однако главная цель этой части книги – изложить основные алгоритмы композиции объектов, управления жизненным циклом и механизма перехвата без необходимости использования конкретного DI-контейнера. Думаю, если бы я использовал конкретный контейнер, то было бы трудно отделить основные принципы от деталей конкретного API.

Глава 7 "Построение объектов" объясняет, как компоновать объекты в таких различных фреймворках, как ASP.NET MVC, WPF, WCF и т.д. Не все фреймворки в равной степени хорошо поддерживают механизм внедрения зависимостей, и даже среди тех, которые его поддерживают, способы, с помощью которых они это выполняют, значительно различаются. Для каждого фреймворка трудно идентифицировать шов, который разрешает механизм внедрения зависимостей в этом фреймворке. Тем не менее, при обнаружении этого шва вы получаете решение для всех приложений, которые используют этот конкретный фреймворк. В главе 7 я выполнил эту задачу для большинства общеизвестных фреймворков .NET приложений. Рассматривайте эту главу как каталог швов для фреймворков.

Даже если выбранный вами конкретный фреймворк не рассматривается здесь, я попытался обратиться ко всем возможным видам ограничений фреймворков, с которыми вы можете столкнуться. Например, с точки зрения механизма внедрения зависимостей, PowerShell – это самый ограниченный тип фреймворка, о котором я только мог подумать, поэтому я использовал его в качестве примера. Вы должны уметь экстраполировать решение для схожих фреймворков, даже если они явно не рассматриваются.

Несмотря на то, что компоновка объектов не усложняется благодаря Poor man's DI, вы должны начать понимать преимущества настоящего DI-контейнера после того, как прочтете о таком аспекте, как управление жизненным циклом. Существует возможность надлежащим образом управлять жизненным циклом различных объектов с помощью диаграммы объектов, но для этого нужно больше пользовательского кода, нежели для композиции объектов, и ни один из этих кодов не добавляет в приложение никакого конкретного бизнес-значения.

Кроме объяснения основных принципов управления жизненным циклом, глава "Жизненный цикл объектов" также содержит каталог универсальных стилей существования. Этот каталог выступает в роли словаря для обсуждения стилей

существования на протяжении всей части "DI-контейнеры", поэтому, несмотря на то, что вам не нужно уметь реализовывать любой из этих стилей вручную, хорошо бы знать, как они работают.

В главе "Механизм перехвата" мы рассматриваем часто встречающуюся проблему реализации сквозных сущностей на основе компонентов. Переходя окончательно от простого приложения паттерна проектирования Decorator к механизму перехвата в рабочей среде, мы рассматриваем способы формирования слабо связанных приложений в модульном исполнении. Я считаю эту главу кульминационной точкой книги – это то место, в котором многие читатели во время начальной программы по управлению доступом высказали мнение, что они начали понимать очертания чрезвычайно мощного способа моделирования программного обеспечения.

Несмотря на то, что я использую Poorman's DI для исследования и объяснения механизма внедрения зависимостей, я не рекомендую использовать его для профессионального использования. Многие отличные DI-контейнеры доступны на .NET и все они являются бесплатными. Таким образом, часть "DI-контейнеры" посвящена подробному рассмотрению API конкретного контейнера.

[7. Построение объектов](#)

[8. Жизненный цикл объектов](#)

[9. Механизм перехвата](#)

7. Построение объектов

Меню:

- Консольные приложения
- ASP.NET MVC
- Windows Communication Foundation
- Windows Presentation Foundation
- ASP.NET (Web Forms)
- PowerShell

Приготовление изысканных блюд из нескольких слоев – смелое начинание, особенно, если вы хотите поучаствовать в поедании этих блюд. Вы не можете хорошо кушать и готовить в одно и то же время, кроме того, многие блюда, чтобы они хорошо получились, необходимо готовить в последнюю минуту.

Профессиональные кулинары знают, как решить большинство из этих задач. Среди многих хитростей они используют в качестве основного принцип "*mise en place*", который можно приблизительно перевести как "все готово": все, что может быть приготовлено хорошо заранее готовится заранее. Чистятся и нарезаются овощи, нарезается мясо, подготавливается инвентарь, разогревается духовка, раскладываются инструменты и т.д.

Подготавливается настолько много компонентов, насколько это возможно. Если мороженое является частью десерта, то его можно сделать за день до приготовления десерта. Если первый слой содержит мидии, их можно почистить за несколько часов до этого. Даже такой недолговечный компонент, как беарнский соус можно приготовить за час до приготовления основного блюда. Когда гости готовы приступить к еде, необходимы только финальные приготовления: подогреть соус во время жарки мяса и т.д. Во многих случаях эта финальная композиция блюда не занимает более 5-10 минут. Рисунок 7-1 иллюстрирует этот процесс.

Рисунок 7-1: Принцип "*mise en place*" подразумевает приготовление всех компонентов блюда заранее для того, чтобы финальная композиция блюда была выполнена настолько быстро и легко, насколько это возможно.



Принцип "*mise en place*" похож на разработку слабо связанного приложения с помощью механизма внедрения зависимостей. Мы можем написать все требуемые компоненты заранее и компоновать их только тогда, когда мы должны это делать.

Примечание

В разделе "Composition Root" я сравнивал Composition Root с таким понятием *Бережливой разработки программного обеспечения* (Lean Software Development), как *Последний от вет ственный момент*. Сравнение Composition Root с принципом "*mise en place*" является схожей аналогией, несмотря на то, что при таком сравнении придается большое значение несколько другому аспекту: композиции.

Как и все аналогии, мы можем использовать их только до настоящего момента. Разница в том, что в кулинарии этапы подготовки и композиции разделены во времени, тогда как в разработке приложений это разделение проявляется в рамках модулей и уровней. Рисунок 7-2 демонстрирует то, как мы компоуем компоненты в Composition Root (часто на уровне пользовательского интерфейса).

Рисунок 7-2: Composition Root соединяет все независимые модули приложения. В противоположность "mise en place" это не происходит настолько поздно, насколько это возможно, но происходит там, где необходима интеграция различных модулей.



Первое, что происходит на этапе выполнения — это композиция объектов. Как только диаграмма объектов подключена, завершается композиция объектов и составные компоненты соединяются.

Несмотря на то, что композиция объектов — это фундаментальная основа механизма внедрения зависимостей, она является самой простой для понимания составляющей. Вы уже знаете, как это делать, поскольку вы все время компоуете объекты при создании объектов, которые содержат в себе другие объекты. В разделе 3.3. "Паттерны DI-контейнеров" мы рассмотрели основные принципы того, когда и как компоновать приложения. Следовательно, я не собираюсь использовать следующие 40 страниц для того, чтобы поведать вам о том, *как* компоновать объекты.

Вместо этого я хочу помочь вам обратиться к тем трудностям, которые могут возникнуть, когда вы будете компоновать объекты. Эти трудности возникают не из-за самой композиции объектов, а из-за фреймворков приложений, в которых вы хотели бы компоновать ваши объекты. Эти проблемы специфичны для каждого фреймворка и то же самое касается и резолюций. В моем случае эти проблемы принимают вид некоторых самых значимых препятствий для успешного применения механизма внедрения зависимостей, поэтому я сконцентрирую свое внимание именно на них. Это делает главу менее теоретической и более практической по сравнению с предыдущими главами.

Подсказка

Если вы хотите почитать только о применении механизма внедрения зависимостей в выбранном вами фреймворке, то вы можете пропустить этот раздел. Каждый раздел является автономным.

Легко формировать целостную иерархию зависимостей приложения, когда мы имеем полный контроль над жизненным циклом приложения (как мы поступаем с приложениями командной строки). Тем не менее, большинство фреймворков (ASP.NET, WCF и т.д.) в .NET включают в себя инверсию управления, которая иногда может усложнять процесс применения механизма внедрения зависимостей. Понимание швов каждого фреймворка является ключевым моментом для применения механизма внедрения зависимостей к конкретному фреймворку. Как иллюстрирует рисунок 7-3, в этой главе мы будем рассматривать то, как реализовать Composition Root'ы в некоторых универсальных фреймворках стандартной библиотеки классов (BCL).

Рисунок 7-3: Структура этой главы принимает форму каталога различных BCL фреймворков и швов, которыми они могут обладать для разрешения механизма внедрения зависимостей. Каждый раздел написан таким образом, что его можно читать независимо от остальных разделов.



Примечание

В связи с пространственными ограничениями я не буду рассматривать приложения Windows Forms. Тем не менее, когда дело касается композиции объектов, то в этом они схожи с WPF приложениями.

Каждый раздел посвящен одному из этих фреймворков и может быть прочитан более или менее независимо от остальных. Я буду начинать каждый раздел с общего введения к применению механизма внедрения зависимостей в этом конкретном фреймворке, затем последует обширный пример, который построен на универсальном примере Commerce, который используется в большей части данной книги.

Начнем мы с самого простого фреймворка, в котором применяется механизм внедрения зависимостей, и постепенно перейдем к более сложным фреймворкам. Когда мы дойдем

до ASP.NET, нам нужно будет пересечь бездну, за пределами которой мы можем применять механизм внедрения зависимостей, только подвергая риску, по крайней мере, несколько наших принципов. Не опровергается тот факт, что такие фреймворки, как ASP.NET и PowerShell являются попросту враждебными средами для применения механизма внедрения зависимостей, поэтому мы должны сделать все так хорошо, насколько мы это можем. Тем не менее, пока мы не достигнем этой точки, нет необходимости в компромиссе.

Консольное приложение, вероятно, является самым простым типом приложения, в котором применяется механизм внедрения зависимостей.

[7.1. Построение консольных приложений](#)

[7.2. Построение ASP.NET MVC приложений](#)

[7.3. Построение WCF приложений](#)

[7.4. Построение WPF приложений](#)

[7.5. Построение ASP.NET приложений](#)

[7.6. Построение PowerShell cmdlets](#)

[7.7. Резюме](#)

7.1. Построение консольных приложений

Консольное приложение – это самый простой для композиции вид приложения. В противоположность большинству других фреймворков BCL приложений, консольное приложение, в сущности, не содержит инверсию управления. Когда процесс выполнения доходит до точки входа в приложение (обычно это метод `Main`), мы берем все в свои руки. Не существует никаких особенных событий, которые нужно было бы описать, интерфейсов, которые нужно было бы реализовать, и мы можем использовать очень ограниченное количество сервисов.

Метод `Main` соответствует `Composition Root`. Первое, что нам нужно сделать в методе `Main`, – скомпоновать модули приложения и позволить им соединиться в единое целое. Это не сложно, но давайте рассмотрим пример.

Пример: актуализация валют

В главе 4 "DI паттерны" мы рассматривали то, как обеспечить возможность конвертации валют в шаблонном приложении `Commerce`. В разделе "Пример: конвертация валюты в корзине" был введен класс `Currency`, который предоставляет курс обмена одной валюты на другие. Поскольку `Currency` является абстрактным классом, мы могли бы создать множество различных реализаций, но в данном примере мы использовали базу данных. Целью примера кода из главы "DI паттерны" было продемонстрировать то, как восстановить и реализовать конвертацию валюты, таким образом, мы никогда не рассматривали то, как актуализировать обменный курс в базе данных.

Для того чтобы продолжить пример, давайте изучим то, как написать простое консольное приложение, которое позволяет администратору или привилегированному пользователю обновлять обменные курсы в базе данных без необходимости прямого взаимодействия с базой данных.

Программа `UpdateCurrency`

Поскольку целью этой программы является актуализация обменных курсов в базе данных, она имеет название `UpdateCurrency.exe`. Она будет принимать в качестве входных данных три аргумента командной строки:

- Код целевой валюты
- Код исходной валюты
- Обменный курс

Может показаться странным, что мы указываем цель до исходных данных, но такой способ является наиболее подходящим для большинства людей. Это говорит вам о том, сколько исходной валюты вам понадобится для того, чтобы купить одну единицу целевой валюты, например, обменный курс доллара к евро выражается как 1 евро за 1,44 доллара.

В командной строке это выглядит примерно так:

```
PS Ploeh:\> .\UpdateCurrency.exe EUR USD "1,44"
```

```
Updated: 1 EUR in USD = 1,44.
```

В результате выполнения программы происходит обновление базы данных и запись обновленных значений обратно в консоль.

Composition Root

UpdateCurrency использует точку входа по умолчанию для консольного приложения: метод Main в классе Program. Это Composition Root для приложения, что продемонстрировано в следующем листинге.

Листинг 7-1: Composition Root консольного приложения

```
public static void Main(string[] args)
{
    var container = new CurrencyContainer();
    container.ResolveCurrencyParser()
        .Parse(args)
        .Execute();
}
```

Единственной ответственностью метода Main является компоновка всех соответствующих модулей, и затем он передает ответственность за функциональность сформированной диаграмме объектов. В данном примере пользовательский контейнер инкапсулирует то, как komponуются модули. Поскольку он выполняет точно такую же функцию, как и DI-контейнер, я решил назвать его *контейнером*, несмотря на то, что он является пользовательским контейнером с жестко подключенными зависимостями. Мы вернемся к нему для того, чтобы вкратце рассмотреть, как он реализован.

После размещения контейнера вы можете теперь попросить его разрешить CurrencyParser, который анализирует входные аргументы и, в итоге, выполняет соответствующую команду.

Подсказка

Composition Root должна делать всего две вещи: настраивать контейнер и разрешать тип, который реализует требуемую функциональность. Как только он это выполнит, он должен сойти с дистанции и оставить остальную работу разрешенному экземпляру.

Подсказка

Используйте для ваших рабочих приложений настоящий DI-контейнер вместо доморощенного пользовательского контейнера.

В этом примере используется пользовательский контейнер, явно созданный для этого приложения, но необходимо прямо заменить его таким настоящим DI-контейнером, как те, что рассматривались в главе "DI паттерны".

Контейнер

Класс CurrencyContainer – пользовательский контейнер, созданный специально для целей подключения всех зависимостей к программе UpdateCurrency. Следующий листинг демонстрирует реализацию.

Листинг 7-2: Пользовательский CurrencyContainer

```
1. public class CurrencyContainer
2. {
3.     public CurrencyParser ResolveCurrencyParser()
4.     {
5.         string connectionString =
6.             ConfigurationManager.ConnectionStrings
7.             ["CommerceObjectContext"].ConnectionString;
8.         CurrencyProvider provider =
9.             new SqlCurrencyProvider(connectionString);
10.        return new CurrencyParser(provider);
11.    }
12. }
13.
```

Строка 5-7: Получает строку соединения из config

В этом примере диаграмма зависимостей довольно поверхностная. Для класса CurrencyParser необходим экземпляр абстрактного класса CurrencyProvider, а в CurrencyContainer вы решаете, что реализацией должен быть SqlCurrencyProvider, который предоставляет необходимое взаимодействие с базой данных.

Класс CurrencyParser использует механизм внедрения через конструктор, поэтому вы передаете в него только что созданный экземпляр SqlCurrencyProvider до того, как вернуть его из метода.

Если вдруг вам станет интересно, то ниже я привожу сигнатуру конструктора CurrencyParser:

```
public CurrencyParser(CurrencyProvider currencyProvider)
```

Помните о том, что CurrencyProvider – это абстрактный класс, который реализуется SqlCurrencyProvider. Несмотря на то, что CurrencyContainer содержит жестко закодированное преобразование CurrencyProvider в SqlCurrencyProvider, остальная часть кода является слабо связанной, поскольку в ней применяется только абстракция.

Этот пример может показаться простым, но в нем komponуются типы из трех различных уровней приложения. Давайте вкратце исследуем то, как эти уровни взаимодействуют в данном конкретном примере.

Иерархическое представление

Composition Root – это место, в котором компоненты всех уровней соединяются вместе. Точка входа и Composition Root образуют единственный исполняемый код. Вся реализация делегирована более низким уровням, как это иллюстрирует рисунок 7-4.

Рисунок 7-4: Композиция компонентов приложения UpdateCurrency. CurrencyParser анализирует аргументы командной строки и возвращает соответствующий ICommand. Если аргументы были понятны, то он возвращает CurrencyUpdateCommand, который использует экземпляр Currency для актуализации обменного курса. Вертикальная линия справа показывает соответствующий уровень приложения. Каждый уровень реализуется в отдельной сборке.

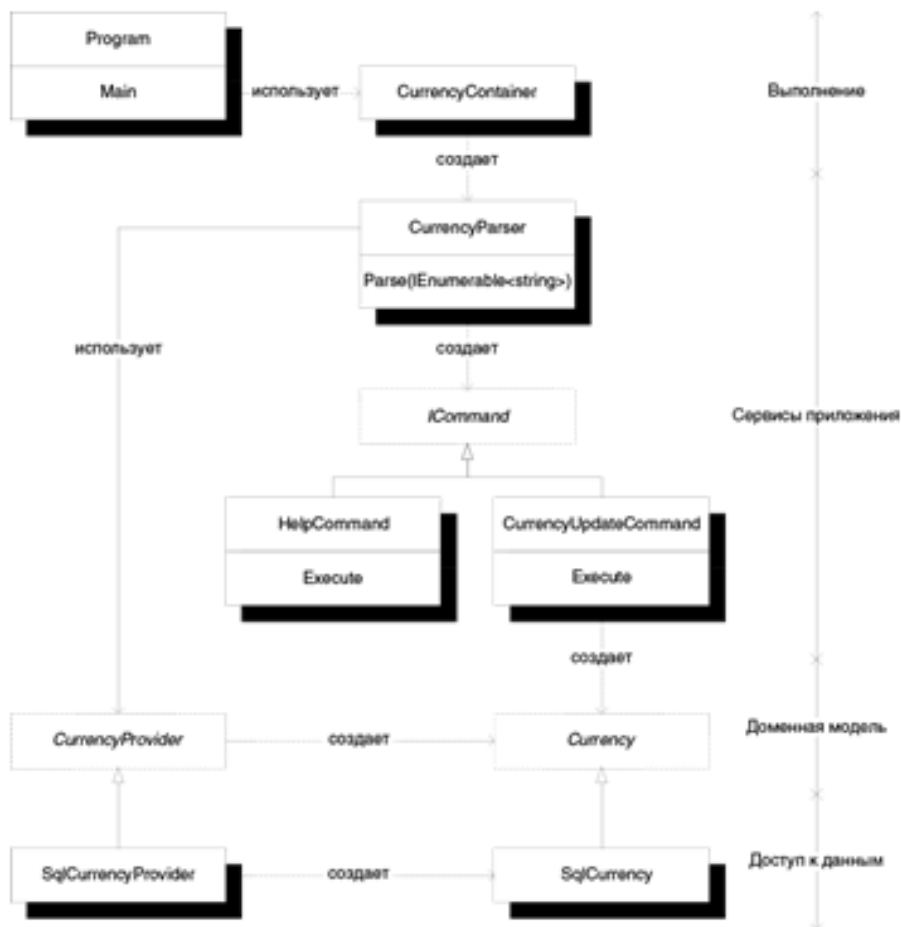


Диаграмма на рисунке 7-4 может казаться сложной, но она представляет почти всю базу кода приложения. Большая часть логики приложения состоит из анализа входных аргументов и выбора корректной команды на основании этих входных данных. Все это имеет место на уровне *Application Services* (сервисы приложения), который взаимодействует напрямую с уровнем доменной модели посредством абстрактных классов `CurrencyProvider` и `Currency`.

`CurrencyProvider` внедряется в `CurrencyParser` с помощью контейнера и впоследствии используется в качестве абстрактной фабрики для создания экземпляра `Currency`, который используется `CurrencyUpdateCommand`.

Уровень доступа к данным предоставляет SQL Server реализации доменных классов. Несмотря на то, что никакие другие классы приложения не обращаются напрямую к этим классам, `CurrencyContainer` преобразует абстракции в конкретные классы.

Использовать механизм внедрения зависимостей в консольном приложении легко, поскольку в действительности в нем нет внешней инверсии зависимостей. .NET Framework просто ускоряет процесс и передает управление методу `Main`.

В большинстве других BCL фреймворков присутствует более высокая степень инверсии управления, которая подразумевает, что нам нужно уметь определять корректные места расширяемости для того, чтобы подключить требуемую диаграмму объектов. Одним из таких фреймворков является ASP.NET MVC.

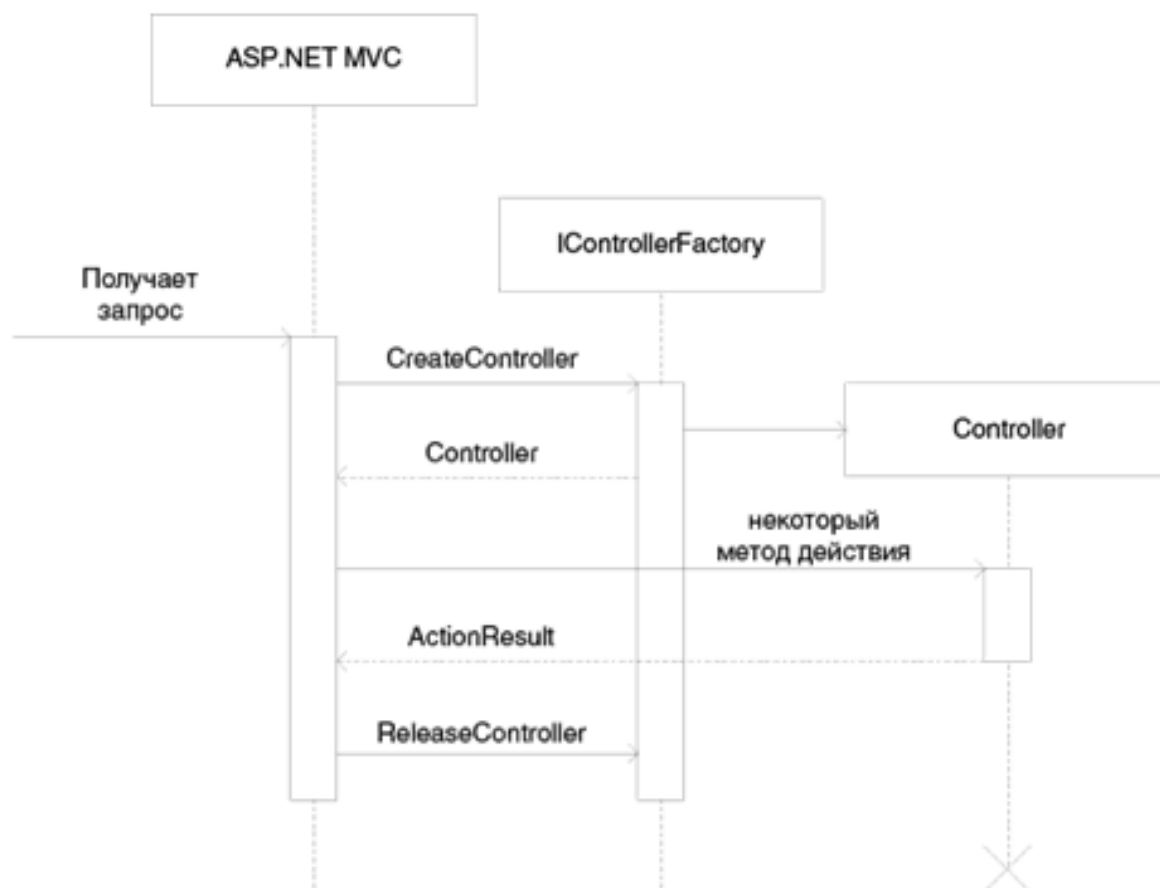
7.2. Построение ASP.NET MVC приложений

ASP.NET MVC был создан с выраженным стремлением стать DI-дружественным, и именно таким он и является. Он не вынуждает использовать механизм внедрения зависимостей, но с легкостью разрешает его, не заставляя нас при этом делать предположения о том, какой вид механизма внедрения зависимостей мы будем применять. Мы можем использовать Poor's Man DI или такой DI-контейнер, какой только пожелаем.

Расширяемость ASP.NET MVC

Как всегда и происходит с механизмом внедрения зависимостей, ключ к его применению заключается в обнаружении корректных мест расширяемости. В ASP.NET MVC таким ключом является интерфейс под названием `ControllerFactory`. Рисунок 7-5 иллюстрирует то, как он вписывается в фреймворк.

Рисунок 7-5: Когда рабочая среда ASP.NET MVC получает запрос, он просит свою фабрику контроллеров создать `Controller` для запрашиваемого URL. Фабрика контроллеров определяет корректный тип контроллера, который используется для данного запроса, создает и возвращает новый экземпляр этого типа. Затем ASP.NET MVC вызывает соответствующий метод действия для экземпляра `Controller`. После создания экземпляра контроллера ASP.NET MVC дает фабрике контроллеров освободить ресурсы путем вызова `ReleaseController`.



Контроллеры являются центральным понятием в ASP.NET MVC. Они управляют запросами и определяют то, как на них откликаться. Если нам нужно сделать запрос к базе данных, проверить и сохранить входные данные, вызвать доменную логику и т.д., то мы инициируем такие действия из контроллера.

Контроллер не должен выполнять такие действия сам, а должен делегировать работу соответствующим зависимостям. Это именно то место, где в игру вступает механизм внедрения зависимостей. Мы хотим уметь снабжать зависимостями данный класс `Controller`, в идеале с помощью внедрения через конструктор. Это можно сделать при помощи пользовательского `IControllerFactory`.

IDependencyResolver

После выхода в 2011 году ASP.NET MVC 3 одной из новых реализованных возможностей стала "поддержка DI". Оказалось, что эта поддержка сосредоточена вокруг нового интерфейса с названием `IDependencyResolver`. Этот интерфейс и тот способ, при помощи которого он используется в фреймворке ASP.NET MVC, являются проблематичными.

На концептуальном уровне предполагалось использовать `IDependencyResolver` в качестве `Service Locator`, и именно так фреймворк его и использует.

На более конкретном уровне интерфейс обладает ограниченной полезностью, поскольку в нем отсутствует метод `Release`. Другими словами, мы не можем правильно управлять жизненным циклом диаграмм объектов посредством этого интерфейса. Для некоторых DI-контейнеров этот факт гарантирует утечку ресурсов.

С учетом его текущего воплощения я считаю, что безопаснее и правильнее будет игнорировать `IDependencyResolver`. Ирония данной ситуации заключается в том, что истинный механизм внедрения зависимостей поддерживался ASP.NET MVC еще со времен первой его версии через интерфейс `IControllerFactory`.

Создание пользовательской фабрики контроллеров

ASP.NET MVC поставляется с `DefaultControllerFactory`, которая требует, чтобы классы `Controller` имели конструктор по умолчанию. Именно разумное поведение по умолчанию не вынуждает нас использовать механизм внедрения зависимостей, если мы этого не хотим. Тем не менее, конструкторы по умолчанию и механизм внедрения через конструктор являются взаимно исключаящими, поэтому нам необходимо изменить это поведение посредством реализации пользовательской фабрики контроллеров.

Это не так уж сложно. Для этого необходимо реализовать интерфейс `IControllerFactory`.

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
        string controllerName);
    SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}
```

Метод `CreateController` предоставляет `RequestContext`, который содержит такую информацию, как `HttpContext`, тогда как `controllerName` указывает на то, какой контроллер запрашивается.

Вы можете решить игнорировать `RequestContext` и использовать только `controllerName` для определения того, какой контроллер необходимо вернуть. Вне зависимости от того, что вы делаете, этот метод является местом, в котором вы получаете шанс подключить все необходимые зависимости и поставить их в контроллер прежде, чем вернуть экземпляр. Вы увидите пример в разделе "Пример: реализация `CommerceControllerFactory`".

Если вы создали какие-либо ресурсы, которые должны быть явно уничтожены, то вы можете сделать это при вызове метода `ReleaseController`.

Подсказка

`DefaultControllerFactory` реализует `IControllerFactory` и имеет несколько виртуальных методов. Вместо того чтобы реализовывать `IControllerFactory` с самого начала, часто легче наследовать его от `DefaultControllerFactory`.

Несмотря на то, что реализация пользовательской фабрики контроллеров является трудной частью, она не будет использоваться, пока мы не скажем о ней ASP.NET MVC.

Регистрация пользовательской фабрики контроллеров

Пользовательские фабрики контроллеров регистрируются как часть последовательности запуска приложения – обычно в файле *Global.asax*. Они регистрируются при помощи вызова `ControllerBuilder.Current.SetControllerFactory`. Ниже приведен фрагмент из шаблонного приложения `Commerce`:

```
var controllerFactory = new CommerceControllerFactory();  
ControllerBuilder.Current.SetControllerFactory(controllerFactory);
```

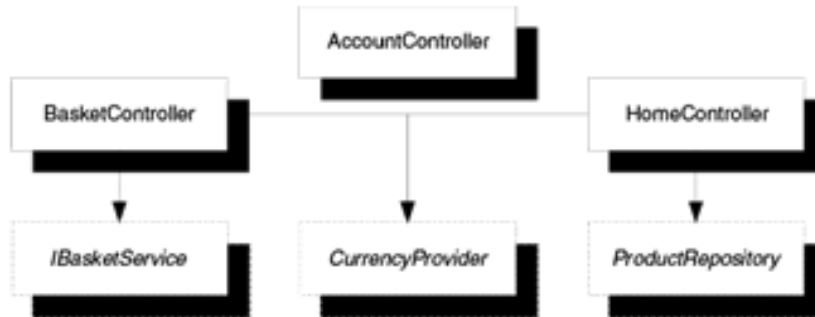
Этот пример создает и определяет новый экземпляр пользовательской `CommerceControllerFactory`. ASP.NET MVC теперь будет использовать экземпляр `controllerFactory` в качестве фабрики контроллеров данного приложения.

Если этот код кажется вам слегка знакомым, то это потому, что вы видели нечто похожее в разделе "Паттерны DI-контейнеров". Тогда я обещал показать вам, как реализовывать пользовательскую фабрику контроллеров, в главе "Построение объектов", и что? Это и есть глава "Паттерны DI-контейнеров".

Пример: реализация `CommerceControllerFactory`

Для шаблонного приложения `Commerce` нужна пользовательская фабрика контроллеров, чтобы подсоединить необходимые зависимости к контроллерам. Несмотря на то, что полноценная диаграмма зависимостей для всех контроллеров значительно глубже, с точки зрения самих контроллеров, соединение всех ближайших зависимостей достигает трех элементов, что продемонстрировано на рисунке 7-6.

Рисунок 7-6: Диаграмма зависимостей для трех контроллеров шаблонного приложения Commerce. Конкретные реализации каждой из трех зависимостей имеют другие зависимости, но они здесь не показаны. `BasketController` и `HomeController` имеют одну общую зависимость `CurrencyProvider`. `AccountController` унаследован неизменным от шаблона ASP.NET MVC по умолчанию; поскольку он использует `Bastard Injection`, он не имеет неразрешенных зависимостей



Несмотря на то, что вы могли бы реализовать `IControllerFactory` напрямую, проще наследовать его от `DefaultControllerFactory` и переопределить его метод `GetControllerInstance`. Это означает, что `DefaultControllerFactory` заботится о преобразовании имени контроллера в тип контроллера, и все, что вам приходится делать — это возвращать экземпляры необходимых типов.

Листинг 7-3: Создание контроллеров

```

1. protected override IController GetControllerInstance(
2.     RequestContext requestContext, Type controllerType)
3. {
4.     string connectionString =
5.         ConfigurationManager.ConnectionStrings
6.         ["CommerceObjectContext"].ConnectionString;
7.     var productRepository =
8.         new SqlProductRepository(connectionString);
9.     var basketRepository =
10.        new SqlBasketRepository(connectionString);
11.     var discountRepository =
12.        new SqlDiscountRepository(connectionString);
13.     var discountPolicy =
14.        new RepositoryBasketDiscountPolicy(
15.            discountRepository);
16.     var basketService =
17.        new BasketService(basketRepository,
18.            discountPolicy);
19.     var currencyProvider = new CachingCurrencyProvider(
20.         new SqlCurrencyProvider(connectionString),
21.         TimeSpan.FromHours(1));
22.     if (controllerType == typeof(BasketController))
23.     {
24.         return new BasketController(
25.             basketService, currencyProvider);
26.     }
27.     if (controllerType == typeof(HomeController))
28.     {
29.         return new HomeController(
30.             productRepository, currencyProvider);
31.     }
32.     return base.GetControllerInstance(
33.         requestContext, controllerType);
34. }

```

Строка 1-2: Переопределяет

Строка 4-21: Создает зависимости

Строка 22-31: Возвращает подключенные контроллеры

Строка 32-33: Использует базу для других контроллеров

Этот метод переопределяет `DefaultControllerFactory.GetControllerInstance` для того, чтобы создать экземпляры необходимых типов контроллера. Если требуемый тип – это `BasketController` или `HomeController`, то вы явно соединяете их с необходимыми зависимостями и возвращаете их. Оба типа используют внедрение через конструктор, поэтому вы поставляете зависимости через их конструкторы.

Для упрощения кода я решил соединить все зависимости до проверки `controllerType`. Очевидно, это означает, что некоторые созданные зависимости не будут использоваться, поэтому это не слишком рациональная реализация. Вы можете выполнить рефакторинг листинга 7-3 в более подходящую (но слегка более сложную) форму.

Для тех типов, которые не обрабатываются явно, вы по умолчанию обращаетесь к базовому поведению, которое заключается в создании необходимого контроллера при помощи его конструктора по умолчанию. Обратите внимание на то, что вы не обрабатываете явно `AccountController`, поэтому вместо этого вы позволите базовому поведению справиться с `AccountController`. `AccountController` является остатком шаблона ASP.NET MVC проекта и использует `Bastard Injection`, который дает ему конструктор по умолчанию.

Примечание

Я считаю `Bastard Injection` анти-паттерном, но я оставляю `AccountController` в таком состоянии, поскольку у меня есть множество других корректных примеров механизма внедрения зависимостей для демонстрации. Я сделал это, в конце концов, потому что это шаблонный код, но я бы никогда не оставил это в подобном состоянии в рабочем коде.

После регистрации экземпляра `CommerceControllerFactory` в файле *Global.asax* он будет корректно создавать все необходимые контроллеры с необходимыми зависимостями.

Подсказка

Подумайте над тем, чтобы не писать пользовательскую фабрику контроллеров самостоятельно. Вместо этого используйте универсальную фабрику контроллеров, которая работает совместно с выбранным вами DI-контейнером. Для вдохновения посмотрите MVC Contrib проект или используйте одну из доступных в нем многократно используемых реализаций. Некоторые DI-контейнеры также имеют "официальную" интеграцию с ASP.NET MVC.

Прекрасно, что ASP.NET MVC был сконструирован таким образом, что уже подразумевал DI, поэтому нам всего лишь нужно знать и использовать единственное место расширяемости для того, чтобы разрешить DI для нашего приложения. В других фреймворках разрешение DI может быть гораздо более сложной задачей. Windows Communication Foundation (WCF), хотя и является расширяемым, выступает в качестве примера такого фреймворка.

7.3. Построение WCF приложений

WCF – одна из самых расширяемых составляющих библиотеки стандартных классов (BCL). Несмотря на то, что начать писать WCF сервисы довольно легко, бесчисленные возможности расширяемости могут усложнить процесс поиска той возможности, которая нужна именно вам. Это еще один случай, когда в игру вступает механизм внедрения зависимостей.

Примечание

Согласно шутке, WCF – это акроним для Windows Complication Foundation. В этом утверждении есть определенная доля правды.

Вас легко можно было бы склонить к тому, что WCF не поддерживает внедрение через конструктор. Если вы будете реализовывать WCF сервис при помощи механизма внедрения через конструктор и без конструктора по умолчанию, то хостинг WCF сервиса в рабочей среде будет выдавать `ServiceActivationException` с сообщением, подобным следующему:

Предоставленный тип сервиса не удалось загрузить в качестве сервиса, поскольку у него отсутствует конструктор по умолчанию (непараметризованный). Для устранения проблемы добавьте к типу конструктор по умолчанию или передайте экземпляр типа в хостинг.

Это сообщение строго указывает на необходимость наличия конструктора по умолчанию. Кажется, что единственный выход – передать уже созданный экземпляр в WCF хостинг, но это приводит к нескольким проблемам:

- Как мы можем это сделать, если мы размещаем сервис на Internet Information Services (IIS)?
- Для этого требуется, чтобы сервис запускался в `Single InstanceContextMode`, который по разным причинам не желателен.

Отличная новость – сообщение об исключении просто вводит нас в заблуждение. Существуют другие способы разрешения внедрения через конструктор в рамках WCF.

Расширяемость WCF

WCF обладает множеством возможностей для расширяемости, но когда дело доходит до механизма внедрения зависимостей, нам всего лишь необходимо иметь представление об интерфейсе `IInstanceProvider` и о поведении контрактов. *Поведение контракта* – это шов в WCF, который позволяет нам изменять то, как ведет себя данный контракт (в данном случае – сервис).

`IInstanceProvider` – это интерфейс, который определяет, как создаются экземпляры сервиса (и высвобождаются). Ниже приведено определение интерфейса во всей его красе:

```
public interface IInstanceProvider
{
    object GetInstance(InstanceContext instanceContext);
    object GetInstance(InstanceContext instanceContext, Message message);
    void ReleaseInstance(InstanceContext instanceContext, object instance);
}
```


Две перегрузки `GetInstance` ответственны за создание соответствующего экземпляра сервиса, а `ReleaseInstance` при необходимости предоставляет возможность для очистки.

Реализация по умолчанию выполняет поиск конструктора по умолчанию для типа сервиса, но мы можем заменить ее какой-либо другой реализацией, использующей механизм внедрения зависимостей. Рисунок 7-7 иллюстрирует весь поток при получении размещенным сервисом сообщения.

Рисунок 7-7: Когда в сервисную операцию поступает сообщение (запрос), WCF определяет, какой тип CLR реализует сервис. WCF просит `ServiceHostFactory` создать соответствующий `ServiceHost`, который может разместить запрашиваемый сервис.

`ServiceHost` выполняет свою часть работы, применяя поведения и создавая запрашиваемый экземпляр.

Когда мы размещаем WCF сервис в IIS, `ServiceHostFactory` обязателен, несмотря на то, что, если мы явным образом не определим альтернативу, будет использоваться реализация по умолчанию. Если мы размещаем сервис вручную, `ServiceHostFactory` все еще может быть полезной, но не является необходимой, потому что мы можем создать соответствующий `ServiceHost`, напрямую в коде.



Когда `ServiceHost` применяет поведения, он собирает их, по крайней мере, из трех различных мест перед тем, как их объединить:

- Атрибуты
- Файл `.config`
- Объекты, хранящиеся в оперативной памяти

Несмотря на то, что мы можем определить поведения в атрибутах, это не очень заманчивая для использования стратегия в тех случаях, когда дело касается механизма внедрения зависимостей, поскольку это означает, что мы компилируем в код конкретную стратегию создания с конкретными зависимостями. Конечный результат почти такой же, как если бы у нас прямо в сервисе присутствовали жестко закодированные зависимости, только гораздо более изощренное.

Конфигурационный файл может казаться максимально гибким, но это не так, потому что он не позволяет нам обязательно сконфигурировать зависимости, если мы хотим это сделать.

Объекты, хранящиеся в оперативной памяти, обеспечивают наилучшую гибкость, потому что мы можем выбрать вариант создания зависимостей напрямую в коде или на основании

настроек конфигурации. Если мы используем DI-контейнер, мы получаем обе опции бесплатно. Это означает, что мы должны создать пользовательский `ServiceHostFactory`, который создает экземпляры пользовательского `ServiceHost`, который снова может соединить нужный сервис со всеми его зависимостями.

Мы можем создать набор универсальных классов, которые делают это на основании выбранного DI-контейнера, или используют один из уже реализованных, многократно используемых `ServiceHostFactory`s. Также мы можем создать специальный `ServiceHostFactory` для конкретного сервиса. Поскольку создание специального `ServiceHostFactory` является наилучшей иллюстрацией процесса, в следующем примере используется специализированная фабрика.

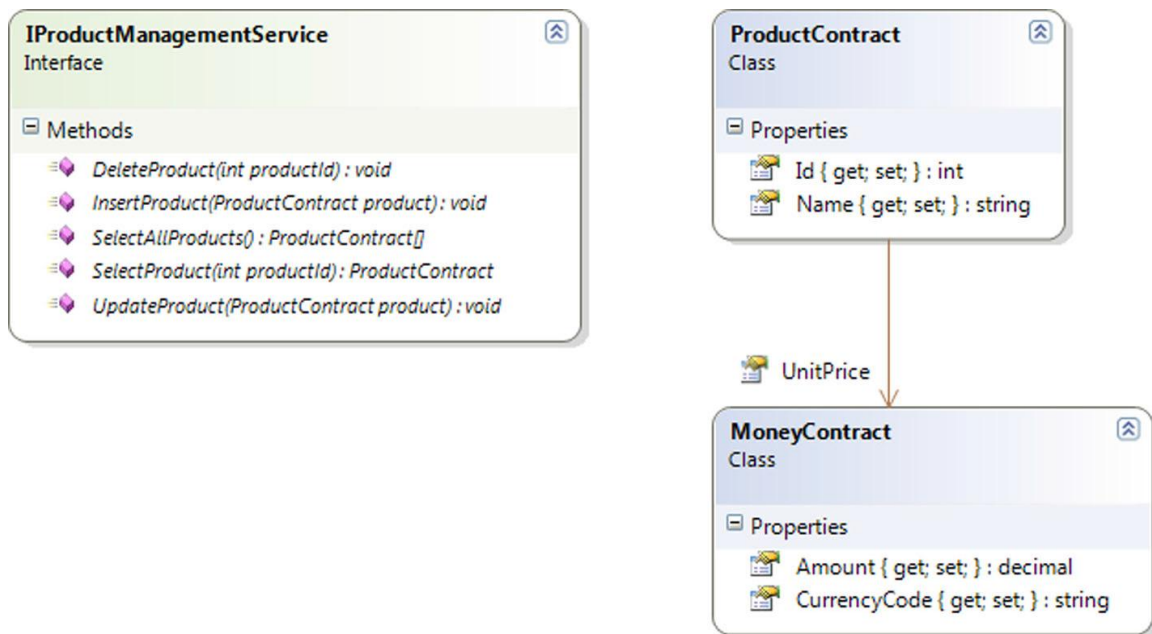
Пример: подключение сервиса управления продуктами

В качестве примера представьте себе, что вас попросили расширить публичное приложение Commerce WCF сервисом, который раскрывает операции, позволяющие другим приложениям управлять данными товара. Это позволяет вам вмонтировать богатого по функциям клиента (вы будете делать это в следующем разделе) или пакетное задание для того, чтобы управлять данными товара.

Знакомство с сервисом управления товарами

Для упрощения примера давайте предположим, что вы хотите раскрыть простые операции Create, Read, Update и Delete (CRUD). Рисунок 7-8 демонстрирует диаграмму сервисов и связанных контрактов данных.

Рисунок 7-8: `IProductManagementService` — это WCF сервис, который определяет простые CRUD операции для товаров. Он использует связанные `ProductContract` и `MoneyContract` для раскрытия этих операций. Несмотря на то, что это не продемонстрировано на диаграмме, все три типа помечены обычными WCF атрибутами: `ServiceContract`, `OperationContract`, `DataContract` и `DataMember`.



Поскольку у вас уже есть существующая доменная модель, вам хотелось бы реализовать этот сервис путем расширения доменной модели, и раскрыть ее операции посредством ее WCF контракта. Точные детали не важны; достаточно сказать, что вы расширяете абстрактный класс `ProductRepository`, который вы видели в предыдущих главах.

Подсказка

Несмотря на то, что я не хочу проводить вас здесь по всему доменному коду, вы можете просмотреть детали в коде, загруженном из книги.

Доменная модель представляет товар как `Entity Product`, а сервисный контракт раскрывает его операции в терминах *Data Transfer Object* (DTO) `ProductContract`. Для преобразований между этими двумя различными типами вы также вводите интерфейс под названием `IMapper`.

Итоговый результат заключается в том, что вы завершаете реализацию сервиса двумя зависимостями, и поскольку они являются обязательными, вам хотелось бы использовать внедрение через конструктор. Ниже приведена сигнатура конструктора сервиса:

```
public ProductManagementService(ProductRepository repository,
    IMapper mapper)
```

До настоящего времени мы счастливо игнорировали слона в комнате: как нам получить WCF для того, чтобы корректно подключить экземпляр `ProductManagementService`?

Присоединение `ProductManagementService` к WCF

Как показано на рисунке 7-7, *Composition Root* в WCF – это триплет `ServiceHostFactory`, `ServiceHost` и `InstanceProvider`. Чтобы подключить к сервису механизм внедрения через конструктор, мы должны обеспечить пользовательские реализации всех трех этих компонентов.

Подсказка

Вы можете написать полностью пригодные для повторного использования реализации, которые заворачивают ваш любимый DI-контейнер в эти три типа и используют их для реализации `InstanceProvider`. Многие люди уже делали это, поэтому вы, скорее всего, можете найти готовый набор для выбранного вами DI-контейнера.

Примечание

В данном примере реализуется контейнер, жестко присоединенный при помощи *Booq Man's DI*. Я решил инкапсулировать жестко закодированные зависимости в пользовательском классе контейнера для того, чтобы дать вам хорошее представление о том, как создать многократно используемое решение на основании конкретного DI-контейнера.

Entity vs. DTO

Предыдущий параграф подбросил вам немного жаргона, поэтому давайте вкратце рассмотрим, что подразумевается под `Entity` и `DTO`.

Entity – это термин проблемно-ориентированного проектирования (*Domain-Driven Design*), который охватывает объект *Domain*, имеющий долгосрочную идентификацию, которая не относится к конкретному экземпляру объекта. Это может показаться абстрактным и теоретическим, но это означает, что *Entity* представляет собой объект, который живет за пределами произвольных битов памяти. Любой экземпляр .NET объекта имеет внутренний адрес (идентификацию), но *Entity* обладает идентификацией, которая обитает по ту сторону жизненного процесса. Мы часто используем базы данных и первичные ключи для идентификации *Entities* и для того, чтобы убедиться, что мы можем сохранять и читать их, даже если хост-компьютер будет перезагружен.

Доменный объект *Product* – это *Entity*, поскольку у сущности товара гораздо более продолжительный жизненный цикл, нежели у единичного процесса, и мы используем ID товара для его идентификации в *ProductRepository*.

Data Transfer Object (DTO), с другой стороны, существует только для того, чтобы быть переданным с одного уровня приложения на другой. Несмотря на то, что *Entity* может инкапсулировать большую часть поведения, DTO – это структура данных без поведения.

При демонстрации доменной модели внешним системам мы часто поступаем так с сервисами и DTOs, поскольку мы никогда не можем быть уверены в том, что другая система сможет использовать нашу типовую систему (она может даже не использовать .NET). В таких ситуациях нам всегда нужно выполнять преобразования между *Entity* и DTOs.

Давайте начнем с пользовательского *ServiceHostFactory*, который является настоящей точкой входа в WCF сервис. Следующий листинг демонстрирует реализацию.

Листинг 7-4: Пользовательский *ServiceHostFactory*

```
1. public class CommerceServiceHostFactory : ServiceHostFactory
2. {
3.     private readonly ICommerceServiceContainer container;
4.     public CommerceServiceHostFactory()
5.     {
6.         this.container =
7.             new CommerceServiceContainer();
8.     }
9.     protected override ServiceHost CreateServiceHost(
10.         Type serviceType, Uri[] baseAddresses)
11.     {
12.         if (serviceType == typeof(ProductManagementService))
13.         {
14.             return new CommerceServiceHost(
15.                 this.container,
16.                 serviceType, baseAddresses);
17.         }
18.         return base.CreateServiceHost(serviceType, baseAddresses);
19.     }
20. }
21.
```

Строка 6-7: Создает экземпляр контейнера

Строка 14-16: Создает пользовательский *ServiceHost*

Пользовательский `CommerceServiceHostFactory` унаследован от `ServiceHostFactory` с единственной целью – присоединить экземпляры `ProductManagementService`. Он использует пользовательский `CommerceServiceContainer` для выполнения реальной работы, поэтому создает экземпляр контейнера в его конструкторе. Вы можете легко расширить этот пример для того, чтобы использовать настоящий DI-контейнер, создавая и конфигурируя вместо этого экземпляр данного контейнера.

При получении запроса о создании `ServiceHost` он возвращает новый `CommerceServiceHost` с настроенным контейнером, если запрашиваемый тип сервиса подходит. `CommerceServiceHost` ответственен за определение подходящих поведений для всех типов сервисов, на которых он размещается. В данном примере вы хотите добавить всего лишь одно поведение, которое передает желаемый `IInstanceProvider` в сервисы. Вы можете завершить всю эту работу в конструкторе, продемонстрированном ниже, а базовый класс позаботится обо всем остальном.

Листинг 7-5: Пользовательский `ServiceHost`

```
1. public class CommerceServiceHost : ServiceHost
2. {
3.     public CommerceServiceHost(ICommerceServiceContainer container,
4.         Type serviceType, params Uri[] baseAddresses)
5.         : base(serviceType, baseAddresses)
6.     {
7.         if (container == null)
8.         {
9.             throw new ArgumentNullException("container");
10.        }
11.        var contracts = this.ImplementedContracts.Values;
12.        foreach (var c in contracts)
13.        {
14.            var instanceProvider =
15.                new CommerceInstanceProvider(
16.                    container);
17.            c.Behaviors.Add(instanceProvider);
18.        }
19.    }
20. }
```

Строка 14-16: Создает `InstanceProvider`

Строка 17: Добавляет `InstanceProvider` в качестве поведения

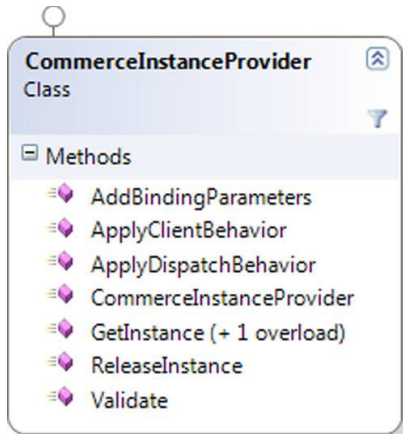
Класс `CommerceServiceHost` наследуется от `ServiceHost`, который является конкретным классом, выполняющим всю тяжелую работу. В большинстве случаев вы будете размещать только один вид сервиса (в данном случае, `ProductManagementService`), но вам разрешено размещать самые разнообразные сервисы; это означает, что вы должны добавить ко всем этим сервисам `IInstanceProvider`. Свойство `ImplementedContracts` – это словарь, поэтому вы можете выполнить цикл по его `Values`, чтобы пометить их всех.

Для каждого вида сервиса вы инициализируете новый экземпляр пользовательского класса `CommerceInstanceProvider` вместе с контейнером. Поскольку он дублируется в качестве поведения, вы можете добавить его в `Behaviors` сервиса.

Последней составляющей пользовательского WCF триплета является `CommerceInstanceProvider`, который дублирует как `IInstanceProvider`, так и

`OperationContractBehavior`. Это простая реализация, но поскольку она реализует два различных интерфейса со сложными сигнатурами, она может выглядеть слегка устрашающей, если вы видите ее впервые. Вместо нее я продемонстрирую код, отнимающий мало времени; рисунок 7-9 предоставляет краткий обзор.

Рисунок 7-9: `CommerceInstanceProvider` реализует как `IInstanceProvider`, так и `OperationContractBehavior`, поэтому вам нужно реализовать семь методов. Вы можете оставить три этих метода пустыми, а остальные четыре являются однострочными.



Листинг 7-6 демонстрирует объявление класса и конструктор. Здесь ничего не происходит, кроме использования `Constructor Injection` для внедрения контейнера. Обычно мы используем механизм внедрения через конструктор, для того, чтобы объявить DI-контейнеру, что классу нужны некоторые зависимости, но в данном случае это делать поздно, поскольку вы внедряете сам контейнер. Это обычно пахнет большим кодом, потому что он чаще всего указывает на намерение использовать анти-паттерн `Service Locator`, но здесь это необходимо, поскольку вы реализуете `Composition Root`.

Листинг 7-6: Объявление класса `CommerceInstanceProvider` и конструктор

```

1. public partial class CommerceInstanceProvider :
2.     IInstanceProvider, IContractBehavior
3.     {
4.         private readonly ICommerceServiceContainer container;
5.         public CommerceInstanceProvider(
6.             ICommerceServiceContainer container)
7.         {
8.             if (container == null)
9.             {
10.                 throw new ArgumentNullException("container");
11.             }
12.             this.container = container;
13.         }
14.     }

```

Строка 2: Реализует WCF интерфейсы

Строка 4-13: `Constructor Injection`

`CommerceInstanceProvider` реализует как `IInstanceProvider`, так и `OperationContractBehavior`. Вы дополняете контейнер посредством стандартного `Constructor Injection`. В данном примере вы используете пользовательский `CommerceServiceContainer`, но замена его универсальным DI-контейнером – обычная практика.

Реализация `IInstanceProvider` в следующем листинге используется рабочей средой WCF для создания экземпляров класса `ProductManagementService`.

Листинг 7-7: Реализация `IInstanceProvider`

```
1. public object GetInstance (InstanceContext instanceContext, Message
   message)
2. {
3.     return this.GetInstance(instanceContext);
4. }
5. public object GetInstance (InstanceContext instanceContext)
6. {
7.     return this.container
8.         .ResolveProductManagementService ();
9. }
10. public void ReleaseInstance (InstanceContext instanceContext,
   object instance)
11. {
12.     this.container.Release(instance);
13. }
14. }
```

Строка 3: Делегирует полномочия перегрузке

Строка 7-8: Использует контейнер для преобразования

Строка 13: Просит контейнер высвободить экземпляр

Рабочая среда WCF вызывает один из методов `GetInstance` для того, чтобы получить экземпляр запрашиваемого вида сервиса, поэтому вы просите контейнер присоединить к `ProductManagementService` всего его необходимые зависимости.

При завершении операции сервиса рабочая среда WCF просит вас высвободить экземпляр, и вы снова делегируете эту работу контейнеру.

Остальная часть `CommerceInstanceProvider` – это реализация `IContractBehavior`. Единственная причина для реализации этого интерфейса – позволить вам добавить его в список поведений, как это продемонстрировано в листинге 7-5. Все методы интерфейса `IContractBehavior` возвращают `void`, поэтому вы можете оставить большинство из них пустыми, поскольку вам не нужно их реализовывать.

Следующий листинг демонстрирует реализацию единственного метода, о котором вам стоит позаботиться.

Листинг 7-8: Основная реализация `IContractBehavior`

```
public void ApplyDispatchBehavior(
    ContractDescription contractDescription, ServiceEndpoint endpoint,
    DispatchRuntime dispatchRuntime)
{
    dispatchRuntime.InstanceProvider = this;
}
```

В этом методе вам нужно сделать всего лишь одну очень простую вещь. Рабочая среда WCF вызывает этот метод и передает экземпляр `DispatchRuntime`, который позволяет вам сказать методу о том, что он должен использовать эту конкретную реализацию `IInstanceProvider` – помните, что `CommerceInstanceProvider` также реализует

`IInstanceProvider`. Рабочая среда WCF теперь знает, какой `IInstanceProvider` использовать, и может впоследствии вызвать метод `GetInstance`, продемонстрированный в листинге 7-7.

Кажется, чтобы разрешить механизм внедрения зависимостей для WCF необходимо реализовать большое количество кода, а я даже не показал вам реализацию `CommerceServiceContainer`.

Подсказка

Помните, что вы можете легко написать многократно используемые версии этих трех классов, обертывающих ваш любимый DI-контейнер и упаковывающих эту реализацию в библиотеку. Это делают многие разработчики, поэтому вы, скорее всего, можете найти подходящую готовую библиотеку в интернете.

Контейнер – это последний фрагмент WCF DI пазла.

Реализация специализированного контейнера

`CommerceServiceContainer` – это специализированный контейнер с единственной целью присоединения класса `ProductManagementService`. Помните, что для этого класса нужны экземпляры `ProductRepository` и `IContractMapper` в качестве зависимостей.

Если исключить всю WCF инфраструктуру, то контейнер свободен для того, чтобы сконцентрироваться на присоединении диаграммы зависимостей.

Примечание

Кроме соблюдения принципа единичной ответственности, такая концепция разделения демонстрирует, что вы можете заменить этот специализированный контейнер универсальным DI-контейнером, поскольку WCF-специфичный код отсутствует.

Метод `ResolveProductManagementService` связывает экземпляр с Poor Man's DI, что продемонстрировано ниже.

Листинг 7-9: Преобразование `ProductManagementService`

```
1. public IProductManagementService ResolveProductManagementService()
2. {
3.     string connectionString =
4.         ConfigurationManager.ConnectionStrings
5.         ["CommerceObjectContext"].ConnectionString;
6.     ProductRepository repository =
7.         new SqlProductRepository(connectionString);
8.     IContractMapper mapper = new ContractMapper();
9.     return new ProductManagementService(repository,
10.        mapper);
11. }
```

Строка 6-7: Создает репозиторий товаров

Строка 8: Создает преобразователь контрактов

В некотором смысле, когда дело доходит до преобразования диаграммы зависимостей, часто бывает выгодно вернуться назад. Вы знаете, что вам нужно вернуть экземпляр `ProductManagementService` с экземплярами `ProductRepository` и `IContractMapper`. Экземпляр `IContractMapper` легко создать, но для того, чтобы создать `ProductRepository` нужно потрудиться побольше.

Вам хотелось бы использовать `SqlProductRepository`, но для этого вам нужна строка соединения, которую вы можете прочесть из файла `web.config`.

Если вы хотите разместить сервис в вашем собственном приложении, то сейчас вы можете это сделать путем создания нового экземпляра класса `CommerceServiceHostFactory` и вызова его метода `CreateServiceHost` с корректными параметрами. Метод вернет экземпляр `CommerceServiceHost`, который вы можете открыть, и будет выполнять остальную часть работы за вас, а также размещать `ProductManagementService`.

Тем не менее, если вы хотите разместить сервис на IIS, то вы должны выполнить еще один шаг.

Хостинг `ProductManagementService` в IIS

В IIS мы не создаем новые экземпляры `CommerceServiceHostFactory` вручную. Вместо этого мы должны сообщить IIS о том, чтобы он сделал это за нас. Это может быть сделано в `.svc` файле путем добавления атрибута `Factory`:

```
<%@ ServiceHost=""
    Factory = "Ploeh.Samples.CommerceService.CommerceServiceHostFactory,
    Ploeh.Samples.CommerceService"
    Service = "Ploeh.Samples.CommerceService.ProductManagementService"
%>
```

Данный `.svc` файл дает IIS указание использовать `CommerceServiceHostFactory` всякий раз, когда ему нужно создавать экземпляр класса `ProductManagementService`. То, что рассматриваемый `ServiceHostFactory` имеет конструктор по умолчанию, является условием, но в данном примере это именно так.

Разрешение DI в WCF сложнее, нежели это должно было быть, но, по крайней мере, это возможно, и конечного результата вполне достаточно. Мы можем использовать любой желаемый DI-контейнер, и завершаем работу, приобретая правильный `Composition Root`.

Некоторые фреймворки не предоставляют нам подходящие швы, которые позволили бы нам такую роскошь. Тем не менее, до того, как мы рассмотрим один такой известный фреймворк, давайте расслабимся и рассмотрим более простой фреймворк.

7.4. Построение WPF приложений

Если вы думали, что создавать WCF сервис было сложно (как думал и я), то вы оцените, что создавать Windows Presentation Foundation (WPF) приложение почти настолько же просто, как и консольное приложение.

Точка входа в WPF приложение является довольно очевидной и не сложной, и, несмотря на то, что она не предоставляет швов, которые явно предназначены для разрешения DI, мы можем с легкостью сформировать приложение любым предпочитаемым нами способом.

WPF композиция

Точка входа в WPF приложение определяется в его классе App. Как и большинство других классов WPF, этот класс разбит на два файла: *App.xaml* и *App.xaml.cs*. Мы можем определить, что происходит на стадии начальной загрузки в обоих файлах в зависимости от наших потребностей.

При создании нового WPF проекта в Visual Studio файл *App.xaml* определяет атрибут `StartupUri`, который устанавливает, какое окно демонстрируется при запуске приложения – в данном примере `Window1`:

```
<Application x:Class="MyWpfApplication.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
</Application>
```

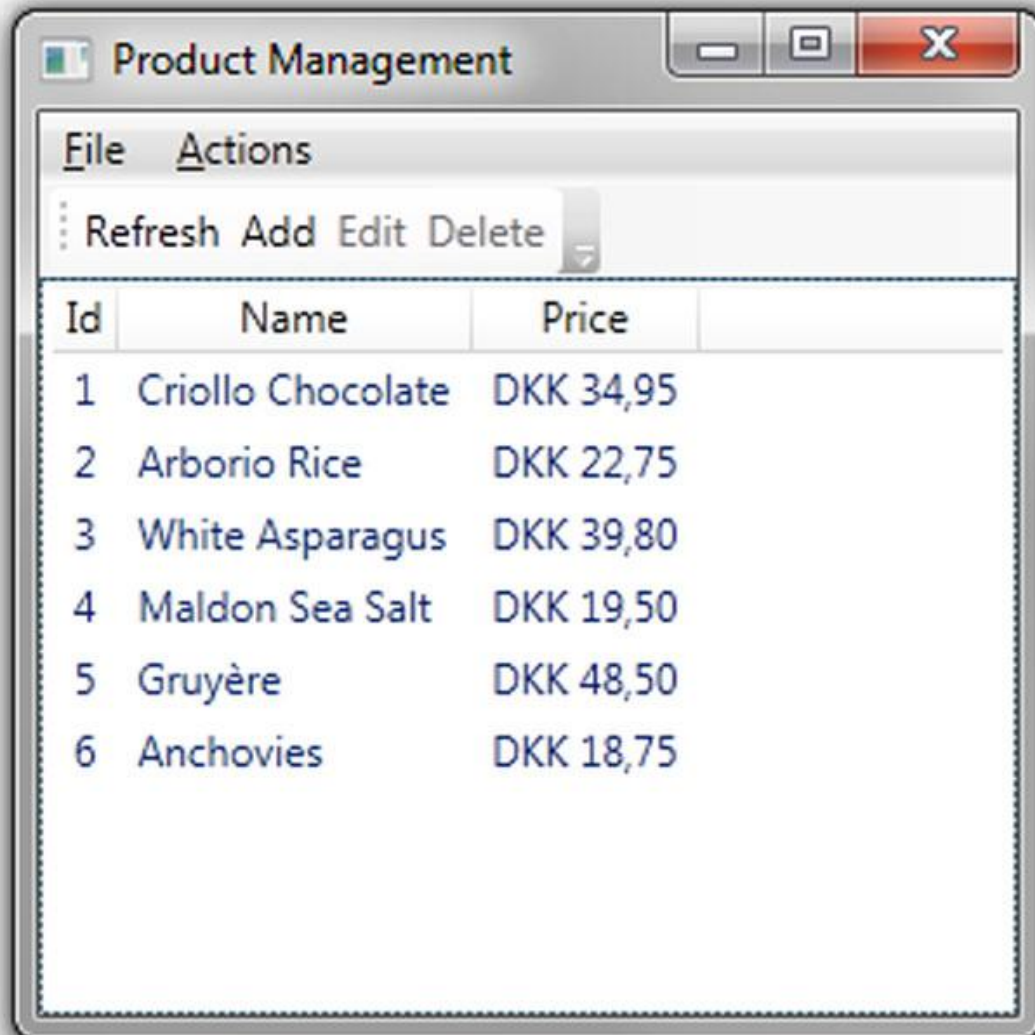
Смысл данного стиля объявления заключается в том, что объект `Window1` создается и демонстрируется без какого-либо дополнительного контекста. Когда вы хотите добавить зависимости к окну, наиболее подходящим может стать более явный подход. Вы можете удалить атрибут `StartupUri` и присоединить окно с помощью переопределения метода `OnStartup`. Это позволяет вам полностью присоединить первое окно до того, как оно будет продемонстрировано, но вам придется за это заплатить: вы должны не забыть явно вызвать метод `Show` для окна.

Метод `OnStartup`, таким образом, становится `Composition Root` приложения. Вы можете использовать DI-контейнер или `Root Man's DI` для создания окна. В следующем примере используется `Root Man's DI` для того, чтобы проиллюстрировать, что вам не приходится полагаться на возможности какого-либо конкретного DI-контейнера.

Пример: присоединение ценного клиента управления товарами

В предыдущем примере разрабатывался веб-сервис, который мы можем использовать для управления каталогом товаров в шаблонном приложении `Commerce`. В данном примере мы создадим WPF приложение, которое использует этот веб-сервис для управления товарами. Рисунок 7-10 демонстрирует скриншот этого приложения.

Рисунок 7-10: Главное окно приложения Product Management – это список товаров. Вы можете добавлять новые товары, редактировать существующие или удалять их. При добавлении или редактировании товаров используется модальное окно редактирования. Все операции реализуются посредством вызова соответствующей операции для веб-сервиса управления товарами из раздела 7.3.2.



Приложение реализует Model View ViewModel (MVVM) подход и содержит три уровня, которые продемонстрированы на рисунке 7-11. Обычно мы держим ту составляющую, в которой находится большая часть логики, изолированно от других модулей – в данном примере `PresentationLogic.ProductManagementClient` – это *humble*-исполнитель (*humble executable*), который выполняет несколько большую работу, нежели просто определяет пользовательский интерфейс и делегирует реализацию другим модулям.

Рисунок 7-11: Приложение состоит из трех отдельных блоков. Блок `ProductManagementClient` – исполняемый, и включает в себя пользовательский интерфейс, реализованный в XAML без выделенного кода (code-behind). Библиотека `PresentationLogic` содержит ViewModels и опорные классы, а библиотека `ProductWcfAgent` включает в себя Adapter между пользовательской абстракцией `IProductManagementAgent` и конкретным WCF прокси, который используется для взаимодействия с веб-сервисом управления товарами. Стрелки-указатели зависимостей означают, что `ProductManagementClient` выступает в роли Composition Root, поскольку он соединяет вместе остальные модули.



Благодаря подходу MVM мы передаем `ViewModel` в свойство `DataContext` главного окна, а механизм связывания данных и движок шаблонизации данных заботятся о корректном представлении данных, как только мы вплетаем новые `ViewModels` или изменяем данные существующих `ViewModels`.

MVVM

Model View ViewModel (MVVM) – это паттерн проектирования, для которого отлично подходит WPF. Он разделяет код пользовательского интерфейса на три отдельных ответственности.

Model – это основополагающая модель приложения. Ею часто, но не всегда, является доменная модель. Она часто состоит из `POCO`-объектов. В текущем примере доменная модель реализуется в веб-сервисе, поэтому на данном уровне у вас нет истинной доменной модели. Тем не менее, приложение функционирует с абстракциями, находящимися поверх прокси веб-сервиса, и это и есть ваша *Model*. Обратите внимание на то, что *Model* обычно выражается UI-нейтральным способом. Это не предполагает, что *Model* будет раскрыта напрямую пользовательским интерфейсом, поэтому она не раскрывает никакой WPF-специфичной функциональности.

View – это рассматриваемый нами пользовательский интерфейс. В WPF мы можем официально выразить *View* в XAML и использовать механизм связывания данных и движок шаблонизации данных для представления данных. Можно выразить *Views* без использования выделенного кода.

ViewModel – мост между View и Model. Каждый *ViewModel* – это класс, который преобразовывает и раскрывает Model конкретным специфическим способом. В WPF это означает, что *ViewModel* может раскрывать списки как *ObservableCollections*, и тому подобное.

Внедрение зависимостей в главный *ViewModel*

MainWindow содержит только XAML разметку и не содержит никакого пользовательского выделенного кода. Зато он использует механизм связывания данных для отображения данных на экране и управления пользовательскими командами. Для того чтобы это позволить мы должны передать *MainWindowViewModel* в его свойство *DataContext*.

MainWindowViewModel раскрывает такие данные, как список товаров, а также команды создания, обновления или удаления товара. Возможность такой функциональности зависит от сервиса, который обеспечивает доступ к каталогу товаров: абстракция *IProductManagementAgent*.

Помимо *IProductManagementAgent* для *MainWindowViewModel* также необходим сервис, который он может использовать для того, чтобы контролировать его *оконную среду* (*windowing environment*), например, демонстрацию модальных диалоговых окон. Эта зависимость называется *IWindow*.

MainWindowViewModel использует паттерн *Constructor Injection* со следующей сигнатурой конструктора:

```
public MainWindowViewModel(IProductManagementAgent agent, IWindow window)
```

Для соединения приложения мы должны создать *MainWindowViewModel* и передать его в свойство *DataContext* экземпляра *MainWindow*.

Соединение *Main Window* и *MainWindowViewModel*

Данному примеру придает остроты тот факт, что для корректной реализации *IWindow*, вам нужен указатель на реальное WPF окно (*MainWindow*); но для *ViewModel* необходим *IWindow*, а свойство *DataContext* экземпляра *MainWindow* должно быть *ViewModel*. Другими словами, вы получаете *циклическую зависимость*.

В главе 6 мы имели дело с циклическими зависимостями и прошли по соответствующей части этого конкретного примера, поэтому я не буду повторять это в данной главе. Достаточно сказать, что вводится *MainWindowViewModelFactory*, который является ответственным за создание экземпляров *MainWindowViewModel*.

Вы используете эту фабрику в рамках реализации *IWindow* под названием *MainWindowAdapter* для того, чтобы создать *MainWindowViewModel* и передать его в свойство *DataContext* экземпляра *MainWindow*:

```
var vm = this.vmFactory.Create(this);  
this.WpfWindow.DataContext = vm;
```

Переменная члена *vmFactory* – это экземпляр *IMainWindowViewModelFactory*, и вы передаете в его метод *Create* экземпляр содержащегося класса, который реализует

IWindow. Итоговый экземпляр ViewModel затем передается в DataContext WpfWindow, который является экземпляром MainWindow.

Примечание

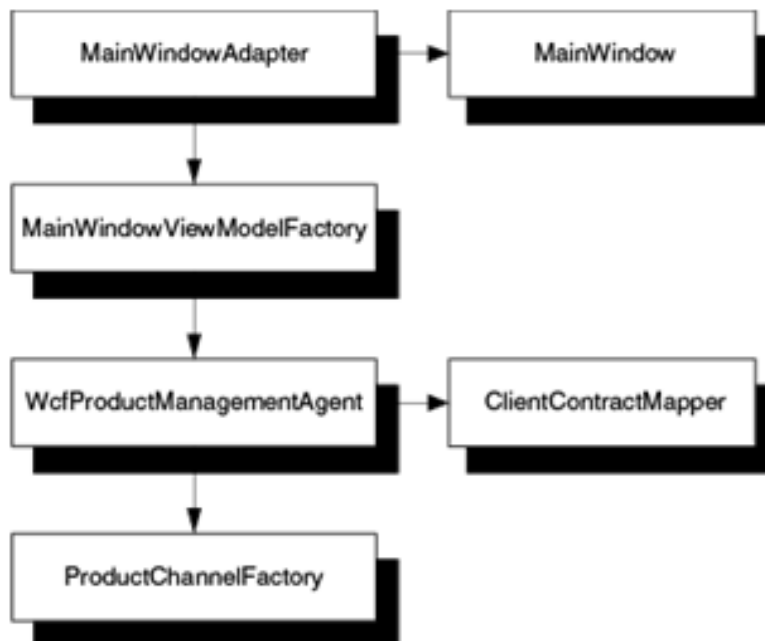
Я не рассматриваю детали, поскольку мы рассматривали их в главе 6. Вернитесь к этой главе и прочтите раздел о циклических зависимостях, если вам нужно освежить в памяти то, что сейчас происходит.

Подсказка

Для механизма связывания WPF данных необходимо, чтобы мы передавали зависимость (ViewModel) в свойство DataContext. По моему мнению, это неправильное использование PropertyInjection, поскольку это сигнализирует о том, что зависимость является необязательной, а это абсолютно не так. Тем не менее, WPF 4 вводит нечто, имеющее название XamlSchemaContext, который может использоваться в качестве шва, который, в свою очередь, дает нам большую гибкость в тех ситуациях, когда дело доходит до создания экземпляров Views на основании разметки.

Рисунок 7-12 демонстрирует окончательную диаграмму зависимостей приложения.

Рисунок 7-12: Диаграмма зависимостей MainWindowAdapter, который является основным объектом приложения. MainWindowAdapter использует MainWindowViewModelFactory для создания соответствующего ViewModel и передачи его в MainWindow. Для того чтобы создать MainWindowViewModel фабрике нужно передать WcfProductManagementAgent во ViewModel. Этот "посредник" является адаптером между IProductManagementAgent и WCF прокси. Он требует, чтобы ProductChannelFactory создал экземпляры WCF прокси, а также IClientContractMapper, который может выполнять преобразование между ViewModels и WCF контрактами данных.



Теперь, когда вы идентифицировали все строительные блоки приложения, вы можете скомпоновать их. Для того чтобы сохранить Poor Man's DI код симметричным, используя

при этом DI-контейнер, я реализовал это в виде `Resolve` метода специализированного класса контейнера. В следующем листинге продемонстрирована реализация.

Листинг 7-10: Композиция главного окна

```
public IWindow ResolveWindow()
{
    IProductChannelFactory channelFactory =
        new ProductChannelFactory();
    IClientContractMapper mapper =
        new ClientContractMapper();
    IProductManagementAgent agent =
        new WcfProductManagementAgent(
            channelFactory, mapper);
    IMainWindowViewModelFactory vmFactory =
        new MainWindowViewModelFactory(agent);
    Window mainWindow = new MainWindow();
    IWindow w =
        new MainWindowAdapter(mainWindow, vmFactory);
    return w;
}
```

В конечном итоге вы возвращаете экземпляр `IWindow`, реализованный `MainWindowAdapter`, а для этого вам нужны `WPFWindow` и `IMainWindowViewModelFactory`. Первым окном, которое вы должны продемонстрировать пользователям, должно быть `MainWindow`, поэтому именно его вы и передаете в `MainWindowAdapter`.

`MainWindowViewModelFactory` использует паттерн `Constructor Injection` для запроса `IProductManagementAgent`, поэтому вы должны скомпоновать `WcfProductManagementAgent` с двумя его зависимостями.

Окончательный `MainWindowAdapter`, возвращаемый из метода, обертывает `MainWindow`, поэтому, когда мы вызываем метод `Show`, он делегирует полномочия методу `Show` `MainWindow`. Это именно то, что вы и будете делать в `Composition Root`.

Реализация `Composition Root`

Теперь, когда вы знаете, как подключить приложение, вам нужно всего лишь сделать это в правильном месте. Как описывалось в предыдущем разделе, вам для начала нужно открыть `App.xaml` и удалить атрибут `StartupUri`, поскольку вы хотите самостоятельно явным образом компоновать начальное окно запуска.

После того, как вы это сделали, вам нужно только переопределить метод `OnStartup` в `App.xaml.cs` и вызвать контейнер.

Листинг 7-11: Реализация `WPF Composition Root`

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    var container =
        new ProductManagementClientContainer();
    container.ResolveWindow().Show();
}
```

В данном примере вы используете специализированный `ProductManagementClientContainer`, но вы также могли использовать и универсальный DI-контейнер, например, `Unity` или `StructureMap`. Вы просите контейнер преобразовать экземпляр `IWindow`, а затем вызвать его метод `Show`. Возвращаемый экземпляр `IWindow` — это `MainWindowAdapter`, когда вы вызываете его метод `Show`, он вызывает метод `Show` инкапсулированного `MainWindow`, который становится причиной того, что желаемое окно демонстрируется пользователю.

WPF предлагает простое место для `Composition Root`. Все, что вам нужно сделать — удалить `StartupUri` из *App.xaml*, переопределить `OnStartup` в *App.xaml.cs* и скомпоновать там приложение.

До настоящего момента вы встречались с примерами, в которых фреймворки предоставляют швы, позволяющие нам перенять жизненный цикл экземпляров ключевых объектов (веб-страниц, экземпляров сервисов, окна и других). Во многих случаях это довольно просто; но даже когда это настолько усложняется, как например, в WCF, мы все еще можем достичь цели и реализовать подлинный механизм внедрения зависимостей, не подвергая при этом риску наши принципы.

Некоторые фреймворки, тем не менее, не предоставляют нам такой роскоши.

7.5. Построение ASP.NET приложений

Некоторые фреймворки настаивают на создании и управлении жизненным циклом написанных нами классов. Самый популярный фреймворк – ASP.NET (Web Forms, в противоположность MVC).

Примечание

Некоторые другие фреймворки, разделяющие эту особенность, – это Microsoft Management Console (MMC), управляемый SDK, и такая недавняя разработка, как PowerShell.

Наиболее очевидным симптомом таких фреймворков является то, что, чтобы им подойти, наши классы должны обладать конструктором по умолчанию. В ASP.NET, например, любой реализуемый нами класс Page должен обладать непараметризованным конструктором. Мы не можем использовать Constructor Injection в рамках этих фреймворков, поэтому давайте рассмотрим наши возможности.

ASP.NET композиция

Паттерн Constructor Injection был бы предпочтительным, поскольку он позволял бы удостовериться, что наши классы Page будут правильно инициализироваться совместно с их зависимостями. Так как это невозможно, мы должны выбирать между следующими альтернативными вариантами:

- Перемещать и дублировать наши Composition Root'ы в пределах каждого класса Page.
- Использовать Service Locator для того, чтобы преобразовать все зависимости в пределах каждого класса Page.

Тем не менее, не забывайте о том, что Service Locator – это анти-паттерн, поэтому такой вариант не желателен. Наилучшая альтернатива – пойти на компромисс в вопросе расположения нашего Composition Root.

В идеале, мы бы предпочли сценарий, продемонстрированный на рисунке 7-13, при котором мы имеем всего один Composition Root для приложения, но в ASP.NET это невозможно, поскольку мы не можем компоновать экземпляры Page извне. Другими словами, фреймворк Web Forms вынуждает нас компоновать приложение в пределах каждого Page.

Рисунок 7-13: В идеальном мире нам хотелось бы уметь компоновать объекты Page из Composition Root приложения. При получении запроса мы должны уметь использовать определенную конфигурацию зависимостей для того, чтобы скомпоновать соответствующий объект Page. Тем не менее, это невозможно, поскольку ASP.NET управляет жизненным циклом объектов Page от нашего имени.



Примечание

До настоящего времени я говорил только об объектах `Page`, но ASP.NET требует, чтобы большинство объектов обладали конструкторами по умолчанию в случае, если мы хотим использовать фреймворк. Еще одним примером является `Object Data Sources`. Обсуждение, имеющее место в данном разделе, в равной степени хорошо применяется ко всем остальным типам, которые должны обладать конструктором по умолчанию.

Чтобы обратиться к этой проблеме, мы должны подвергнуть риску наши идеалы, но компромисс в вопросе расположения наших `Composition Root`'ов кажется мне более безопасным, чем позволить `Service Locator` вступить в игру.

По существу мы преобразуем каждый `Page` в `Composition Root`, как это продемонстрировано на рисунке 7-14. Принцип единичной ответственности напоминает нам о том, что каждый класс должен иметь только одну ответственность; теперь, когда мы используем `Page` для того, чтобы скомпоновать все необходимые зависимости, мы должны делегировать ответственность за реализацию *реализатору* (*implementer*). Такой подход эффективно преобразует `Page` в *humble*-объект ("скромный" объект), раскрывая другие члены, такие, как обработчики события нажатия кнопки только для того, чтобы делегировать полномочия реализатору преобразованного `Page`.

Рисунок 7-14: В ASP.NET мы можем использовать точку входа в приложение (*global.asax*) для того, чтобы конфигурировать зависимости, но потом до того, как мы сможем продолжить композицию, нам придется ждать, пока фреймворк не создаст новый объект `Page`. В пределах каждого `Page` мы можем использовать сконфигурированные зависимости для того, чтобы скомпоновать реализатор, который реализует все поведение класса `Page`.



Различие между вариантом перемещения `Composition Root` в каждый класс и вариантом использования `Service Locator` – трудно уловимо. Разница заключается в том, что благодаря `Service Locator` мы можем преобразовывать зависимости каждого класса `Page` в индивидуальном порядке, и использовать их напрямую в пределах класса `Page`. Как обычно бывает с `Service Locator`, он склонен к размытию фокуса класса. Кроме того, довольно заманчиво сохранить контейнер и использовать его для того, чтобы преобразовать остальные зависимости как надо.

Чтобы противодействовать этой тенденции, важно использовать наш контейнер только для преобразования реализатора, а затем – забыть о нем. Это позволяет нам руководствоваться подходящими DI-паттернами (например, `Constructor Injection`) для остальной части кода приложения.

Несмотря на то, что это только теория, вы расслабитесь, услышав, что это легко реализовать. Лучше всего это иллюстрируется на примере.

Пример: подключение CampaignPresenter

Шаблонное приложение Commerce, которое вы знаете и любите, поддерживает такие возможности, как *скидки на товары* и *список рекомендуемых товаров*, но до настоящего момента вы не обеспечивали потребителей приложением, которое управляло бы этими аспектами. В данном примере мы рассмотрим, как компоновать ASP.NET приложение (продемонстрировано на рисунке 7-15), которое позволяет потребителю обновлять данные об акциях для товара.

Рисунок 7-15: Приложение CampaignManagement позволяет промышленным потребителям редактировать данные об акциях (*Featured* (Рекомендуемые) и *Discount Price* (Цена со скидкой)) для товара. Это ASP.NET приложение, построенное с элементом управления GridView, привязанным к элементу управления ObjectDataSource.

	Product Id	Product Name	Unit Price	Featured	Discount Price
Edit	1	Criollo Chocolate	34.9500	☒	
Edit	2	Arborio Rice	22.7500	☒	19.5000
Edit	3	White Asparagus	39.8000	☒	
Edit	4	Maldon Sea Salt	19.5000	☐	15.8500
Edit	5	Gruyère	48.5000	☒	
Edit	6	Anchovies	18.7500	☒	
Update Cancel	12	White Truffles	500.0000	☒	

Говоря простыми словами, это приложение состоит из единственного элемента управления, привязанного к ObjectDataSource. Источник данных – это класс конкретного приложения, который делегирует свое поведение доменной модели и, в конечном счете, передает его в библиотеку доступа к данным, которая хранит данные в базе данных SQL Server.

Вы все еще можете использовать *global.asax* для конфигурирования зависимостей, но вы должны отложить компоновку приложения до тех пор, пока не будут созданы Page и его ObjectDataSource. Конфигурирование зависимостей схоже с предыдущими примерами.

Конфигурирование зависимостей в ASP.NET

В ASP.NET точкой входа в приложение является файл *global.asax*, и, несмотря на то, что вы ничего не можете компоновать в этой точке, вы можете создать свой *mise en place*, подготавливая все для запуска приложения:

```
protected void Application_Start(object sender, EventArgs e)
{
    this.Application["container"] =
        new CampaignContainer();
}
```

Единственное, что вы здесь делаете, – создаете ваш контейнер и сохраняете его в Application Context, поэтому вы можете использовать его, когда вам это нужно. Это позволяет вам совместно использовать контейнер в рамках отдельных веб-запросов, которые являются существенными, если вам нужно сохранить некоторые зависимости на время жизненного цикла процесса (более подробно о жизненных циклах мы поговорим в главе 8).

Примечание

Как и во всех остальных примерах данной главы, я использую Poof Man's DI для того, чтобы продемонстрировать основные рассматриваемые принципы. CampaignContainer – это пользовательский класс, созданный специально для этого примера, но вы можете легко заменить его выбранным вами DI-контейнером.

Большинство различных Page и объектов источников данных могут совместно использовать один и тот же контейнер посредством обращения к Application Context. Тем не менее, этот подход несет за собой опасность неправильного использования его в качестве Service Locator, поскольку любой класс может потенциально получить доступ к Application Context. Таким образом, важно делегировать реализацию классам, которые не могут получить доступ к Application Context. На практике это означает делегирование полномочий классам, реализованным в других отдельных библиотеках, которые не ссылаются на ASP.NET.

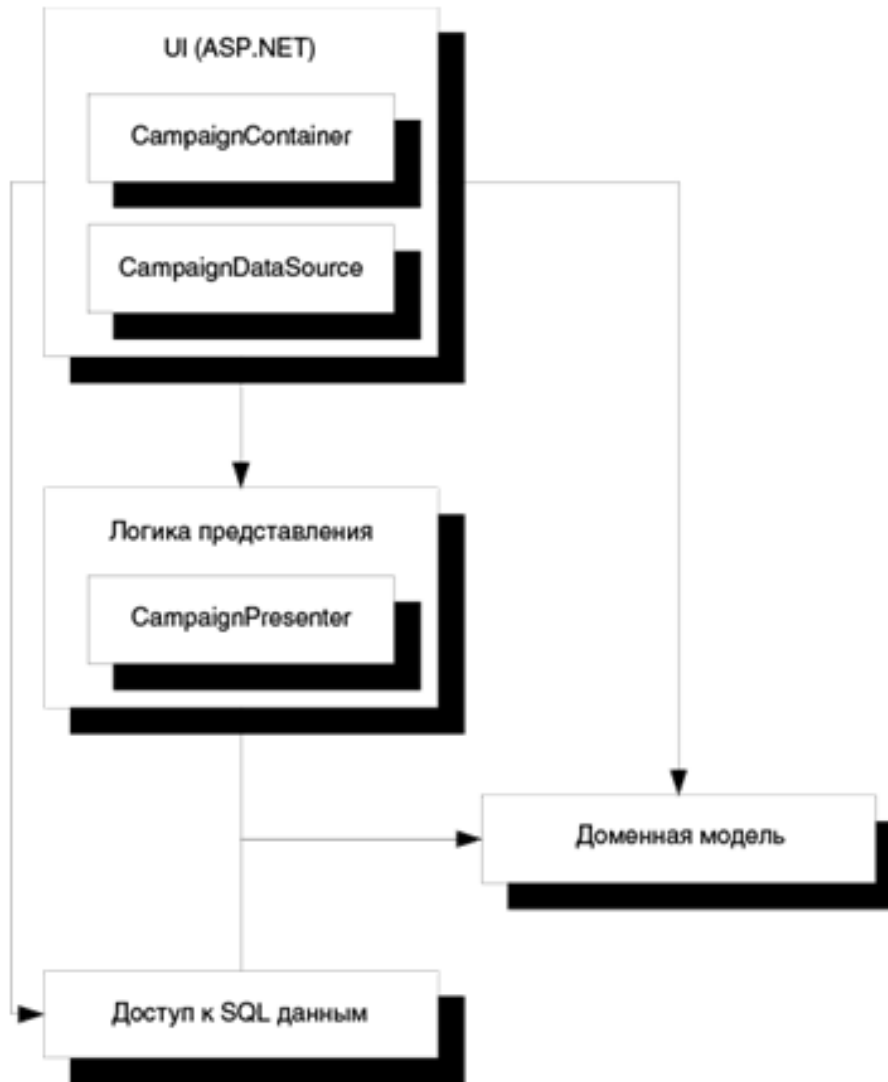
Примечание

Мы можем также продолжить свой путь слегка дисциплинированно, сдерживая себя от обращения к Application Context, за исключением реализации Composition Root. Это может хорошо подходить в тех случаях, когда все разработчики имеют опыт в написании слабо связанного кода; но если мы думаем, что некоторые члены команды могут не до конца понимать рассматриваемую проблему, мы можем удачно защитить код посредством использования отдельных библиотек. Раздел 6.5 описывает, как это сделать.

В текущем примере вы будете делегировать всю реализацию отдельной библиотеке логики отображения для того, чтобы убедиться, что никакие классы не обращаются напрямую к Application Context. Вы не позволяете библиотеке ссылаться на какую-либо сборку ASP.NET (например, System.Web).

Рисунок 7-16 демонстрирует частичное представление архитектуры приложения. Основной момент – это тот факт, что вы используете классы центральной части приложения (Default Page и CampaignDataSource) в качестве Composition Root'ов, которые преобразуют классы уровня "Логика представления" совместно с их зависимостями.

Рисунок 7-16: Центральная часть приложения CampaignManagement — это единственная составляющая приложения, ссылающаяся на ASP.NET. Класс CampaignDataSource имеет конструктор по умолчанию, но действует как Composition Root или *humble-объект*, который делегирует все вызовы метода CampaignPresenter. Обычно стрелки обозначают указатели, а центральная часть приложения ссылается на все остальные модули приложения, поскольку она соединяет их вместе. Как модуль логики представления, так и модуль доступа к данным ссылаются на библиотеку доменной модели. Не все рассматриваемые классы продемонстрированы на этом рисунке.



Вооруженные знанием диаграммы зависимостей приложения, мы теперь можем реализовать Composition Root для кадра, продемонстрированного на рисунке 7-15.

Компоновка ObjectDataSource

Default Page, продемонстрированный на рисунке 7-15, состоит из элемента управления GridView и связанного с ним элемента управления ObjectDataSource. Как и в случае с классами Page, класс, используемый для ObjectDataSource также должен обладать конструктором по умолчанию. Для достижения этой цели вы специально создадите класс, продемонстрированный в следующем листинге.

Листинг 7-12: Компоновка `Presenter` в качестве источника данных

```
1. public class CampaignDataSource
2. {
3.     private readonly CampaignPresenter presenter;
4.     public CampaignDataSource()
5.     {
6.         var container =
7.             (CampaignContainer)HttpContext.Current
8.             .Application["container"];
9.         this.presenter = container.ResolvePresenter();
10.    }
11.    public IEnumerable<CampaignItemPresenter> SelectAll()
12.    {
13.        return this.presenter.SelectAll();
14.    }
15.    public void Update(CampaignItemPresenter item)
16.    {
17.        this.presenter.Update(item);
18.    }
19. }
```

Строка 9: Формирует `Presenter`

Строка 13, 17: Делегирует полномочия `Presenter`

Класс `CampaignDataSource` имеет конструктор по умолчанию, поскольку того требует ASP.NET. Близкий по духу принципу *Fail Fast* (принцип быстрого отказа), он незамедлительно пытается извлечь контейнер из Application Context и преобразовать экземпляр `CampaignPresenter`, который будет выступать в роли реальной реализации.

Все члены класса `CampaignDataSource` делегируют вызов преобразованному предьявителю, таким образом, действуя как *hum ble*-объект.

Примечание

Для приверженцев паттернов проектирования класс `CampaignDataSource` очень похож либо на Decorator, либо на Adapter. Он не реализует строго типизированный интерфейс, но обертывает соответствующую реализацию в класс, который соответствует требованиям предьявляемых ASP.NET.

Вам может быть интересно, что мы приобретаем благодаря этому дополнительному уровню преобразования. Если вы привыкли к разработке через тестирование, то это должно быть вам понятно: `HttpContext.Current` недоступен во время модульного тестирования, поэтому вы не можете выполнить модульное тестирование `CampaignDataSource`. Это важная причина того, почему вы должны сохранять его *hum ble*-объектом.

Несмотря на то, что эта конструкция, в лучшем случае, неудобна, она позволяет вам руководствоваться соответствующими DI-паттернами из класса `CampaignPresenter` и в дальнейшем проходить уровни приложения один за одним.

Компоновка Presenter

Я не буду знакомить вас с подробной информацией о CampaignPresenter, но стоит рассмотреть сигнатуру его конструктора, поскольку он использует Constructor Injection:

```
public CampaignPresenter(CampaignRepository repository,
    IPresentationMapper mapper)
```

Зависимостями CampaignPresenter являются абстрактный класс CampaignRepository и интерфейс IPresentationMapper. Как раз то, *что* делают эти абстракции, менее важно, чем то, *как* вы их компонуете. Это является задачей CampaignContainer из следующего листинга. Вы можете вспомнить, что вы конфигурировали его в *global.asax* и регистрировали в Application Context.

Листинг 7-13: Преобразование CampaignPresenter

```
1. public CampaignPresenter ResolvePresenter()
2. {
3.     string connectionString =
4.         ConfigurationManager.ConnectionStrings
5.         ["CommerceObjectContext"].ConnectionString;
6.     CampaignRepository repository =
7.         new SqlCampaignRepository(connectionString);
8.     IPresentationMapper mapper =
9.         new PresentationMapper();
10.    return new CampaignPresenter(repository, mapper);
11. }
```

Строка 6-7: Создает репозиторий

Строка 8-9: Создает преобразователь

Строка 10: Формирует Presenter

Ответственность метода ResolvePresenter – скомпоновать экземпляр CampaignPresenter. Из конструктора вы знаете, что для него нужен CampaignRepository, поэтому вы преобразовываете его в экземпляр SqlCampaignRepository. Другой зависимостью является IPresentationMapper, и вы преобразуете ее в конкретный класс PresentationMapper.

Вооружившись всеми необходимыми зависимостями, вы можете впоследствии вернуть новый экземпляр CampaignPresenter.

Использование механизма внедрения зависимостей в рамках ASP.NET невозможно. Главным недостатком использования каждого Page и источника данных объекта в виде объединенного Composition Root и *humble*-объекта является то, что для этого необходимо дублирование большинства членов класса.

Обратили ли вы внимание на то, как каждый член CampaignDataSource делегирует свою реализацию схожему по названию методу CampaignPresenter? Вам придется повторять эту идиому кода на протяжении всего ASP.NET приложения. Для каждого обработчика события нажатия кнопки вам необходимо определить и поддерживать в работоспособном состоянии связанный метод класса Presenter и тому подобное.

Как мы уже обсуждали в главе 3, я приравниваю понятие Composition Root к такому понятию *Бережливой разработки программного обеспечения* (Lean Software Development), как *последний ответственный момент*. В рамках таких фреймворков, как ASP.NET MVC и WCF, мы можем отложить композицию приложения вплоть до точки входа в приложение, но в ASP.NET так не делается. Не важно, как упорно мы стараемся, мы можем отложить только принятие решений о композиции объектов, пока не столкнемся с требованием о необходимости наличия конструктора по умолчанию.

Потом это становится "самым возможным местом", в котором мы можем компоновать объекты. Несмотря на то, что мы считаем, что пришли к компромиссу, мы все еще следуем всеобщему духу Composition Root. Мы компоновем иерархии объектов настолько близко к верхним уровням приложения, насколько это возможно, и разрешаем корректные DI-паттерны как здесь, так и на более низких по иерархии уровнях.

ASP.NET все еще предоставляет нам небольшую роскошь: мы можем использовать один экземпляр контейнера в рамках Application Context. Некоторые фреймворки не допускают даже это.

7.6. Построение PowerShell cmdlets

Некоторые фреймворки совсем не предоставляют швов, которые позволяют нам управлять жизненным циклом основных элементов фреймворка. Windows PowerShell является одним из таких фреймворков.

Примечание

Прочитайте этот раздел, даже если вы не интересуетесь PowerShell. Я выбрал его, главным образом, в качестве примера последнего испытания механизма внедрения зависимостей. Я также мог остановить свой выбор на Managed MMCSDK, но он в столь многих других моментах не приятен для использования, что я предпочел использовать в качестве примера PowerShell.

Важным элементом PowerShell является cmdlet (предполагаю, что это слово произносится как *commandlet*, но я видел только, что он пишется как *cmdlet*). Вы можете считать cmdlet продвинутой утилитой командной строки.

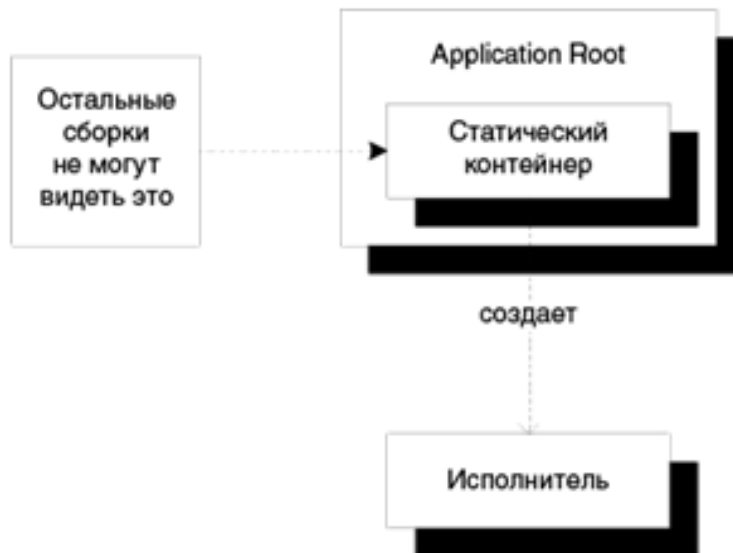
cmdlet — это класс, унаследованный от *Cmdlet*, и он должен иметь конструктор по умолчанию. Как и для ASP.NET это требование эффективно исключает любое использование Constructor Injection. Решение проблемы тоже аналогично: мы перемещаем Composition Root в конструктор каждого *cmdlet*. Единственное отличие — отсутствует встроенный Application Context, поэтому мы должны прибегнуть к самому низшему универсальному деноминатору: статическому классу.

Примечание

Я предпочитаю *code smell* (гнилой код) любому использованию ключевого слова *static*, но по сравнению с анти-паттернами *code smell*ы указывают только на потенциальные недостатки проектирования. В некоторых особых случаях использование подобных идиом оправдано, и это именно тот случай.

Вас может заинтересовать то, как это все отличается от анти-паттерна Service Locator. Как и для ASP.NET главное отличие заключается не в структуре кода, а в использовании паттерна. Вместо того чтобы пытаться использовать статический Service Locator в качестве виртуального ключевого слова *new*, для каждого *cmdlet* мы используем его только один раз. Чтобы защитить себя в дальнейшем от неправильного использования, мы можем сделать *Composer* внутренним и использовать его только для преобразования типов из различных сборок, как это продемонстрировано на рисунке 7-17.

Рисунок 7-17: Когда нет выхода для статического контейнера, мы можем сделать его внутренним и переместить в корневую сборку приложения. Все методы `Resolve` возвращают классы, которые определены в других сборках. Таким образом, как только контейнер преобразовал реализатор, ни один из классов преобразованной иерархии зависимостей не имеет доступа к статическому контейнеру, поскольку все они находятся за пределами центральной сборки приложения, а контейнер находится внутри.



Результатом преобразования диаграммы зависимостей является класс, определенный в другой сборке, и этот класс не имеет доступа к статическому контейнеру, поскольку он находится внутри центральной сборки приложения. Реализатору `cmdlet` необходимо использовать соответствующие DI-паттерны, как например, `Constructor Injection`, для того, чтобы применять любые зависимости, и мы эффективно защищаем себя от угроз `Service Locator`'а.

Давайте рассмотрим пример, который иллюстрирует этот принцип.

Пример: построение `cmdlet`'ов управления корзиной

Этот пример возвращает нас к шаблонному коммерческому приложению. Как и большинство других коммерческих приложений, это приложение имеет функциональность *корзины покупок*. Среди пользователей общепринятой является такая ситуация: добавлять элементы в свою корзину покупок, но потом покидать сайт и никогда больше сюда не заходить.

Несмотря на то, что хранение в эти дни обходится дешево, заинтересованные стороны проекта попросили, чтобы вы предоставили им гибкую возможность очищать осиротевшие корзины на основании различных критериев. Они хотят иметь возможность выбирать осиротевшие корзины на основании того, когда они в последний раз обновлялись. Если корзина содержит значительное количество товаров, то ее не следует удалять (или, может быть, ей необходимо предоставить более длительный льготный период), но общее количество должно быть рассчитано согласно всем текущим бизнес-правилам.

Кажется, что API написания сценариев отлично подошло бы для этого, поскольку администратор смог бы определять и составлять расписание простых скриптов очистки. PowerShell здесь наиболее подходит благодаря своей продвинутой функциональности фильтрации и транспортировки.

Вы можете реализовать желаемое API с помощью двух cmdlet'ов: один – для того, чтобы извлечь все корзины, а другой – для удаления корзины данного пользователя. Следующий листинг является примером того, как это могло бы выглядеть в интерактивной сессии.

Листинг 7-14: Удаление корзин, хранящихся более одного месяца

```
1. PS C:\> Get-Basket
2. LastUpdated Owner Total
3. -----
4. 19.03.2010 20:5... ploeh 89,4000
5. 22.01.2010 19:5... ndøh 199,0000
6. 21.03.2010 09:1... fnaah 171,7500
7. PS C:\> $now = [System.DateTime]::Now
8. PS C:\> $month = [System.TimeSpan]::FromDays(30)
9. PS C:\> $old = $now - $month
10. PS C:\> Get-Basket | ? { $_.LastUpdated -lt $old } |
11. Remove-Basket
12. PS C:\> Get-Basket
13. LastUpdated Owner Total
14. -----
15. 19.03.2010 20:5... ploeh 89,4000
16. 21.03.2010 09:1... fnaah 171,7500
17. PS C:\>
18.
```

Строка 5: Старая корзина

Строка 7-9: Рассчитывает дату закрытия

Строка 10-11: Удаляет старые корзины

Перед тем как вы начнете удалять корзины, вам хотелось бы просмотреть текущие корзины системы. Вы можете использовать пользовательский cmdlet Get-Basket для того, чтобы перечислить все корзины. Обратите внимание на то, что каждая корзина имеет три свойства, которые сообщают вам о том, *когда* корзина в последний раз обновлялась, *кто* является владельцем корзины, а также об *общей стоимости* (включая скидки) корзины.

Текущей датой выполнения этой конкретной сессии была 22 марта 2010 года. Обратите внимание на то, что второй корзине уже более 30 дней. Теперь вы можете рассчитать дату закрытия на основании текущей даты и использовать ее в выражении фильтра. Вы можете удалить все старые корзины путем передачи результата Get-Basket в фильтр, а затем передавая результат отфильтрованных корзин в cmdlet Remove-Basket. Если бы вы захотели выполнить фильтрацию по свойству Total, то вы также смогли бы это сделать тем же способом.

В итоге, вы перечислите все корзины, чтобы удостовериться, что все старые корзины удалены

Примечание

Не переживайте, если не поняли всех деталей выражения фильтра. Это книга не о PowerShell, поэтому я не буду задерживаться на этой теме.

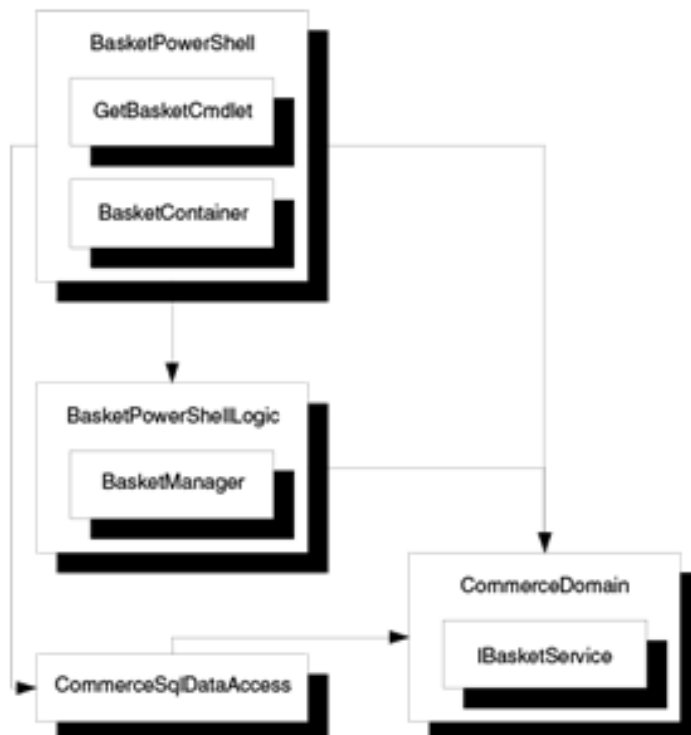
Для продвижения этого API написания сценариев вам необходимо реализовать два пользовательских cmdlet'a. Поскольку одним из требований является то, что Total должен принимать во внимание все соответствующие бизнес-правила, вам необходимо компоновать cmdlet'ы в рамках уровня Domain Model.

Построение GetBasketCmdlet

Давайте рассмотрим, как реализуется cmdlet `Get-Basket`. `Remove-Basket` реализуется похожим способом, поэтому я не буду рассматривать его реализацию.

Для того чтобы избежать соблазна статического контейнера, вы будете реализовывать полноценный мост между cmdlet'ом PowerShell и Domain Model в отдельной библиотеке, которая носит название `BasketPowerShellLogic`. Рисунок 7-18 демонстрирует, как компонуется приложение в пределах библиотек.

Рисунок 7-18: Библиотека `BasketPowerShell` содержит только инфраструктуру, необходимую для того, чтобы ошастливить PowerShell – это *humble*-объект. Как только `BasketContainer` преобразовал `BasketManager`, все дальнейшие реализации происходят в других сборках. Класс `BasketManager` не имеет доступа к внутреннему `BasketContainer`, но использует `IBasketService` из Domain Model. Обычно стрелки обозначают указатели. Не все рассматриваемые классы продемонстрированы на рисунке.



Примечание

Если вы думаете, что рисунок 7-18 очень похож на рисунок 7-16, то вы начинаете понимать паттерн.

Примечание

Вы можете вспомнить `IBasketService` из главы 2, раздела 2.3.2.

Класс `GetBasketCmdlet` должен иметь конструктор по умолчанию для того, чтобы соответствовать `PowerShell`, поэтому вы используете его в качестве `Composition Root` и оставляете его в виде *humble*-объекта. Следующий листинг демонстрирует только то, насколько он "скромен".

Листинг 7-15: Реализация `GetBasketCmdlet`

```
1. [Cmdlet(VerbsCommon.Get, "Basket")]
2. public class GetBasketCmdlet : Cmdlet
3. {
4.     private readonly BasketManager basketManager;
5.     public GetBasketCmdlet()
6.     {
7.         this.basketManager =
8.             BasketContainer.ResolveManager();
9.     }
10.    protected override void ProcessRecord()
11.    {
12.        var baskets =
13.            this.basketManager.GetAllBaskets();
14.        this.WriteObject(baskets, true);
15.    }
16. }
```

Строка 7-8: `Composition Root`

Строка 12-13: Делегирование полномочий реализатору

В требуемом конструкторе по умолчанию вы используете статический контейнер для того, чтобы преобразовать `BasketManager`, который выступает в роли реализации. `BasketManager` использует `Constructor Injection` для запроса экземпляра `IBasketService`. К настоящему моменту вы должны уже хорошо знать этот паттерн, а также реализацию `BasketContainer`, продемонстрированную в следующем листинге.

Листинг 7-16: Преобразование `BasketManager`

```
1. internal static BasketManager ResolveManager()
2. {
3.     BasketRepository basketRepository =
4.         new SqlBasketRepository(
5.             BasketContainer.connectionString);
6.     DiscountRepository discountRepository =
7.         new SqlDiscountRepository(
8.             BasketContainer.connectionString);
9.     BasketDiscountPolicy discountPolicy =
10.        new RepositoryBasketDiscountPolicy(
11.            discountRepository);
12.     IBasketService basketService =
13.        new BasketService(basketRepository,
14.            discountPolicy);
15.     return new BasketManager(basketService);
16. }
```

Строка 1: Внутренний метод

Строка 15: Возвращает `basket manager`

Метод, так же, как и весь класс, является внутренним, что делает возможным вызов его из `GetBasketCmdlet`, как это продемонстрировано в листинге 7-15, но невозможно случайно использовать его из `BasketManager` или из его зависимостей.

Теперь реализация метода должна быть вам понятной. И снова я считаю, что проще всего отойти от результата. Для класса `BasketManager` необходим экземпляр `IBasketService`, и поэтому вы используете класс `BasketService` (других реализаций, которые вы могли бы выбрать, у вас нет).

Для `BasketService` необходимы `BasketRepository` и `BasketDiscountPolicy`. Для `BasketDiscountPolicy` вы используете `RepositoryBasketDiscountPolicy`. Для этого класса требуется еще одна абстракция репозитория, а для этих двух репозиториев вы используете реализации на основе SQL Server.

Реализация `BasketManager` является стандартной, поэтому я не буду ее показывать. Все, что она делает – выражает необходимые операции в терминах доменной модели.

`Remove-Basket cmdlet` руководствуется тем же паттерном: он использует статический, но внутренний `BasketContainer` для того, чтобы преобразовать экземпляр `BasketManager`, а потом делегировать реализацию преобразованному экземпляру. Оба `cmdlet`'а выступают в роли сочетания `Composition Root` и *humble*-объекта.

Класс `BasketManager` реализуется в другой сборке. Как только код уходит от `cmdlet`'ов, риск, что какая-либо из основополагающих реализаций будет использовать статический контейнер в качестве `Service Locator`, исчезает, поскольку он является внутренним по отношению к сборке, содержащей `cmdlet`'ы.

Примечание

Очевидно, основополагающий код ничего не будет делать случайно, но разработчик, пишущий код, может. Мы защищаем статический контейнер от остальной части кода для того, чтобы защитить себя от совершения ошибок.

Фреймворк, подобный PowerShell, является самым DI-недружественным. Использование простой технологии превращения каждого элемента фреймворка в `Composition Root` и *humble*-объект дает вам простой способ решения этой проблемы.

7.7. Резюме

Композиция объектов— один из трех важных аспектов механизма внедрения зависимостей (двумя другими являются управление жизненным циклом и перехват). В данной главе я продемонстрировал то, как компоновать приложения из слабо связанных модулей во множестве различных сред.

Некоторые фреймворки облегчают этот процесс. При написании консольных приложений и Windows клиентов (WPF или Windows Forms) мы более или менее напрямую контролируем то, что происходит в точке входа в приложение. Это обеспечивает нас различными и легко реализуемыми Composition Root в точке входа.

Другие фреймворки, например, ASP.NET MVC и WCF, заставляют нас поработать немного усерднее, но они все же предоставляют швы, которые мы можем использовать для того, чтобы определить то, как приложение должно быть скомпоновано. ASP.NET MVC уже был создан с замыслом механизма внедрения зависимостей, поэтому построение приложения такой же простой процесс, как и реализация пользовательского `IServiceFactory` и регистрация его с помощью фреймворка. Кажется, что в WCF шов находится почти случайно, но, несмотря на то, что это более обходной путь, нежели реализация единичного интерфейса, мы все еще можем достичь всех ценных свойств механизма внедрения зависимостей, которые только могли бы пожелать.

Остальные фреймворки являются явно DI-недружественными и требуют от нас использования конструктора по умолчанию для соответствия им. ASP.NET (Web Forms) является самым известным из них, но еще одними примерами являются также и PowerShell, и Managed MMC SDK. Эти фреймворки управляют жизненными циклами классов, которые мы предоставляем, поэтому единственный вариант — рассматривать каждый класс как отдельный Composition Root. Это требует много затрат, поэтому я лично предпочитаю использовать DI-дружественные фреймворки, если у меня есть выбор.

Без композиции объектов нет и механизма внедрения зависимостей, но вы, возможно, еще не осознали полностью роль жизненного цикла объектов, когда мы переносили создание объектов из используемых классов. Вам может показаться само собой разумеющимся, что внешний объект, выполняющий вызов (чаще всего DI-контейнер) создает новые экземпляры зависимостей — но когда высвобождаются внедряемые зависимости? И что если внешний объект, выполняющий вызов, решит не создавать новые экземпляры все время, а вместо этого передаст вам существующий экземпляр? Это темы обсуждения для следующей главы.

8. Жизненный цикл объектов

Меню:

- Управление жизненным циклом зависимостей
- Устраняемые зависимости
- Singleton
- Transient
- Per Graph
- Web Request Context
- Pooled

Истечение срока действия основательно влияет на большинство продуктов и напитков, но последствия сильно различаются. Лично я считаю 12-месячный *Gruyère* (швейцарский сыр) гораздо более интересным, нежели 6-месячный *Gruyère*, но предпочитаю, чтобы аспарагус был свежее 12-месячного и 6-месячного *Gruyère*. В большинстве случаев оценить точный возраст элемента довольно просто; но в определенных случаях этот процесс становится очень сложным. Истечение срока действия наиболее значимо, когда дело касается вина (см. рисунок 8-1).

Рисунок 8-1: Вино, сыр и аспарагус. Несмотря на то, что, если смешать эти ингредиенты мы получим не совсем хороший результат, возраст этих ингредиентов значительно влияет на их общее качество.



С годами вина становятся лучше – до тех пор, пока они не станут слишком старыми и не потеряют свой аромат. Это зависит от множества факторов, включая происхождение и урожай вина. Несмотря на то, что я интересуюсь винами, я даже и не надеюсь, что смогу определить, когда вино достигнет своего полного расцвета. В этом вопросе я полагаюсь на экспертов: на книги – дома и на сомелье – в ресторанах. Они разбираются в винах лучше меня, так как это их специальность, поэтому всякий раз, когда я им доверяюсь в вопросе выбора вина, я с удовольствием позволяю им взять все под свой контроль.

За исключением того случая, когда вы перешли к данной главе, не читая при этом ни одной из предыдущих глав, вы понимаете, что возможность уйти от контроля является ключевой концепцией механизма внедрения зависимостей. Это проблема инверсии управления, но она подразумевает больше, чем просто позволение кому-то отобрать реализацию необходимой абстракции. Когда мы позволяем `Composer` (Компоновщик) создавать зависимость, мы также должны принять тот факт, что не сможем контролировать ее жизненный цикл.

Поскольку сомелье хорошо знает содержания винных подвалов ресторана и может принять гораздо более информативное решение, нежели бы это сделали мы, способность контролировать жизненный цикл зависимостей мы должны с наибольшей эффективностью доверять `Composer` (Компоновщик), чем потребителю. Компоновка и управление компонентами – это единичная ответственность.

Определение

`Composer` – в том смысле, в котором я его использую здесь, это унифицированный термин, обозначающий любой объект или метод, который компонует зависимости. Чаще всего это DI-контейнер, но может быть и любым методом, который используется в `Roog Man's DI`, например, метод `Main` консольного приложения.

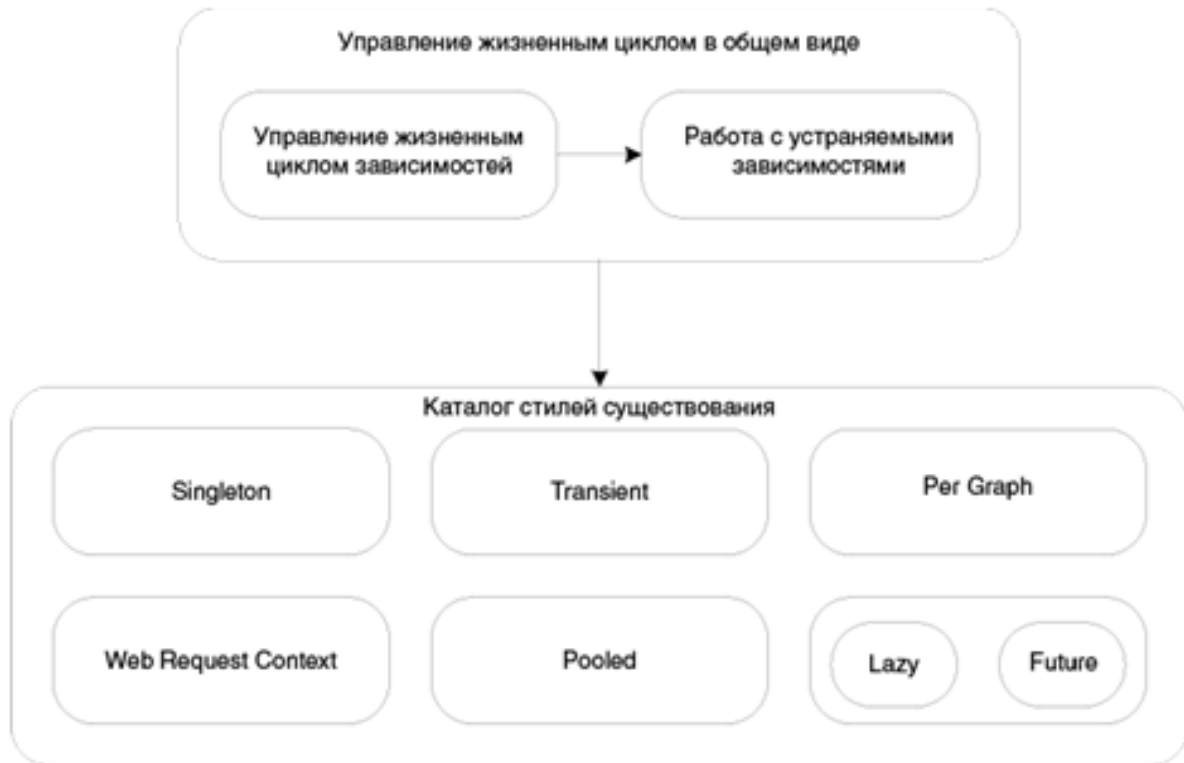
В данной главе мы будем рассматривать процесс управления жизненным циклом зависимостей. Понимание этой темы важно, поскольку точно так же, как вы можете приобретать неудовлетворительный опыт при употреблении вина неправильного возраста, вы можете получить ухудшенное выполнение, неправильно сконфигурировав жизненный цикл зависимостей. Даже хуже того, вы можете получить аналог испортившегося продукта для механизма управления жизненным циклом: утечки ресурсов. Понимание принципов корректного управления границами жизненных циклов компонентов должно дать вам возможность принимать правильные решения и корректно конфигурировать ваши приложения.

Примечание

На всем протяжении данной главы я использую термины *тип стиля существования*, *стратегия жизненного цикла*, *границы жизненного цикла* и другие редко встречающиеся сочетания взаимозаменяемо.

Как иллюстрирует рисунок 8-2, мы начнем с общего введения к концепции, за которым следует обсуждение устраняемых зависимостей. Первая часть главы предназначена для того, чтобы предоставить вам всю теоретическую информацию и необходимые вам ведущие принципы для того, чтобы принимать умное, взвешенное решение по поводу конфигурации границ жизненного цикла ваших собственных приложений.

Рисунок 8-2: Полная структура данной главы. Мы начнем с общего обсуждения процесса управления жизненным циклом зависимостей, включая конкретное обсуждение работы с устранимыми объектами. Нам необходимы эти фундаментальные знания для того, чтобы эффективно обсуждать универсальные паттерны небольшого каталога стилей существования, рассматриваемого далее. Мы начнем с рассмотрения некоторых универсальных и очень полезных паттернов и закончим кратким обзором некоторых более экзотичных стилей существования для того, чтобы дать вам ощущение широты данной темы



После этого мы будем использовать остальную часть главы для рассмотрения различных стратегий жизненного цикла. Эта часть книги принимает форму каталога доступных стилей существования. В большинстве случаев один из этих шаблонных паттернов стилей существования будет отлично подходить для решения конкретной задачи, поэтому заблаговременное понимание этих стилей снабжает вас достаточными знаниями для того, чтобы справиться со многими сложными ситуациями. Когда мы закончим изучение этой главы, вы должны будете хорошо разбираться в управлении жизненным циклом и в универсальных жизненных циклах.

Для начала давайте рассмотрим жизненный цикл объектов и то, каким образом он в общем смысле относится к механизму внедрения зависимостей.

[8.1. Управление жизненным циклом зависимостей](#)

[8.2. Работа с устранимыми зависимостями](#)

[8.3. Каталог стилей существования объектов](#)

[8.4. Резюме](#)

8.1. Управление жизненным циклом зависимостей

До настоящего момента мы, главным образом, обсуждали то, как механизм внедрения зависимостей позволяет нам компоновать зависимости. В предыдущей главе рассматривался этот вопрос чрезвычайно подробно, но как я говорил в разделе 1-4, композиция объектов является всего лишь одним из аспектов механизма внедрения зависимостей. Управление жизненным циклом объектов является еще одним его аспектом.

Примечание

В .NET жизненный цикл объекта достаточно прост: объект создается, используется и уничтожается *сборщиком мусора* (garbage collector). Присутствие `IDisposable` слегка все усложняет, но жизненный цикл от этого не становится более сложным. При обсуждении жизненного цикла объектов мы говорим о том, как мы управляем жизненными циклами объектов.

В первый раз, когда я познакомился с идеей о том, что в сферу механизма внедрения зависимостей входит управление жизненным циклом, я не понимал глубокой связи между композицией объектов и жизненным циклом объектов. В конце концов, я это осознал, и это оказалось довольно просто, поэтому давайте рассмотрим это вместе с вами.

В данном разделе я буду знакомить вас с механизмом управления жизненным циклом и с тем, как применить его к зависимостям. Мы начнем с рассмотрения общего случая компоновки объектов и того, как компоновка влияет на жизненные циклы зависимостей. От этой темы мы перейдем к изучению того, как DI-контейнеры могут управлять жизненным циклом зависимостей. Несмотря на то, что большинство из примеров являются специализированным кодом, который имеет дело с конкретными конфигурациями, мы также сделаем краткий обзор публичного DI-контейнера для того, чтобы получить представление о том, как может выглядеть конфигурация жизненного цикла.

Для начала мы исследуем, почему композиция объектов влияет на жизненный цикл.

Знакомство с механизмом управления жизненным циклом

Когда мы принимаем тот факт, что нам следует выбросить из головы наши физиологические потребности для того, чтобы контролировать зависимости и, кроме того, запрашивать их с помощью Constructor Injection или одного из других DI-паттернов, мы должны выбросить это из головы окончательно. Для того чтобы понять, почему, мы рассмотрим этот вопрос постепенно. Давайте начнем с обзора того, что означает стандартный жизненный цикл объекта для зависимостей. Вы уже должны это знать, но потерпите, пока я не пройду следующую половину страницы, и не установлю контекст.

Простой жизненный цикл зависимостей

Вы знаете, что механизм внедрения зависимостей подразумевает, что мы позволяем сторонним объектам выступать для нас в роли необходимых нам зависимостей. Это также означает, что мы должны позволить им управлять жизненными циклами зависимостей.

Проще всего понять это, когда дело доходит до создания объекта. Ниже приведен фрагмент кода Composition Root шаблонного приложения Commerce (окончательный пример вы можете увидеть в листинге 7-3).

```
var discountRepository =  
    new SqlDiscountRepository(connectionString);  
var discountPolicy =  
    new RepositoryBasketDiscountPolicy(discountRepository);
```

Надеюсь, это очевидно, что класс `RepositoryBasketDiscountPolicy` не контролирует то, когда создается `discountRepository`. В данном случае это, скорее всего, происходит в пределах той же миллисекунды; но для чистоты эксперимента мы могли бы вставить вызов `Thread.Sleep` между этими двумя строками кода для того, чтобы продемонстрировать, что мы можем условно разделить их на промежутки времени. Это было бы достаточно непонятно для исполнения, но вы все поняли.

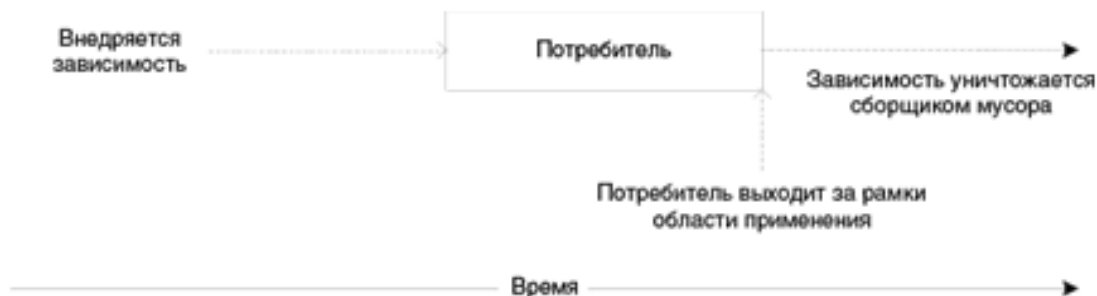
Потребители не контролируют *создание* своих зависимостей, но что насчет *уничтожения*? Как правило, мы не контролируем то, когда в .NET уничтожаются объекты. *Сборщик мусора* (garbage collector) собирает неиспользуемые объекты, но до тех пор, пока мы не будем работать с устранимыми объектами, мы не сможем явно уничтожить объект.

Примечание

Я использую термин *устраняемый объект* (disposable object) в качестве условного обозначения экземпляров объектов типов, которые реализуют интерфейс `IDisposable`.

Объекты уничтожаются сборщиком мусора, когда они выходят за рамки области применения. Наоборот, они действуют, пока кто-нибудь еще ссылается на них. Несмотря на то, что потребитель не может явно уничтожить объект, он может сохранять объект, продолжая ссылаться на него. Именно это мы и делаем при использовании `Constructor Injection`, поскольку мы сохраняем зависимость в приватном поле. Но как демонстрирует рисунок 8-3, когда потребитель выходит за рамки области применения, то же самое может сделать и зависимость.

Рисунок 8-3: Кто бы ни внедрял зависимость в потребителя, он решает, когда создается эта зависимость, но потребитель может сохранять зависимость, продолжая ссылаться на нее. Когда потребитель выходит за границы области применения, зависимость может удовлетворять условиям, необходимым для работы сборщика мусора.



Даже когда потребитель выходит за границы области применения, зависимость может существовать, если другие объекты содержат ссылку на нее. В противном случае она будет уничтожена сборщиком мусора. Поскольку вы являетесь опытным .NET разработчиком, это довольно не ново для вас, но сейчас обсуждение станет для вас более интересным.

Усложнение жизненного цикла зависимости

До настоящего момента наш анализ жизненного цикла зависимостей был обыденным, но мы можем добавить в него некоторую сложность. Что происходит, когда для более чем одного потребителя необходима одна и та же зависимость? Один из вариантов – дополнить каждого потребителя своим собственным экземпляром, как это продемонстрировано в следующем листинге.

Листинг 8-1: Компоновка с помощью различных экземпляров одной и той же зависимости

```
1. var repositoryForPolicy =  
2.   new SqlDiscountRepository(connectionString);  
3. var repositoryForCampaign =  
4.   new SqlDiscountRepository(connectionString);  
5. var discountPolicy =  
6.   new RepositoryBasketDiscountPolicy(  
7.     repositoryForPolicy);  
8. var campaign =  
9.   new DiscountCampaign(repositoryForCampaign);
```

Строка 6-7, 9: Внедряет соответствующий репозиторий

В данном примере для двоих потребителей необходим экземпляр `DiscountRepository`, поэтому вы присоединяете два отдельных экземпляра с одинаковой строкой соединения. Теперь вы способны передать `repositoryForPolicy` в новый экземпляр `RepositoryBasketDiscountPolicy`, а `repositoryForCampaign` в новый экземпляр `DiscountCampaign`.

Когда дело доходит до жизненных циклов каждого репозитория из листинга 8-1, то по сравнению с предыдущим примером ничего не изменилось. Каждый из них выходит за рамки области применения и уничтожается сборщиком мусора, когда потребитель выходит за рамки области применения. Это может происходить в разное время, но ситуация лишь в малой степени отличается от той, что была до этого.

Несколько другой была бы ситуация, если бы оба потребителя были обязаны совместно использовать одну и ту же зависимость, как это продемонстрировано в примере:

```
var repository =  
    new SqlDiscountRepository(connectionString);  
var discountPolicy =  
    new RepositoryBasketDiscountPolicy(repository);  
var campaign = new DiscountCampaign(repository);
```

Вместо создания двух различных экземпляров `SqlDiscountRepository`, вы создаете единственный экземпляр, который внедряете в оба потребителя. Оба потребителя сохраняют ссылку для последующего использования.

Примечание

Потребители совершенно не осведомлены о том, что зависимость используется ими совместно. Поскольку они оба принимают любую передаваемую им зависимость, для размещения этого изменения в конфигурации зависимости не нужно никакой модификации исходного кода. Это результат принципа подстановки Барбары Лисков.

Принцип подстановки Барбары Лисков

Как утверждалось ранее, принцип подстановки Барбары Лисков является теоретической и абстрактной сущностью. Но в объектно-ориентированном программировании мы можем перефразировать этот принцип следующим образом: *Методы, использующие абстракции, должны уметь использовать любой унаследованный класс, ничего при этом не зная о нем.* Другими словами, мы должны уметь заменять абстракцию произвольной реализацией, не изменяя при этом точность системы

Ситуация, связанная с жизненным циклом зависимости `repository`, отчетливо изменилась по сравнению с предыдущим примером. Оба потребителя должны выйти за рамки области применения до того, как `repository` сможет приобрести право быть уничтоженным сборщиком мусора, и сделать они это могут в разное время. Ситуация становится менее предсказуемой, когда зависимость достигает момента завершения своего жизненного цикла, и эта особенность только усиливается при увеличении числа потребителей.

При достаточном количестве потребителей, скорее всего, поблизости всегда будет находиться один из них, который сохраняет зависимость "живой". Это может казаться проблемой, но такое редко случается: вместо множества схожих экземпляров, мы имеем только один, который сохраняет память. Это настолько завидное качество, что мы формализуем его в паттерне стиля существования `Singleton`. Несмотря на их схожесть, не путайте его с паттерном проектирования `Singleton`. Более подробно этот вопрос мы рассмотрим в разделе "`Singleton`".

Ключевым моментом, который стоит принять во внимание, является тот факт, что `Composer` обладает большей степенью влияния на жизненный цикл зависимостей, нежели любой единичный потребитель. `Composer` решает, когда создаются зависимости, и на основании своего выбора, использовать ли экземпляры совместно или нет, определяет, выходит ли зависимость за рамки области применения единственным потребителем, или все потребители должны выйти за рамки области применения до того, как могут быть освобождены зависимости.

Это сравнимо с посещением ресторана в компании хорошего сомелье. Сомелье проводит значительную часть дня, управляя винным подвалом и совершенствуя его содержимое, покупая новые вина, пробуя вино из доступных бутылок для отслеживания того, как развивается это вино, а также, работая с шеф-поварами, чтобы определить оптимальное соответствие подаваемым блюдам. При ознакомлении с винной картой мы видим только то, что сомелье посчитал нужным представить к продаже. Мы вольны выбрать вино в соответствии с нашим личным вкусом, но мы не допускаем, что знаем больше сомелье о выборке вин ресторана, и о том, как эти вина сочетаются с блюдами.

Сомелье будет часто принимать решение о хранении большого количества бутылок в хранилище на протяжении многих лет, и как вы увидите в следующем разделе, `Composer` может принять решение о хранении экземпляров "живыми", продолжая поддерживать ссылки на эти экземпляры.

Управление жизненным циклом с помощью контейнера

В предыдущем разделе объяснялось, как мы можем варьировать композицию зависимостей для того, чтобы влиять на их жизненный цикл. В данном разделе мы будем рассматривать то, как DI-контейнер может обращаться к этим вариациям.

Примечание

В данном разделе обсуждаются принципы, лежащие в основе управления жизненными циклами с помощью DI-контейнера, поэтому я не буду подробно рассматривать конкретные контейнеры. Как и на протяжении всей части 3, я использую Poor Man's DI для иллюстрации этих сущностей.

Мы начнем с рассмотрения того, как контролировать жизненный цикл зависимостей с помощью пользовательских контейнеров, а затем перейдем к легко реализуемому примеру задания стилей существования в реальном DI-контейнере.

Управление стилями существования с помощью специализированного контейнера

В главе 7 мы создали специализированные контейнеры для построения приложений. Одним из таких контейнеров был `CommerceServiceContainer`. Листинг 7-9 демонстрирует реализацию его метода `ResolveProductManagementService`; и, как показывает рисунок 8-4, этот метод является единственным кодом в данном классе.

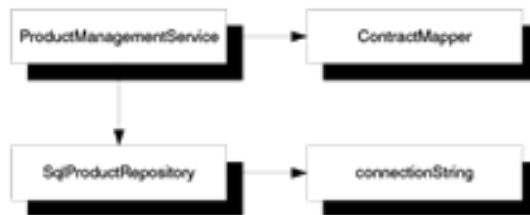
Рисунок 8-4: Вся реализация класса `CommerceServiceContainer` в настоящий момент располагается в методе `ResolveProductManagementService`. Метод `Release` абсолютно ничего не делает, кроме того, в классе нет ни полей, ни свойств. Если вам интересно, почему здесь присутствует метод `Release`, то мы вернемся к этому вопросу в разделе "Управление устраняемыми зависимостями".



Как вы можете помнить из листинга 7-9, метод `Resolve` создает полноценную диаграмму зависимостей, каждый раз, когда он вызывается. Другими словами, каждая зависимость является приватной по отношению к рассматриваемому `IProductManagementService`, и какие-либо связи отсутствуют. Когда экземпляр `IProductManagementService` выходит за рамки области применения (что происходит всякий раз, когда сервис отвечает на запрос), все зависимости также выходят за рамки области применения. Это часто называют стилем существования `Transient` (кратковременным), но подробнее о нем мы поговорим в разделе "Transient".

Давайте проанализируем диаграмму объектов, созданную `CommerceServiceContainer` и продемонстрированную рисунком 8-5, на факт существования возможности для совершенствования.

Рисунок 8-5: Диаграмма объектов, созданная `CommerceServiceContainer`. Каждый созданный экземпляр `ProductManagementService` содержит свой собственный `ContractMapper` и свой собственный `SqlProductRepository`, который, в свою очередь, содержит собственную строку соединения. Зависимости, показанные справа, являются неизменными.



Класс `ContractMapper` является совершенно не сохраняющим свое состояние сервисом, поэтому нет необходимости создавать новый экземпляр всякий раз, когда нам нужно обслужить запрос. Строка соединения также, скорее всего, не изменяется, поэтому мы можем также решить повторно использовать ее в рамках запросов.

Класс `SqlProductRepository`, с другой стороны, полагается на Entity Framework `ObjectContext`, и считается хорошим тоном использовать для каждого запроса новый экземпляр.

При данной конкретной конфигурации наилучшая реализация `CommerceServiceContainer` снова использовала бы те же самые экземпляры как `ContractMapper`, так и строки соединения, при создании новых экземпляров `SqlProductRepository`. Короче говоря, вам следует сконфигурировать `ContractMapper` и строку соединения таким образом, чтобы они использовали стиль существования `Singleton` и `SqlProductRepository` в виде `Transient`. Следующий листинг демонстрирует, как реализовать данное изменение.

Листинг 8-2: Управление жизненным циклом с помощью контейнера

```

1. public partial class LifetimeManagingCommerceServiceContainer :
2.     ICommerceServiceContainer
3.     {
4.         private readonly string connectionString;
5.         private readonly IContractMapper mapper;
6.         public LifetimeManagingCommerceServiceContainer()
7.         {
8.             this.connectionString =
9.                 ConfigurationManager.ConnectionStrings
10.                ["CommerceObjectContext"].ConnectionString;
11.             this.mapper = new ContractMapper();
12.         }
13.         public IProductManagementService
14.             ResolveProductManagementService()
15.         {
16.             ProductRepository repository =
17.                 new SqlProductRepository(
18.                     this.connectionString);
19.             Return new ProductManagementService(
20.                 repository, this.mapper);
21.         }
22.     }
  
```

Строка 8-11: Создает `Singleton` зависимости

Строка 16-18: Создает `Transient` зависимость

Поскольку вы хотите повторно использовать строку соединения и `ContractMapper` в рамках всех запросов, вы сохраняете их в приватных полях и инициализируете в конструкторе. Ключевое слово `readonly` обеспечивает дополнительную гарантию того, что, будучи единожды заданными, эти `Singleton` экземпляры остаются неизменными и не могут быть заменены, но кроме этой дополнительной гарантии, `readonly` никоим образом не требуется при реализации стиля существования `Singleton`.

Каждый раз, когда контейнер просят создать новый экземпляр, он создает `Transient` экземпляр `SqlProductRepository` с помощью `Singleton` строки соединения. В конечном счете, контейнер использует этот `Transient repository` вместе с `Singleton mapper` для того, чтобы скомпоновать и вернуть экземпляр `ProductManagementService`.

Примечание

Код в листинге 8-2 функционально эквивалентен коду из листинга 7-9, но только слегка более эффективен.

Продолжая ссылаться на создаваемые им зависимости, контейнер может сохранять их в жизнеспособном состоянии столь долго, сколько он того хочет. В предыдущем примере он создает обе зависимости, как только они инициализируются, но он мог использовать и `Lazy` инициализацию.

Данный пример должен дать вам представление о том, как DI-контейнеры управляют жизненными циклами. Поскольку DI-контейнер является повторно используемой библиотекой, мы не можем изменять его исходный код каждый раз, когда нам хочется переконфигурировать стиль существования. В следующем разделе мы вкратце рассмотрим то, как конфигурировать стили существования для шаблонного контейнера.

Управление стилем существования с помощью Autofac

Время от времени на протяжении этой книги я делаю передышку от `Poor Man's DI`, чтобы предоставить пример того, как мы можем достичь результата с помощью шаблонного DI-контейнера. Каждый DI-контейнер имеет свой собственный конкретный API для выражения множества различных признаков; но, несмотря на то, что детали различаются, принципы остаются теми же. Это справедливо и для механизма управления жизненным циклом.

Примечание

Даже термин "управление жизненным циклом" не является вездесущим. Например, `Autofac` называет этот процесс *Областью применения экземпляра* (`Instance Scope`).

В данном разделе мы вкратце будем рассматривать конфигурирование жизненных циклов с помощью `Autofac`.

Примечание

Нет какой-то конкретной причины того, почему для данного примера я предпочел `Autofac` другим DI-контейнерам. Таким же образом я мог выбрать и любой другой DI-контейнер.

Следующий листинг демонстрирует, как сконфигурировать `Autofac` с помощью простых `Transient` зависимостей аналогично примеру из листинга 7-9.

Листинг 8-3: Конфигурирование Autofac с помощью Transient зависимостей

```
var builder = new ContainerBuilder();
builder.RegisterType<ContractMapper>()
    .As<IContractMapper>();
builder.Register((c, p) =>
    new SqlProductRepository(
        ConfigurationManager
            .ConnectionStrings["CommerceObjectContext"]
            .ConnectionString))
    .As<ProductRepository>();
builder.RegisterType<ProductManagementService>()
    .As<IProductManagementService>();
var container = builder.Build();
```

Одной из особенностей Autofac является то, что вы не конфигурируете сам контейнер, а конфигурируете `ContainerBuilder` и используете его для создания контейнера при завершении конфигурации.

Самая простая форма регистрации – это когда вам нужно определить только преобразование между абстракцией и конкретным типом, например, преобразование `IContractMapper` в `ContractMapper`. Обратите внимание на то, что конкретный тип указан перед абстракцией, что является порядком, противоположным тому, который используется большинством DI-контейнеров.

Несмотря на то, что Autofac так же, как и другие DI-контейнеры поддерживает автоматическую интеграцию, внедрение таких примитивных типов, как строки, всегда представляет собой особый случай, поскольку здесь может потенциально использоваться множество различных строк. В данном случае вы имеете всего одну строку соединения, но вам нужно еще передать ее в `SqlProductRepository`, который вы на данный момент регистрируете. Вы можете сделать это с помощью лямбда-выражения, которое будет выполняться при запросе типа `ProductRepository`.

Использование лямбда-выражений – одна из заявок Autofac на успех. Несмотря на то, что большинство DI-контейнеров на данный момент обладают похожим свойством, Autofac был одним из первых контейнеров, познакомивших нас с лямбда-выражениями. Вы можете использовать лямбда, чтобы указать, как создается класс `SqlProductRepository`, и что еще более специфично, вы выглядываете параметр конструктора `connectionString` из конфигурации приложения.

Преимущество использования лямбда-выражений заключается в том, что они безопасны относительно типов, поэтому вы получаете статическую верификацию создания `SqlProductRepository`. Недостаток – вы не получаете автоматическую интеграцию, поэтому до тех пор, пока вам не нужно явно указывать параметр конструктора, предпочтительнее всего является более простое преобразование с помощью `RegisterType`. Это и есть то, как вы преобразуете `IProductManagementService` в `ProductManagementService`, поэтому он может воспользоваться преимуществом автоматической интеграции.

Теперь вы можете использовать экземпляр `container` для создания новых экземпляров `IProductManagementService`, подобных следующему:

```
var service = container.Resolve<IProductManagementService>();
```

Но постоит, а что насчет управления жизненным циклом? Большинство DI-контейнеров обладают стилем существования по умолчанию. В случае Autofac используемый по умолчанию стиль называется Per Dependency, что то же самое, что и стиль существования Transient. Поскольку он является используемым по умолчанию, вам не нужно было указывать его, но если вы захотите, то можете сделать это следующим образом:

```
builder.RegisterType<ContractMapper>()  
    .As<IContractMapper>()  
    .InstancePerDependency();
```

Обратите внимание на то, что вы используете свободный интерфейс регистрации для того, чтобы определить *область применения* экземпляра (термин Autofac используемый вместо термина "стиль существования") при помощи метода InstancePerDependency.

Также существует Single Instance Scope (единичная область применения), которая соответствует стилю существования Singleton. Вооружившись этими знаниями, вы можете создать Autofac – аналог лисгинга 8-2:

```
builder.RegisterType<ContractMapper>()  
    .As<IContractMapper>()  
    .SingleInstance();  
builder.Register((c, p) =>  
    new SqlProductRepository(connectionString))  
    .As<ProductRepository>();  
builder.RegisterType<ProductManagementService>()  
    .As<IProductManagementService>();
```

Вы хотите, чтобы ContractMapper имел стиль существования Singleton, поэтому вы определяете это путем вызова метода SingleInstance. Когда дело касается SqlProductRepository, все становится немного сложнее, поскольку экземпляр SqlProductRepository должен быть Transient, но внедренная строка соединения должна быть Singleton. Вы можете достичь этого, извлекая connectionString из конфигурации приложения (не продемонстрировано, но похоже на то, как это делалось ранее) и используя эту внешнюю переменную в рамках *замыкания* (closure), которое вы применяете для определения конструктора. Поскольку connectionString является внешней переменной, она остается неизменной в рамках множества вызовов конструктора. Обратите внимание на то, как безоговорочно вы сделали и SqlProductRepository, и ProductManagementService, не указывая при этом стиль существования.

Несмотря на то, что этот пример описывает, как определять стили существования с помощью Autofac, другие DI-контейнеры имеют для той же цели приблизительно такое же API.

Возможность точной настройки стиля существования каждой зависимости важна с точки зрения представления, но также важна и с точки зрения функциональности. Например, паттерн проектирования Mediator (Посредник) полагается на совместно используемый директор, посредством которого взаимодействуют несколько компонентов. Это работает только, когда Mediator совместно используется рассматриваемыми сотрудниками.

До настоящего момента мы обсуждали то, как инверсия управления намекает на то, что потребители не могут управлять жизненными циклами своих зависимостей, поскольку они, очевидно, не контролируют процесс создания объектов, и поскольку .NET использует

механизм сборки мусора (garbage collection) пользователи не могут явным образом уничтожать объекты.

Таким образом, вопрос "а как насчет устранимых зависимостей?" остается без ответа. Сейчас мы обратим свой взор к этому деликатному вопросу.

8.2. Работа с устранимыми зависимостями

Несмотря на то, что .NET – это управляемая платформа, имеющая сборщик мусора, она все еще может взаимодействовать с *неуправляемым кодом* (unmanaged code). Когда это происходит, .NET код взаимодействует с неуправляемой памятью, которая не уничтожается сборщиком мусора. Для предотвращения утечки памяти нам нужен механизм, с помощью которого мы бы детерминированно выпускали неуправляемую память. Это и является ключевой целью интерфейса `IDisposable`.

Вероятнее всего, что некоторые реализации зависимостей будут содержать в себе неуправляемые ресурсы. К примеру, ADO.NET связи являются устранимыми, поскольку они стремятся использовать неуправляемую память, поэтому реализации, связанные с базами данных, например, репозитории, скорее всего, сами являются устранимыми.

Как нам следует моделировать устранимые зависимости? Должны ли мы также позволять абстракциям быть устранимыми? Все это могло бы выглядеть так

```
public interface IMyDependency : IDisposable { }
```

Технически это возможно, но не особенно предпочтительно, поскольку является примером smell(дурно пахнущее)-проектирования, который указывает на leaky-абстракцию ("дырявую" абстракцию).

Интерфейс ..., в большинстве случаев, не должен быть устранимым. Не существует способа, с помощью которого тот, кто определяет интерфейс, смог бы предвидеть все возможные реализации этого интерфейса – вы всегда можете столкнуться с устранимой реализацией практически любого интерфейса.

Николас Блумхардт, участник форума Common Context Adapters

Если вы стремитесь добавить `IDisposable` в ваш интерфейс, то это возможно, поскольку вы имеете ввиду конкретную реализацию. Но вы не должны использовать это знание при проектировании интерфейса. Все это усложнило бы реализацию интерфейса для других классов и внесло бы в абстракцию неопределенность. Кто отвечает за уничтожение устранимых зависимостей? Может ли за это отвечать потребитель?

Использование устранимых зависимостей

В целях рассуждений представьте себе, что у нас есть устранимая абстракция, подобная абстрактному классу `OrderRepository`:

```
public abstract class OrderRepository : IDisposable
```

Как класс `OrderRepository` должен взаимодействовать с зависимостью? Большинство руководств по проектированию (включая `FxCop` и встроенный в `Visual Studio Code Analysis`) настаивало бы на том, что, если бы класс содержал устранимый ресурс в качестве члена класса, то он сам должен был бы реализовывать `IDisposable` и избавляться от ресурса, подобного следующему:

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        this.repository.Dispose();
    }
}
```

Но, оказывается, что это совсем плохая идея, поскольку член `repository` был внедрен первоначально, и он может совместно использоваться другими потребителями, как это продемонстрировано на рисунке 8-6.

Рисунок 8-6: Единичный экземпляр `SqlOrderRepository` внедряется и в `OrderService`, и в `SupplierReorderPolicy`. Эти два экземпляра используют одну и ту же зависимость. Если `OrderService` уничтожит свой внедренный `OrderRepository`, то он разрушит зависимость `SupplierReorderPolicy`, и когда `SupplierReorderPolicy` попытается использовать эту зависимость, возникнет исключение.



Было бы менее опасным не удалять внедренный репозиторий, но это означает, что мы, по существу, игнорируем тот факт, что абстракция является устранимой. Другими словами, объявление абстракции как унаследованной от `IDisposable` не приносит никакой пользы.

Кроме того, могут встречаться сценарии, в которых нам нужно сообщать о начале и окончании краткосрочных областей применения, а `IDisposable` иногда используется для этой цели. Перед тем, как рассмотреть то, как `Composer` может управлять жизненным циклом устранимой зависимости, нам следует обсудить то, как работать с такими недолговечными устранимыми объектами.

Создание недолговечных устранимых объектов

Многие API стандартной библиотеки классов .NET используют `IDisposable`, чтобы сообщить о том, что конкретная область применения перестала существовать. Одним из самых выдающихся примеров является WCF-прокси.

WCF-прокси и `IDisposable`

Все авто-генерируемые WCF-прокси реализуют `IDisposable`, поэтому важно не забывать вызывать метод `Dispose` (или `Close`) для прокси, как только это становится возможным. Многие связывания при отправке первого запроса автоматически создают сессию для сервиса, и эта сессия задерживается в сервисе до тех пор, пока не заканчивается ее время, или до тех пор, пока она явным образом не уничтожается.

Если мы забудем уничтожить наши WCF-прокси после использования, количество сессий будет увеличиваться до тех пор, пока мы не достигнем лимита одновременных

соединений из одного и того же источника. Когда мы достигнем лимита, возникнет исключение. Слишком большое количество сессий также привносит в сервис лишнюю нагрузку, поэтому уничтожение WCF-прокси по мере возможности является очень важным.

Чтобы быть полностью технически корректными, нам не приходится вызывать метод `Dispose` для WCF-прокси. Использование метода `Close` приведет к тому же самому результату.

Важно не забывать, что при использовании `IDisposable` для таких целей не нужно указывать *leak*-абстракцию, поскольку эти типы не всегда в первую очередь являются абстракциями. С другой стороны, некоторые из них являются *leak*-абстракциями; а в таких случаях, как должны мы поступать с ними?

К счастью, после уничтожения объекта мы не можем его заново использовать. Это означает, что если мы хотим вызвать то же самое API снова, мы должны создать новый экземпляр. К примеру, это хорошо подходит, когда мы используем WCF-прокси или ADO.NET команды: мы создаем прокси, вызываем его операции и избавляемся от него, как только работа с ним завершается. Как мы можем совместить это с механизмом внедрения зависимостей, если мы считаем, что устранимые абстракции являются *leak*-абстракциями?

Как и всегда может быть полезным скрытие ненужных деталей в интерфейсе. Если мы обратимся к WPF приложению из раздела 7.4, то в данном случае мы спрятали WCF-прокси в интерфейсе `IProductManagementAgent`.

Примечание

Интерфейс `IProductManagementAgent` наиболее значителен в листинге 7-10, но если не брать во внимание этот листинг, мы не рассматривали этот интерфейс подробно. В сущности, такой агент занимает такое же самое место, как и репозиторий, но много лет назад я приобрел привычку называть компоненты доступа к данным *Smart Clients* агентами, а не репозиториями.

С точки зрения `MainViewModel` ниже приведено то, как вы удаляете товар:

```
this.agent.DeleteProduct(productId);
```

Вы просите внедренный `agent` удалить товар. `MainViewModel` может безопасно хранить ссылку на агента, поскольку интерфейс `IProductManagementAgent` не унаследован от `IDisposable`.

При рассмотрении WCF реализации этого интерфейса формируется другая картина. Ниже приведена реализация метода `DeleteProduct`:

```
public void DeleteProduct(int productId)
{
    using (var channel = this.factory.CreateChannel())
    {
        channel.DeleteProduct(productId);
    }
}
```

Класс `WcfProductManagementAgent` не является изменчивым, но имеет внедренную абстрактную фабрику, которую вы можете использовать для создания канала. *Канал* – это еще одно слово для обозначения WCF-прокси, и это еще и авто-генерируемый клиентский интерфейс, который вы получаете бесплатно, когда создаете ссылку на сервис с помощью Visual Studio или `svcutil.exe`. Поскольку этот интерфейс унаследован от `IDisposable`, вы можете вернуть его в оператор `using`.

Вы используете канал для удаления товара. Когда вы покидаете область применения `using`, канал уничтожается.

Но постойте! Не утверждал ли я, что устранимые абстракции являются leaky-абстракциями? Да, утверждал, но мне приходится сопоставлять прагматические сущности с принципами. В данном случае `WcfProductManagementAgent`, абстрактная фабрика `IProductChannelFactory` и `IProductManagementServiceChannel` определены в одной и той же WCF-специфичной библиотеке, выделенной на рисунке 8-7.

Рисунок 8-7: Помимо других типов библиотека `ProductWcfAgent` содержит реализацию `IProductManagementAgent` и поддерживаемые им типы. `WcfProductManagementAgent` использует `IProductChannelFactory` для создания экземпляров `IProductManagementServiceChannel`, которые являются устранимыми. Несмотря на то, что они могут рассматриваться, как leak-абстракции, они не "протекают" слишком далеко, поскольку все потребители и реализаторы находятся в той же самой сборке.



Каждый раз при вызове метода для класса `WcfProductManagementAgent` он незамедлительно открывает новый канал и уничтожает его после использования. Его жизненный цикл весьма краток, поэтому я называю такую устранимую абстракцию недолговечным устранимым объектом.

Обратите внимание на то, что недолговечный устранимый объект никогда не внедряется в потребителя. Вместо него внедряется абстрактная фабрика, и вы используете эту фабрику для того, чтобы контролировать жизненный цикл недолговечных устранимых объектов.

Обобщая все выше сказанное, устранимые абстракции являются leaky-абстракциями. Иногда мы должны принимать такую "протечку" для того, чтобы избежать багов (например, отвергнутые WCF соединения); но если мы так поступаем, то можем сделать все возможное, чтобы эта "протечка" не распространилась по всему приложению.

На данный момент мы рассмотрели то, как использовать устранимые зависимости. Давайте обратим наше внимание на то, как мы можем обслуживать их и управлять ими вместо потребителей.

Управление устранимыми зависимостями

Поскольку я столь непреклонно настаиваю на том, что устранимые абстракции являются leaky-абстракциями, можно сделать вывод, что абстракции не должны быть устранимыми. С другой стороны, реализации иногда бывают устранимыми, и если мы не уничтожим их правильно, в нашем приложении будет происходить утечка ресурсов. Кто-то *должен* их удалять.

Подсказка

Старайтесь реализовывать сервисы так, чтобы они не содержали ссылок на устранимые объекты, а создавали и уничтожали их по требованию, как это проиллюстрировано на рисунке 6-3. Это упрощает процесс управления памятью, поскольку сервис может быть уничтожен сборщиком мусора наряду с другими объектами.

Как всегда данная ответственность ложится на *Composer* (например, DI-контейнер). *Composer* лучше всех остальных знает, в какой момент он создает устранимый экземпляр, поэтому он также знает, что этот экземпляр необходимо уничтожить. Для *Composer* проще хранить ссылку на устранимый экземпляр и вызывать его метод *Dispose* в подходящий момент.

Проблема заключается в определении подходящего времени для уничтожения. Как мы узнаем, в какой момент все потребители вышли за рамки области применения?

До тех пор, пока кто-нибудь не скажет нам, что все потребители вышли за рамки области применения, мы это не узнаем, но чаще всего наш код располагается внутри некоторого рода контекста с вполне определенным жизненным циклом и событиями, которые и сообщают нам, в какой момент завершается конкретная область применения. Таблица 8-1 демонстрирует области применения технологий, которые мы рассматривали в главе 7.

Таблица 8-1: Точки входа и выхода для различных .NET Framework'ов

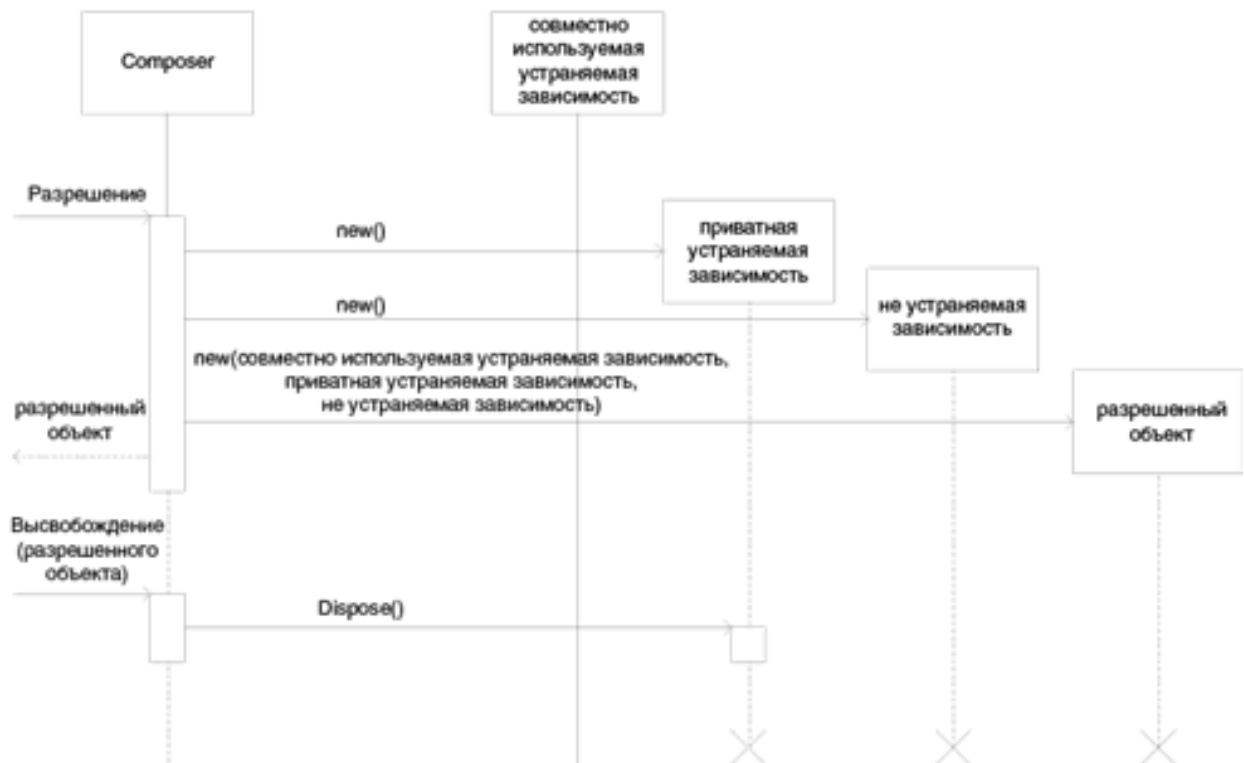
Технология	Точка входа	Точка выхода
Консольное приложение	Main	Main
ASP.NET MVC	<code>IControllerFactory.CreateController</code>	<code>IControllerFactory.ReleaseController</code>
WCF	<code>IInstanceProvider.GetInstance</code>	<code>IInstanceProvider.ReleaseInstance</code>
WPF	<code>Application.OnStartup</code>	<code>Application.OnExit</code>
ASP.NET	Constructors**, <code>Page_Load</code>	<code>IDisposable.Dispose**</code> , <code>Page_Unload</code>
PowerShell	Constructors**	<code>IDisposable.Dispose**</code>

Мы можем использовать различные точки выхода для того, чтобы сообщить `Composer`, что ему необходимо уничтожить все зависимости данного объекта. Поэтому задача отслеживания таких зависимостей и их стилей существования, а также принятие решения по вопросу, должны ли те или иные зависимости высвобождаться, остается в компетенции `Composer`.

Высвобождение зависимостей

Высвобождение диаграммы объектов – это не то же самое, что и ее уничтожение. Это сигнал, сообщаемый `Composer`, что центральная часть диаграммы выходит за рамки области применения, поэтому если сама центральная часть реализует `IDisposable`, то она должна быть уничтожена. Но зависимости этой центральной части могут использоваться также и другими центральными частями, поэтому `Composer` может принять решение о сохранении некоторых из них, поскольку он знает, что другие объекты все еще полагаются на эти зависимости. Рисунок 8-8 иллюстрирует данную последовательность событий.

Рисунок 8-8: Когда `Composer` получает запрос о преобразовании объекта, он собирает все зависимости запрашиваемого объекта. В данном примере запрашиваемый объект имеет три зависимости, и две из них являются устранимыми. Одна из этих устранимых зависимостей также используется и другими потребителями, поэтому она является повторно используемой, тогда как остальные зависимости проиллюстрированы только в одном месте. При получении запроса о высвобождении объекта `Composer` уничтожает приватную устранимую зависимость и разрешает неустранимую зависимость и самому объекту выйти за рамки области применения. Единственной взаимосвязью с повторно используемой зависимостью остается тот факт, что она внедрена в запрашиваемый объект; но так как она является повторно используемой, она пока не уничтожается.



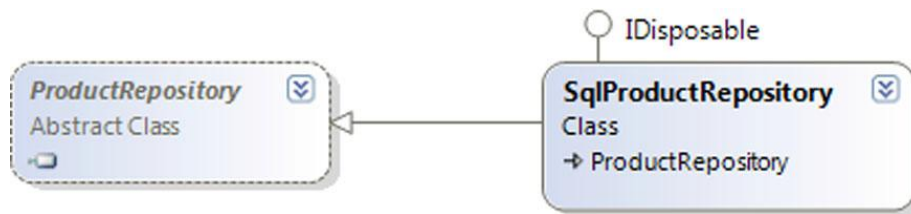
Чтобы высвободить зависимости, `Composer` должен отследить все устраняемые зависимости, которые он когда-либо обслуживал, и тех потребителей, которые используют эти зависимости, таким образом, чтобы он мог уничтожить их, когда будет высвобожден последний потребитель.

Подсказка

Если вы когда-либо работали со счетчиками референсов (или имели опыт работы с багами, являющимися следствием плохой реализации), то вы оцените, каким сложным может быть процесс сохранения обозначений для всех зависимостей и их потребителей. Это именно тот момент, когда вам может пригодиться DI-контейнер, поскольку он берет заботу обо всем этом на себя. Используйте DI-контейнер, а не разрабатывайте свой собственный код отслеживания жизненного цикла. Реализация процесса управления жизненным циклом DI-контейнером гарантированно является более протестированной, нежели все то, что вы можете создать в пределах разумных временных рамок.

Давайте вернемся к примеру WCF сервиса из раздела "Управление жизненным циклом с помощью контейнеров". Оказывается, в листинге 8-2 есть баг, потому что, как демонстрирует рисунок 8-9, `SqlProductRepository` реализует `IDisposable`.

Рисунок 8-9: `SqlProductRepository` реализует `IDisposable`, потому что содержит устраняемый ресурс. Он также наследуется от абстрактного класса `ProductRepository`, который не реализует `IDisposable`.



Код из листинга 8-2 создает новые экземпляры `SqlProductRepository`, но никогда не высвобождает эти экземпляры. Это будет приводить к утечке ресурсов, поэтому давайте устраним этот баг с помощью новой версии специализированного контейнера.

Для начала учтите, что контейнер должен уметь обслуживать множество одновременных запросов, поэтому ему приходится связывать каждый экземпляр `SqlProductRepository` с создаваемым им `IProductManagementService`. Контейнер использует `Dictionary<IProductManagementService, SqlProductRepository>`, называемые репозиториями, для того, чтобы отслеживать эти связи. Следующий листинг демонстрирует, как контейнер преобразовывает запросы экземпляров `IProductManagementService`.

Листинг 8-4: Связывание устраняемых зависимостей с разрешенной центральной частью

```
public IProductManagementService ResolveProductManagementService()
{
    var repository = new SqlProductRepository(this.connectionString);
    var srvc = new ProductManagementService(repository, this.mapper);
    lock (this.syncRoot)
    {
        this.repositories.Add(srvc, repository);
    }
    return srvc;
}
```

Метод начинается с преобразования всех зависимостей. Оно аналогично реализации из листинга 8-2. Но перед тем как вернуть разрешенный сервис, контейнер должен вспомнить связь между сервисом и репозиторием.

В WCF приложении присутствует только один экземпляр контейнера, и поскольку, скорее всего, он будет получать одновременные запросы, вам необходимо заблокировать словарь перед тем, как добавить в него репозиторий. Добавление элементов в словарь не является полностью безопасной операцией, поэтому вам необходимо выполнить блокировку, чтобы убедиться, что все репозитории сохранены для последующих даже внутренних одновременных вызовов.

Если вы обратитесь снова к листингу 7-7, вы заметите, что реализация `IInstanceProvider` уже вызывает метод `Release` для контейнера. До настоящего момента вы не реализовывали этот метод, полагаясь на то, что сборщик мусора выполнит эту работу, но для устранимых зависимостей существенно, чтобы вы осознали эту возможность уничтожения. Ниже приведена эта реализация.

Листинг 8-5: Высвобождение устранимых зависимостей

```
1. public void Release(object instance)
2. {
3.     var srvc = instance as IProductManagementService;
4.     if (srvc == null)
5.     {
6.         return;
7.     }
8.     lock (this.syncRoot)
9.     {
10.         SqlProductRepository repository;
11.         if (this.repositories.TryGetValue(srvc, out repository))
12.         {
13.             repository.Dispose();
14.             this.repositories.Remove(srvc);
15.         }
16.     }
17. }
```

Строка 13: Уничтожает репозиторий

Строка 14: Удаляет репозиторий из словаря

Поскольку метод `Release` принимает любые типы объектов, вам для начала потребуется *граничный оператор* (*Guard Clause*), чтобы убедиться, что `instance` является `IProductManagementService`.

Параллельные цепочки могут вызывать метод `Release` одновременно, поэтому вы должны еще раз сериализовать обращение к словарю `repositories`, чтобы убедиться в том, что параллельные цепочки не искажают состояние этого словаря. Если бы репозитории не были удалены из словаря, то это могло бы привести к утечке памяти.

Переменная `srvc` выступает в роли ключа к словарю, поэтому вы можете использовать ее для поиска устранимой зависимости. Когда вы найдете такую зависимость, вы можете уничтожить ее и удалить из словаря, чтобы удостовериться в том, что контейнер не оставляет ее "живой".

Примеры, продемонстрированные в листингах 8-4 и 8-5, используются специально для работы с одной конкретной устраняемой зависимостью: `SqlProductRepository`. Довольно банально было бы, если бы мы расширили код для того, чтобы иметь возможность работать с зависимостями любого вида, но после этого ситуация усложнилась бы. Представьте, что вам приходится работать с многочисленными устраняемыми зависимостями одного и того же объекта, или вложенными устраняемыми зависимостями, причем часть из них должны быть Singleton'ами, а некоторые должны быть Transient – и это мы еще даже не начинали обсуждать более сложные стили существования!

Подсказка

Окажите себе самому услугу и воспользуйтесь DI-контейнером вместо того, чтобы пытаться решить все эти проблемы в пользовательском коде. Единственная причина, почему я привожу этот пользовательский код – объяснить принципы управления жизненным циклом.

DI-контейнеры могут работать со сложными сочетаниями стилей существования, и предлагают возможности (например, метод `Release`) для явного высвобождения компонентов после завершения работы с ними. Мы должны не забывать использовать эти методы для того, чтобы избежать утечки памяти, особенно в тех случаях, когда одна или более одной зависимости из сконфигурированных являются устраняемыми.

На данный момент мы обсудили в некоторых подробностях механизм управления жизненным циклом. Будучи потребителями, мы не можем управлять жизненным циклом внедренных зависимостей; эта ответственность ложится на плечи `Composer`, который может выбрать между двумя вариантами: разделить один экземпляр между многими потребителями или наделить каждого потребителя своим собственным приватным экземпляром. Эти Singleton и Transient стили существования являются всего лишь самыми универсальными представителями огромного набора стилей существования, и оставшуюся часть главы мы будем использовать для рассмотрения каталога стратегий жизненного цикла.

8.3. Каталог стилей существования объектов

Теперь, когда в предыдущих разделах мы рассмотрели принципы, лежащие в основе механизма управления жизненным циклом, мы потратим оставшуюся часть главы на рассмотрение паттернов универсальных стилей существования.

Примечание

Я буду использовать аналогичные примеры в рамках всего раздела. Но для того чтобы позволить нам с вами сконцентрироваться на самых главных моментах, я буду составлять поверхностные иерархии и иногда игнорировать проблему, связанную с устраняемыми зависимостями, для того, чтобы избежать этой дополнительной сложности.

Поскольку вы уже сталкивались и с Singleton, и с Transient, мы начнем с этих стилей существования, а затем раскроем остальные типы. По мере того, как мы будем продвигаться по этим стилям существования, мы будем переходить от общепринятых стилей существования к более экзотичным, как это описано в таблице 8-2.

Таблица 8-2: Паттерны стилей существования, рассматриваемые в данном разделе

Название	Описание
Singleton	Один экземпляр постоянно повторно используется
Transient	Всегда используются новые экземпляры
Per Graph	Один экземпляр повторно используется в пределах каждой диаграммы объектов
Web Request Context	В большинстве случаев для одного веб-запроса используется один экземпляр каждого типа
Pooled	Используются экземпляры из пула готовых объектов
Lazy	Зависимость, требующая больших затрат, создается и используется в замедленном темпе
Future	Зависимость станет доступна в будущем

Несмотря на то, что вы можете использовать такие стили существования, как Pooled, довольно редко, хорошо бы о них все-таки знать, а этот список должен дать вам хорошее представление о широком ряде доступных стилей существования. По сравнению с более сложными стилями существования, Singleton может казаться более приземленным, но он, тем не менее, является универсальной и уместной стратегией жизненного цикла.

Singleton

В этой книге мы время от времени косвенным образом использовали стиль существования Singleton. Его название одновременно является и достаточно понятным, и несколько запутанным. Оно имеет множество смыслов, поскольку результирующее поведение похоже на паттерн проектирования Singleton, но структура отличается.

Предупреждение

Не путайте стиль существования Singleton с паттерном проектирования Singleton.

В пределах области применения одного Composer компонент со стилем существования Singleton ведет себя приблизительно как Singleton. Всякий раз, когда потребитель выполняет запрос компонента, используется один и тот же экземпляр.

Но на этом схожесть заканчивается. Потребитель не может обратиться к зависимости с Singleton областью применения посредством статического члена, и если мы просим два разных Composer'a передать нам экземпляры, то мы получим два разных экземпляра.

Подсказка

Используйте стиль существования Singleton всякий раз, когда это можно сделать.

Поскольку используется только один экземпляр, стиль существования Singleton в основном потребляет минимальное количество памяти. Единственный момент, когда так не происходит, – это когда экземпляр редко используется, но потребляет чрезмерное количество памяти. В таких случаях стиль существования Lazy с примыкающим экземпляром Transient может стать наилучшей конфигурацией (но я долго пыгался найти разумные примеры таких ситуаций).

Когда использовать Singleton

При возможности используйте стиль существования Singleton. Главной проблемой, которая может не дать вам использовать Singleton, может стать то, что компонент является *поток-безопасным*. Поскольку экземпляр Singleton используется совместно потенциально большим количеством потребителей, он должен уметь управлять одновременными обращениями.

Все сервисы, которые не сохраняют свое состояние, по определению являются потоко-безопасными, как и неизменные типы, и очевидные классы, специально созданные потоко-безопасными. В таких случаях нет причины не делать их Singleton'ами.

Вдобавок к аргументу об эффективности, некоторые зависимости могут работать так, как это было задумано, только если они используются кем-то совместно. Например, это касается реализаций паттерна проектирования Circuit Breaker, а также кэш-оперативной памяти. В этих случаях то, что реализации являются потоко-безопасными, как раз-таки существенно.

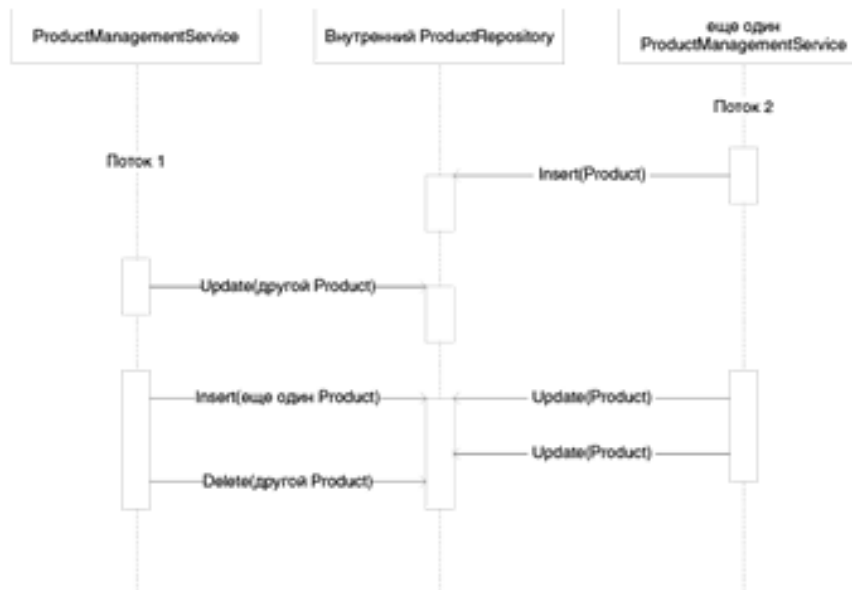
Давайте поближе рассмотрим репозиторий, находящийся в оперативной памяти.

Пример: Использование потоко-безопасного репозитория, находящегося в оперативной памяти

Давайте еще раз обратим наше внимание на реализацию `ICommerceServiceContainer` таким образом, как это описано в разделах "Пример: подключение сервиса управления продуктами", "Управление жизненным циклом с помощью контейнера" и "Управление устраняемыми зависимостями". Вместо использования `ProductRepository`, базирующегося на SQL Server, мы могли бы решить, использовать потоко-безопасную реализацию в оперативной памяти. Для того чтобы хранилище данных, находящееся в

оперативной памяти, имело какой-либо смысл, оно должно совместно использоваться всеми запросами, поэтому оно должно быть потоко-безопасным, как это проиллюстрировано на рисунке 8-10.

Рисунок 8-10: Когда составные экземпляры `ProductManagementService`, идущие отдельными потоками, обращаются к совместно используемому ресурсу, например, находящемуся в оперативной памяти `ProductRepository`, мы должны убедиться, что совместно используемый ресурс является потоко-безопасным.



Вместо явной реализации такого репозитория, как `Singleton`, мы можем использовать конкретный класс и соответствующим образом его ограничить. Единственное требование — он должен быть потоко-безопасным.

Листинг 8-6 демонстрирует, как контейнер может возвращать новые экземпляры всякий раз, когда его просят разрешить `IProductManagementService`, в то время, как `ProductRepository` используется всеми экземплярами совместно.

Листинг 8-6: Управление Singleton'ами

```

1. public class SingletonContainer : ICommerceServiceContainer
2. {
3.     private readonly ProductRepository repository;
4.     private readonly IMapper mapper;
5.     public SingletonContainer()
6.     {
7.         this.repository =
8.             new InMemoryProductRepository();
9.         this.mapper = new IMapper();
10.    }
11.    public IProductManagementService
12.    ResolveProductManagementService()
13.    {
14.        return new ProductManagementService(
15.            this.repository, this.mapper);
16.    }
17.    public void Release(object instance) { }
18. }

```


Строка 3-4: Экземпляры Singleton

Строка 7-9: Создает Singleton'ы

Строка 14-15: Создает сервис

Строка 17: Ничего не делает

Жизненный цикл Singleton довольно легко реализовать: вы храните ссылку на каждую зависимость на протяжении всего жизненного цикла контейнера. Обратите внимание на то, что вы используете ключевое слово `readonly`, чтобы убедиться, что вы не можете случайно изменить ссылки поздней датой. Строгой необходимости в этом для реализации стиля существования Singleton нет, но это обеспечивает некоторую дополнительную безопасность, цена которой – написание восьми букв.

Всякий раз, когда контейнер просит разрешить экземпляр `IProductManagementService`, он создает Transient экземпляр с внедренными в него Singleton'ами. В данном примере и `repository`, и `mapper` являются Singleton'ами, но при желании вы можете смешивать стили существования.

Стиль существования Singleton – один из самых простых для реализации стилей. Все, что для него нужно, – хранить ссылку на объект и использовать этот же объект каждый раз, когда его запрашивают. Экземпляр не выходит за рамки области применения до тех пор, пока не выйдет за рамки области применения `Composer`. Когда это происходит, `Composer` должен избавиться от объекта, если он принадлежит к устранимому типу.

Еще одним простым для реализации стилем существования является стиль Transient.

Transient

Стиль существования Transient включает в себя возврат нового экземпляра всякий раз, когда этот экземпляр запрашивается. До тех пор пока возвращаемый экземпляр не реализует `IDisposable`, следить не за чем. Наоборот, когда экземпляр реализует `IDisposable`, `Composer` должен иметь его ввиду и явным образом избавляться от него, когда его просят высвободить подходящую диаграмму объектов.

Стоит заметить, что в настольных и простых приложениях мы стремимся разрешать полноценную иерархию объектов всего единожды: при запуске приложения. Это означает, даже для Transient компонентов, что будет создано только несколько экземпляров, и что они могут существовать длительное время. В худшем случае, когда на одну зависимость приходится только один потребитель, конечный результат разрешения диаграммы истинных Transient компонентов эквивалентен разрешению диаграммы истинных Singleton'ов или любого их сочетания. Это все потому, что диаграмма разрешается всего лишь раз, поэтому разницы в поведении никогда не возникает.

Когда использовать Transient

Стиль существования Transient – самый безопасный из всех стилей существования, но в то же время и один из самых неэффективных, поскольку он может привести к тому, что будет создано и уничтожено сборщиком мусора бесчисленное множество экземпляров, в то время как одного экземпляра было бы достаточно. Но если вы сомневаетесь насчет

потокowej безопасности компонента, стиль существования `Transient` безопасен, поскольку каждый потребитель имеет свой собственный экземпляр зависимости.

В большинстве случаев мы можем безопасно менять стиль существования `Transient` на такой контекстно-ограниченный стиль существования, как `Web Request Context`, в котором доступ к зависимости также гарантированно сериализуется, но это зависит от среды выполнения (использование `Web Request Context` в настольных приложениях не имеет смысла).

Пример: разрешение разнообразных репозиториев

Ранее в этой главе вы видели несколько примеров использования стиля существования `Transient`. В листинге 8-2 `repository` создается и внедряется в методе, выполняющем разрешение, а контейнер не содержит ссылок на него. Затем в листингах 8-4 и 8-5 вы увидели, как работать с устранимым `Transient` компонентом.

В этих примерах вы могли заметить, что `mapper` повсюду остается Singleton'ом. Это истинный, не сохраняющий свое состояние сервис, поэтому нет необходимости создавать новый экземпляр для каждого вновь созданного `ProductManagementService`. Что действительно заслуживает внимания, так это тот факт, что вы можете сочетать зависимости с различными стилями существования.

Когда разнообразным компонентам нужна одна и та же зависимость, каждый из этих компонентов получает отдельный экземпляр этой зависимости. Следующий листинг демонстрирует метод, который выполняет разрешение ASP.NET MVC контроллера.

Листинг 8-7: Разрешение `Transient DiscountRepository`s

```
1. public IController ResolveHomeController()
2. {
3.     var connStr = ConfigurationManager
4.         .ConnectionStrings["CommerceObjectContext"]
5.         .ConnectionString;
6.     var discountCampaign =
7.         new DiscountCampaign(
8.             new SqlDiscountRepository(connStr));
9.     var discountPolicy =
10.        new RepositoryBasketDiscountPolicy(
11.            new SqlDiscountRepository(connStr));
12.     return new HomeController(
13.         discountCampaign, discountPolicy);
14. }
```

Строка 8: Новый экземпляр `SqlDiscountRepository`

Строка 11: Еще один экземпляр `SqlDiscountRepository`

Как для класса `DiscountCampaign`, так и для класса `RepositoryBasketDiscountPolicy` нужна зависимость `DiscountRepository`. В случае, когда `DiscountRepository` является `Transient`, каждый потребитель получает свой собственный приватный экземпляр, поэтому `DiscountCampaign` получает один экземпляр, а `RepositoryBasketDiscountPolicy` — другой.

Стиль существования `Transient` означает, что каждый потребитель получает приватный экземпляр зависимости даже в тех случаях, когда разнообразные потребители в одной и

той же диаграмме объектов обладают одной и той же зависимостью (как в случае с листингом 8-7). Если множество потребителей совместно используют одну и ту же зависимость, то данный подход будет неэффективным, но если реализация является потоко-безопасной, то в этом случае наиболее эффективный стиль существования Singleton не подходит. В таких ситуациях может больше подойти стиль существования Per Graph.

Per Graph

Singleton – это наиболее эффективный стиль существования, а Transient – самый безопасный, но можем ли мы разработать такой стиль существования, который сочетал бы в себе преимущества этих двух стилей? Несмотря на то, что мы не можем получить самое лучшее от этих двух стилей, в некоторых случаях имеет смысл распределить единичный экземпляр по всей единичной разрешенной диаграмме. Мы можем рассматривать это, как некоторого рода локально-ограниченный Singleton. Мы можем использовать общий экземпляр в рамках единичной диаграммы объектов, но не разделяем этот экземпляр с другими диаграммами.

Каждый раз при разрешении диаграммы объектов мы создаем только один экземпляр каждой зависимости. Если эта зависимость используется несколькими потребителями, они разделяют между собой один и тот же экземпляр; но при разрешении новой диаграммы объектов мы создаем новый экземпляр.

Когда использовать Per Graph

В большинстве случаев мы можем использовать стиль существования Per Graph там, где могли бы использовать и Transient. Обычно мы допускаем, что поток, разрешающий диаграмму объектов, также является единственным потребителем этой диаграммы объектов. Даже когда рассматриваемая зависимость не является потоко-безопасной, мы можем использовать стиль существования Per Graph, поскольку общий экземпляр совместно используется только потребителями, которые проходят по одному и тому же потоку.

В редких случаях, когда один или более одного потребителя проносятся по новым потокам и используют зависимость из этих потоков, Transient все еще остается самым безопасным стилем существования, но такое случается редко. Могут встречаться и такие ситуации, когда зависимость представляет собой изменчивый ресурс, а для каждого потребителя необходимо его собственное приватное состояние. В этом случае Transient будет корректным жизненным циклом, поскольку он дает гарантию того, что экземпляры никогда не будут использоваться коллективно.

По сравнению с Transient, при использовании Per Graph не возникает никаких дополнительных издержек, поэтому мы часто можем использовать его в качестве замены Transient. Несмотря на то, что издержки отсутствуют, нам также не гарантировано и какое-либо преимущество. Мы получаем только некоторое усиление эффективности в случае, если диаграмма единичного объекта содержит многочисленных потребителей одной и той же зависимости. В этом случае мы можем разделить экземпляр между этими потребителями; но если совместно используемые зависимости отсутствуют, то будет нечего делить, и поэтому не будет никаких преимуществ.

Примечание

В большинстве случаев `Per Graph` лучше `Transient`, но не многие DI-контейнеры его поддерживают.

В случаях, когда реализация является потоко-безопасной, наиболее эффективным вариантом все еще остается стиль существования `Singleton`.

Пример: Совместное использование `Repository` в рамках диаграммы

В листинге 8-7 вы видели, как каждый потребитель получал свой собственный приватный экземпляр `SqlDiscountRepository`. Этот класс не является потоко-безопасным, поэтому вам не следует конфигурировать его в виде `Singleton`. Но вы не рассчитываете на то, что разнообразные потоки будут обращаться к индивидуальным экземплярам `HomeController`, поэтому разделение экземпляра `SqlDiscountRepository` между двумя потребителями будет безопасным. Следующий листинг демонстрирует, как создать единственный экземпляр `Per Graph` в методе `ResolveHomeController`.

Листинг 8-8: Разрешение единичного `per graph repository`

```
1. public IController ResolveHomeController()
2. {
3.     var connStr = ConfigurationManager
4.         .ConnectionStrings["CommerceObjectContext"]
5.         .ConnectionString;
6.     var repository =
7.         new SqlDiscountRepository(connStr);
8.     var discountCampaign =
9.         new DiscountCampaign(repository);
10.    var discountPolicy =
11.        new RepositoryBasketDiscountPolicy(repository);
12.    return new HomeController(discountCampaign, discountPolicy);
13. }
```

Строка 6-7: Совместно используемый экземпляр `SqlDiscountRepository`

Строка 8-11: Внедрение совместно используемого экземпляра

Вместо того чтобы создавать отдельные экземпляры для всех потребителей, вы создаете один экземпляр, который можете разделить между всеми потребителями. Вы внедряете этот единственный экземпляр как в `DiscountCampaign`, так и в `RepositoryBasketDiscountPolicy`. Обратите внимание на то, что по сравнению с `Singleton`'ами, в которых совместно используемый экземпляр является приватным членом контейнера, экземпляр `repository` является локальным по отношению к методу `ResolveHomeController`; при следующем вызове метода будет создан новый экземпляр, а затем он будет разделен между двумя потребителями.

Стиль существования `Per Graph` является хорошей альтернативой стилю `Transient` в тех случаях, когда единственной причиной отказа от использования `Singleton` является тот факт, что реализация не потоко-безопасна. Несмотря на то, что `Per Graph` предлагает в целом удобное решение для разделения зависимостей в рамках четко определенных границ, существуют другие, более специализированные альтернативы.

Web Request Context

Как пользователям приложения нам хотелось бы получать ответ от этого приложения как можно быстрее, даже в тех случаях, когда в это же самое время оно используется и другими пользователями. Нам бы не хотелось, чтобы наш запрос был поставлен в очередь наряду с запросами всех остальных пользователей. Возможно, нам пришлось бы чрезмерно долго ждать ответа, если бы перед нашим запросом было бы много других запросов.

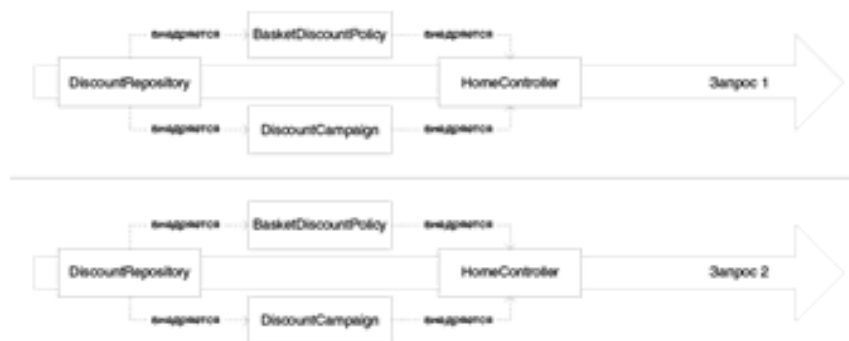
Чтобы решить эту проблему, веб-приложения управляют запросами одновременно. .NET инфраструктура защищает нас от этого, позволяя каждому запросу выполняться в своем собственном контексте и со своим собственным экземпляром Controller (если вы используете ASP.NET MVC) или Page (если вы используете ASP.NET Web Forms).

В связи с параллельным выполнением зависимости, которые не являются потоко-безопасными, не могут использоваться как Singleton'ы. С другой стороны, использование их в виде Transient может быть неэффективным или даже совершенно проблематичным в тех случаях, если нам нужно разделить зависимость между различными потребителями в рамках одного и того же запроса.

Несмотря на то, что ASP.NET движок не гарантирует, что один запрос будет выполняться исключительно в одном потоке, он гарантирует, что код выполняется последовательно. Это означает, что если мы хотим разделить зависимость только в рамках одного запроса, то потоко-безопасность в данном случае не является проблемой.

Рисунок 8-11 демонстрирует, как работает стиль существования Web Request Context. Зависимости ведут себя как Singleton'ы в пределах одного запроса, но не разделены между запросами. Каждый запрос имеет свой собственный набор связанных зависимостей.

Рисунок 8-11: Стиль существования Web Request Context указывает на то, что мы создаем не более одного экземпляра на один запрос. Экземпляр `DiscountRepository` совместно используется `BasketDiscountPolicy` и `DiscountCampaign`, но только в рамках запроса 1. Запрос 2 использует ту же самую конфигурацию, но экземпляры ограничены этим запросом.



Любые устранимые компоненты должны быть уничтожены после окончания запроса.

Когда использовать WebRequest Context

Очевидно, что стиль существования Web Request Context имеет смысл только в веб-приложении. Даже в пределах веб-приложения он может быть использован только в

запросах. Несмотря на то, что запросы составляют большую часть веб-приложения, стоит отметить, что если мы проносимся по исходному потоку с целью асинхронной обработки, то этот стиль существования не применим, поскольку исходный поток не будет синхронизироваться с веб-запросом.

Стиль существования `Web Request Context` предпочтительнее `Transient`, но стиль существования `Singleton` все еще более эффективен. Используйте `Web Request Context` только в тех ситуациях, в которых `Singleton` не работает.

Примечание

Если вы следуете общему совету разрешать только одну диаграмму объектов для одного веб-запроса, то стили существования `Web Request Context` и `Per Graph` функционально эквивалентны.

Подсказка

Если вам когда-нибудь понадобится сформировать в веб-запросе `Entity FrameworkObjectContext`, то самый лучший для этого стиль существования – `Web Request Context`. Экземпляры `ObjectContext` не являются потоко-безопасными, но в данном случае на один веб-запрос должен быть только один `ObjectContext`.

Не все DI-контейнеры поддерживают данный стиль существования, поэтому очевидно мы можем использовать его только, если он доступен.

Подсказка

Некоторые DI-контейнеры позволяют вам создавать свои собственные расширения стилей существования, поэтому, возможно, это вариант в тех случаях, если выбранный вами контейнер не поддерживает стиль существования `Web Request Context`. К тому же это может быть не тривиальной затеей.

Что касается других стилей существования, то мы можем сочетать стили существования, например, таким образом, что некоторые из них конфигурируются как `Singleton`'ы, а другие совместно используются в рамках веб-запросов.

Пример: компоновка `HomeControllerRepository`, разделенным между веб-запросами

В данном примере вы увидите, как компоновать `ASP.NET MVC` экземпляр `HomeController` с зависимостями, если для двух этих зависимостей необходим `DiscountRepository`. Эта ситуация обрисована на рисунке 8-11: для `HomeController` нужны `BasketDiscountPolicy` и `DiscountCampaign`, а для этих двух зависимостей, в свою очередь, нужен `DiscountRepository`.

Примечание

Шаблонный код данного раздела сложнее того, что гарантируется одноразовым решением. Я не думаю, что вам когда-нибудь придется писать пользовательский код

жизненного цикла Web Request Context, подобный данному, но мне хочется показать вам, как он работает. Вместо написания такого кода используйте DI-контейнер, который поддерживает этот стиль существования.

Вам хотелось бы использовать совместный `SqlDiscountRepository`, но так как этот класс не является потоко-безопасным, вы не можете коллективно использовать его в качестве Singleton. Вместо этого вы будете использовать его совместно в рамках каждого веб-запроса. Специализированный контейнер компонует экземпляры `HomeController` так, как это продемонстрировано в следующем листинге.

Листинг 8-9: Компоновка `HomeController`

```
1. public IController ResolveHomeController()
2. {
3.     var discountPolicy =
4.         new RepositoryBasketDiscountPolicy(
5.             this.ResolveDiscountRepository());
6.     var campaign = new DiscountCampaign(
7.         this.ResolveDiscountRepository());
8.     return new HomeController(
9.         campaign, discountPolicy);
10. }
```

Строка 5,7: Делегирует резолюцию репозитория

Строка 8-9: Возвращает сформированный `HomeController`

На данный момент с большинством операторов этого метода вы должны быть уже знакомы. Единственное, что стоит отметить, – это то, что вы делегируете резолюцию `DiscountRepository` отдельному методу. Данный метод гарантирует, что на один веб-запрос разрешается хотя бы один экземпляр.

При запросе разрешения `DiscountRepository` контейнер должен проверить, существует ли уже экземпляр, связанный с веб-запросом. Если он существует, то возвращается данный экземпляр; иначе экземпляр создается и связывается с веб-запросом перед тем, как будет возвращен. Как показывает следующий листинг, в ASP.NET (как в MVC, так и в Web Forms) вы можете использовать текущий `HttpContext` для поддержания этой связи в работоспособном состоянии.

Листинг 8-10: Разрешение зависимости, ограниченной контекстом веб-запроса

```
1. protected virtual DiscountRepository ResolveDiscountRepository()
2. {
3.     var repository = HttpContext.Current
4.         .Items["DiscountRepository"]
5.         as DiscountRepository;
6.     if (repository == null)
7.     {
8.         var connStr = ConfigurationManager
9.             .ConnectionStrings["CommerceObjectContext"]
10.            .ConnectionString;
11.         repository = new SqlDiscountRepository(connStr);
12.         HttpContext.Current
13.             .Items["DiscountRepository"] = repository;
14.     }
15.     return repository;
16. }
```

Строка **3-5**: Выполняет поиск репозитория в контексте запроса

Строка **12-13**: Сохраняет репозиторий в контексте запроса

Особенностью стиля существования Web Request Context является повторное использование экземпляров уже связанных с текущим запросом, поэтому первое, что необходимо сделать, – проверить, существует ли уже необходимый экземпляр. Если он существует, то вы можете его вернуть. Если экземпляр не найден, то вы должны создать его и связать с текущим веб-запросом перед тем, как вернуть этот экземпляр.

При первом вызове метода `ResolveDiscountRepository` он создает репозиторий и связывает его с запросом таким образом, чтобы каждый последующий вызов повторно использовал один и тот же экземпляр.

При завершении запроса вы могли бы оставить устраняемую зависимость в веб-запросе, что привело бы к утечке памяти, поэтому при завершении запроса вы должны убедиться еще и в том, что все зависимости высвобождены. Один из способов осуществления этого – зарегистрировать пользовательский `IHttpModule`, который присоединяется к событию `EndRequest` для того, чтобы надлежащим образом уничтожить все устраняемые зависимости. В следующем листинге продемонстрирован пример реализации.

Листинг 8-11: Высвобождение устраняемых зависимостей, ограниченных контекстом веб-запроса

```
1. public class DiscountRepositoryLifestyleModule : IHttpModule
2. {
3.     public void Init(HttpApplication context)
4.     {
5.         context.EndRequest += this.OnEndRequest;
6.     }
7.     public void Dispose() { }
8.     private void OnEndRequest(object sender, EventArgs e)
9.     {
10.         var repository = HttpContext.Current
11.             .Items["DiscountRepository"];
12.         if (repository == null)
13.         {
14.             return;
15.         }
16.         var disposable = repository as IDisposable;
17.         if (disposable != null)
18.         {
19.             disposable.Dispose();
20.         }
21.         HttpContext.Current
22.             .Items.Remove("DiscountRepository");
23.     }
24. }
```

Строка **10-11**: Выполняет поиск репозитория в контексте запроса

Строка **16-20**: Уничтожает репозиторий

Строка **21-22**: Удаляет репозиторий из контекста запроса

При завершении веб-запроса вы предпринимаете попытку поиска репозитория в контексте запроса. При обнаружении репозитория вы можете уничтожить его, если это можно

сделать. Независимо от того, является ли этот репозиторий устранимым или нет, вы должны не забыть удалить его из контекста запроса.

Стиль существования `Web Request Context` связывает зависимость с текущим запросом, сохраняя и извлекая этот запрос посредством `HttpContext.Current`. Данный пример продемонстрировал специализированное решение, но методика может быть обобщена таким образом, чтобы произвольное количество зависимостей множества различных типов можно было связывать с контекстом запроса. Это относится к сфере соответствующего DI-контейнера.

Вариация: `Session Request Context`

Редко встречающейся вариацией стиля существования `Web Request Context` является стиль существования, при котором область применения жизненного цикла зависимостей связана не с конкретным запросом, а с сессией. Это намного более экзотичный стиль существования, и если вы решите его использовать, то вам следует делать это с чрезвычайной осторожностью.

С технической точки зрения может показаться, что он похож на `Web Request Context`, но наиболее важное отличие заключается в том, что, несмотря на то, что HTTP-запрос имеет вполне определенный жизненный цикл, сессии не обладают таким определенным жизненным циклом. Сессия в редких случаях заканчивается явным образом, а чаще всего завершается после некоторого периода бездействия. Это означает, что все зависимости, зарегистрированные данным образом, вероятнее всего, длительное время сохраняются там, где они не используются. Все это время они занимают память, что может сильно повлиять на производительность приложения.

Предупреждение

Используйте стиль существования `Session Request Context` только, если он вам действительно необходим. Вероятнее всего, использование данного стиля понизит производительность вашей системы.

Подсказка

Если вам нужно привязать некие зависимости к сессии, то вам лучше сконфигурировать их с помощью `Web Request Context` и посредством фабрики, которая присоединяет каждый экземпляр на основании соответствующего ключа сессии. Данный подход позволяет вам более явным образом управлять жизненным циклом зависимости, несмотря на то, что она все еще связана с сессией.

Еще одна проблема, с которой мы сталкиваемся, – это то, что состояние сессии может быть сохранено во внешнем хранилище, например на отдельном сервере сессий или на `SQL Server session state`. При таких конфигурациях должны быть сериализованы все данные сессии, а также зависимости, на которые оказывается влияние. Сериализация типа может быть так же проста, как и наделение этого типа атрибутом `[Serializable]`, но существует еще кое-что, что мы должны не забыть сделать.

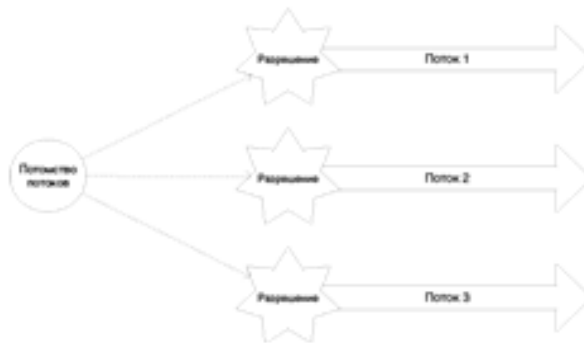
В целом, я считаю `Session Request Context` непривлекательным и не припомню, чтобы когда-либо видел, как он используется.

Вариация: Thread Context

Еще одна, более привлекательная вариация стиля существования Web Request Context, – это ассоциирование зависимости с конкретным потоком. Сущность остается той же: в каждом потоке управление зависимостью ведется как с Singleton, но для каждого потока существует свой экземпляр зависимости.

Данный подход наиболее полезен в сценариях, при которых мы проносимся по многочисленным эквивалентным рабочим потокам и используем начало каждого потока в качестве Composition Root. Эта ситуация проиллюстрирована на рисунке 8-12.

Рисунок 8-12: Когда приложение незамедлительно проносится по некоторому количеству параллельных задач и разрешает зависимости в рамках каждого потока, мы можем использовать стиль существования Thread Context для того, чтобы убедиться, что любые зависимости, не являющиеся потоко-безопасными, могут быть разделены между любым количеством потребителей одного и того же потока. Каждый поток имеет свои собственные экземпляры.



Чтобы реализовать стиль существования Thread Context, мы можем поискать запрашиваемую зависимость в *Thread Local Storage* (TLS). Если мы ее там обнаружим, то можем повторно ее использовать; иначе, мы создаем эту зависимость и сохраняем ее в TLS.

Между тем, как Session Request Context может быть совершенно опасным, а Thread Context – слегка экзотичным, стиль существования Web Request Context полезен. Он позволяет нам разделять зависимости в пределах веб-запроса, не беспокоясь при этом о том, являются ли они потоко-безопасными. Web Request Context – это нечто среднее между Singleton и Transient.

Web Request Context является более эффективной альтернативой стилю существования Transient, но мы можем использовать его только в веб-приложениях. Если мы имеем зависимости, для управления которыми требуются большие затраты, в других типах приложений, то мы можем обратиться к другим возможностям оптимизации.

Pooled

Иногда для создания компонентов необходимы большие затраты. Универсальное решение этой проблемы – возможность легкого доступа к пулу уже созданных компонентов. Хорошо известный пример – подключения баз данных, которые практически всегда находятся в пуле. Мы автоматически применяем организацию пулов подключений баз данных, и можем использовать ту же самую технологию, если у нас есть пользовательские компоненты, создание которых требует больших затрат.

Несмотря на то, что общая концепция объектов в пуле должна быть вам знакома, в таблице 8-3 перечислены некоторые вариации реализации.

Таблица 8-3: Варианты реализации пулов объектов

Вариант	Описание
Подготовка пула	Как мы подготавливаем пул? Создаем ли мы все объекты в пуле заранее или наполняем его постепенно по мере поступления запросов? Для того чтобы заполнить пул заранее, необходимо, чтобы мы знали, по крайней мере, первоначальный размер пула Возможно это тоже затратная операция, поскольку целью пула является обеспечение легкого доступа к объектам, требующим больших затрат. Но преимущество такого подхода заключается в том, что объекты становятся доступными для быстрого доступа. Может быть, есть возможность предварительно заполнить пул из исходного потока таким образом, чтобы он мог начать быть полезным объектам во время его наполнения. В противном случае мы можем начать с пустого пула и заполнять его постепенно по необходимости. Это приводит к тому, что в начале время доступа замедляется, но такой подход может помочь хранить пул в нужном размере.
Минимальный размер	Мы можем решить проблему подготовки пула путем ввода настраиваемого минимального размера. Если мы задаем минимальный размер, отличающийся от нуля, то перед тем, как объекты смогут его использовать, пул должен заполниться до этого размера. С другой стороны, при минимальном размере, равном нулю, объекты могут начать использовать его, как только он начнет заполняться.
Максимальный размер	Каков максимальный размер пула?
Поведение пула при достижении максимального размера	Что происходит, когда мы достигаем максимального размера пула? Позволяем ли мы пулу вырасти в размере? Если мы позволим ему это сделать, то мы рискуем попасть в ситуацию, когда у нас закончится память. Если не позволим, то, как, в таком случае, мы должны обрабатывать дополнительные запросы объектов? Один из вариантов – заблокировать вызов до тех пор, пока объект не станет доступным. Но если мы это сделаем, то мы должны, по крайней мере, предоставить вызывающему оператору возможность указания задержки. Еще один вариант – незамедлительно выдавать исключение.
Очистка пула	Сохраняем ли мы пул заполненным до того момента, как приложение начнет закрываться, или же мы начинаем очищать его, если замечаем, что он начинает превышать свой размер?

В таблице представлены все важные концепции, касающиеся пулов объектов. Но как в случае со стилем существования `WebRequest Context` нам не следует разрабатывать свои собственные пользовательские пулы объектов, а следует использовать те, которые предоставляются DI-контейнерами.

Примечание

Не все DI-контейнеры предоставляют стиль существования `Pooled`, поэтому очевидно, что мы можем выбрать этот стиль существования только, если он поддерживается нашим DI-контейнером.

При использовании стиля существования `Pooled`, предоставляемого DI-контейнером, все варианты, описанные в таблице 8-3, могут быть недоступны. Нам придется работать только с тем, что доступно.

Когда использовать стиль существования `Pooled`

Стиль существования `Pooled` вступает в игру в тех ситуациях, когда у нас есть специфичные компоненты, часто используемые, но для создания которых необходимы большие затраты. Даже если создание компонента трудозатратно, нам все еще стоит предпочесть стиль существования `Singleton`, если это возможно, поскольку он позволяет нам работать с единственным экземпляром и расплачиваться за создание объекта всего лишь единожды.

Из этого следует, что организация пула применима только тогда, когда рассматриваемый компонент не должен использоваться коллективно, что часто происходит в тех случаях, когда данный компонент не является потоко-безопасным. Если мы заглянем в веб-приложение, то стиль существования `WebRequest Context` мог бы стать разумной альтернативой; мы должны были понять, что стиль существования `Pooled` используется вне веб-приложений.

Обратите внимание на то, что тот факт, что рассматриваемый компонент можно использовать повторно, является необходимым условием. Если этот компонент обладает обычным жизненным циклом, который исключает возможность повторного использования, то мы не можем организовать его пул. Одним из примером этого является WCF интерфейс `ICommunicationObject`, который имеет вполне определенный жизненный цикл. Когда `ICommunicationObject` находится в состоянии `Closed` или `Faulted`, то он может, по определению, никогда не покидать это состояние. Такой тип объекта не пригоден для организации пула. Мы должны уметь возвращать объект обратно в пул в первоначальном состоянии.

Пример: Повторное использование затратных репозитория

Однажды я работал над проектом, в котором нам нужно было из .NET кода взаимодействовать с мейнфреймом. Ранее консультанты создали неуправляемую COM библиотеку, которая могла взаимодействовать с некоторой конечной точкой мейнфрейма, и мы решили обернуть эту библиотеку в управляемую сборку.

COM библиотека взаимодействовала с мейнфреймом посредством секретного протокола сетевых соединений. Чтобы его использовать, нам приходилось открывать соединение и проходить через квитирование установления связи. После открытия соединения мы могли

передавать сообщения с умеренной скоростью, но открытие соединения занимало много времени.

Давайте посмотрим, как создать пул экземпляров `ProductRepository`, которые могут взаимодействовать посредством такого протокола. В проекте, в который я был вовлечен, мы называли COM библиотеку *Xfer* (видовой), поэтому давайте создадим пул экземпляров `XferProductRepository`.

Примечание

Как и в примере стиля существования `Web Request Context`, я не жду, что вы будете создавать пользовательские системы управления жизненным циклом объектов в пуле. Несмотря на то, что для управления пулами объектов вам следует использовать соответствующий DI-контейнер, я хочу вам показать упрощенный пример для того, чтобы продемонстрировать, как это работает.

Предупреждение

Следующий пример не является потоко-безопасным. Я убрал из него код синхронизации, чтобы сохранить в примере разумный уровень сложности, а потоко-безопасную реализацию я оставляю читателям для самостоятельной практики (всегда хотел так написать).

Данный пример является еще одной вариацией `IServiceContainer`, несколько вариаций которого вы уже видели в этой главе. В следующем листинге продемонстрирована база контейнера.

Листинг 8-12: Схема базы для организации пула контейнера

```
public partial class PooledContainer : IServiceContainer
{
    private readonly IMapper mapper;
    private readonly List<XferProductRepository> free;
    private readonly List<XferProductRepository> used;
    public PooledContainer()
    {
        this.mapper = new IMapper();
        this.free = new List<XferProductRepository>();
        this.used = new List<XferProductRepository>();
    }
    public int MaxSize { get; set; }
    public bool HasExcessCapacity
    {
        get
        {
            return this.free.Count + this.used.Count < this.MaxSize;
        }
    }
}
```

Несмотря на то, что вы планируете организовать пул экземпляров `XferProductRepository`, вы все равно конфигурируете `IMapper` в виде `Singleton`, поскольку он является сервисом, несохраняющим свое состояние.

Для отслеживания процесса организации пула вы используете две коллекции: в одной из них хранятся доступные репозитории, а другая содержит репозитории, которые

используются на текущий момент. При создании и освобождении компонентов вы перемещаете репозитории между этими двумя коллекциями.

Свойство `MaxSize` позволяет вам определить максимальный размер пула, а свойство `HasExcessCapacity` по существу является инкапсулированным вычислением, которое вы можете использовать в условных выражениях для определения того, превышает ли вы по-прежнему размер пула.

При такой вариации пула вы будете наполнять пул постепенно, по мере поступления запросов до тех пор, пока не превысите максимальный размер. Как демонстрирует следующий листинг, вы выдаете исключение, если превышаете максимальный размер и все еще получаете запросы.

Листинг 8-13: Разрешение репозитория из пула

```
1. public IProductManagementService ResolveProductManagementService()
2. {
3.     XferProductRepository repository = null;
4.     if (this.free.Count > 0)
5.     {
6.         repository = this.free[0];
7.         this.used.Add(repository);
8.         this.free.Remove(repository);
9.     }
10.    if (repository != null)
11.    {
12.        return this.ResolveWith(repository);
13.    }
14.    if (!this.HasExcessCapacity)
15.    {
16.        throw new InvalidOperationException(
17.            "The pool is full.");
18.    }
19.    repository = new XferProductRepository();
20.    this.used.Add(repository);
21.    return this.ResolveWith(repository);
22. }
23. private IProductManagementService ResolveWith(
24.     ProductRepository repository)
25. {
26.     return new ProductManagementService(repository,
27.         this.mapper);
28. }
```

Строка 4-9: При возможности выбирает из пула

Строка 10-13: Возвращает из пула

Строка 19-20: Добавляет новый репозиторий

Для того чтобы разрешить экземпляру `IProductManagementService`, вы начинаете работу с проверки того, доступен ли повторно используемый репозиторий. Если он доступен, то вы выбираете репозиторий из коллекции свободных репозиториях и перемещаете его в список используемых. Если вы смогли найти повторно используемый репозиторий, то вы можете незамедлительно вернуть сервис.

Если вы не можете найти доступный репозиторий в пуле, то существует две возможные причины такой ситуации: пул полностью заполнен и используются все репозитории, или вы все еще заполняете пул. Если вы можете обойти граничный оператор, который выполняет проверку заполненности пула, то вы создадите новый экземпляр репозитория и добавляете его в коллекцию используемых репозиториях перед тем, как вернуть скомпонованный сервис.

`ResolveProductManagementService` всего лишь перемещает репозитории из коллекции `free` в коллекцию `used`, поэтому важно освободить сервисы после использования. В следующем листинге продемонстрировано, как это сделать.

Листинг 8-14: Возврат репозитория в пул

```
1. public void Release(object instance)
2. {
3.     var service = instance as ProductManagementService;
4.     if (service == null)
5.     {
6.         return;
7.     }
8.     var repository = service.Repository
9.         as XferProductRepository;
10.    if (repository == null)
11.    {
12.        return;
13.    }
14.    this.used.Remove(repository);
15.    this.free.Add(repository);
16. }
```

Строка 4-7, 10-13: Граничные операторы

Строка 14-15: Возвращает репозиторий в пул

Вернуть репозиторий в пул легко: вы перемещаете его из коллекции используемых репозиториях в коллекцию доступных репозиториях.

Обратите внимание на то, что, даже если этот пример может показаться сложным, я не затронул несколько моментов:

- Пример определенно не является потоко-безопасным. Рабочая реализация должна позволять нескольким потокам разрешать и освобождать экземпляры параллельно.
- Поскольку класс `XferProductRepository` инкапсулирует неуправляемый код, он реализует `IDisposable`. Пока вы продолжаете повторно использовать экземпляры, вам не нужно их уничтожать, но вам точно следует это сделать в тех ситуациях, когда контейнер выходит за рамки области применения. Таким образом, сам контейнер должен реализовывать `IDisposable` и удалять все репозитории с помощью метода `Dispose`.

Организация пула объектов – хорошо известный паттерн проектирования, но он часто инкапсулируется в существующие API; например, ADO.NET использует пулы соединений, но мы не сталкиваемся с ним явно. Только когда нам точно нужно оптимизировать доступ к затратным ресурсам, стиль существования `Pooled` начинает иметь смысл.

Стиль существования Pooled помогает справиться с ситуацией, когда нам нужно оптимизировать использование затратных ресурсов. Это самый последний из универсальных типов стилей существования.

Другие стили существования

Типы стилей существования, рассматриваемые в данной главе, являются самыми универсальными типами, но ваши нужды могут быть более экзотичными, и рассматриваемые стили могут не удовлетворять вашим нуждам. Когда я попал в такую ситуацию, первой моей реакцией было чувство чрезвычайной гордости от того, что я обнаружил редкий и очень краеугольный случай, который требовал от меня использования экзотического элемента из моего программистского инструментария.

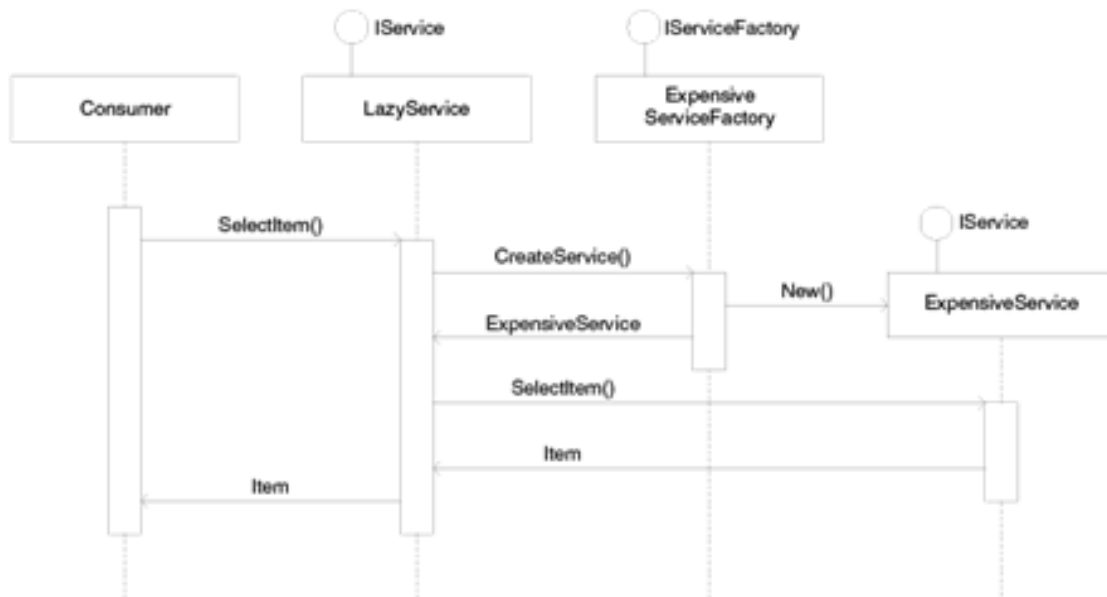
Следующей моей реакцией было осознание того, что мой подход абсолютно неверен, и если я слегка изменю свою модель, то все можно будет вполне подогнать к стандартным паттернам. Такая реализация часто является ухудшением, но приводит к лучшему и более легко сопровождаемому коду. Суть заключается в том, что если вы чувствуете необходимость реализовать пользовательский стиль существования, то, прежде всего, вам необходимо со всей серьезностью пересмотреть вашу модель.

Было сказано, что некоторые DI-контейнеры предоставляют возможности расширяемости, которые позволяют вам разрабатывать пользовательские стили существования. Давайте вкратце рассмотрим два, технически возможных, но довольно экзотичных стиля существования. В обоих случаях я предоставляю только краткую схему того, как работал бы жизненный цикл. Я не отдаю под это целые разделы, поскольку мне было бы трудно придумать разумный сценарий их применения.

Lazy

Стиль существования Lazy или Delayed – это Virtual Proxy более затратной зависимости. Смысл в том, что если у нас есть требующая больших затрат зависимость, которую мы не планируем часто использовать, то мы можем отложить создание затратной зависимости до тех пор, пока она нам не понадобится. Рисунок 8-13 иллюстрирует, как можно внедрить потребителя с легковесным дублером для существующей, более затратной реализации.

Рисунок 8-13: Потребителю необходима зависимость `IService`, но если он использует эту зависимость лишь в небольших фракциях своего жизненного цикла, то он может долгое время просуществовать до того момента, когда ему понадобятся сервисы `IService`. Когда он, в конце концов, вызывает `IService.SelectItem()`, `LazyService` использует его внедренный `IServiceFactory` для создания экземпляра другого `IService`. К данному моменту `ExpensiveService` еще не создан. При создании `ExpensiveService` все последующие вызовы могут быть делегированы ему.



Имеет смысл использовать такой стиль существования только, если потребитель использует затратную зависимость лишь в небольших фракциях своего собственного жизненного цикла, или если мы можем трезво предположить, что до момента вызова зависимости пройдет значительное количество времени. Если зависимость вызывается незамедлительно или часто, то Lazy Decorator ни на что не влияет, но использует дополнительные ресурсы.

Если это возможно, требующая больших затрат зависимость должна регистрироваться как Singleton, чтобы нам нужно было расплачиваться за ее создание лишь единожды. Если это невозможно по причинам потоко-безопасности, мы часто можем разрешить эту головоломку путем организации пула затратного компонента. Даже если чаще всего у нас может быть только один экземпляр, пул этого экземпляра в сочетании с задержкой доступа предоставит нам сериализованный доступ к зависимости.

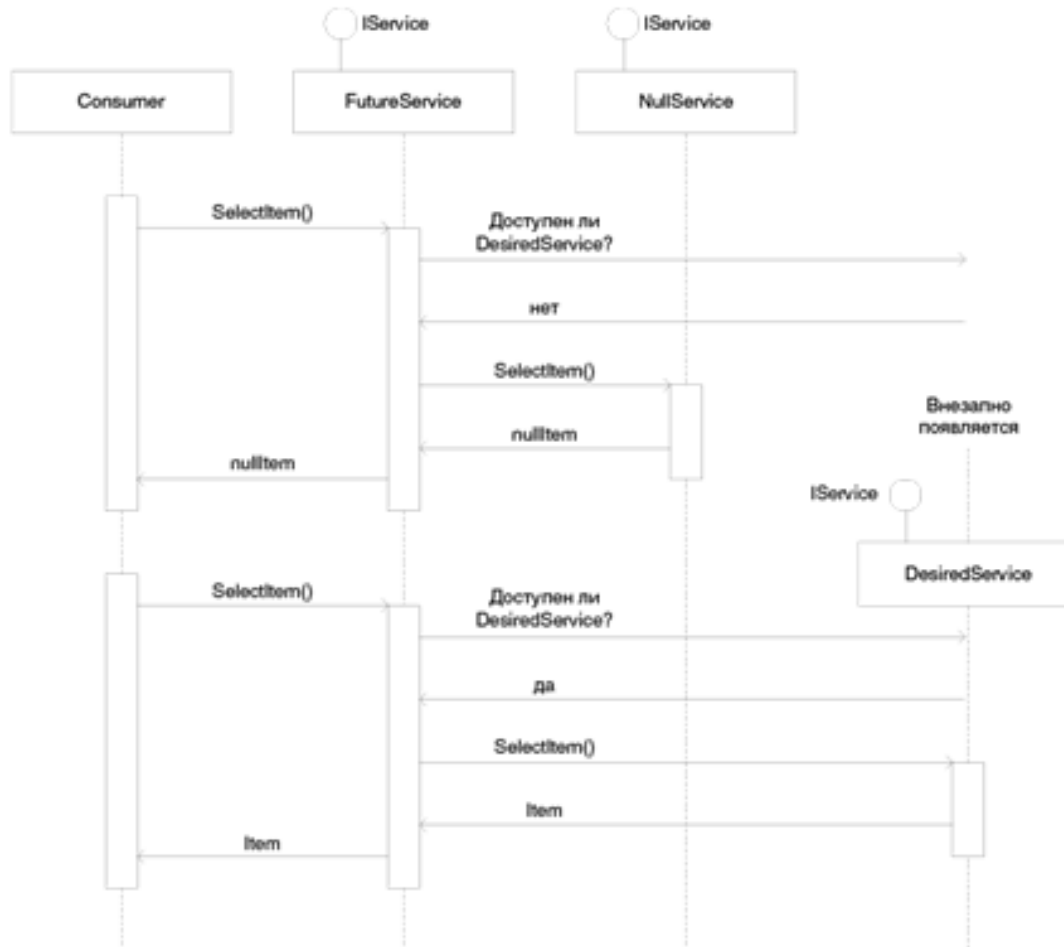
Стиль существования Lazy более интересен с технической точки зрения, нежели практически полезная стратегия жизненного цикла; если вам интересно, то я даю ссылки на рекомендуемую литературу, связанную с данной книгой. (Mark Seemann, "Rebuttal: Constructor over-injection anti-pattern," 2010, <http://blog.ploeh.dk/2010/01/20/RebuttalConstructorOverInjectionAntipattern.aspx>)

Future

Стиль существования Future даже более экзотичен. Смысл в том, что мы можем захотеть использовать зависимость, которая недоступна на данный момент, но которую мы будем использовать, когда она станет доступной.

Наилучший способ реализации такого стиля существования похож на стиль существования Lazy: мы можем использовать Decorator, который делегирует полномочия первоначальной реализации до тех пор, пока нужная зависимость не станет доступной. Рисунок 8-14 иллюстрирует концептуальное взаимодействие между компонентами. Первоначальная реализация, используемая в качестве дублера до тех пор, пока Future Decorator не дожидается нужной зависимости, часто является приложением паттерна проектирования Null Object.

Рисунок 8-14: Потребителю необходима зависимость IService, но DesiredService может быть еще недоступным. В этом случае мы можем инкапсулировать NullService в виде дублера, который будет использоваться до тех пор, пока мы находимся "в ожидании Годо". FutureService – это установленная машина, которая выполняет опрос с целью определения того, стал ли доступным DesiredService. Когда DesiredService недоступен, FutureService Decorator ничего не остается, как использовать резервную реализацию, обеспечиваемую NullService. Когда DesiredService становится окончательно доступным, все последующие запросы направляются к нему.



Я должен согласиться, что я с трудом выжал из себя разумный пример того, когда зависимость может стать доступной после того, как мы присоединили полноценную диаграмму объектов. Это может казаться слегка похожим на ситуацию, когда мы полагаемся на внешний ресурс, например, базу данных или веб-сервис, но не забывайте, что даже если действующий ресурс недоступен, все еще существует программная зависимость; например, веб-сервис может не работать, но WCF прокси, который мы используем для взаимодействия с этим веб-сервисом все еще доступен.

Лучше мы будем иметь дело с недоступными внешними ресурсами, используя паттерн Circuit Breaker, который мы рассмотрим в следующей главе. До тех пор, пока кто-нибудь не предоставит мне разумный сценарий, я считаю стратегию жизненного цикла Future как технически интересную.

На данный момент мы рассмотрели широкий спектр доступных стилей существования зависимостей – от универсальных до более экзотичных.

8.4. Резюме

Когда мы применяем инверсию управления к зависимостям, мы инвертируем управление не только над выбором типа, но также над процессом управления жизненным циклом. Когда потребитель больше не создает свои собственные зависимости, он уже не может постановить, когда была создана зависимость, и была ли она разделена с другими потребителями.

Composer'ы могут принять решение, позволяющее множеству потребителей совместно использовать один и тот же экземпляр, или позволяющие каждому потребителю обладать своим собственным экземпляром. Кроме того, в игру вступают более продвинутые стратегии.

Несмотря на то, что Composer'ы отлично управляют тем, когда создаются объекты, управляемая модель памяти .NET подразумевает, что в большинстве случаев они лишь слегка влияют на то, когда *разрушаются* объекты. Зависимости могут выйти за рамки области применения и быть утилизированы с помощью сборщика мусора. Но особое место отведено компонентам, которые также реализуют `IDisposable`, поскольку мы должны убедиться, что все неуправляемые ресурсы очищены – иначе наши приложения вскоре начнут испытывать утечку памяти.

Равносильно вызову метода `Resolve` (или какое название он имеет) мы должны всегда не забывать вызывать метод `Release`, когда разрешенная диаграмма объектов выходит за рамки области применения. Это дает Composer возможность уничтожить любые устранимые компоненты, которые становятся неиспользуемыми.

Каждая диаграмма объектов может иметь смесь множества различных стилей существования, и нам, кроме того, нужно следить за тем, являются ли компоненты устранимыми. Добавьте к этой смеси потоко-безопасность, и станет сложно отслеживать все эти вещи. Именно здесь и расцветает только что распустившийся DI-контейнер, и это одна из причин того, что нам следует использовать DI-контейнер вместо Poor Man's DI.

Каждый из множества доступных DI-контейнеров предлагает свой собственный доступный стиль существования. Некоторые из них поддерживают только несколько стилей существования, другие DI-контейнеры поддерживают большинство или все стили существования, но многие из них предоставляют возможности расширяемости, которые позволяют нам реализовывать свои собственные стили существования.

Самый безопасный стиль существования – `Transient`, так как экземпляры не делятся с кем-либо еще. Также это самый неэффективный стиль существования, поскольку, скорее всего, в оперативной памяти находится множество экземпляров одного и того же типа.

Наиболее эффективный стиль существования – `Singleton`, потому что в оперативной памяти находится всего один единственный экземпляр (то есть на один контейнер). Но для этого стиля необходимо, чтобы компонент был потоко-безопасным, поэтому не всегда есть возможность использовать данный стиль существования.

Стили существования `WebRequest Context` и `Pooled` обеспечивают хорошую альтернативу `Singleton` и `Transient`, но в более ограниченных сценариях.

Возможны и более экзотичные стили существования. Стил ь существования Future может с первого взгляда казаться отличным способом управления недоступными ресурсами, но как вы увидите в следующей главе, лучше мы обратимся к такой теме, как механизм перехвата.

9. Механизм перехвата

Меню:

- Сквозные сущности
- Аспектно-ориентированное программирование
- Динамический перехват

В кулинарии одно из самых интересных – это то, как мы можем смешивать между собой множество ингредиентов; некоторые из них сами по себе не очень вкусные, а их смесь намного вкуснее. Часто мы начинаем с простого ингредиента, составляющего основу блюда, а затем изменяем и приукрашиваем его до тех пор, пока, в конечном итоге, не получим превосходное блюдо.

Представьте себе говяжью отбивную. Если бы мы находились в полном отчаянии, то могли бы съесть ее сырой, но в большинстве случаев мы бы предпочли ее пожарить. Тем не менее, если мы просто шлепнем ее на горячую сковороду, то результат будет менее выдающимся. За исключением подгоревшего привкуса вы больше ничего не почувствуете.

К счастью, есть множество шагов, которые мы можем предпринять, чтобы повысить свою квалификацию:

- Жарка отбивной в масле предотвратит мясо от подгорания, но вкус все равно останется слабым.
- Добавление соли усилит вкус мяса.
- Добавление других специй, например, перца, сделает вкус мяса более богатым.
- Панировка отбивной в смеси из соли и специй не только добавит ей вкусовых качеств, но также придаст первоначальному ингредиенту новую структуру. На данном этапе мы приближаемся к тому, что называется котлетой.
- Прорезание в отбивной кармашка и добавление в него ветчины, сыра и чеснока перед панировкой поднимет нас на пьедестал почета. Мы получили кордон блю, самое превосходное блюдо.

Разница между подгоревшей отбивной и кордон блю значительна, но исходный ингредиент этих блюд один и тот же. Вариативность вызвана теми ингредиентами, которые мы добавляем к первоначальному. Подав говяжью отбивную, мы можем приукрасить ее, не изменяя при этом главный ингредиент, и создать другое блюдо.

С помощью слабого связывания мы можем выполнить аналогичный трюк при разработке программного обеспечения. Когда мы программируем на основе интерфейсов, мы можем преобразовать или улучшить основную реализацию, завернув ее в другие реализации данного интерфейса. Вы уже видели часть этой методики в действии в разделе 8.3.6 "Другие стили существования", где мы использовали ее для изменения жизненного цикла затратной зависимости, обертывая ее в Proxy.

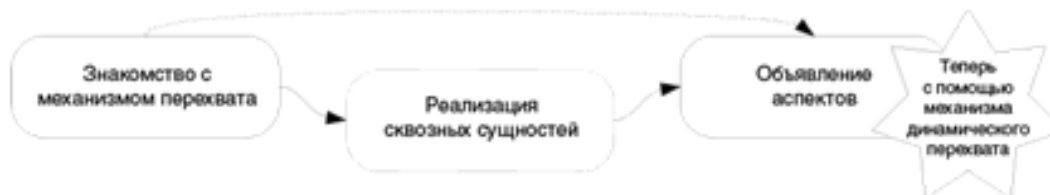
Этот подход может быть обобщен, что предоставляет нам возможность перехватить запрос потребителя к сервису; именно это мы и будем рассматривать в данной главе. Подробно говяжьей отбивной мы начинаем с основного ингредиента и добавляем больше ингредиентов для его улучшения, но, не меняем при этом его первоначальную суть. Механизм перехвата – одна из самых мощных возможностей, полученных нами благодаря

слабому связыванию. Он позволяет нам с легкостью применять принцип единичной ответственности и концепцию разделения.

В предыдущих главах мы потратили множество усилий на перебрасывание нашего кода на ту позицию, где он действительно слабо связан. В данной главе мы начнем пожинать преимущества данного вложения.

Рисунок 9-1 демонстрирует обзор структуры данной главы. После окончания изучения главы вы должны уметь использовать механизм перехвата для разработки слабо связанного кода согласно установленным принципам объектно-ориентированного проектирования. В частности вы должны приобрести способность успешно соблюдать концепцию разделения и применять сквозные сущности, при этом сохраняя код в отличном состоянии.

Рисунок 9-1: Вся структура данной главы достаточно линейна. Мы начнем с ознакомления с механизмом перехвата, включая пример. Далее мы перейдем к разговору о сквозных сущностях. Данный раздел главы не обременен теорией, но наполнен примерами, поэтому, если вы уже знакомы с данной темой, вы можете перейти прямо к последнему разделу, касающемуся аспектов. Этот раздел является кульминационной точкой главы, так как знакомит вас с усовершенствованной и, кроме того, гибкой концепцией механизма перехвата.



Поскольку понять, как работает механизм перехвата, не сложно, мы начнем с небольшого примера для того, чтобы определить контекст. Для того чтобы полноценно оценить возможности, мы должны изучить некоторые связанные понятия, например, *аспектно-ориентированное программирование* (AOP) и *SOLID принципы*, и связать их с механизмом перехвата с помощью примеров. В конечном счете, вы увидите, как можно использовать DI-контейнер для того, чтобы обобщить механизм перехвата и облегчить его применение.

Поскольку в основе механизма перехвата лежат хорошо известные паттерны проектирования и принципы объектно-ориентированного проектирования, данная глава наполнена различными примерами. Маршрут главы вполне прямолинеен, начинается с вводного примера и вырастает до более сложных понятий и примеров. Последнее и более продвинутое понятие можно быстро объяснить с помощью абстракции, но, поскольку это понятие, скорее всего, будет вам понятно только на примере, глава завершается примером, занимающим несколько страниц, который демонстрирует, как все работает.

Но прежде чем мы до этого доберемся, мы должны начать с самого начала.

[9.1. Знакомство с механизмом перехвата](#)

[9.2. Реализация сквозных сущностей](#)

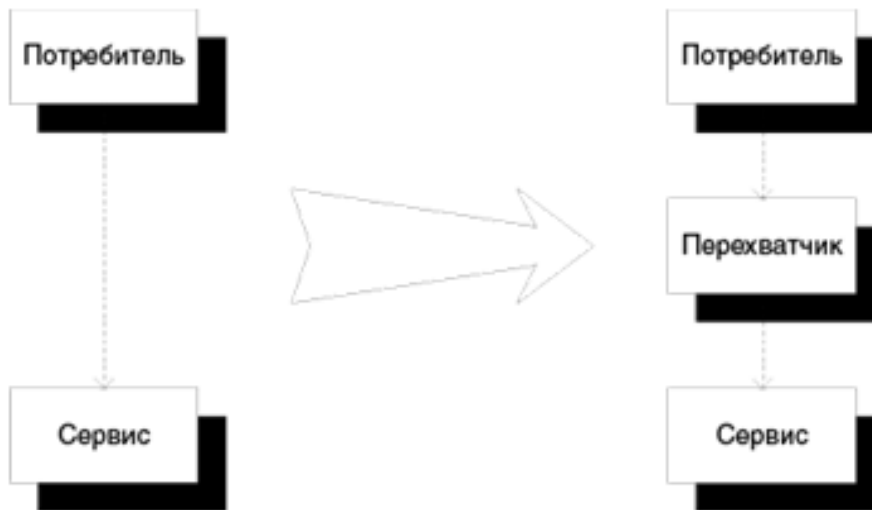
[9.3. Объявление аспектов](#)

[9.4. Резюме](#)

9.1. Знакомство с механизмом перехвата

Концепция механизма перехвата довольно проста: нам хотелось бы уметь перехватывать сигнал между потребителем и сервисом, и исполнять некоторый код до или после вызова данного сервиса. На рисунке 9-2 обычный сигнал от потребителя к сервису перехватывается посредником, который может исполнить свой собственный код до или после передачи сигнала к фактическому сервису.

Рисунок 9-2: Механизм перехвата в двух словах. Мы можем сконвертировать простой сигнал от потребителя к сервису в более сложное взаимодействие путем вставки фрагмента кода посредника. Посредник получает первоначальный сигнал и передает его в действующую реализацию, при этом действуя на сигнал так, чтобы он выполнял то, что ему требуется делать.



В данном разделе мы собираемся познакомиться с механизмом перехвата и узнать, каким образом, по своей сути, он является применением паттерна проектирования Decorator. Если вы не знакомы с паттерном Decorator, то мы разберем его в рамках беседы, и когда мы это сделаем, вы должны будете получить хорошее понимание того, как он работает. Мы начнем с рассмотрения простого примера, который демонстрирует паттерн, и перейдем к обсуждению, как механизм перехвата относится к паттерну Decorator.

Пример: реализация аудита

В данном примере мы будем реализовывать аудит для `ProductRepository`. Аудит – это универсальный пример концепции сквозных сущностей: он может потребоваться, но не должен оказывать влияние на основную функциональность чтения и редактирования товаров. Поскольку принцип единичной ответственности предполагает, что мы не должны позволять `ProductRepository` самому реализовывать аудит, использование паттерна Decorator является наилучшим вариантом.

Реализация `AuditingProductRepository`

Реализовать аудит для `ProductRepository` мы можем путем введения нового класса `AuditingProductRepository`, который обергивает другой `ProductRepository` и реализует аудит. Рисунок 9-3 иллюстрирует то, как типы связаны друг с другом.

Рисунок 9-3: AuditingProductRepository наследуется от абстрактного класса ProductRepository и обортывает экземпляр любой другой реализации ProductRepository. AuditingProductRepository делегирует всю работу расширенному ProductRepository, но добавляет аудит на соответствующие места. Сможете ли вы увидеть планировку?



Помимо расширенного ProductRepository для AuditingProductRepository также нужен сервис, который реализует аудит. В следующем листинге роль такого сервиса играет интерфейс IAuditor.

Листинг 9-1: Объявление AuditingProductRepository

```

1. public partial class AuditingProductRepository :
2.     ProductRepository
3.     {
4.         private readonly ProductRepository
5.             innerRepository;
6.         private readonly IAuditor auditor;
7.         public AuditingProductRepository(
8.             ProductRepository repository,
9.             IAuditor auditor)
10.        {
11.            if (repository == null)
12.            {
13.                throw new ArgumentNullException("repository");
14.            }
15.            if (auditor == null)
16.            {
17.                throw new ArgumentNullException("auditor");
18.            }
19.            this.innerRepository = repository;
20.            this.auditor = auditor;
21.        }
22.    }

```

Строка 2, 4-5, 8, 19: Наследуется и обортывает ProductRepository

Строка 6, 9, 20: Сервис аудита

AuditingProductRepository наследуется от той же самой абстракции, которую он расширяет. AuditingProductRepository использует стандартный Внедрение в конструктор (Constructor Injection) для запроса ProductRepository, который он может обернуть и которому он может делегировать свою основную реализацию. Помимо расширенного репозитория для AuditingProductRepository также требуется IAuditor, который он может использовать для отслеживания операций, реализованных расширенным репозиторием.

Следующий листинг демонстрирует шаблонные реализации двух методов `AuditingProductRepository`.

Листинг 9-2: Реализация `AuditingProductRepository`

```
public override Product SelectProduct(int id)
{
    return this.innerRepository.SelectProduct(id);
}
public override void UpdateProduct(Product product)
{
    this.innerRepository.UpdateProduct(product);
    this.auditor.Record(
        new AuditEvent("ProductUpdated", product));
}
```

Не для всех операций нужен аудит. Универсальным требованием является аудит всех операций `Create`, `Update` и `Delete`, и игнорирование операций `Read`. Поскольку метод `SelectProduct` является истинной операцией `Read`, вы делегируете вызов расширенного репозитория и незамедлительно возвращаете результат.

Метод `UpdateProduct`, с другой стороны, должен подвергаться аудиту. Вы все еще делегируете реализацию расширенному репозиторию, но после того, как делегированный метод возвращается, вы используете внедренный `IAuditor` для отслеживания операции.

`Decorator`, подобно `AuditingProductRepository`, является своего рода *панировкой говяжьей от бивной*: он приукрашивает основной ингредиент, не изменяя его. Сама по себе панировка не является просто пустой оболочкой, а содержит собственный список ингредиентов. Настоящая панировка делается из панировочных сухарей и специй; подобным образом `AuditingProductRepository` содержит `IAuditor`.

Обратите внимание на то, что внедренный `IAuditor` сам по себе является абстракцией, что означает, что вы можете варьировать реализацию независимо от `AuditingProductRepository`. Все, что делает класс `AuditingProductRepository`, — координирует действия расширенного `ProductRepository` и `IAuditor`.

Вы можете создать любую реализацию `IAuditor`, какую только пожелаете, но реализация, основанная на `SQL Server`, — универсальный вариант. Давайте посмотрим, как вы можете подключить все соответствующие зависимости для выполнения этой работы.

Компоновка `AuditingProductRepository`

Несмотря на то, что многие приложения используют класс `ProductRepository` для извлечения информации о товаре, и в связи с тем, что WCF веб-сервис `CommerceService` из раздела 7.3.2 "Пример: подключение сервиса управления продуктами" раскрывает CRUD-операции для `Products`, это подходящее место для начала работы.

В главе 8 вы видели несколько примеров того, как компоновать экземпляры `ProductManagementService`. Листинги 8-4 и 8-5 предоставляли самую корректную реализацию, но в следующем листинге мы проигнорируем тот факт, что `SqlProductRepository` является устранимым, для того, чтобы сконцентрироваться на компоновке `Decorator`'ов.

Листинг 9-3: Компоновка Decorator

```
1. public IProductManagementService ResolveProductManagementService ()
2. {
3.     string connectionString =
4.         ConfigurationManager.ConnectionStrings
5.         ["CommerceObjectContext"].ConnectionString;
6.     ProductRepository sqlRepository =
7.         new SqlProductRepository(connectionString);
8.     IAuditor sqlAuditor =
9.         new SqlAuditor(connectionString);
10.    ProductRepository auditingRepository =
11.        new AuditingProductRepository(
12.            sqlRepository, sqlAuditor);
13.    IContractMapper mapper = new ContractMapper();
14.    return new ProductManagementService(
15.        auditingRepository, mapper);
16. }
```

Строка 6-7: Внутренний ProductRepository

Строка 10-12: Decorator

Строка 14-15: Внедряет Decorator

Как и в листинге 7-9, поскольку вам хочется использовать ProductRepository, базирующийся на SQL Server, вы создаете новый экземпляр SqlProductRepository. Но вместо того, чтобы напрямую внедрять его в экземпляр ProductManagementService, вы будете обертывать его в AuditingProductRepository.

Вы внедряете и SqlProductRepository, и базирующуюся на SQL Server реализацию IAuditor в экземпляр AuditingProductRepository. Обратите внимание на то, как и sqlRepository, и auditingRepository объявлены в виде экземпляров ProductRepository.

Теперь вы можете внедрить auditingRepository в новый экземпляр ProductManagementService и вернуть его. ProductManagementService видит только auditingRepository и ничего не знает о sqlRepository.

Предупреждение

Листинг 9-3 – это упрощенный пример, в котором игнорируются проблемы жизненного цикла. Поскольку и SqlProductRepository, и SqlAuditor являются устранимыми типами, данный код станет причиной утечки ресурсов. Более корректной реализацией была бы интерполяция листинга 9-3 с листингами 8-4 и 8-5 – но я уверен, вы поймете, что в данном случае реализация начнет усложняться.

Подсказка

Вместо того, чтобы вручную бороться с перестановкой местами композиции объектов, управления жизненным циклом и механизма перехвата, используйте DI-контейнер.

Заметьте, что вы могли бы добавить поведение в ProductRepository, не изменяя при этом исходного кода существующих классов. Для того чтобы добавить возможность аудита,

нам не пришлось изменять `SqlProductRepository`. Это желанное свойство, известное как *принцип от крытости/закрытости*.

Обязательная пищевая аналогия

Думаю, это относится к покрытию говяжьей отбивной панировкой. Несмотря на то, что мы изменили отбивную, мы оставили ее в том же размере вместо того, чтобы разрезать ее и потушить.

Теперь, когда вы увидели пример перехватывания конкретного `SqlProductRepository` с помощью расширенного `AuditingProductRepository`, давайте немного вернемся назад и изучим паттерны и принципы, лежащие в основе механизма перехвата.

Паттерны и принципы механизма перехвата

Как и в случае со многими другими DI-паттернами, паттерн Decorator является давним и хорошо описанным паттерном проектирования, который имел место еще за несколько лет до появления механизма внедрения зависимости. Он является настолько фундаментальной частью механизма перехвата, что независимо от того, знаком ты с ним близко или нет, он гарантированно о себе напоминает.

Вы могли заметить, что такие термины, как *принцип единственной ответственности* и *принцип от крытости/закрытости*, используются чаще обычного. Это составляющие части пятиэлементного меню SOLID.

Все эти паттерны и принципы считаются ценным руководством по чистому коду. Основная цель данного раздела – связать это заданное руководство с механизмом внедрения зависимостей для того, чтобы продемонстрировать, что механизм внедрения зависимостей является всего лишь средством достижения цели. Мы используем DI как средство разрешения поддерживаемого кода.

Все потребители зависимостей должны при вызове своих зависимостей соблюдать *принцип подстановки Барбары Лисков*. Это позволяет нам заменить первоначально запланированную реализацию другой реализацией той же самой абстракции. Поскольку Decorator реализует такую же абстракцию, как и класс, который он обертывает, мы можем заменить первоначальный класс на Decorator.

Именно это вы и делали в листинге 9-3, когда заменяли первоначальный `SqlProductRepository` на `AuditingProductRepository`. Вы могли бы сделать это, не изменяя код `ProductManagementService`, так как `ProductManagementService` следует принципу подстановки Барбары Лисков: ему нужен экземпляр `ProductRepository`, и тогда любая реализация будет выполнена.

Возможность расширения поведения класса без изменения его кода называется *принципом от крытости/закрытости*, и является одним из пяти принципов, зашифрованных сущностью под названием *SOLID*.

SOLID

Кому не хотелось бы создавать надежное программное обеспечение? Программное обеспечение, которое могло бы выдержать тест временем и остаться полезным для своих

пользователей, – стоящая цель; введение SOLID в качестве акронима разработки качественного программного обеспечения имеет смысл.

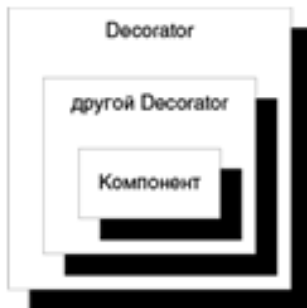
Decorator

Паттерн Decorator впервые был описан в книге "Паттерны проектирования". Цель этого паттерна – "динамически присоединить к объекту дополнительные ответственности. Decorator'ы являются гибкой альтернативой деления на подклассы, которое выполняется для расширения функциональности."

Работа Decorator заключается в обертывании одной реализации абстракции в другую реализацию. Объект, выполняющий обертывание, делегирует операции находящейся в нем реализации, добавляя при этом поведение до или после вызова обернутого объекта.

Decorator может обертывать другой Decorator, который обертывает еще одного Decorator'a, и т.д. Следующий рисунок демонстрирует, как Decorator'ы могут обертывать друг друга. В самом центре должна находиться независимая реализация, которая выполняет необходимую работу.

Decorator обертывает другого Decorator'a, обертывающего, в свою очередь, самостоятельного компонента. Когда вызывается член самого дальнего Decorator'a, он делегирует сигнал обернутому им компоненту. Поскольку обернутый компонент сам по себе является Decorator'ом, он делегирует сигнал содержащемуся в нем компоненту. При каждом вызове Decorator имеет возможность использовать входное или выходное значение содержащегося в нем компонента для того, чтобы выполнить дополнительную работу



Когда в Decorator поступает вызов одного из членов реализуемой им абстракции, он может просто делегировать вызов, ничего при этом не делая:

```
public string Greet(string name)
{
    return this.innerComponent.Greet(name);
}
```

Кроме того, перед тем, как делегировать вызов, он может изменить входные данные:

```
public string Greet(string name)
{
    var reversed = this.Reverse(name);
    return this.innerComponent.Greet(reversed);
}
```

Похожим образом он может изменить возвращаемое значение перед тем, как его вернуть:

```
public string Greet(string name)
{
    var returnValue = this.innerComponent.Greet(name);
    return this.Reverse(returnValue);
}
```

Имея два предыдущих примера, мы можем обернуть последний из них в предшествующий для того, чтобы создать комбинацию, которая изменяет как входные, так и выходные данные.

Decorator также может решить не вызывать приведенную ниже реализацию:

```
public string Greet(string name)
{
    if (name == null)
    {
        return "Hello world!";
    }
    return this.innerComponent.Greet(name);
}
```

В данном примере граничный оператор предоставляет поведение по умолчанию для входных данных типа null, при котором обернутый компонент вообще не вызывается.

То, что отличает Decorator от любого класса, содержащего зависимости, – расширенный объект реализует ту же самую абстракцию, что и Decorator. Это позволяет Composer заменить первоначальный компонент Decorator'ом, не изменяя при этом потребителя. Расширенный объект часто внедряется в Decorator, объявленный в виде абстрактного типа, причем в этом случае Decorator должен соблюдать принцип подстановки Барбары Лисков и относиться ко всем расширенным объектам одинаково.

В некоторых местах данной книги вы уже видели Decorator'ы в действии. В примере раздела 9.1.1 "Пример: реализация аудита" использовался Decorator, как и в разделе 4.4.4.

Под акронимом SOLID мы понимаем пять принципов объектно-ориентированного программирования, которые оказываются полезными при написании поддерживаемого кода. В таблице 9-1 перечислены эти принципы.

Таблица 9-1: Пять принципов SOLID

Принцип	Описание	Как связан с механизмом внедрения зависимостей
Принцип единственной ответственности (SRP)	Класс должен иметь только одну ответственность. Он должен выполнять только одну задачу, но выполнять ее хорошо. Противоположностью данного принципа является антипаттерн под названием God Class, в котором один класс может делать все, включая приготовление кофе.	Придерживаясь данного принципа может быть сложновато, но одним из многочисленных преимуществ Constructor Injection является тот факт, что данный принцип становится очевидным всякий раз, когда мы его не соблюдаем. В примере раздела 9.1.1 "Пример: реализация аудита", вы могли соблюдать принцип единственной ответственности путем разделения

Принцип	Описание	Как связан с механизмом внедрения зависимостей
		ответственностей на отдельные типы <code>SqlProductRepository</code> имеет дело только с хранением и извлечением данных о товарах, тогда как <code>SqlAuditor</code> сконцентрирован на сохранении пути аудита в базе данных. Единственной ответственностью класса <code>AuditingProductRepository</code> является координация действий <code>ProductRepository</code> и <code>IAuditor</code>.
Принцип открытости/закрытости (OCP)	Класс должен быть открыт для расширяемости, но закрыт для модификации. Это означает, что должна присутствовать возможность добавлять к существующему классу поведение, не изменяя при этом код этого класса. Этого не так просто достичь, но принцип единственной ответственности, по крайней мере, упрощает данный процесс, поскольку, чем проще код, тем проще обнаружить потенциальные "Швы".	Существует много способов сделать класс расширяемым, включая виртуальные методы, внедрение стратегий и применение Decorator'ов – но детали не важны, механизм внедрения зависимостей предоставляет данную возможность, позволяя нам компоновать объекты.
Принцип подстановки Барбары Лисков (LSP)	Клиент должен одинаково относиться ко всем реализациям абстракций. Мы должны уметь заменять любую реализацию другой реализацией, не разрушая при этом потребителя.	Принцип подстановки Барбары Лисков является основой механизма внедрения зависимостей. Если потребитель не соблюдает данный принцип, то вы не можете заменять зависимости, и мы теряем какие-то (если не все) преимущества механизма внедрения зависимостей.
Принцип разделения интерфейсов (ISP)	Интерфейсы должны проектироваться дифференцированно. У нас нет желания смешивать слишком большое количество ответственностей в одном интерфейсе, поскольку становится слишком тяжело	Поначалу, кажется, что принцип разделения интерфейсов лишь отдаленно связан с механизмом внедрения зависимостей. Но он важен, поскольку интерфейс, который моделирует все, включая кухонную раковину, подталкивает вас в направлении конкретной реализации. Часто это пахнет

Принцип	Описание	Как связан с механизмом внедрения зависимостей
	его реализовывать. Я считаю, что принцип разделения интерфейсов является концептуальной основой принципа единственной ответственности. Согласно принципу ISP интерфейсы должны моделировать только одну сущность, тогда как принцип SRP утверждает, что реализации должны иметь только одну ответственность.[/p]	<i>leaky</i> -абстракцией и значительно усложняет процесс замены зависимостей, поскольку некоторые члены интерфейса могут не иметь смысла в контексте, отличающемся от того, который первоначально запускал проектирование.
Принцип инверсии зависимостей (DIP)	Еще один термин крылатой фразы <i>программирование на основании интерфейсов, а не на основании конкретной реализации</i> .	Принцип инверсии зависимостей (DIP) – это принцип, являющийся <i>основанием</i> механизма внедрения зависимостей.

Примечание

Ни один из принципов, инкапсулированных в акрониме SOLID, не представлен абсолютным образом. Они являются директивами, которые могут помочь нам в написании чистого кода. Для меня они являются целями, которые помогают мне решить, по какому направлению я должен направить свои API. Я всегда радуюсь своему успеху, но бывает и не радуюсь.

Decorator (и все паттерны проектирования в целом) и такие директивы, как SOLID принципы, существовали на протяжении многих лет, и в целом их применение считалось благотворным. В данном разделе я попытался дать вам подсказку, как они связаны с механизмом внедрения зависимостей.

SOLID принципы релевантны на протяжении всех глав данной книги, и вы могли заметить, что я то тут, то там упоминал о некоторых из них. Но только когда мы приступаем к разговору о механизме внедрения зависимостей, и о том, как они связаны с Decorator'ами, начинает проступать связь с SOLID. Некоторые из них более неуловимы, но добавление поведения (например, аудита) с помощью Decorator является чистейшим применением принципа открытости/закрытости с не отстающим от него принципом единственной ответственности, так как принцип открытости/закрытости позволяет нам создавать реализации, имеющие специально заданные области применения.

В данном разделе мы попутешествовали по паттернам и принципам для того, чтобы понять взаимосвязь механизма внедрения зависимостей с другими установленными директивами. Вооружившись этим дополнительным знанием, давайте вернемся к цели данной главы, которая заключается в написании чистого и поддерживаемого кода вопреки противоречивым и изменяющимся требованиям, и необходимости обращаться к концепции сквозных сущностей.

9.2. Реализация сквозных сущностей

Большинство приложений должно обращаться к аспектам, которые не связаны напрямую ни с какой конкретной возможностью, а вместо этого обращаются к более широкой сущности. Эти сущности часто касаются множества других несвязанных областей кода, находящихся даже в различных модулях и на различных уровнях. Поскольку они охватывают большую область базы кода, мы называем их *сквозными сущностями*. В следующей таблице приведены некоторые примеры. Данная таблица не является всеобъемлющим списком всех доступных аспектов; это всего лишь наглядный пример.

Таблица 9-2: Универсальные примеры сквозных сущностей

Аспект	Описание
Аудит	Любая операция, подразумевающая изменение данных, должна оставлять путь аудита, включая временную отметку, идентификатор пользователя, внесшего изменения, и информацию о том, что изменилось. Пример этого вы видели в разделе 9.1.1 "Пример: реализация аудита".
Вход в систему	Слегка отличающийся от аудита вход в систему фокусируется на записи событий, которые отражают состояние приложения. Это могут быть события, интересные IT-отделу, или, возможно, бизнес-события.
Контроль производительности	Слегка отличается от аспекта входа в систему, поскольку имеет дело больше с записью производительности, а не с конкретными событиями. Если у вас есть <i>соглашения о качестве предоставляемых услуг</i> (Service Level Agreements), которые нельзя контролировать посредством стандартной инфраструктуры, вы должны реализовать пользовательский контроль производительности. Пользовательские счетчики производительности Windows хорошо подходят для этого, но вам еще нужно добавить некоторый код, который захватывает данные.
Безопасность	Некоторые операции должны быть разрешено выполнять только конкретным пользователям, и вы должны претворить это в жизнь.
Кэширование	Довольно часто вы можете увеличить производительность за счет реализации кэшов, но другой причины, согласно которой конкретный компонент доступа к данным должен иметь дело с этим аспектом, не существует. Возможно, вы захотите иметь возможность разрешать и блокировать кэширование различных реализаций доступа к данным. Мы уже наблюдали намек на реализацию кэширования с помощью Decorator'ов в разделе 4.4.4.
Обработка ошибок	Возможно, мы захотим обрабатывать определенные исключения и либо заносить их в лог, либо демонстрировать пользователю сообщение. Для того чтобы обрабатывать ошибки должным образом, мы можем использовать обработчик ошибок Decorator.
Отказоустойчивость	Внешние ресурсы гарантированно время от времени будут недоступны. Вы можете реализовать паттерны отказоустойчивости, например, Circuit Breaker, с помощью Decorator.

При рисовании диаграмм архитектуры многоуровневого приложения сквозные сущности чаще всего представляются в виде вертикальных блоков, размещенных около уровней, как это показано на рисунке 9-4.

Рисунок 9-4: Чаще всего мы представляем сквозные сущности на диаграммах архитектуры приложения с помощью вертикальных блоков, которые охватывают все уровни. В данном примере безопасность является сквозной сущностью.



В этом разделе мы рассмотрим некоторые примеры, которые иллюстрируют, как мы можем использовать механизм перехвата в форме Decorator'ов для реализации сквозных сущностей. Мы выберем несколько аспектов из таблицы 9-2, чтобы попробовать реализовать их с помощью SOLID принципов, но рассмотрим только небольшое их количество. Как и в случае со многими другими сущностями, механизм перехвата проще понять с помощью абстракции, но вся сложность заключается именно в деталях. Он принимает на себя воздействие, чтобы надлежащим образом впитать приемы, и лучше я приведу вам слишком много примеров, нежели совсем мало. Когда мы закончим с этими примерами, вы должны будете приобрести ясную картину того, что такое механизм перехвата, и как вы можете его применить.

Поскольку вы уже видели ознакомительный пример раздела 9.1.1 "Пример: реализация аудита", мы рассмотрим более сложный пример, чтобы проиллюстрировать, как можно использовать механизм перехвата в рамках сколь угодно сложной логики. Как только мы это сделаем, мы изучим пример, который приведет нас прямо к более декларативному подходу.

Осуществление перехвата с помощью Circuit Breaker

Любое приложение, взаимодействующее с внешними ресурсами, может сталкиваться с ситуациями, когда ресурс недоступен. Разрываются сетевые соединения, переходят в автономный режим базы данных, веб-сервисы засоряются *DDos-атаками* (Distributed Denial of Service). В таких случаях приложение, передающее сигнал, должно уметь восстанавливаться и незамедлительно решать проблему.

Большинство .NET API имеют время ожидания по умолчанию, гарантирующее, что внешний вызов не заблокирует навсегда потребляющий поток. Однако в ситуации, когда вы только что получили исключение, касающееся времени ожидания, как вам относиться к следующему вызову ресурса, который является виновником возникшего исключения? Попытаться ли вам вызвать ресурс снова? Поскольку время ожидания чаще всего указывает на то, что другая сторона либо находится в автономном режиме, либо засорена запросами, осуществление нового блокирующего вызова может стать не очень хорошей

идеей. Будет лучше допустить самое худшее и незамедлительно выдать исключение. Это и есть логическое обоснование паттерна Circuit Breaker.

Circuit Breaker – *стабильный паттерн*, так как он добавляет устойчивости приложению, быстро прерываясь, вместо того, чтобы зависать и потреблять ресурсы во время зависания. Это отличный пример нефункционального требования и истинной сквозной сущности, поскольку этому паттерну приходится совсем немного работать с той возможностью, которая реализуется с помощью внешнего вызова.

Сам по себе паттерн Circuit Breaker слегка сложный и его может быть сложно реализовывать, но нам нужно сделать эти вложения всего лишь раз. Мы даже могли бы при желании реализовать его в повторно используемой библиотеке. Имея повторно используемый Circuit Breaker, мы можем с легкостью применить его к разнородным компонентам, используя паттерн Decorator.

Circuit Breaker

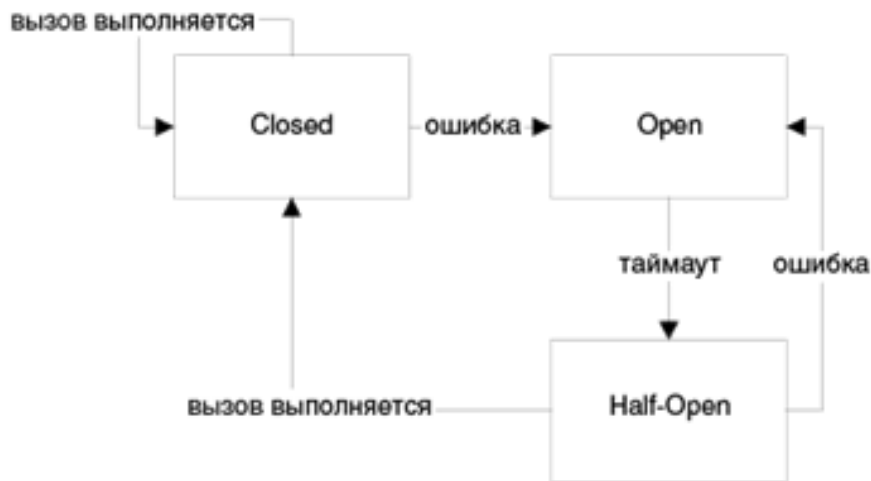
Паттерн проектирования Circuit Breaker получил свое название от электрического выключателя с таким же именем. Он создан для размыкания соединения при появлении неисправности с целью предотвращения распространения неисправности.

В прикладном программном обеспечении при возникновении таймаутов или похожих ошибок взаимодействия, ситуация может ухудшиться, если вы будете продолжать долбить упавшую систему. Если удаленная система засорена, множественные повторные вызовы могут привести к падению системы – при этом пауза может дать системе шанс восстановиться. На уровне осуществления вызова потоки, заблокировавшие ожидание таймаутов, могут сделать приложение-потребитель не реагирующим на вызов. Лучше обнаружить разрыв взаимодействия и быстро прерваться на время.

Паттерн Circuit Breaker решает этот вопрос путем выключения выключателя при возникновении ошибки. Обычно в этот паттерн входит время ожидания, что заставляет его повторить соединение через некоторое время, таким образом, он может автоматически восстанавливаться, когда удаленная система становится резервной.

Следующий рисунок иллюстрирует упрощенное представление переключения состояний в Circuit Breaker.

Упрощенная диаграмма переключения состояний паттерна Circuit Breaker. Паттерн Circuit Breaker начинается с состояния Closed, указывающего на то, что цепь закрыта, и сообщения могут поступать. При возникновении ошибки отключается прерыватель, и состояние переключается на Open. В данном состоянии прерыватель не позволяет ни одного вызова удаленной системы; вместо этого он незамедлительно выдает исключение. После таймаута состояние переключается на Half-Open (Полуоткрытое), при котором разрешается прохождение только одного удаленного вызова. При успешном выполнении состояние возвращается к Closed, но если вызов не проходит, прерыватель возвращается к состоянию Open, начиная новый таймаут.



Вы можете захотеть сделать Circuit Breaker более сложным, нежели это описано здесь. Для начала вы, возможно, не захотите выключать прерыватель всякий раз при возникновении случайной ошибки, а захотите использовать барьер. Во-вторых, вы должны выключать прерыватель только при возникновении ошибок определенных типов. Таймауты и исключения, касающиеся процесса взаимодействия, хорошо, но `NullReferenceException`, скорее всего, указывает на баг, а не на скачкообразную ошибку.

Давайте рассмотрим пример, который демонстрирует, как паттерн Decorator можно использовать для добавления поведения Circuit Breaker к существующему внешнему компоненту. В этом примере мы будем концентрироваться на применении повторно используемого Circuit Breaker, а не на том, как он реализован.

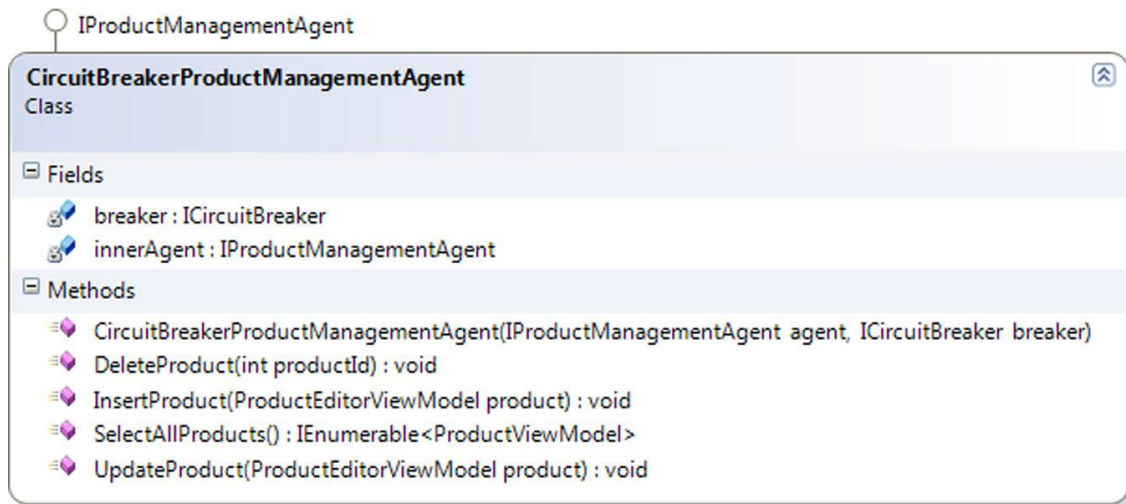
Пример: Реализация Circuit Breaker

В разделе 7.4.2 "Пример: присоединение ценного клиента управления товарами" мы создали WPF приложение, которое взаимодействует с WCF сервисом при помощи интерфейса `IProductManagementAgent`. Несмотря на то, что мы возвращались к этому приложению в разделе 8.2.1 "Использование устранимых зависимостей", мы никогда не изучали его подробно.

В предыдущих примерах вы использовали `WcfProductManagementAgent`, который реализует интерфейс посредством вызова операций WCF сервиса. Поскольку данная реализация не обладает возможностью явной обработки ошибок, любая ошибка взаимодействия будет передаваться вызывающему объекту.

Это отличная ситуация для Circuit Breaker. Вам хотелось бы незамедлительно прерываться, как только начинают возникать исключения; таким образом, вы не будете блокировать вызывающий поток и засорять сервис. Как демонстрирует рисунок 9-5, вы начинаете объявления Decorator'a для `IProductManagementAgent` и запроса необходимых зависимостей посредством Constructor Injection.

Рисунок 9-5: `CircuitBreakerProductManagementAgent` — это `Decorator` для `IProductManagementAgent`: обратите внимание на то, как он реализует интерфейс, а также на то, что он содержит экземпляр, внедренный через конструктор. Еще одна зависимость — `ICircuitBreaker`, которую мы можем использовать для реализации паттерна `Circuit Breaker`.



Теперь вы можете обернуть любой вызов расширенного `IProductManagementAgent` подобно примеру, продемонстрированному в следующем листинге.

Листинг 9-4: Расширение с помощью `Circuit Breaker`

```

public void InsertProduct(ProductEditorViewModel product)
{
    this.breaker.Guard();
    try
    {
        this.innerAgent.InsertProduct(product);
        this.breaker.Succeed();
    }
    catch (Exception e)
    {
        this.breaker.Trip(e);
        throw;
    }
}

```

Первое, что вам нужно сделать перед тем, как вы попытаетесь вызвать расширенного агента — проверить состояние `Circuit Breaker`. Метод `Guard` позволит вам пройти, если `Circuit Breaker` находится в состоянии `Closed` или `Half-Open`, тогда как этот же метод выдаст исключение, если `Circuit Breaker` находится в состоянии `Open`. Это гарантирует, что вы незамедлительно прерветесь, если у вас есть причины полагать, что вызов не приведет к успешному завершению.

Если вы сможете пройти мимо метода `Guard`, то можете попытаться вызвать расширенного агента. Обратите внимание на то, что вызов обернут в блок `try`: если вызов не выполнится, вы отключаете прерыватель. В данном примере все просто, но в настоящей реализации вам следует схватить и отключить прерыватель от выбора типа исключений. Поскольку `NullReferenceExceptions` или похожие типы исключений редко указывают на скачкообразные ошибки, нет причины отключать прерыватель в таких случаях.

Как при состоянии `Closed`, так и при состоянии `Half-Open`, отключение прерывателя вернет нас в состояние `Open`. При состоянии `Open` время ожидания определяет, когда мы вернемся к состоянию `Half-Open`.

Наоборот, вы вызываете `Circuit Breaker` в случае успешного вьзова. Если вы уже находитесь в состоянии `Closed`, вы в нем и остаетесь. Если вы находитесь в состоянии `Half-Open`, вы переходите обратно к состоянию `Closed`. При нахождении `Circuit Breaker` в состоянии `Open` невозможно сигнализировать об успешном выполнении, поскольку метод `Guard` будет гарантировать, что вы никогда не зайдете так далеко.

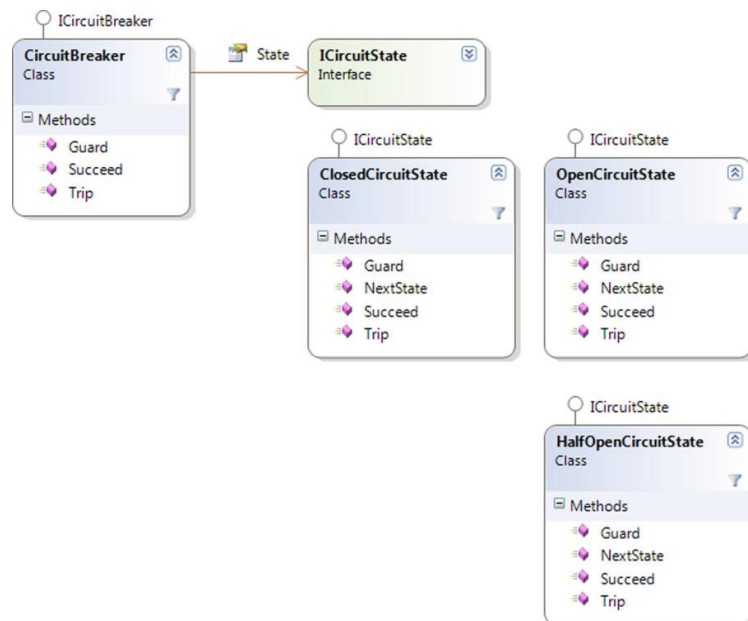
Все остальные методы `IProductManagementAgent` выглядят также с одной лишь разницей, заключающейся в методе, который они вызывают для `innerAgent`, и дополнительной строке кода для метода, который возвращает значение. Вы можете увидеть данную вариацию внутри блока `try` метода `SelectAllProducts`:

```
var products = this.innerAgent.SelectAllProducts();
this.breaker.Succeed();
return products;
```

Поскольку вы должны сигнализировать `Circuit Breaker` об успехе, вам приходится сохранять возвращаемое значение расширенного агента перед тем, как вернуть самого агента; но это единственное отличие между методами, которые возвращают значение, и методами, не возвращающими значение.

На данном этапе вы оставили реализацию `ICircuitBreaker` открытой, но реальная реализация является полностью повторно используемым комплексом классов, применяющих паттерн проектирования "Состояние" (State). Рисунок 9-6 демонстрирует включенные в этот комплекс классы.

Рисунок 9-6: Класс `CircuitBreaker` реализует интерфейс `ICircuitBreaker` путем применения паттерна State. Все три метода реализованы путем делегирования полномочий члену State, подразумевающему возможность реконфигурации, который изменяется подобно переходам состояний от одного к другому.



Несмотря на то, что в этой книге мы не собираемся погружаться вглубь реализации `CircuitBreaker`, важно, что вы можете выполнить перехват с помощью условно сложного кода.

Подсказка

Если вы интересуетесь реализацией класса `CircuitBreaker`, то она доступна в коде, присоединенном к данной книге.

Чтобы скомпоновать `ProductManagementAgent` с помощью добавленной функциональности, вы можете обернуть ее в другой реализации:

```
var timeout = TimeSpan.FromMinutes(1);
ICircuitBreaker breaker = new CircuitBreaker(timeout);
IProductManagementAgent circuitBreakerAgent =
    new CircuitBreakerProductManagementAgent(wcfAgent, breaker);
```

В листинге 7-10 вы компоновали WPF приложение из нескольких зависимостей, включая экземпляр `WcfProductManagementAgent`. Вы можете дополнить эту переменную `wcfAgent` путем внедрения ее в экземпляр `CircuitBreakerProductManagementAgent`, который реализует тот же интерфейс. В данном конкретном примере вы создаете новый экземпляр класса `CircuitBreaker` всякий раз при разрешении зависимостей, а это соответствует стилю существования `Transient`.

В WPF приложении, в котором вы разрешаете зависимости всего лишь раз, использование `Transient Circuit Breaker` не является проблемой, но в целом это не есть оптимальный стиль существования для такой функциональности. На другой стороне будет только один веб-сервис. Если данный сервис станет недоступным, `Circuit Breaker` должен прервать все попытки подключения к этому сервису. Если используется несколько экземпляров `CircuitBreakerProductManagementAgent`, то это должно выполняться для каждого из них.

Более компактный `ICircuitBreaker`

Как представлено в данном разделе, интерфейс `ICircuitBreaker` содержит три члена: `Guard`, `Succeed` и `Trip`. В альтернативном определении интерфейса мог использоваться стиль передачи продолжений для того, чтобы сократить объем до нескольких единожды используемых методов:

```
public interface ICircuitBreaker
{
    void Execute(Action action);
    T Execute<T>(Func<T> action);
}
```

Это позволило бы нам более сжато применять `ICircuitBreaker` в каждом методе, подобно тому, как это показано ниже:

```
public void InsertProduct(ProductEditorViewModel product)
{
    this.breaker.Execute(() =>
        this.innerAgent.InsertProduct(product));
}
```

Я решил использовать более явную и старомодную версию `ICircuitBreaker`, поскольку мне хочется, чтобы вы могли сконцентрироваться на текущей теме – механизм перехвата. Несмотря на то, что лично мне нравится стиль передачи продолжений, я считаю лямбда и дженерики по-своему продвинутыми тематиками, и думаю, что в данном контексте они могли бы скорее отвлечь наше внимание, нежели стать полезными.

Предпочтем ли мы, в конечном итоге, одно определение интерфейса другому, не меняет результат текущей главы

Это очевидный случай для задания стиля существования `Singleton` для `CircuitBreaker`, но также это означает, что `CircuitBreaker` должен быть потоко-безопасным. Согласно своей сущности `CircuitBreaker` сохраняет состояние; потоко-безопасность должна быть реализована явным образом. Это делает реализацию даже более сложной.

Несмотря на сложность `CircuitBreaker`, вы с легкостью можете перехватить экземпляр `IProductManagementAgent` с помощью `Circuit Breaker`. Хотя первый пример механизма перехвата в разделе 9.1.1 "Пример: реализация аудита" был довольно простым, пример `Circuit Breaker` демонстрирует, что вы можете перехватить класс с помощью сквозных сущностей, чья реализация просто более сложна, чем первоначальная реализация.

Паттерн `Circuit Breaker` гарантирует, что приложение незамедлительно прервет свое выполнение вместо того, чтобы связывать драгоценные ресурсы; но в идеале приложение не будет полностью разрушено. Для того чтобы справиться с данной проблемой, вы можете реализовать некоторые виды обработки ошибок с помощью механизма перехвата.

Обработка исключений

Вероятно, зависимости время от времени выдают исключения. Даже написанный наилучшим образом код будет (и должен) выдавать исключения, если сталкивается с ситуациями, с которыми не может справиться. Клиенты, использующие внешние ресурсы, попадают в данную категорию. Класс, подобный классу `WcfProductManagementAgent` из шаблонного WPF-приложения, является одним из примеров таких зависимостей. Если веб-сервис недоступен, агент начинает выдавать исключения.

`Circuit Breaker` не изменяет этому фундаментальному свойству. Несмотря на то, что он перехватывает WCF клиента, он все равно выдает исключения.

Вместо сбоя приложения, вы могли бы выбрать возможность предоставления окна сообщения, которое сообщало бы вам о том, что операция не завершилась, и что вам необходимо повторить попытку позднее.

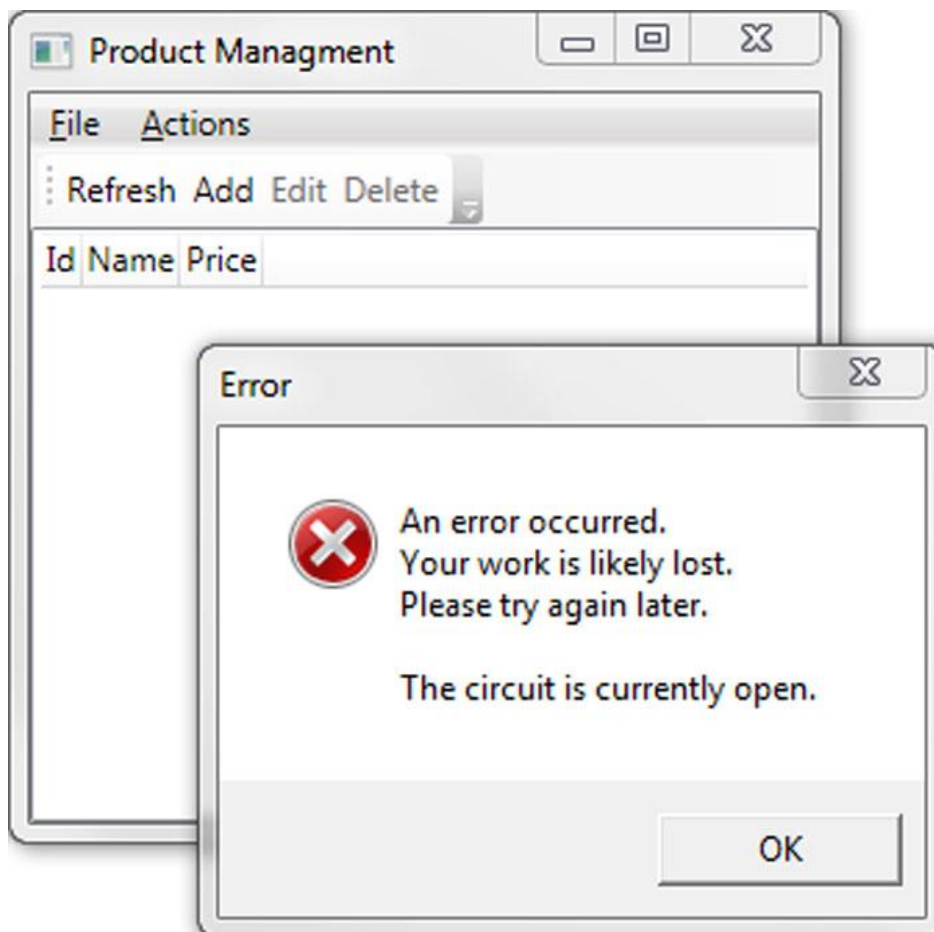
Вы можете использовать механизм перехвата для того, чтобы добавить обработку ошибок в стиле SOLID. Вы не хотите нагружать зависимость обработкой ошибок. Поскольку зависимость должна рассматриваться как повторно используемый компонент, который можно использовать во множестве различных сценариев, было бы невозможно добавить в саму зависимость стратегию обработки исключений, которая бы подходила для всех сценариев. Если бы вы так сделали, то это противоречило бы принципу единственной ответственности.

Используя механизм перехвата для обработки исключений, вы соблюдаете принцип открытости/закрытости. Это позволяет вам реализовать самую лучшую стратегию обработки ошибок для любой конкретной ситуации. Давайте рассмотрим пример.

Пример: обработка исключений

В предыдущем примере вы обертывали `WcfProductManagementAgent` в `Circuit Breaker` для использования в клиентском приложении `Product Management`, впервые введенном в разделе 7.4.2 "Пример: присоединение ценного клиента управления товарами". `Circuit Breaker` справляется с ошибками, определяя, что клиент незамедлительно прервет свое выполнение, но он все равно будет выдавать исключения. Оставшись необработанными эти ошибки приведут к сбою приложения, поэтому вам следует реализовать `Decorator`, который знает, как обрабатывать некоторые из этих ошибок. При выдаче исключения должно всплывать сообщение, как это показано на рисунке 9-7.

Рисунок 9-7: Приложение `Product Management` обрабатывает исключения, касающиеся взаимодействия, путем предоставления пользователю сообщения. Обратите внимание, что в данном примере сообщение об ошибке порождается `Circuit Breaker`, а не вышеупомянутым нарушением взаимодействия.



Реализовать такое поведение легко. Таким же образом, как вы делали это в разделе 9.2.1 "Осуществление перехвата с помощью `Circuit Breaker`", вы добавляете новый класс `ErrorHandlingProductManagementAgent`, который дополняет интерфейс `IProductManagementAgent`. Следующий листинг демонстрирует шаблон одного из методов данного интерфейса, но все эти методы похожи друг на друга.

Листинг 9-5: Обработка исключений

```
1. public void InsertProduct(ProductEditorViewModel product)
2. {
3.     try
4.     {
5.         this.innerAgent.InsertProduct(product);
6.     }
7.     catch (CommunicationException e)
8.     {
9.         this.AlertUser(e.Message);
10.    }
11.    catch (InvalidOperationException e)
12.    {
13.        this.AlertUser(e.Message);
14.    }
15. }
```

Строка 5: Делегирует полномочия расширенному агенту

Строка 9, 13: Предупреждает пользователя об опасности

Метод `InsertProduct` — это представитель целостной реализации класса `ErrorHandlingProductManagementAgent`. При возникновении исключения вы пытаетесь вызвать расширенного агента и предупредить пользователя об опасности с помощью сообщения об исключении. Обратите внимание, что вы обрабатываете только определенный набор известных исключений, потому что пресекать все исключения было бы опасно.

Предупреждение пользователя об опасности включает в себя форматирование строки и демонстрацию этой строки пользователю посредством метода `MessageBox.Show`.

И снова вы добавили функциональность к первоначальной реализации (`WcfProductManagementAgent`) путем реализации Decorator'a. Вы строго соблюдаете как принцип единственной ответственности, так и принцип открытости/закрытости, последовательно добавляя новые типы вместо модификации существующего кода. К настоящему моменту вы должны были уже начать видеть паттерн, который подразумевает более общую систематизацию, нежели Decorator.

Для данной сквозной сущности реализация, основанная на Decorator, склонна к повторяемости. Реализация Circuit Breaker включает в себя применение одного и того же шаблона кода ко всем методам интерфейса `IProductManagementAgent`. Если бы вы хотели добавить Circuit Breaker в другую абстракцию, то вам пришлось бы применить этот же самый код еще и к другим методам. Несмотря на то, что шаблоны отличаются, то же самое справедливо и для кода обработки исключений, который мы только что рассматривали.

Для того чтобы закрепить этот вопрос, давайте вкратце рассмотрим реализацию аспекта безопасности. Данный аспект будет предполагать более общий подход к компоновке сквозных сущностей, который мы в дальнейшем рассмотрим в разделе 9.3.

Добавление функциональности обеспечения безопасности

Безопасность – это еще одна универсальная сквозная сущность. Мы хотели бы, насколько это возможно, обезопасить наши приложения от несанкционированного доступа к важной функциональности.

Примечание

Безопасность – это обширная тема для обсуждения, которая охватывает множество областей, включая раскрытие важной информации и взлом сетей. В данном разделе я лишь кратко коснусь такой темы, как авторизация – т.е. возможность убедиться в том, что только авторизованные пользователи (или системы) могут выполнять определенные действия.

Аналогично тому, как мы использовали `Circuit Breaker`, нам хотелось бы перехватить вызов метода и проверить, должен ли этот вызов быть разрешен. Если не должен, то вместо разрешения вызова будет выдано исключение. Принцип тот же: разница заключается в критерии, который мы используем для определения обоснованности вызова.

Универсальный подход к реализации логики авторизации – применить защиту на основании ролей с помощью `Thread.CurrentPrincipal`. Вы могли бы начать с `Decorator`'а `SecureProductRepository`. Поскольку, как вы уже видели в предыдущих разделах, все методы похожи друг на друга, следующий листинг демонстрирует всего лишь реализацию шаблонного метода.

Листинг 9-6: Явная проверка авторизации

```
public override void InsertProduct(Product product)
{
    if (!Thread.CurrentPrincipal.IsInRole("ProductManager"))
    {
        throw new SecurityException();
    }
    this.innerRepository.InsertProduct(product);
}
```

Метод `InsertProduct` начинается с граничного оператора, который явным образом вызывает `Thread.CurrentPrincipal` и запрашивает, обладает ли он ролью `ProductManager`. Если он не обладает данной ролью, то он незамедлительно выдает исключение. Только если вызываемый `IPrincipal` имеет требуемую роль, вы позволяете ему обойти граничный оператор и вызвать расширенный репозиторий.

Примечание

Запомните, что `Thread.CurrentPrincipal` – это пример паттерна `Ambient Context`.

То, что `Thread.CurrentPrincipal` инкапсулируется в классе `System.Security.Permissions.PrincipalPermission`, является всеобщей идиомой кодирования; поэтому вы могли бы написать предыдущий пример более сжато:

```
public override void InsertProduct(Product product)
{
    new PrincipalPermission(null, "ProductManager").Demand();
    this.innerRepository.InsertProduct(product);
}
```

Класс `PrincipalPermission` инкапсулирует запрос о том, имеет ли текущий `IPrincipal` определенную роль. Вызов метода `Demand` приведет к выдаче исключения, если `Thread.CurrentPrincipal` не имеет ролей `ProductManager`. Данный пример функционально эквивалентен листингу 9-6.

Когда единственное, что вы требуете, – чтобы текущий `IPrincipal` имел определенную роль, вы можете перейти к чисто декларативному стилю.

```
[PrincipalPermission(SecurityAction.Demand, Role = "ProductManager")]
public override void InsertProduct(Product product)
{
    this.innerRepository.InsertProduct(product);
}
```

Атрибут `PrincipalPermission` предлагает ту же самую функциональность, что и класс `PrincipalPermission`, но раскрывается в виде атрибута. Поскольку `.NET Framework` понимает этот атрибут, где бы он его ни встречал, он выполняет соответствующее требование `PrincipalPermission`.

На данном этапе наличие отдельного `Decorator` только с целью применения атрибута выглядит слегка уничтожающим. Почему бы не применить атрибут напрямую к самому первоначальному классу? Несмотря на то, что это кажется довольно привлекательным, существует несколько причин, почему вы можете не захотеть поступить именно так

- Использование атрибутов исключает более сложную логику. Что если бы вы захотели разрешить большинству пользователей обновлять описание товаров, но обновлять цену – только `ProductManager`'ам? Такая логика может быть выражена в императивном коде, но с помощью атрибутов сделать это легко не получится.
- Что если бы вы захотели убедиться, что правила разрешения доступа используются независимо от того, какую реализацию `ProductRepository` вы используете? Поскольку атрибуты конкретных классов не могут повторно использоваться в рамках реализаций, это привело бы к нарушению принципа "не повторяйся".
- Вы не смогли бы варьировать логику обеспечения безопасности независимо от `ProductRepository`.

Однако идея обращения к сквозным сущностям в декларативной манере не нова. Поскольку она часто применяется в аспектно-ориентированном программировании, она подходит нам только в том плане, что мы более близко рассматриваем саму эту идею и то, как она приводит нас к слабо связанному механизму перехвата.

9.3. Объявление аспектов

В предыдущих разделах мы рассматривали паттерны механизма перехвата и то, как они могут помочь вам обратиться к сквозным сущностям с помощью SOLID принципов. В разделе 9.2.3 "Добавление функциональности обеспечения безопасности" вы увидели, как можно было сократить объем реализации такого аспекта, как проверка безопасности, до чисто декларативного подхода.

Применение атрибутов для объявления аспектов является универсальным приемом аспектно-ориентированного программирования (AOP). Но сколь заманчивым бы он ни казался с первого взгляда, использование атрибутов сопровождается некоторого рода встроенными проблемами, которые делают применение атрибутов менее чем идеальным решением. Первую часть данного раздела я буду использовать для того, чтобы дать обзор этой сущности и ее малоизвестных недостатков.

Примечание

Я периодически использую термин *атрибут аспекта*, обозначающий пользовательский атрибут, реализующий или выражающий аспект.

Поскольку мы, в сущности, отбросили в сторону идею об использовании атрибутов для объявления аспектов, остальную часть данной главы мы проведем за рассмотрением динамического перехвата с помощью DI-контейнера, который предоставляет лучшую альтернативу.

Использование атрибутов для объявления аспектов

Атрибуты имеют общую с Decorator'ами черту: несмотря на то, что они могут добавлять или подразумевать изменение поведения члена, сигнатуру они оставляют неизменной. Как вы видели в разделе 9.2.3 "Добавление функциональности обеспечения безопасности", вы можете заменить явный, императивный код авторизации на атрибут. Вместо того чтобы написать одну или более одной строки явного кода, вы могли бы достичь того же результата, применяя атрибут `[PrincipalPermission]`.

Довольно заманчиво было бы экстраполировать эту концепцию в другие сквозные сущности. Было бы это здорово, если бы вы могли отметить метод или класс атрибутом `[HandleError]` или даже пользовательским атрибутом `[CircuitBreaker]`, и, таким образом, применить аспект при помощи единственной строки декларативного кода?

Возможно, это было бы здорово, но существует несколько проблем, связанных с данным подходом, которые вы должны понять.

В первую очередь это проблема, которая вырастает из того факта, что атрибуты, по своему существу, являются пассивными. Несмотря на то, что определение пользовательского атрибута и применение его столь же просто, как наследование класса от `System.Attribute` и наделение других классов пользовательским атрибутом, он находится там, ничего при этом не делая.

Но подождите! Разве атрибут `[PrincipalPermission]` не изменил поведение метода? Да, это так, но этот атрибут (и некоторые другие атрибуты, доступные в стандартной

библиотеке классов) – особенный. .NET Framework понимает этот атрибут и влияет на него, но .NET Framework не будет так делать для всякого пользовательского атрибута, который вам захотелось бы ввести.

Если вы хотите разрешить пользовательским атрибутам изменять поведение приложения, то у вас есть два варианта:

- Изменить шаг компиляции
- Ввести пользовательский хост рабочей среды

Давайте вкратце исследуем каждый вариант.

Модификация компиляции

Работа одного из самых популярных фреймворков аспектно-ориентированного программирования, PostSharp, заключается именно в разрешении вам добавлять пользовательские атрибуты в код. Эти атрибуты должны наследоваться от специального атрибута, который определен в PostSharp SDK, предоставляющем виртуальные методы, которые вы можете переопределить для того, чтобы задать желаемое вами поведение аспекта. Затем вы можете применить эти атрибуты к своим классам или членам класса. Рисунок 9-8 демонстрирует, что случится потом.

Рисунок 9-8: Работа PostSharp заключается в добавлении шага посткомпиляции после того, как будет завершена обычная компиляция. Поскольку пользовательские PostSharp атрибуты в вашем коде трактуются обычным компилятором (например, *csc.exe*) так же, как и любые другие атрибуты, выходные результат – это обычная сборка с пассивными атрибутами. В PostSharp входит шаг посткомпиляции, на котором PostSharp собирает скомпилированную сборку и чередует код ваших пользовательских атрибутов напрямую с атрибутивным кодом. Результатом является новая .NET сборка с вложенными аспектами.



PostSharp полагается на посткомпиляцию для того, чтобы преобразовать пассивные атрибуты в активный код. PostSharp процессор присматривает за атрибутами, унаследованными от PostSharp атрибутов, и чередует код этих атрибутов с кодом, расширенным этими атрибутами. Результат – новая сборка, в которой соответствующий код аспекта чередуется с первоначальным кодом.

Эта сборка – абсолютно обычная сборка, которая запускается всюду, где запускается весь остальной .NET код. Для этого не требуется никакой специальной рабочей среды.

Среди преимуществ данного подхода можно отметить тот факт, что для него не требуется никак особых усилий с вашей стороны. Механизм внедрения зависимостей не необходим, хотя и не исключается. Я уверен, что есть и другие преимущества.

Один из недостатков данного подхода – тот факт, что запускаемый код отличается от написанного вами кода. Если вы захотите отладить код, то вам нужно будет выполнить особые шаги, и, несмотря на то, что поставщик с радостью предоставляет средства,

позволяющие вам выполнить только отладку, он к тому же направляет вас к антипаттерну Vendor Lock-In.

Но самый большой недостаток заключается в использовании самих атрибутов. Этот недостаток идет в одном ряду с использованием пользовательского хоста для активации атрибутов. Давайте сделаем обзор данного варианта перед тем, как рассмотреть недостатки атрибутов.

Использование пользовательского хоста

Еще один вариант активации атрибутов заключается в требовании, чтобы весь код активировался или инициализировался при помощи пользовательского хоста или фабрики. Такая фабрика смогла бы проверять все атрибуты классов, которые она инициализирует, и действовать соответствующим образом.

Мы знаем данный прием из многочисленных .NET технологий, которые полагаются на атрибуты. Примерами этого приема являются:

- В WCF входит множество атрибутов, например, `[ServiceContract]`, `[OperationContract]` и т.д. Эти атрибуты приобретают поведение только, когда вы размещаете сервис в экземпляре `ServiceHost` (то же самое для вас делает и IIS).
- ASP.NET MVC дает вам возможность указать, какой HTTP verbs вы допустите с атрибутом `[AcceptVerbs]`, а также дает вам возможность обрабатывать исключения с помощью атрибута `[HandleError]`, и еще несколько других возможностей. Это возможно, поскольку ASP.NET MVC – это один большой пользовательский хост, и он управляет жизненными циклами своих контроллеров.
- Все .NET фреймворки модульного тестирования, о которых я осведомлен, используют атрибуты для идентификации тестовых сценариев. Фреймворк модульного тестирования инициализирует тестовые сценарии и интерпретирует атрибуты, чтобы указать, какой из тестов необходимо выполнить.

Компоновка объектов с помощью DI-контейнера аналогична этим примерам. Поскольку DI-контейнер инициализирует экземпляры включенных в него классов, он имеет возможность просматривать каждый класс в поисках пользовательских атрибутов.

Вас не должно удивлять то, что многие DI-контейнеры обладают возможностями, которые позволяют вам делать именно это. Если вы уже решили использовать DI-контейнер, должны ли вы идти до конца и определять, и применять пользовательские атрибуты?

Я могу подразумевать только одно преимущество, которое такое поведение дает нам касательно динамического перехвата: поскольку атрибут очень легко обнаружить, даже если он предлагает достаточно привлекательный уровень преобразования данных, вы все равно получаете ценный намек на то, что что-то происходит, нежели тот намек, который дает рассматриваемое вами тело метода.

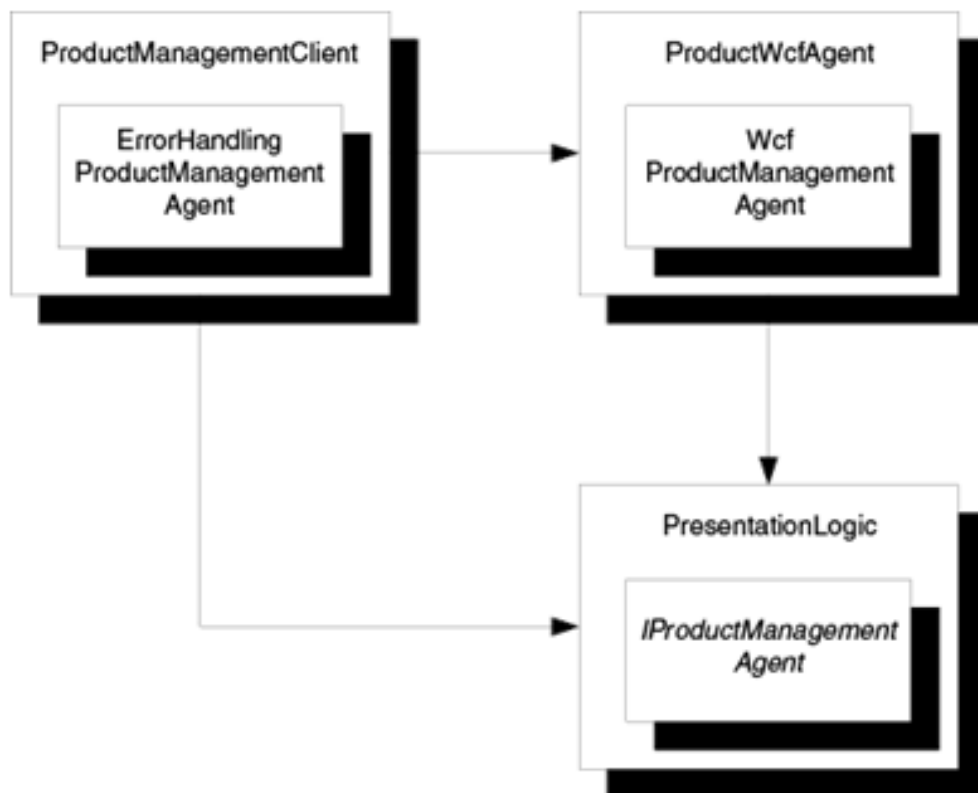
Но есть и недостатки применения сквозных сущностей с помощью атрибутов. Эти недостатки являются общими для пост-компиляции и пользовательских хостов.

Недостатки атрибутов аспектов

Насколько бы привлекательной ни казалась возможность реализации аспектов в виде пользовательских атрибутов, существуют некоторые недостатки такого подхода.

Во-первых, атрибуты компилируются совместно с кодом, который они расширяют. Это означает, что вы не сможете так просто изменить ход своих мыслей. Рассмотрим в качестве примера процесс обработки ошибок. В разделе 9.2.2 "Обработка исключений" вы видели, как можно использовать паттерн проектирования Decorator для того, чтобы реализовать обработку ошибок для любого `IProductManagementAgent`. Интересно то, что упомянутый выше `WcfProductManagementAgent` ничего не знает об `ErrorHandlingProductManagementAgent`. Как проиллюстрировано на рисунке 9-9, они реализованы даже в разных библиотеках.

Рисунок 9-9: Как `ErrorHandlingProductManagementAgent`, так и `WcfProductManagementAgent` реализуют `IProductManagementAgent`, но определены они в двух различных библиотеках. Поскольку сборка `ProductManagementClient` содержит `Composition Root`, она зависит от `ProductWcfAgent` и `PresentationLogic`.



Основная реализация, предложенная `WcfProductManagementAgent`, не содержит явной функциональности обработки ошибок, поскольку истинный процесс обработки исключений является контексто-зависимым. В GUI приложениях, например, в WPF приложении, которое мы не так давно использовали в качестве примера, хорошей стратегией может стать диалоговое сообщение, но в консольном приложении вместо диалогового сообщения вы, вероятнее всего, предпочли бы создавать выходной поток ошибок, а автоматизированный сервис мог бы перемещать операцию с целью повторения очереди и продолжать выполнять что-то еще.

Для того чтобы сохранять свою открытость и гибкость, библиотека `ProductWcfAgent` не должна включать в себя обработку ошибок. Но, если вы применяете атрибут аспекта к `WcfProductManagementAgent` (или, что еще хуже, к `IProductManagementAgent`), то вы сильно привязываете этот аспект к реализации (или даже к абстракции). Если вы так

поступаете, то вы навязываете `WcfProductManagementAgent` определенную стратегию обработки ошибок, и теряете способность варьировать аспекты независимо от реализации.

Вторая проблема, связанная с атрибутами аспектов, заключается в том, что, у вас есть лишь ограниченное количество вариантов применения атрибутов, которые могут использоваться только на следующих уровнях:

- Параметры, включая возвращаемые значения
- Члены, например, методы, свойства и поля
- Типы, например, классы и интерфейсы
- Библиотеки

Несмотря на то, что атрибуты аспектов предоставляют вам широкий круг возможностей, вы не можете с легкостью выразить большее количество конфигураций, основанных на соглашениях, например, "Я хочу применить аспект `Circuit Breaker` ко всем типам, имена которых начинаются с `Wcf`". Вместо этого вам пришлось бы применить гипотетический атрибут `[CircuitBreaker]` ко всем подходящим классам, нарушающим *принцип DRY* (Don't Repeat Yourself – Не повторяйся).

Последний недостаток атрибутов аспектов – атрибуты должны обладать простым конструктором. Если вам нужно использовать зависимости от аспекта, вы можете сделать это только с помощью `Ambient Context`. Вы уже видели пример этого в разделе 9.2.3 "Добавление функциональности обеспечения безопасности", где `Thread.CurrentPrincipal` является `Ambient Context`. Но данный паттерн в редких случаях является наиболее подходящим, и делает процесс управления жизненным циклом более трудным. К примеру, совместное использование `ICircuitBreaker` в рамках многочисленных WCF клиентов неожиданно становится более сложным.

Несмотря на все эти недостатки, привлекательность атрибутов аспектов заключается в том, что вам приходится реализовывать код аспекта только в одном единственном месте. В следующем разделе вы увидите, как можно использовать возможности перехвата DI-контейнеров для того, чтобы достичь этой цели, не связывая при этом сильно атрибуты аспектов.

Применение динамического перехвата

До настоящего момента вы видели, как можно использовать `Decorator`'ы для обращения к сквозным сущностям и их реализации. Данный прием удовлетворяет SOLID принципу, но нарушает DRY принцип. Возможно, это и не видно из примеров данной главы, но применение аспекта путем ручной разработки расширений классов включает в себя огромное количество повторяющегося кода.

Повторяемость `Decorator`'ов

Примеры из разделов 9.1.1 "Пример: реализация аудита" и 9.2.1 "Осуществление перехвата с помощью `Circuit Breaker`" демонстрируют только репрезентативные методы, поскольку каждый метод реализуется одинаково, а мне не хотелось загружать несколько страниц практически идентичным кодом, так как это отвлекало бы нас от того, что мы рассматриваем. Следующий листинг демонстрирует, как похожи методы `CircuitBreakerProductmanagementAgent`. Этот листинг демонстрирует только два метода

интерфейса `IProductManagementAgent`, но я уверен, что вы сможете экстраполировать их и представить, как выглядит остальная реализация.

Листинг 9-7: Нарушение DRY принципа

```
1. public void DeleteProduct(int productId)
2. {
3.     this.breaker.Guard();
4.     try
5.     {
6.         this.innerAgent.DeleteProduct(productId);
7.         this.breaker.Succeed();
8.     }
9.     catch (Exception e)
10.    {
11.        this.breaker.Trip(e);
12.        throw;
13.    }
14. }
15. public void InsertProduct(ProductEditorViewModel product)
16. {
17.     this.breaker.Guard();
18.     try
19.     {
20.         this.innerAgent.InsertProduct(product);
21.         this.breaker.Succeed();
22.     }
23.     catch (Exception e)
24.     {
25.         this.breaker.Trip(e);
26.         throw;
27.     }
28. }
```

Строка 6, 20: Единственное отличие

Так как вы уже видели метод `InsertProduct` в листинге 9-4, цель данного примера кода – проиллюстрировать повторяющуюся сущность Decorator'ов, используемых в виде аспектов. Единственное отличие между методами `DeleteProduct` и `InsertProduct` – это то, что каждый из них вызывает свой собственный соответствующий метод расширенного агента.

Даже если мы успешно делегировали реализацию `Circuit Breaker` отдельному классу посредством интерфейса `ICircuitBreaker`, этот код инфраструктуры явно нарушает DRY принцип. Он может быть склонен к разумной неизменности, но все равно обязателен. Всякий раз, когда вам хочется добавить новый член к типу, который вы расширяете, или когда вам хочется применить `Circuit Breaker` к новой абстракции, вы должны применять этот же самый код инфраструктуры.

В качестве одного из способов решения данной проблемы вы могли бы рассматривать применение таких генераторов кода, как `Text Template Transformation Toolkit (T4)` от `Visual Studio`, но многие DI-контейнеры предлагают лучший вариант решения этой проблемы – посредством динамического перехвата.

Автоматизация Decorator'ов

Код в каждом методе листинга 9-7 очень похож на шаблон. Самая сложная составляющая процесса реализации Decorator'a в виде аспекта – проектирование этого шаблона, но после того, как шаблон спроектирован, остальное – это уже просто механический процесс:

- Создать новый класс
- Унаследовать от нужного интерфейса
- Реализовать каждый член интерфейса путем применения шаблона

Данный процесс является настолько механическим, что вы можете использовать какой-либо инструмент для его автоматизации. Такой инструмент использовал бы рефлексии и схожие API для того, чтобы найти все члены, которые нужно реализовать, а затем применял бы шаблон ко всем этим членам. Рисунок 9-10 демонстрирует, как эту процедуру можно применить с помощью T4 шаблона.

Рисунок 9-10: T4 делает возможной автоматическую генерацию кода Decorator'a из шаблонов. Стартовая точка – прототип шаблона, который понимает основную концепцию Decorator'a. Прототип шаблона содержит код генерации кода, который будет генерировать границы расширяемого класса, но не определяет никакого кода аспекта. Из прототипа шаблона разрабатывается шаблон аспекта, который описывает, как должен применяться конкретный аспект (например, Circuit Breaker) при расширении любого интерфейса. Результатом является специализированный шаблон (*SomeAspect.tt*) этого конкретного аспекта, который можно использовать для генерации Decorator'ов конкретных интерфейсов. Результат – обычный файл кода (*SomeAspectDecorator.cs*), который обычно компилируется вместе с другими файлами кода.



Несмотря на то, что генераторы кода позволяют вам справляться с симптомами повторяющегося кода, они все равно на своем пути оставляют большое количество повторяющегося кода. Если вы верите, что код обязателен, то больший объем кода приведет к большим затратам, независимо от того, автоматически он генерируется или нет.

Даже если вы не купитесь на этот аргумент, у вас все еще есть статический набор автоматически генерируемых Decorator'ов. Если вам понадобится новый Decorator для данной комбинации аспекта и абстракции, то вы должны явным образом добавить данный класс. Он может генерироваться автоматически, но вам все равно нужно не забыть создать его и присоединить. Более условный подход в данном случае не возможен.

Некоторые DI-контейнеры предоставляют вам вариант, который лучше автоматически генерируемого кода: автоматически генерируемые классы. Возможно, разница между этими вариантами кажется скрытой, но продолжайте читать дальше.

Динамический перехват

Одной из множества значительных возможностей .NET Framework является возможность динамически порождать типы. Помимо автоматической генерации кода во время проектирования, также возможно написать код, который порождает полнофункциональный класс во время выполнения. Такой класс не имеет основообразующего файла исходного кода, но компилируется напрямую из некоторой абстрактной модели.

Таким же образом, как вы можете автоматизировать генерацию Decorator'ов в файлах исходного кода, вы можете автоматизировать генерацию Decorator'ов, которые порождались бы прямо во время процесса выполнения. Как демонстрирует рисунок 9-11, это как раз то, что и позволяет вам выполнить динамический перехват.

Примечание

Не все DI-контейнеры поддерживают перехват во время выполнения; если вам нужна данная возможность, позаботьтесь о том, чтобы ваш DI-контейнер был подобран соответствующим образом.

Рисунок 9-11: Некоторые DI-контейнеры позволяют нам определять аспекты в виде перехватчиков. Перехватчик – фрагмент кода, который реализует аспект и взаимодействует с контейнером. Регистрация перехватчика с помощью контейнера позволяет контейнеру динамически создавать и порождать Decorator'ы, которые содержат поведение аспекта. Эти классы существуют только во время выполнения.



Для применения динамического перехвата вам все равно нужно написать код, который реализует аспект. Это мог бы быть код инфраструктуры, необходимый для аспекта Circuit Breaker, что продемонстрировано в листинге 9-7. Как только вы написали этот код, вы должны сообщить DI-контейнеру об этом аспекте и о том, когда ему следует его применять.

Во время выполнения DI-контейнер будет динамически порождать новые классы в выполняемом AppDomain, основанном на зарегистрированных аспектах. Самое лучшее в данном подходе – это то, что вы можете использовать конфигурацию на основе соглашений для того, чтобы определить, как применяются аспекты, и можете решить использовать разные соглашения в разных приложениях (например, несмотря на то, что вы можете совместно использовать множество библиотек, в WPF приложении и PowerShell приложении у вас могут быть разные стратегии обработки ошибок).

Примечание

В аспектно-ориентированном программировании соглашение, которое сопоставляет аспекты с классами и членами, называется *Pointcut*.

Хватит теории – давайте рассмотрим пример.

Пример: перехват с помощью Windsor

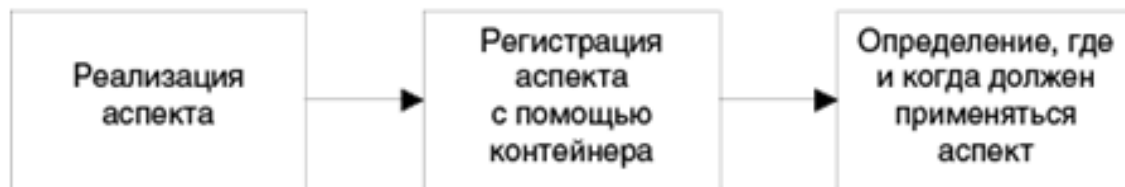
Благодаря их повторяющемуся коду аспекты *Circuit Breaker* и обработчик ошибок из разделов 9.2.1 "Осуществление перехвата с помощью *Circuit Breaker*" и 9.2.2 "Обработка исключений" являются отличными кандидатами для динамического перехвата. В качестве примера давайте рассмотрим, как можно получить DRY, SOLID код с помощью возможностей перехвата *Castle Windsor*.

Примечание

Вместо *Castle Windsor* я мог бы выбрать и другой DI-контейнер, но определенно не любой. Некоторые DI-контейнеры поддерживают механизм перехвата, а остальные не поддерживают – в части 4 рассматриваются возможности конкретных DI-контейнеров.

В данном примере вы будете реализовывать и регистрировать перехватчики как для обработчика ошибок, так и для *Circuit Breaker*. Добавление аспекта в *Windsor* – это процесс, состоящий из трех шагов, как это продемонстрировано на рисунке 9-12.

Рисунок 9-12: Три шага, составляющие процесс добавления аспекта в *Windsor*



В данном примере вы будете выполнять эти шаги для обоих аспектов. Обработку ошибок реализовать проще всего, поскольку у нее нет зависимостей; давайте начнем с этого аспекта.

Реализация перехватчика обработки ошибок

Реализация перехватчика для *Windsor* требует от нас реализации интерфейса *IInterceptor*, который имеет всего один метод. Следующий листинг демонстрирует, как реализовать стратегию обработки ошибок из листинга 9-5, но в отличие от листинга 9-5 следующий листинг демонстрирует весь класс.

Листинг 9-8: Реализация перехватчика обработки ошибок

```
1. public class ErrorHandlerInterceptor : IInterceptor
2. {
3.     public void Intercept(IInvocation invocation)
4.     {
5.         try
6.         {
7.             invocation.Proceed();
8.         }
```

```

9.         catch (CommunicationException e)
10.        {
11.            this.AlertUser(e.Message);
12.        }
13.        catch (InvalidOperationException e)
14.        {
15.            this.AlertUser(e.Message);
16.        }
17.    }
18.    private void AlertUser(string message)
19.    {
20.        var sb = new StringBuilder();
21.        sb.AppendLine("An error occurred.");
22.        sb.AppendLine("Your work is likely lost.");
23.        sb.AppendLine("Please try again later.");
24.        sb.AppendLine();
25.        sb.AppendLine(message);
26.        MessageBox.Show(sb.ToString(), "Error",
27.            MessageBoxButton.OK, MessageBoxImage.Error);
28.    }
29. }

```

Строка 1: Реализация `IInterceptor`

Строка 5-6, 8-16: Реализация аспекта

Строка 7: Вызов расширенного метода

Строка 18: Демонстрация диалогового окна

Для того чтобы реализовать перехватчик, вы должны выполнить наследование от интерфейса `IInterceptor`, определенного Windsor. Нужно реализовать всего один метод, и вы делаете это путем применения того же самого кода, который вы не раз использовали, когда реализовывали `ErrorHandlingProductManagementAgent`.

Единственное отличие от листинга 9-5 – вместо того, чтобы делегировать вызов метода конкретному методу, вы должны обобщить все, поскольку применяете этот код к потенциально любому методу. Вы даете Windsor указание, позволить вызову пройти дальше к расширенной строке, вызывая для этого метод `Proceed` для входного параметра `invocation`.

Интерфейс `IInvocation`, переданный в метод `Intercept` в качестве параметра, представляет собой вызов метода. К примеру, он мог бы представлять вызов метода `InsertProduct`. Метод `Proceed` – один из ключевых членов данного интерфейса, поскольку он дает нам возможность позволить вызову пройти к следующей реализации стека.

Интерфейс `IInvocation` также дает вам возможность установить выходное значение перед тем, как разрешить вызову пройти дальше, а также обеспечивает доступ к подробной информации о вызове метода. Из параметра `invocation` можно получить информацию об имени метода и значениях параметра, а также множество другой информации о текущем вызове метода.

Реализация перехватчика – сложный шаг. Следующий шаг – полегче.

Регистрация перехватчика обработки ошибок

Перехватчик необходимо зарегистрировать с помощью контейнера до того, как его можно будет использовать. Данный шаг не устанавливает правила, управляющие тем, *как* и *когда* активизируется перехватчик (*Pointcut*), а только делает его доступным в виде компонента.

Примечание

Вы можете рассматривать этот шаг как формальность для соответствия требованиям Windsor. Одна из причуд Windsor заключается в том, что каждый компонент должен быть зарегистрирован явным образом, даже когда он является конкретным типом, обладающим конструктором по умолчанию. Не все DI-контейнеры работают таким образом, но в Windsor такое поведение преднамеренно.

Зарегистрировать класс `ErrorHandlingInterceptor` легко (`container` – это экземпляр `IWindsorContainer`):

```
container.Register(Component.For<ErrorHandlingInterceptor>());
```

Регистрация класса `ErrorHandlingInterceptor` ничем не отличается от регистрации любого другого компонента с помощью Windsor, и вы даже могли бы решить использовать подход основанный на соглашениях для того, чтобы зарегистрировать все реализации `IInterceptor`, обнаруженные в конкретной сборке. Возможно, это похоже на шаблонный код из раздела 3.2 "Конфигурирование DI-контейнеров".

Последний шаг активации перехватчика – определить правила того, когда и как применяется этот перехватчик, но поскольку эти правила также должны затрагивать перехватчик `Circuit Breaker`, мы отложим этот шаг до тех пор, пока также не будет готов и другой перехватчик.

Реализация перехватчика `Circuit Breaker`

Перехватчик `Circuit Breaker` несколько сложнее, поскольку для него необходима зависимость `ICircuitBreaker`, но как демонстрирует следующий листинг, вы решаете этот вопрос путем применения стандартного `Constructor Injection`. Когда дело доходит до компоновки класса, Windsor обращается с ним как с любым другим компонентом: так как он может разрешать зависимость, то все в порядке.

Листинг 9-9: Реализация перехватчика `Circuit Breaker`

```
1. public class CircuitBreakerInterceptor : IInterceptor
2. {
3.     private readonly ICircuitBreaker breaker;
4.     public CircuitBreakerInterceptor(
5.         ICircuitBreaker breaker)
6.     {
7.         if (breaker == null)
8.         {
9.             throw new ArgumentNullException(
10.                "breaker");
11.         }
12.         this.breaker = breaker;
13.     }
14.     public void Intercept(IInvocation invocation)
15.     {
```

```

16.         this.breaker.Guard();
17.         try
18.         {
19.             invocation.Proceed();
20.             this.breaker.Succeed();
21.         }
22.         catch (Exception e)
23.         {
24.             this.breaker.Trip(e);
25.             throw;
26.         }
27.     }
28. }

```

Строка 3-13: Constructor Injection

Строка 16-18, 20-26: Реализация аспекта

Строка 19: Вызов расширенного метода

Для `CircuitBreakerInterceptor` нужна зависимость `ICircuitBreaker`, а внедрение зависимостей в `IInterceptor` выполняется с помощью `Constructor Injection`, как и в любых других сервисах.

Как вы видели в листинге 9-8, вы реализуете интерфейс `IInterceptor` путем применения шаблона, предложенного предыдущей, повторяющейся реализацией из листинга 9-4. Еще раз вместо вызова конкретного метода вы вызываете метод `Proceed` для того, чтобы дать перехватчику указание, позволить обработке продолжить свое выполнение для следующего компонента стека `Decorator`.

К данному моменту вы должны начать понимать формирование паттерна. Вместо того чтобы повторять код инфраструктуры `Circuit Breaker` для каждого метода абстракции, вы можете определить его всего один раз в перехватчике.

Также вам необходимо зарегистрировать класс `CircuitBreakerInterceptor` с помощью контейнера; так как у класса есть зависимость, для этого потребуется не одна, а две строки кода.

Регистрация перехватчика `Circuit Breaker`

Для перехватчика обработки ошибок нужна была только одна строка кода регистрации, но, поскольку `CircuitBreakerInterceptor` зависит от `ICircuitBreaker`, вы должны зарегистрировать и эту зависимость:

```

container.Register(Component
    .For<ICircuitBreaker>()
    .ImplementedBy<CircuitBreaker>()
    .DependsOn(new
    {
        timeout = TimeSpan.FromMinutes(1)
    }));
container.Register(Component.For<CircuitBreakerInterceptor>());

```

Вы преобразуете интерфейс `ICircuitBreaker` в конкретный класс `CircuitBreaker`, для которого нужен такой параметр конструктора, как время ожидания.

Когда оба перехватчика находятся на своих местах, единственное, что нам еще нужно – определить правила того, когда они будут активизироваться.

Активация перехватчиков

К настоящему моменту перехватчики реализованы и зарегистрированы с помощью контейнера Windsor, но вам еще нужно определить, когда они будут активироваться. Если вы не сделаете это, они останутся всего лишь пассивными регистрациями в контейнере и даже не будут вызываться.

Вы можете рассматривать этот шаг как аналог применения атрибутов аспектов. Если мы применяем к методу гипотетический атрибут `[CircuitBreaker]`, то мы соединяем аспект `Circuit Breaker` с этим методом. Определение и применение пользовательских атрибутов – один из способов, с помощью которых мы можем активизировать перехватчиков Windsor, но у нас еще есть несколько других, более подходящих доступных вариантов.

Самый гибкий – реализовать и зарегистрировать интерфейс `IModelInterceptorsSelector`. Это дает нам возможность написать императивный код, который решает, какой перехватчик к каким типам и членам применять. Поскольку мы можем написать условно сложный код, мы имеем возможность применить наши аспекты гораздо более основанным на соглашениях способом.

В следующем листинге вы используете простую реализацию такого Pointcut.

Листинг 9-10: Реализация Pointcut

```
1. public class ProductManagementClientInterceptorSelector :
2.     IModelInterceptorsSelector
3.     {
4.         public bool HasInterceptors(ComponentModel model)
5.         {
6.             return typeof(IProductManagementAgent)
7.                 .IsAssignableFrom(model.Service);
8.         }
9.         public InterceptorReference[]
10.            SelectInterceptors(ComponentModel model,
11.            InterceptorReference[] interceptors)
12.        {
13.            return new[]
14.            {
15.                InterceptorReference
16.                    .ForType<ErrorHandlingInterceptor>(),
17.                InterceptorReference
18.                    .ForType<CircuitBreakerInterceptor>()
19.            };
20.        }
21.    }
```

Строка 6-7: Применение перехватчиков к `IProductManagementAgent`

Строка 15-18: Возврат перехватчиков

Интерфейс `IModelInterceptorsSelector` руководствуется паттерном `Tester-Doer`. Windsor сначала вызовет метод `HasInterceptors`, чтобы узнать, имеет ли данный компонент, который он собирается инициализировать, какие-либо перехватчики. В данном примере вы отвечаете на этот вопрос положительно, при этом компонент

реализует интерфейс `IProductManagementAgent`, но вы могли бы написать условно сложный код, если бы захотели реализовать более эвристический подход.

Когда метод `HasInterceptors` вернет значение `true`, будет вызван метод `SelectInterceptors`. Благодаря этому методу вы возвращаете ссылки на перехватчиков, которые вы уже зарегистрировали. Обратите внимание, что вы возвращаете не экземпляры перехватчиков, а ссылки на перехватчики, которые вы уже реализовали и зарегистрировали.

Это позволяет Windsor автоматически интегрировать любые перехватчики, которые могут иметь свои собственные зависимости (например, `CircuitBreakerInterceptor` из листинга 9-9).

Знаете, что! Вам еще нужно зарегистрировать класс `ProductManagementClientInterceptorSelector` с помощью контейнера. Это делается немного по-другому, но все равно вкладывается в одну строку кода:

```
container.Kernel.ProxyFactory.AddInterceptorSelector(  
    new ProductManagementClientInterceptorSelector());
```

Регистрация `ProductManagementClientInterceptorSelector` окончательно активизирует перехватчиков так, что когда вы разрешаете приложение с помощью Windsor, они автоматически активизируются там, где они должны активизироваться.

Вы можете подумать, что этот многостраничный анализ перехватчиков Windsor довольно сложен, но вам следует кое-что иметь в виду:

- Вы реализовали два, а не один, перехватчика.
- Я повторил некоторый фрагмент кода из предыдущих примеров для того, чтобы показать, как его можно приспособить. Не важно, решите ли вы писать Decorator'ы вручную, использовать фреймворк аспектно-ориентированного программирования или использовать динамический перехват, вам всегда нужно будет писать код, который реализует аспект.

Динамический перехват предоставляет множество преимуществ. Он позволяет нам иметь дело со сквозными сущностями, соблюдая при этом и SOLID, и DRY принципы. Он дает нам по-настоящему слабо связанные аспекты и возможность применять соглашения или сложные эвристические правила для определения того, когда и где какие аспекты применять. Это максимальный уровень свободы и гибкости.

Вы можете беспокоиться о результатах исполнения компиляции и порождения пользовательских типов "на лету", но насколько я смог определить, Windsor делает это только один раз и повторно использует данный тип для последующих вызовов. Я сделал несколько неформальных отметок, не регистрируя никакого значительного ухудшения исполнения.

Еще одна проблема – добавленный уровень преобразования. Вы могли бы поспорить, что, применяя атрибуты аспектов, мы все равно оставляем в основном методе заметный след того, что поведение, изменяющее аспекты, приходится к месту. Благодаря Decorator'ам и динамическому перехвату в частности такой след отсутствует. Начинаящие разработчики могли бы теоретически напороться на это полу-магическое поведение, завязнув в нем на несколько дней, пока кто-нибудь не поможет им, объяснив концепцию.

В некоторых инфраструктурах это является реальной проблемой. Подумайте, как бы вы справились с этой проблемой, если бы решили применить динамический перехват.

9.4. Резюме

Механизм внедрения зависимостей по-настоящему расцветает, когда дело доходит до применения объектно-ориентированных принципов, например, SOLID. В частности, слабо связанная природа механизма внедрения зависимостей позволяет нам использовать паттерн Decorator для того, чтобы соблюдать принцип открытости/закрытости, а также принцип единственной ответственности. Эта возможность ценна в широком круге ситуаций, поскольку позволяет нам оставлять наш код чистым и хорошо-структурированным, но применяется особенно хорошо, когда дело доходит до сквозных сущностей.

Сквозные сущности традиционно принадлежат области аспектно-ориентированного программирования, но также могут с большим успехом разрешаться с помощью механизма внедрения зависимостей. Паттерн проектирования Decorator – это основной паттерн, позволяющий нам обертывать существующую функциональность в дополнительные уровни поведения, не изменяя первоначальный код.

Но, несмотря на свои многочисленные достоинства, главная проблема, связанная с реализацией Decorator'ов, – это то, что они склонны к многословности и повторяемости. Даже если вы можете соблюдать SOLID принципы проектирования, мы прекращаем нарушать DRY принцип, поскольку нам нужно писать снова и снова один и тот же код инфраструктуры – для каждого члена каждого интерфейса, который мы хотим расширить с помощью конкретного аспекта.

Кажется, что атрибуты похожи на привлекательную альтернативу Decorator'ов, поскольку они позволяют нам добавлять аспекты более сжатым способом. Но, поскольку атрибуты компилируются в код, который они расширяют, они приводят к сильному связыванию, и их применения нужно избегать.

Некоторые DI-контейнеры предлагают более привлекательную альтернативу благодаря возможности динамически порождать Decorator'ы во время выполнения. Эти динамические Decorator'ы обеспечивают перехват, который соблюдает как SOLID, так и DRY принципы.

Интересно отметить, что динамический перехват – единственная особенность DI-контейнеров, которая не имеет прямого эквивалента в Poor Man's DI. С этой точки зрения, в части 3 вы видели, как обращаться с композицией объектов и механизмом управления жизненным циклом с помощью благоразумного применения паттернов, но когда дело доходит до механизма перехвата, самое близкое, что мы получаем – это множество Decorator'ов.

Несмотря на то, что понятие Decorator сравнимо с механизмом перехвата, предпочтителен скачок от множества закодированных вручную Decorator'ов к одному, а также DRY перехватчик. Скачок от множества Decorator'ов к одному может привести к вспышке повторяющегося кода инфраструктуры, тогда как использование DRY перехватчика предлагает возможность обращения к сквозным сущностям с помощью нескольких строк кода и в качестве дополнительного бонуса – возможность использовать применение аспектов на основе соглашений.

Именно здесь, в заключении части 3, мы, наконец, достигаем той области, где DI-контейнеры бесспорно оставляют Poor Man's DI позади. Даже без механизма перехвата DI-контейнер может гораздо лучше управлять сложностью, вложенной в процесс преобразования абстракций к конкретным типам, а также управлять их жизненными циклами; но когда мы добавим к этому сочетанию механизм перехвата, мы уже не сможем разрушить комбинацию.

На этой ноте мы можем с радостью оставить Poor Man's DI за частью 3 и перейти к чтению информации о конкретных DI-контейнерах в части 4.

DI-контейнеры

Предыдущие части этой книги были посвящены разнообразным принципам и паттернам, которые совместно определяют механизм внедрения зависимостей. Как объяснялось в главе 3, DI-контейнер – это необязательный инструмент, который можно использовать для реализации большого количества общецелевой инфраструктуры, которую вам пришлось бы реализовать иным образом, если бы вы использовали Poor Man's DI.

На протяжении всей книги я стремился по возможности сохранять ее, независимой от контейнеров. Не принимайте это в качестве рекомендации Poor Man's DI; наоборот, я хотел, чтобы вы увидели механизм внедрения зависимостей в его истинной форме, незапятнанным API какого-либо конкретного контейнера (возможно, необычным).

Есть небольшая причина того, чтобы потратить свое время на изучение Poor Man's DI, так как в .NET доступны многие превосходные DI-контейнеры. В данной части 4 представлен набор из 6 конкретных бесплатных контейнеров. В каждой главе я предоставляю детальный обзор API конкретного контейнера, как он относится к масштабам, охватываемым в части 3, а также другие разнообразные проблемы, которые являются причинами бед начинающих разработчиков.

В части 4 рассматриваются такие контейнеры, как Castle Windsor, StructureMap, Spring.NET, Autofac, Unity и MEF. Существуют и другие фреймворки контейнеров, которые я, по той или иной причине, не включил в данную книгу: Ninject, Hilo, Funq, LinFu, Ocinject и т.д. Пожалуйста, обратите внимание на то, что содержание данной книги само по себе не является предвестником одобрения. Несмотря на то, что многие из включенных в эту книгу DI-контейнеров обладают превосходным качеством, это не было единственным критерием отбора. Существуют конкретные контейнеры, которые я не включил в эту книгу всего лишь по причине той доли, в которой они представлены на рынке. Между тем я решил описать остальные, поскольку они обеспечивают отличный контраст благодаря их различным философиям проектирования и целям. Это также означает, что мне пришлось исключить некоторые превосходные контейнеры. Мне, конечно, хотелось бы включить в эту книгу все DI-контейнеры, но, увы, поскольку мое пространство ограничено, это не возможно.

Каждая глава руководствуется общим шаблоном. Чтение одного и того же предложения по 6 раз может дать вам определенное ощущение дежавю, но я считаю это достоинством, поскольку такая структура глав должна дать вам возможность быстро находить аналогичные разделы в рамках разных глав, если вы захотите сравнить, как конкретная деталь раскрывается в рамках двух или более контейнеров. Для быстрого сравнения обращайтесь к следующей таблице:

DI-контейнер	Достоинства	Недостатки
Castle Windsor	Завершенность Понимает Decorator Типизированные фабрики Доступно коммерческое сопровождение	Местами извилистое API

DI-контейнер	Достоинства	Недостатки
StructureMap	Просто работает в большинстве случаев	Не поддерживает механизм перехвата
Spring.NET	Механизм перехвата Исчерпывающая документация Доступно коммерческое сопровождение	Очень завязан на XML Не поддерживает API, основанное на соглашениях Не поддерживает пользовательские жизненные циклы Ограниченные возможности автоматической интеграции
Autofac	Простое для изучения API Доступно коммерческое сопровождение	Не поддерживает механизм перехвата Частично поддерживает пользовательские жизненные циклы
Unity	Механизм перехвата Хорошая документация Неизменное API	Небольшие возможности управления жизненным циклом Не поддерживает API, основанное на соглашениях
MEF	Доступен в .NET 4/Silverlight 4 BCL Коммерческое сопровождение	Не настоящий DI-контейнер Конфигурация, основанная на статических атрибутах Не поддерживает XML Не поддерживает технологию использования кода в качестве конфигурации Не поддерживает API, основанное на соглашениях Не поддерживает пользовательские жизненные циклы Не поддерживает механизм перехвата

Многие из описанных здесь контейнеров – это проекты с открытым исходным кодом, имеющие быстрые циклы выпуска. На момент написания книги информация, представленная в части 4, была актуальна, но всегда старайтесь также консультироваться с более современными источниками.

Эти главы рассматриваются в качестве стимула. Если вы еще не выбрали тот контейнер, который вам больше всего нравится, то можете прочитать эти шесть глав, чтобы сравнить все контейнеры, но можете прочитать одну или две главы, которые действительно вас интересуют.

[10. Castle Windsor](#)

[11. StructureMap](#)

[12. Spring.NET](#)

[13. Autofac](#)

[14. Unity](#)

15. MEF

10. Castle Windsor

Меню:

- Знакомство с Castle Windsor
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В предыдущих девяти главах мы обсуждали паттерны и принципы, которые применяются к механизму внедрения зависимостей в общем, но за исключением нескольких примеров мы еще не рассматривали подробно то, как применять эти принципы и паттерны при использовании конкретного DI-контейнера. В этой главе вы увидите, как все эти общие паттерны применяются к Castle Windsor, для того чтобы эта глава принесла вам полноценную пользу, вам нужно ознакомиться с материалом предыдущих глав.

Castle Windsor – второй по старшинству DI-контейнер .NET. Он является частью более крупного проекта с открытым исходным кодом, известного как Castle Project, который предоставляет повторно используемые библиотеки для различных целей. Windsor является составляющей DI-контейнера в Castle Project, но может использоваться и независимо от любых других компонентов Castle. В этой главе мы рассмотрим его как отдельный компонент.

Помимо того, что он является одним из самых старых DI-контейнеров, Castle Windsor еще и один из самых зрелых контейнеров, и, если верить нескольким совсем не научным интернет-опросам, один из самых популярных контейнеров. Несмотря на то, что начать работать с Windsor достаточно легко, он предлагает богатое и расширяемое API.

В данной главе мы попутешествуем по Castle Windsor. Когда мы завершим свой тур, вы должны будете уже иметь достаточные знания об этом контейнере, чтобы суметь незамедлительно применять его. Мы не собираемся рассматривать продвинутые сценарии расширяемости, а вместо этого сконцентрируемся на главных паттернах применения. Рисунок 10-1 демонстрирует структуру главы.

Рисунок 10-1: Структура данной главы похожа на дерево. Первый раздел знакомит вас с контейнером Castle Windsor и объясняет, как сконфигурировать и разрешить компоненты. Остальные разделы, базирующиеся на введении, можно читать последовательно или более или менее независимо. В последнем разделе используется синтаксис и несколько методов, которые в первый раз встречаются в разделе, посвященном составным компонентам, поэтому, если вы захотите пропустить предпоследний раздел, можете случайно вернуться к нему снова.



Первый раздел обеспечивает полноценное знакомство с Castle Windsor и демонстрирует, как конфигурировать и разрешать компоненты. Следующие три раздела имеют дело с паттернами применения, которые требуют излишнего внимания; вы можете прочитать их все по порядку или пропустить некоторые и читать только те, которые вас интересуют.

Данная глава должна дать вам возможность приступить к работе, а также коснуться наиболее универсальных проблем, которые могут возникнуть при повседневном использовании Castle Windsor. Глава 10 не является законченной трактовкой Castle Windsor – на рассмотрение его могла бы уйти вся эта книга.

Вы можете прочитать эту главу изолированно от остальных глав части 4, чтобы изучить конкретно Castle Windsor, или можете прочитать ее совместно с другими главами, чтобы сравнить DI-контейнеры. Цель данной главы – показать, какое отношение Castle Windsor имеет к паттернам и принципам, описанным в предыдущих девяти главах, и как он их реализует.

[10.1. Знакомство с Castle Windsor](#)

[10.2. Управление жизненным циклом](#)

[10.3. Работа с составными компонентами](#)

[10.4. Конфигурирование сложных API](#)

[10.5. Резюме](#)

10.1. Знакомство с Castle Windsor

Из этого раздела вы узнаете, где можно взять Castle Windsor, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим универсальные варианты конфигурации, а также то, как упаковать настройки конфигурации в повторно используемые компоненты. В таблице 10-1 содержится фундаментальная информация, которая, скорее всего, понадобится вам для того, чтобы приступить к работе.

Как показывает рисунок 10-2, в Castle Windsor присутствует простая цикличность: конфигурирование контейнера путем добавления компонентов, а затем разрешение необходимых компонентов.

Рисунок 10-2: Общий паттерн применения Castle Windsor прост: сначала мы конфигурируем контейнер, затем разрешаем компоненты из этого контейнера. В большинстве случаев мы создаем экземпляр `WindsorContainer` и полностью конфигурируем его перед тем, как начать разрешать компоненты из этого контейнера. Мы разрешаем компоненты из того экземпляра, который конфигурируем.



Таблица 10-1: Краткая информация о Castle Windsor

Вопрос	Ответ
Откуда мне его получить?	Перейти на сайт http://www.castleproject.org/download/ и нажать на ссылку соответствующего релиза. Из Visual Studio 2010 можно получить его посредством NuGet. Имя пакета – <code>Castle.Windsor</code> .
Что находится в загруженном файле?	Можно загрузить zip-файл, содержащий предварительно скомпилированные бинарные файлы. Кроме того, можно получить текущий исходный код и скомпилировать его самостоятельно. Бинарные файлы – это dll-файлы, которые можно размещать там, где захочется, и ссылаться на них из собственного кода.
Какие платформы поддерживаются?	.NET 3.5 SP1, .NET 4 Client Profile, .NET 4, Silverlight 3, Silverlight 4.
Сколько он стоит?	Нисколько. Это программное обеспечение с открытым исходным кодом, обладающее мягкой лицензией.
Где мне получить помощь?	Коммерческое сопровождение можно получить от Castle Stronghold. Больше информации по этому вопросу можно получить на сайте www.castlestronghold.com/services/support .

Вопрос	Ответ
	Помимо коммерческого сопровождения, Castle Windsor остается программным обеспечением с открытым исходным кодом, имеющим бурно развивающуюся экосистему, поэтому, скорее всего (но не гарантированно), вы получите помощь на официальном форуме http://groups.google.com/group/castle-project-users . Stack Overflow (http://stackoverflow.com/) – еще одно место, где можно задать вопросы.
На какой версии Castle Windsor основана данная глава?	2.5.2

После окончания изучения данного раздела вы должны будете уже хорошо разбираться в Castle Windsor, а также уметь использовать его в сценариях, в которых все компоненты руководствуются соответствующими DI-паттернами, например, Constructor Injection. Мы начнем с самого простого сценария и увидим, как можно разрешать объекты с помощью контейнера Windsor.

Разрешение объектов

Основная цель каждого DI-контейнера – разрешать объекты посредством подключения к ним всех их зависимостей. Castle Windsor предоставляет простое API для разрешения сервисов, но перед тем, как вы сможете разрешить сервис, его необходимо зарегистрировать с помощью контейнера. Ниже приведено самое простое возможное применение Windsor:

```
var container = new WindsorContainer();
container.Register(Component.For<SauceBéarnaise>());
SauceBéarnaise sauce = container.Resolve<SauceBéarnaise>();
```

Перед тем, как попросить WindsorContainer *что-то* разрешить, вы должны явным образом зарегистрировать соответствующие компоненты. В данном примере вы можете достичь этого, регистрируя единственный конкретный тип. Тем не менее, как вы увидите далее, чаще всего вы будете регистрировать преобразования от абстракции к конкретному типу.

После соответствующей конфигурации контейнера можно разрешить тип `SauceBéarnaise` для того, чтобы получить его экземпляр. Вам не придется выполнять проверку на `null`-значение, поскольку `WindsorContainer` выдаст исключение, если не сможет выполнить автоматическую интеграцию и вернуть экземпляр запрашиваемого типа.

Примечание

Для Windsor необходимо, чтобы все запрашиваемые компоненты были зарегистрированы, даже если они являются конкретными типами. Это сделано намеренно, но данная схема не используется в других DI-контейнерах.

Этот первый пример функционально эквивалентен прямому созданию экземпляра класса `SauceBéarnaise` с помощью ключевого слова `new`: ничего нового еще не приобретено. Вспомните, что механизм внедрения зависимостей – средство достижения результата, а

результатом является слабое связывание. Для того чтобы достичь слабого связывания, вы должны преобразовать абстракции к конкретным типам.

Преобразование абстракций к конкретным типам

Несмотря на то, что иногда нужно регистрировать сам конкретный класс, гораздо более универсальное требование – преобразовать абстракцию к конкретному классу. В конце концов, это и есть основная услуга, предлагаемая DI-контейнерами.

Ниже вы преобразуете интерфейс `IIngredient` к конкретному классу `SauceBéarnaise`, который позволяет вам успешно разрешать `IIngredient`:

```
var container = new WindsorContainer();
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
IIngredient ingredient = container.Resolve<IIngredient>();
```

Вместо регистрации конкретного типа вы преобразуете абстракцию к конкретному типу. Когда вы позднее запросите экземпляр `IIngredient`, контейнер вернет экземпляр `SauceBéarnaise`.

Строго типизированное свободное API, доступное через класс `Component` (`Castle.MicroKernel.Registration.Component`, а не `System.ComponentModel.Component`), помогает предотвратить ошибки конфигурации, поскольку метод `ImplementedBy` имеет *generic*-ограничитель, который гарантирует, что тип, указанный в аргументе типа, реализует аргумент типа абстракция, заданный в методе `For`. То есть, предыдущий код примера компилируется, потому что `SauceBéarnaise` реализует `IIngredient`.

Во многих случаях строго типизированное API – это все, что вам нужно, и, поскольку оно обеспечивает желаемую проверку во время компиляции, вы должны использовать его там, где это возможно. Кроме того, существуют ситуации, в которых вам необходим слабо типизированный способ разрешения сервисов. Это также возможно.

Разрешение слабо типизированных сервисов

В некоторых случаях мы не можем писать *generic*-код для разрешения типа, так как можем даже не знать точный тип абстракции на этапе проектирования. Хороший пример такой ситуации – `DefaultControllerFactory` в ASP.NET MVC, который мы обсуждали в разделе 7.2 "Построение ASP.NET MVC приложений". Соответствующая часть этого класса – виртуальный метод `GetControllerInstance`:

```
protected internal virtual IController GetControllerInstance(
    RequestContext requestContext, Type controllerType);
```

В этом API нет строго типизированных *generic*-ов. Вместо них нам предоставляют `Type` и просят вернуть экземпляр `IController`. Класс `WindsorContainer` также обладает слабо типизированной версией метода `Resolve`. Вы можете использовать этот метод для реализации `GetControllerInstance`:

```
return (IController) this.container.Resolve(controllerType);
```

Обратите внимание на то, что в этом примере вы передаете аргумент `controllerType` в метод `Resolve`. Поскольку слабо типизированная версия метода `Resolve` возвращает экземпляр `System.Object`, вы должны явным образом выполнить приведение к `IController` перед тем, как вернуть результат.

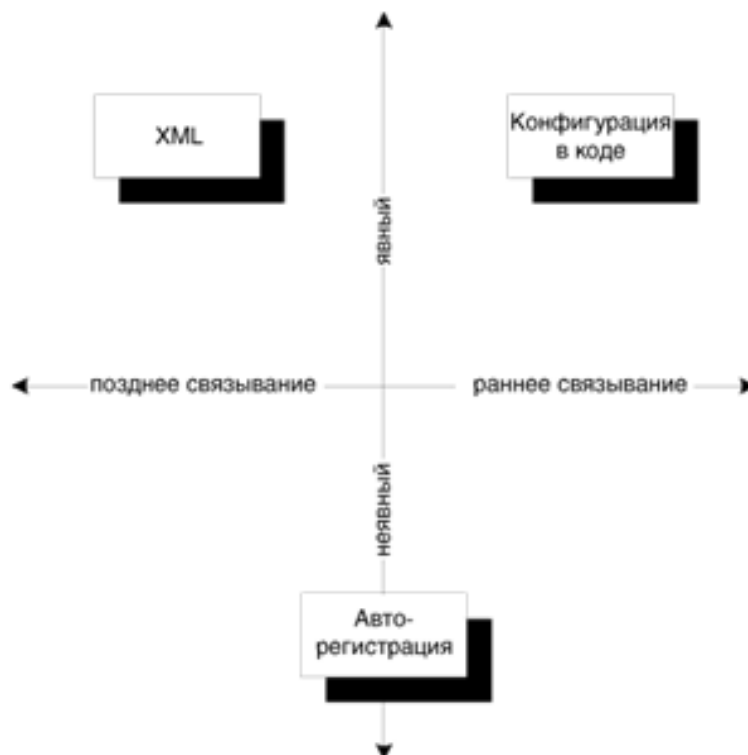
Не важно, какую перегрузку метода `Resolve` вы используете, Windsor гарантирует, что она вернет экземпляр требуемого типа или выдаст исключение, если существуют зависимости, не удовлетворяющие данным условиям. После правильной регистрации всех необходимых зависимостей Windsor будет автоматически интегрировать необходимый тип на основании его конфигурации.

В предыдущем примере `this.container` — это экземпляр `IWindsorContainer`. Для того чтобы иметь возможность разрешать требуемые типы, необходимо перед этим зарегистрировать все типы и их зависимости. Существует множество способов конфигурации контейнера Windsor, и в следующем разделе рассматриваются наиболее универсальные из этих способов.

Конфигурирование контейнера

Как обсуждалось в разделе 3.2 "Конфигурирование DI-контейнеров", существует несколько, концептуально разных способов конфигурирования DI-контейнера. На рисунке 10-3 представлен обзор возможных вариантов.

Рисунок 10-3: Концептуально разные варианты конфигурирования. Использование кода в качестве конфигурации подразумевает строгую типизированность и явное объявление. XML, с другой стороны, — позднее связывание, но все равно явное объявление. Автоматическая регистрация полагается на соглашения, которые могут быть строго типизированными и более слабо определенными.



Как и другие, имеющие длинную историю DI-контейнеры, Castle Windsor сначала использовал XML в качестве главного источника конфигурации. Но многие команды разработчиков вскоре поняли, что определение регистрации типов с помощью XML – чрезвычайно хрупкий способ. На сегодняшний момент мы предпочитаем строго типизированную конфигурацию. Это можно сделать, используя код в качестве конфигурации, но чаще всего более эффективно делать это при помощи автоматической регистрации, которая в большей степени основывается на соглашениях.

Castle Windsor поддерживает все три подхода и даже позволяет нам сочетать их в пределах одного и того же контейнера. В этом отношении, Castle Windsor дает нам все, что мы только могли бы попросить. В данном разделе вы увидите, как можно использовать каждый из этих трех типов источников конфигурации.

Code as Configuration

В главе 3 вы видели примеры API Castle Windsor, использующего код в качестве конфигурации. Каждая регистрация инициализируется с помощью метода `Register` и обычно указывается при помощи Fluent API.

Мы конфигурируем `WindsorContainer` методом `Register`, который в качестве входных данных принимает массив `IRegistration`. На первый взгляд все это выглядит довольно абстрактным. Но вместо того, чтобы возлагать на нас обязанность определения того, какую реализацию `IRegistration` использовать, Castle Windsor предоставляет `Fluent Registration API`, которое позволяет создавать экземпляры `IRegistration` с более понятным синтаксисом.

Для того чтобы применять `Fluent Registration API`, мы используем статический класс `Component` в качестве точки входа.

Предупреждение

Не путайте `Castle.MicroKernel.Registration.Component` с `System.ComponentModel.Component` из стандартной библиотеки классов.

Как вы уже видели ранее, самая простая возможная регистрация – регистрация конкретного типа:

```
container.Register(Component.For<SauceBéarnaise>)();
```

Данный код регистрирует класс `SauceBéarnaise` с помощью контейнера, но не обеспечивает никакого преобразования. Даже если `SauceBéarnaise` будет реализовывать `IIngredient`, контейнер выдаст исключение, если вы попросите его разрешить `IIngredient`:

```
container.Resolve<IIngredient>()
```

Для возможности такого более релевантного сценария вы должны преобразовать конкретный тип в абстракцию:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

Обратите внимание на то, что теперь вместо класса `SauceBéarnaise` вы регистрируете интерфейс `IIngredient`. Это позволит вам разрешить `IIngredient`, но, что может показаться слегка удивительным, вы при этом потеряли способность разрешать конкретный класс `SauceBéarnaise`. Изредка тот факт, что код слабо связан, становится проблемой, но в исключительных ситуациях, когда вам нужно уметь разрешать оба типа, вы можете обеспечить это с помощью перегрузки метода `For`:

```
container.Register(Component
    .For<SauceBéarnaise, IIngredient>());
```

Данный код регистрирует компонент `SauceBéarnaise`, в то же самое время перенаправляя регистрацию к интерфейсу `IIngredient`. Это означает, что и `SauceBéarnaise`, и `IIngredient` регистрируются как разрешаемые типы. В обоих случаях реализация обеспечивается с помощью `SauceBéarnaise`. Заметьте, что при использовании этой перегрузки вам не нужно неявно использовать метод `ImplementedBy`.

Очевидно, вы можете регистрировать составные типы при помощи последовательных вызовов метода `Register`:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<Course>());
```

Данный код регистрирует и интерфейс `IIngredient`, и интерфейс `ICourse`, а также преобразует их к конкретным типам. Тем не менее, регистрация одной и той же абстракции несколько раз имеет приводит к некоторым интересным последовательностям:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

В этом примере вы регистрируете `IIngredient` дважды. Если вы разрешаете `IIngredient`, то получаете экземпляр `Steak`. Выигрывает первая регистрация, но последующие регистрации не забыты. `Castle Windsor` имеет изощренную модель работы с составными регистрациями. Мы вернемся к этому в разделе 10.3 "Работа с составными компонентами".

В `Fluent Registration API` доступны более продвинутые варианты, но данным способом мы можем сконфигурировать все приложение. Тем не менее, для того чтобы уберечь нас от слишком явного сопровождения конфигурации контейнера, мы можем рассмотреть в большей степени основанный на соглашениях подход, в котором используется автоматическая регистрация.

Автоматическая регистрация

Во многих случаях большинство регистраций похожи друг на друга. Такие регистрации трудно сопровождать, а явная регистрация каждого компонента часто сокращает производительность.

Давайте рассмотрим библиотеку, которая содержит множество реализаций `IIngredient`. Вы можете регистрировать каждый класс индивидуально, но это приведет к многочисленным, схожим вызовам метода `Register`. Что еще хуже, каждый раз при добавлении новой реализации `IIngredient` вы должны также явно регистрировать эту реализацию с помощью контейнера, если хотите, чтобы эта реализация была доступной. Было бы более продуктивно установить, что все реализации `IIngredient`, обнаруженные в данной сборке, должны быть зарегистрированы.

Сделать это можно путем применения статического класса `AllTypes`, который играет роль, аналогичную той роли, которую играет класс `Component`. Мы можем использовать один из методов класса `AllTypes` для того, чтобы просмотреть сборку на факт наличия типов, удовлетворяющих определенному критерию. Приведенный ниже код регистрирует все реализации `IIngredient` за один шаг:

```
container.Register(AllTypes
    .FromAssemblyContaining<Steak>()
    .BasedOn<IIngredient>());
```

Класс `AllTypes` предоставляет множество методов, которые дают нам возможность обратиться к конкретной сборке, но я считаю генерический метод `FromAssemblyContaining` наиболее кратким: предоставь ему тип представителя в качестве параметра типа, и он будет использовать сборку, содержащую данный тип. Существуют также и другие методы, предоставляющие возможность обеспечения `Assembly` с помощью других средств.

В предыдущем примере вы, безусловно, регистрируете все реализации интерфейса `IIngredient`, но можете задать другие критерии отбора, либо сузив отбор, либо выполняя отбор не на основании интерфейсов и базовых классов, а на основании других критериев. Ниже представлена основанная на соглашениях регистрация, при которой вы добавляете все классы, имя которых начинается с `Sauce`, и регистрируете их относительно всех интерфейсов, которые они реализуют:

```
container.Register(AllTypes
    .FromAssemblyContaining<SauceBéarnaise>()
    .Where(t => t.Name.StartsWith("Sauce"))
    .WithService.AllInterfaces());
```

Обратите внимание на то, что вы применяете предикат к методу `Where`, который выполняет фильтрацию по типу имени. Любой тип, имя которого начинается с `Sauce`, будет отобран из сборки, содержащей класс `SauceBéarnaise`. Свойство `WithService` позволяет задать правило регистрации типа. В данном примере вы регистрируете все типы относительно всех интерфейсов, которые они реализуют.

Благодаря регистрации на основании соглашений вы можете перейти от строгого контроля типов к той области, где безопасность типов, кажется, исчезает. Пример, подобный предыдущему, компилируется, но при этом не гарантируется, что будет регистрироваться вообще любой тип. Это зависит от того, существуют ли какие-либо типы, удовлетворяющие критерию отбора. Вы можете переименовать все `sauce`-классы во что-то еще и останетесь без `sauce`.

В классе `AllTypes` существует даже метод, который принимает в качестве входного параметра имя сборки. Он будет использовать *Fusion* (средство загрузки сборки .NET

Framework) для обнаружения соответствующей сборки. Сочетая сборку с поздним связыванием и нетипизированный предикат, можно продвинуться вглубь территории позднего связывания. Такая возможность могла бы стать полезным приемом реализации дополнений, поскольку Castle Windsor также может просматривать все сборки в директории.

Еще один способ регистрации дополнений и других сервисов с поздним связыванием – применение возможности XML конфигурации Castle Windsor.

XML конфигурация

Когда вам нужно уметь изменять конфигурацию без повторной компиляции приложения, XML конфигурация – наилучший вариант.

Подсказка

Используйте XML конфигурацию только для тех типов, которые вам нужно изменить без повторной компиляции приложения. Для остальных случаев используйте автоматическую регистрацию или технологию использования кода в качестве конфигурации.

Мы можем вставить XML конфигурацию в обычные файлы .NET конфигурации или импортировать XML из специализированных файлов. В Castle Windsor ничего не происходит, пока мы об этом явно не попросим, поэтому мы должны также указать, хотим ли загрузить конфигурацию из XML.

Сделать это можно несколькими способами, но рекомендуемый способ – использовать метод `Install` (подробнее об `Installer`'ах мы поговорим в разделе 10.1.3 "Пакетирование конфигурации"):

```
container.Install(Configuration.FromAppConfig());
```

Метод `FromAppConfig` возвращает экземпляр `ConfigurationInstaller`, который читает XML конфигурацию Castle Windsor из конфигурационного файла приложения и преобразует ее в объекты, понятные контейнеру.

Для того чтобы разрешить размещать конфигурацию Castle Windsor в конфигурационном файле, вы должны сначала добавить секцию конфигурации:

```
<configSections>
  <section name="castle"
    type="Castle.Windsor.Configuration.AppDomain
      ➡.CastleSectionHandler, Castle.Windsor" />
</configSections>
```

Данный код позволяет вам добавить секцию конфигурации `castle` в конфигурационный файл. Ниже приведен простой пример, который преобразует интерфейс `IIngredient` в класс `Steak`:

```
<castle>
  <components>
    <component id="ingredient.sauceBéarnaise"
      service="IIngredient"
      type="Steak"/>
  </components>
</castle>
```


Обратите внимание на то, что вам не приходится применять квалифицированное имя типа сборки для сервиса или для класса. Пока имена в рамках всех загруженных сборок будут уникальными, они будут корректно разрешаться – но если бы вам захотелось получить конфликт имен, вы могли бы все равно использовать квалифицированные имена типов.

Очевидно, вы можете добавить столько компонентов, сколько вам потребуется.

`ConfigurationInstaller` преобразует эту XML конфигурацию в объекты регистрации, которые конфигурируют контейнер, и вы можете последовательно разрешать сконфигурированные типы.

XML конфигурация – это хороший вариант в тех ситуациях, когда вам нужно изменить конфигурацию одного или более одного компонента без повторной компиляции приложения. Тем не менее, поскольку это может быть довольно хрупким вариантом, то для таких случаев вам следует сделать резервную копию и использовать либо автоматическую регистрацию, либо использовать код в качестве конфигурации для основной части конфигурации контейнера.

Подсказка

Помните, каким образом выигрывает первая конфигурация? Вы можете использовать это поведение для того, чтобы перезаписать жестко-закодированную конфигурацию XML конфигурацией. Для этого вы должны не забыть установить `ConfigurationInstaller` до того, как установите любые другие компоненты.

В данном разделе мы главным образом рассматривали различные API конфигурации `Castle Windsor`. Несмотря на то, что наверняка можно написать один большой блок неструктурированного кода конфигурации, лучше всего разделить конфигурацию на модули. Для этих целей у нас есть `Windsor Installer`'ы.

Пакетирование конфигурации

Иногда хочется упаковать логику конфигурации в повторно используемые группы, и даже когда повторное использование само по себе не является для нас самым приоритетным, мы можем пожелать обеспечить некоторого рода структуру в случае, если нам нужно сконфигурировать большое и сложное приложение.

Благодаря `Castle Windsor` мы можем упаковать конфигурацию в `Installer`'ы. `Installer` – это класс, который реализует интерфейс `IWindsorInstaller`:

```
public interface IWindsorInstaller
{
    void Install(IWindsorContainer container, IConfigurationStore store);
}
```

Все, что вы делали до настоящего момента, вы также можете сделать и внутри `Installer`. Следующий листинг демонстрирует `Installer`, который регистрирует все реализации `IIngredient`.

Листинг 10-1: Реализация Windsor Installer

```
public class IngredientInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container,
        IConfigurationStore store)
    {
        container.Register(AllTypes
            .FromAssemblyContaining<Steak>()
            .BasedOn<IIngredient>());
    }
}
```

`IngredientInstaller` реализует интерфейс `IWindsorInstaller` посредством использования точно такого же API, которое вы видели ранее, для регистрации всех реализаций `IIngredient`.

Для того чтобы зарегистрировать `Installer`, вызовите метод `Install`:

```
container.Install(new IngredientInstaller());
```

Несмотря на то, что метод `Install` можно вызывать бесчисленное множество раз, в документации к `Castle Windsor` рекомендуется выполнять всю конфигурацию в единственном вызове метода `Install`. Метод `Install` принимает в качестве параметра массив экземпляров `IWindsorInstaller`:

```
public IWindsorContainer Install(params IWindsorInstaller[] installers);
```

Подсказка

`Windsor Installer`'ы позволяют вам упаковывать и структурировать код конфигурации вашего контейнера. Используйте их вместо встроенной конфигурации: это сделает вашу `Composition Root` более читабельной.

Подсказка

Кроме преимуществ, которые `Installer`'ы предлагают для вашего собственного кода, `Castle Windsor` также движется по направлению оптимизации большинства своих API вокруг `Installer`'ов. Это своеобразный и рекомендуемый способ конфигурирования в большей степени `Castle Windsor`, нежели других контейнеров.

Кроме того, вы можете задать один или более одного `Installer`'а в XML, и загрузить конфигурационный файл, как это было описано ранее:

```
<installers>
  <install type="IngredientInstaller" />
</installers>
```

Используя `Installer`'ы вы можете конфигурировать `WindsorContainer` таким способом, каким вам только захочется – используя код в качестве конфигурации, автоматическую регистрацию или XML – или же вы можете сочетать все три подхода. После того, как контейнер сконфигурирован, вы можете попросить его разрешить сервисы

Данный раздел познакомил вас с DI-контейнером `Castle Windsor` и продемонстрировал основные принципы: как сконфигурировать контейнер и впоследствии использовать его

для того, чтобы разрешать сервисы. Выполнять разрешение сервисов легко с помощью единичного вызова метода `Resolve`, поэтому вся сложность заключается в конфигурировании контейнера. Это можно сделать несколькими различными способами, включая императивный код и XML. До настоящего момента мы рассматривали только самое основное API. Есть и более перспективные области, которые нам необходимо рассмотреть. Одна из наиболее важных тем – как управлять жизненным циклом компонентов.

10.2. Управление жизненным циклом

В главе 8 мы обсуждали механизм управления жизненным циклом, в том числе самые универсальные, принципиальные стили существования, например, Singleton и Transient. Castle Windsor поддерживает множество различных стилей существования и позволяет вам сконфигурировать жизненный цикл всех сервисов. Стили существования, продемонстрированные в таблице 10-2, доступны в виде составляющей API Castle Windsor.

Таблица 10-2: Стили существования Castle Windsor

Название	Комментарии
Singleton	Этот стиль существования используется в Castle Windsor по умолчанию.
Transient	Каждый раз создается новый экземпляр, но экземпляр все равно отслеживается контейнером.
PerThread	На один поток создается один экземпляр.
PerWebRequest	Необходима регистрация в web.config (см. раздел 10.2.2 "Использование продвинутых стилей существования").
Pooled	Чаще всего будет уместно конфигурировать размер пула (см. раздел 10.2.2).
Custom	Создайте свой собственный пользовательский стиль существования (см. раздел 10.2.3 "Разработка пользовательского стиля существования").

Некоторые из встроенных стилей существования полностью эквивалентны основным паттернам стилей существования, описанным в главе 8. Это, в частности, справедливо и для стилей существования Singleton и Transient, поэтому в этой главе я не буду выделять для их описания какое-то специальное пространство.

Примечание

Стиль существования, используемый в Castle Windsor по умолчанию, – это Singleton. Это отличает его от многих других контейнеров. Как мы уже обсуждали в главе 8, Singleton – самый эффективный, но, тем не менее, не всегда самый безопасный из всех стилей существования. В Castle Windsor по умолчанию эффективность имеет приоритет над безопасностью.

В этом разделе вы увидите, как можно конфигурировать стили существования для компонентов и как использовать некоторые более специфичные стили существования, например, PerWebRequest и Pooled. Мы также рассмотрим реализацию пользовательского стиля существования для того, чтобы продемонстрировать, что мы не ограничены использованием только встроенных стилей существования. После прочтения данного раздела вы должны будете уметь использовать стили существования Castle Windsor в своем собственном приложении.

Давайте начнем с обзора того, как конфигурировать стили существования для компонентов.

Конфигурирование стиля существования

В данном разделе мы разберем то, как управлять стилями существования компонентов с помощью Castle Windsor. Стил ь существования конфигурируется в виде составляющей части регистрации компонента, поэтому нам доступны те же самые возможности, что и при выполнении конфигурации в целом: код или XML. Мы будем рассматривать каждую из этих возможностей поочередно.

Конфигурирование стиля существования с помощью кода

Стил ь существования конфигурируется с помощью Fluent Registration API, которое мы используем для регистрации компонентов. Это настолько просто, как и представленный ниже код:

```
container.Register(Component
    .For<SauceBéarnaise>()
    .LifeStyle.Transient);
```

Обратите внимание на то, что вы определяете стил ь существования при помощи свойства `LifeStyle`. В данном примере вы устанавливаете в качестве стили ь существования стил ь `Transient`. Таким образом, каждый раз, когда разрешается `SauceBéarnaise`, возвращается новый экземпляр.

Вы все равно можете явным образом определить стил ь существования `Singleton`, даже если он и является используемым по умолчанию. Два примера, приведенные ниже, являются эквивалентными:

```
container.Register(Component
    .For<SauceBéarnaise>()
    .LifeStyle.Singleton);
```

и

```
container.Register(Component
    .For<SauceBéarnaise>());
```

Поскольку `Singleton` является используемым по умолчанию стилем существования, вам не нужно явно определять его, но вы можете это сделать, если пожелаете.

Точно так же, как мы можем конфигурировать компоненты в коде и XML, мы можем конфигурировать стили ь существования и в коде, и в XML.

Конфигурирование стиля существования с помощью XML

В разделе 10.1.2 "Конфигурирование контейнера" вы видели, как конфигурировать компоненты с помощью XML, но не применяли никакого стили ь существования. Как и в случае с конфигурированием компонента при помощи Fluent Registration API, `Singleton` является используемым по умолчанию стилем существования, но при необходимости вы можете явно определить другой стил ь существования.

```
<component id="ingredient.sauceBéarnaise"
    service="IIIngredient"
    type="Steak"
    lifestyle="transient" />
```

Единственное отличие от примера из раздела 10.1.2 "Конфигурирование контейнера" заключается в добавленном атрибуте `lifestyle`. Как вы видите, определение стиля существования легко выполняется и с помощью кода, и при помощи XML.

Высвобождение компонентов

Как мы уже обсуждали в разделе 8.2.2 "Управление устраняемыми зависимостями", важно высвободить объекты после того, как мы завершили работу с ними. Это также просто, как и вызов метода `Release`:

```
container.Release(ingredient);
```

Данный код будет высвобождать экземпляр, предоставленный в метод `Release` (переменная `ingredient` из предыдущего примера), а также все те зависимости экземпляра, жизненный цикл которых завершился. То есть, если экземпляр обладает `Transient` зависимостью, то эта зависимость будет высвобождена (и возможно уничтожена), тогда как `Singleton` зависимость останется в контейнере.

Подсказка

Castle Windsor отслеживает все, даже `Transient` компоненты, поэтому важно, не забывать высвобождать все разрешенные экземпляры для того, чтобы избежать утечек памяти.

Подсказка

Высвобождайте явно то, что вы явно разрешаете.

Подсказка

Не забывайте уничтожать сам контейнер при закрытии приложения. Такое поведение приведет к уничтожению `Singleton` компонентов и даст гарантию того, что приложение правильно очищается.

Теперь давайте обратим наше внимание на некоторые из тех стилей существования, для которых нужно немного больше конфигурации, нежели простой оператор.

Использование продвинутых стилей существования

В данном разделе мы рассмотрим два стиля существования Castle Windsor, для которых необходимо больше конфигурации, нежели простое объявление: `Pooled` и `PerWebRequest`.

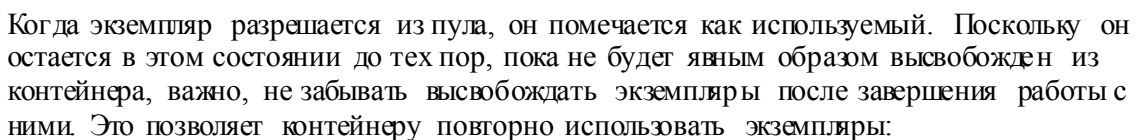
Использование стиля существования Pooled

В разделе 8.3.5 "Pooled" мы рассматривали общую концепцию стиля существования `Pooled`. В данном разделе мы увидим, как использовать реализацию Windsor. Стиль существования `Pooled` в Castle Windsor уже идет с заданным по умолчанию размером пула, но, поскольку оптимальный размер пула всецело зависит от обстоятельств, вам следует явно сконфигурировать размер пула. Вы можете определить стиль существования `Pooled` с заданными по умолчанию размерами таким же способом, как вы конфигурируете любой другой стиль существования.

Тем не менее, данный код не передает размер пула. Несмотря на то, что я не смог найти какую-либо *документацию*, в которой указано, каковы значения пула по умолчанию, исходный код Castle Windsor 2.1.1 показывает, что по умолчанию первоначальный размер пула равен 5, а максимальный размер – 15. Для меня эти значения кажутся довольно произвольными, что является еще одной причиной определения размера явным образом.

```
container.Register(Component
    .For<IIIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.PooledWithSize(10, 50));
```

Рисунок 10-4: Прогрессия размера пула с первоначальным размером, равным 3, и максимальным размером, равным 5. Даже если первоначальный размер равен 3, пул остается пустым до тех пор, пока не разрешается первый экземпляр. На данном этапе созданы все три экземпляра для первоначального размера, и один из них непосредственно используется. Когда экземпляр высвобождается, он возвращается в пул. Пул увеличивается в размере, если необходимо больше экземпляров, чем для первоначального размера. Обратите внимание на то, что можно превысить максимальный размер, но что излишние экземпляры не используются повторно при высвобождении.



383

Примечание

То, что происходит, когда пул полностью утилизируется, довольно неожиданно. Вместо того чтобы выдавать исключение или блокировать вызов, создаются избыточные экземпляры. После использования они отбрасываются вместо того, чтобы повторно использоваться.

Несмотря на то, что стиль существования Pooled более продвинут, чем Singleton или Transient, его все равно легко использовать. Единственное дополнительное усилие, которое вам необходимо сделать – передать два дополнительных числа для того, чтобы сконфигурировать размеры пула. Стиль существования PerWebRequest не сильно отличается от Pooled, но его несколько сложнее конфигурировать.

Использование стиля существования PerWebRequest

Как и подразумевает его имя, работа стиля существования PerWebRequest заключается в создании экземпляра для веб-запроса. Объявление этого стиля столь же просто, как и объявление стиля существования Transient:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.PerWebRequest);
```

Тем не менее, если мы попытаемся использовать его без дальнейшей конфигурации, то получим исключение:

Looks like you forgot to register the HTTP module Castle.MicroKernel.Lifestyle.PerWebRequestLifestyleModule

Add '<add name="PerRequestLifestyle" type="Castle.MicroKernel.Lifestyle.PerWebRequestLifestyleModule, Castle Windsor" />' to the <httpModules> section on your web.config. If you're running IIS7 in Integrated Mode you will need to add it to <modules> section under <system.webServer>

(Похоже, вы забыли зарегистрировать HTTP-модуль Castle.MicroKernel.Lifestyle.PerWebRequestLifestyleModule

Добавьте '<add name="PerRequestLifestyle" type="Castle.MicroKernel.Lifestyle.PerWebRequestLifestyleModule, Castle Windsor" />' в раздел <httpModules> вашего файла web.config. Если IIS7 запущен в интегрированном режиме, вам нужно будет добавить этот код в раздел <modules> после <system.webServer>)

Поскольку выдаются сообщения об ошибках, приведенное выше исключение является показательным. Оно точно указывает, что вам необходимо сделать.

В любом случае, важно отметить, что стиль существования PerWebRequest использует HTTP-модуль для того, чтобы указать, какой веб-запрос на данный момент выполняется. Одно из последствий этого – нам необходимо зарегистрировать HTTP-модуль так, как я это описывал выше, другое – данный конкретный стиль существования работает *только* в

рамках веб-запросов. Если мы попытаемся использовать его в других типах приложений, мы получим то же самое исключение, что и в предыдущем примере.

Как и стиль существования Pooled, так и стиль существования PerWebRequest, требуют использования немного большего разнообразия задач, нежели простое объявление, но при этом их все равно легко конфигурировать и использовать. Встроенные стили существования Castle Windsor предоставляют обширный и полезный набор стилей существования, который удовлетворяет большинству сценариев. Однако если ни один из этих стилей существования не удовлетворяет специализированным нуждам, мы можем создать пользовательский стиль существования.

Разработка пользовательского стиля существования

В большинстве случаев для наших нужд должны подходить встроенные стили существования Castle Windsor, но если нам нужно что-то особенное, мы можем создать пользовательский стиль существования.

В данном разделе вы увидите, как это сделать. Для начала мы вкратце рассмотрим соответствующий шов, который делает возможным создание пользовательского стиля существования, но быстро перейдем к рассмотрению примера.

Понимание API стиля существования

Вы можете создать пользовательский стиль существования путем реализации интерфейса `ILifestyleManager`:

```
public interface ILifestyleManager : IDisposable
{
    void Init(IComponentActivator componentActivator,
             IKernel kernel, ComponentModel model);
    bool Release(object instance);
    object Resolve(CreationContext context);
}
```

Одним из немного странных условий реализации `ILifestyleManager` является тот факт, что он должен иметь конструктор по умолчанию. То есть использовать Constructor Injection при реализации `ILifestyleManager` запрещено. Вместо этого нам предоставляется одна из относительно редких возможностей использовать Method Injection. Будет вызван метод `Init`, предоставляя среди прочих параметров экземпляр `IKernel`, который можно использовать в качестве Service Locator. Это мне совершенно не подходит, и когда мы будем рассматривать некоторые примеры кода, вы увидите, что такой подход в большей степени усложняет реализацию, чем, если бы было возможным использование Constructor Injection.

Другие методы интерфейса `ILifestyleManager` — `Resolve` и `Release`, но нам следует использовать их в качестве перехватчиков, а не предоставлять свои собственные реализации `Resolve` и `Release` — за это отвечает `IComponentActivator`, переданный нам в методе `Init`. Рисунок 10-5 демонстрирует, что для обеспечения возможности управления жизненным циклом каждого компонента мы должны использовать эти методы только для перехвата вызовов `Resolve` и `Release`.

Рисунок 10-5: `ILifestyleManager` выступает в роли некоторого рода перехватчика, который вызывается вместо упомянутого выше `IComponentActivator`. Предполагается, что реализация `ILifestyleManager` использует предоставленный `IComponentActivator` для создания экземпляров объектов. Поскольку `ILifestyleManager` располагается посередине, он получает возможность перехватывать каждый вызов и исполнять свою собственную логику стиля существования. Можно повторно использовать экземпляры вместо того, чтобы каждый раз вызывать `IComponentActivator`.



Castle Windsor обеспечивает реализацию по умолчанию `ILifestyleManager` в виде класса `AbstractLifestyleManager`. Он реализует интерфейс и предоставляет обоснованную реализацию по умолчанию для большинства методов. Именно этот класс вы будете использовать для реализации шаблонного стиля существования.

Разработка стиля существования Caching

Поскольку Castle Windsor предлагает исчерпывающий набор стандартных стилей существования, трудно придумать хороший пример. Тем не менее, представьте себе, что вам захотелось разработать стиль существования `Caching`, который держит экземпляр под рукой в течение определенного времени, а затем высвобождает его. Это хороший пример, поскольку он достаточно сложен для демонстрации различных аспектов реализации пользовательского стиля существования, но не так сложен, чтобы на его рассмотрение выделять несколько страниц.

Примечание

Стиль существования `Caching` – выдуманный пример. Существуют лучшие способы реализации функциональности кэширования, поскольку обычно вы хотите кэшировать не сервисы, а данные, которыми управляют эти сервисы.

Предупреждение

Продemonстрированный в данном разделе шаблонный код не принимает во внимание потоко-безопасность, но реальная реализация `ILifestyleManager` должна быть потоко-безопасной.

Самый легкий способ реализации пользовательского стиля существования – наследование от `AbstractLifestyleManager`, что продемонстрировано в следующем листинге.

Листинг 10-2: Определение пользовательского стиля существования

```
1. public partial class CacheLifestyleManager :
2.     AbstractLifestyleManager
3. {
4.     private ILease lease;
5.     public ILease Lease
6.     {
7.         get
8.         {
9.             if (this.lease == null)
10.            {
11.                this.lease = this.ResolveLease();
12.            }
13.            return this.lease;
14.        }
15.    }
16.    private ILease ResolveLease()
17.    {
18.        var defaultLease = new SlidingLease(TimeSpan.FromMinutes(1));
19.        if (this.Kernel == null)
20.        {
21.            return defaultLease;
22.        }
23.        if (this.Kernel.HasComponent(typeof(ILease)))
24.        {
25.            return this.Kernel.Resolve<ILease>();
26.        }
27.        return defaultLease;
28.    }
29. }
```

Строка 2: Наследование от `AbstractLifestyleManager`

Строка 9-13: "Ленивая" загрузка

Строка 23-26: Пыгается определить местоположение `ILease`

Одна из тех вещей, которые вы получаете благодаря наследованию от `AbstractLifestyleManager`, – реализация метода `Init`. Вы можете переопределить его, но в данном случае это не нужно. Все, что он делает, – сохраняет внедренные сервисы таким образом, чтобы вы могли в дальнейшем получить доступ к ним посредством защищенных свойств.

Для того чтобы реализовать функциональность, которая решает, когда заканчивается время кэшированного объекта, вам нужен экземпляр `ILease`. Если бы вы могли использовать `Constructor Injection`, то вы бы запросили `ILease` с помощью конструктора, состоящего из трех строк кода (включая граничный оператор). На данный момент вам необходимо 12 строк кода, поскольку вам приходится иметь дело со множеством потенциальных состояний `CacheLifestyleManager`: вызывался ли уже метод `Init`? Обладает ли `Kernel` экземпляром `ILease`?

Вы справляетесь с этим при помощи свойства `Lease`, имеющего отложенную загрузку (*lazy-loaded property*). При первом его прочтении он вызывает метод `ResolveLease`, который выясняет, каким должен быть срок аренды. Он использует срок аренды, заданный по умолчанию, но пытается искать альтернативный срок посредством `Kernel` – если `Kernel` вообще существует. Я думаю, что это довольно хорошая иллюстрация

недостатков Method Injection. Обратите внимание на то, что, если кто-либо прочитает свойство `Lease` до вызова метода `Init`, то будет использоваться срок аренды по умолчанию даже в случае, если `Kernel` содержит компонент `ILease`. Тем не менее, поскольку `Castle Windsor` ничего не знает о свойстве `Lease`, при обычном его применении этого не происходит.

Примечание

Интерфейс `ILease`, используемый в данном примере, является пользовательским интерфейсом, определенным для конкретной цели. Это не `System.Runtime.Remoting.Lifetime.ILease`, который имеет аналогичное, но не намного более сложное API.

По сравнению со всеми трудностями, которые вам приходится преодолеть для того, чтобы внедрить зависимость в ваш пользовательский стиль существования, реализация метода `Resolve` намного проще. Увидеть это вы можете из приведенного ниже листинга.

Листинг 10-3: Реализация метода `Resolve`

```
1. private object obj;
2. public override object Resolve(CreationContext context)
3. {
4.     if (this.Lease.IsExpired)
5.     {
6.         base.Release(this.obj);
7.         this.obj = null;
8.     }
9.     if (this.obj == null)
10.    {
11.        this.Lease.Renew();
12.        this.obj = base.Resolve(context);
13.    }
14.    return this.obj;
15. }
```

Строка 4-8: Выносит объект на просрочку

Строка 11: Продляет срок аренды

Каждый раз, когда `CacheLifestyleManager` просят разрешить компонент, он начинает с проверки того, просрочен ли текущий срок аренды. Если срок аренды просрочен, `CacheLifestyleManager` высвобождает текущий кэшированный экземпляр и обнуляет его. Метод `Release` явным образом вызывается для базового класса и через него – для `IComponentActivator`, что продемонстрировано на рисунке 10-5. Это важно выполнить, потому что это дает вышеупомянутой реализации возможность уничтожить экземпляр, если он реализует `IDisposable`.

Следующее, что необходимо сделать, – проверить, имеет ли кэшированный экземпляр значение `null`. Он может иметь `null`, если он был высвобожден перед этим, но также может иметь `null` при первом вызове метода `Resolve`. В обоих случаях вы продлите срок аренды и просите базовую реализацию разрешить компонент для вас. Именно здесь базовый класс вызывает соответствующий метод для `IComponentActivator`.

В данном пользовательском стиле существования вы переопределяете метод `Release` так, чтобы он ничего не делал:

```
public override bool Release(object instance)
{
    return false;
}
```

Это может показаться странным, но является вполне нормальным. Вы должны принять во внимание тот факт, что метод `Release` — это *Hook*-метод, который является частью *Seam* стиля существования *Castle Windsor*. Вы проинформированы о том, что компонент может быть высвобожден, но это не означает, что вам придется это делать. Для примера стиль существования *Singleton* по определению никогда не высвобождает свой экземпляр, поэтому он имеет такую же реализацию метода `Release`, как и продемонстрированная ранее.

В случае с `CacheLifestyleManager` вы время от времени высвобождаете кэшированный экземпляр, но, как показано в листинге 10-3, вы делаете это в рамках метода `Resolve` по необходимости.

`CacheLifestyleManager` кэширует разрешенный экземпляр до тех пор, пока не истечет срок аренды, а затем разрешает новый экземпляр и продлевает срок аренды. Существует несколько способов реализации логики аренды, но мы рассмотрим только один из них.

Реализация `Lease`

Вам нужна, по крайней мере, одна реализация `ILease` для `CacheLifestyleManager`. Срок класса `SlidingLease` истекает по окончании фиксированного промежутка времени, но вы могли бы создать другие реализации, срок которых истекает фиксированное количество раз в день, или после того, как они разрешили компонент заданное количество раз.

Примечание

Интерфейс `ILease` и класс `SlidingLease`, продемонстрированные в данном разделе, ничего не делают в рамках *Castle Windsor*, но я хотел продемонстрировать их ради полноты знаний. Вы можете пропустить этот раздел и приступить к чтению информации о том, как зарегистрировать пользовательский жизненный цикл, если вас не интересует `SlidingLease`.

Следующий листинг демонстрирует реализацию `SlidingLease`.

Листинг 10-4: Реализация `SlidingLease`

```
1. public class SlidingLease : ILease
2. {
3.     private readonly TimeSpan timeout;
4.     private DateTime renewed;
5.     public SlidingLease(TimeSpan timeout)
6.     {
7.         this.timeout = timeout;
8.         this.renewed = DateTime.Now;
9.     }
10.    public TimeSpan Timeout
11.    {
12.        get { return this.timeout; }
13.    }
14.    public bool IsExpired
15.    {
16.        get
```

```

17.     {
18.         return DateTime.Now >
19.             this.renewed + this.timeout;
20.     }
21. }
22. public void Renew()
23. {
24.     this.renewed = DateTime.Now;
25. }
26. }
27.

```

Строка 16-17: Выносит на просрочку в случае истечения времени ожидания

Строка 22: Продление срока

Класс `SlidingLease` реализует `ILease` путем отслеживания того, когда был продлен срок аренды. Всякий раз, когда вы запрашиваете у `SlidingLease` информацию о том, был ли просрочен срок аренды, он сравнивает текущее время со временем продления и ожидания. При продлении срока аренды `SlidingLease` устанавливает текущее время в качестве времени продления. Я мог бы использовать `TimeProvider Ambient Context` из раздела 4.4.4 вместо `DateTime.Now`, но решил упростить все, насколько это возможно.

Теперь, когда вы знаете, как реализовать пользовательский стиль существования и любые пользовательские зависимости, которые он может иметь, вам осталось узнать только то, как их использовать.

Конфигурирование компонентов с помощью пользовательского стиля существования

Применять `CacheLifestyleManager` к компоненту легко, и выполняется это таким же образом, как и определение любого другого стиля существования:

```

container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.Custom<CacheLifestyleManager>()));

```

Вы применяете генерис-перегрузку метода `Custom` для того, чтобы определить, какой тип `ILifestyleManager` использовать. Но существует также перегрузка, которая принимает экземпляр `Type` в качестве входного параметра.

До тех пор, пока вы не будете забывать регистрировать также и `ILease`, `CacheLifestyleManager` будет использовать заданный по умолчанию срок аренды `SlidingLease` с одноминутным таймаутом. Ниже приведен способ регистрации пользовательского `ILease`:

```

container.Register(Component
    .For<ILease>()
    .Instance(new SlidingLease(TimeSpan.FromHours(1))));

```

При помощи данного кода выполняется регистрация экземпляра `SlidingLease` с временным интервалом в один час. Вы должны не забывать регистрировать `ILease` перед использованием пользовательского стиля существования, поскольку иначе будет использоваться срок аренды по умолчанию.

Разработка пользовательского стиля существования для Castle Windsor не особенно сложна. В большинстве случаев класс `AbstractLifestyleManager` предоставляет хорошую стартовую точку, и нам нужно только переопределить методы, которые нам особенно важны. Чаще всего это будет метод `Resolve`, хотя для некоторых других методов мы можем оставить их реализации по умолчанию. Только в редких случаях нам нужно будет создавать пользовательский стиль существования, поскольку стандартный набор стилей существования Castle Windsor довольно исчерпывающий.

На этом мы заканчиваем свое путешествие по управлению жизненным циклом с помощью Castle Windsor. Компоненты можно конфигурировать при помощи сочетания стилей существования, и это справедливо даже в тех случаях, когда мы регистрируем составные реализации одной и той же абстракции. Нам еще придется рассмотреть то, как работать с составными компонентами, поэтому давайте сейчас перенаправим наше внимание в это направление.

10.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями
- Используются Decorator'ы

В данном разделе мы рассмотрим каждую из этих ситуаций и увидим, как Castle Windsor поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь регистрировать и разрешать компоненты даже тогда, когда в дело вступают составные реализации одной и той же абстракции.

Давайте сначала рассмотрим то, как можно обеспечить более разветвленное управление, нежели то, которое предоставляет *автоматическая интеграция* (Auto-Wiring).

Выбор из составных кандидатов

Автоматическая интеграция – удобный и мощный инструмент, но предоставляет нам меньшие возможности контроля. Пока все абстракции преобразуются в конкретные типы отдельно друг от друга, никаких трудностей не возникает, но как только мы вводим больше реализаций для одного и того же интерфейса, возникает неопределенность.

Давайте для начала повторим, как Castle Windsor работает с составными регистрациями одной и той же абстракции.

Регистрация составных реализаций одного и того же сервиса

Как вы видели в разделе 10.1.2 "Конфигурирование контейнера", вы можете регистрировать составные компоненты одного и того же сервиса:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

Данный пример регистрирует как класс `Steak`, так и класс `SauceBéarnaise` для сервиса `IIngredient`. Выигрывает первая регистрация, поэтому, если вы будете разрешать `IIngredient` при помощи `container.Resolve<IIngredient>()`, вы получите экземпляр `Steak`. Тем не менее, в результате вызова `container.ResolveAll<IIngredient>()`

возвращается массив `IIngredient`, содержащий как `Steak`, так и `SauceBéarnaise`. То есть, последующие регистрации не позабыты, но их трудно получить.

Подсказка

Побеждает первая регистрация данного типа. Она определяет регистрацию, используемую для него по умолчанию.

Предупреждение

Если существуют регистрации данного типа, которые не могут быть разрешены в связи с отсутствием зависимостей, `ResolveAll` молча игнорирует их и возвращает только те, которые может разрешить. Поскольку не выдается никаких исключений, все это иногда может приводить к некоторым труднопонимаемым багам.

Следующий листинг демонстрирует способ, благодаря которому вы можете предоставлять подсказки, которые могут впоследствии использоваться для осуществления выбора из составных кандидатов.

Листинг 10-5: Присваивание имен компонентам

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>()
    .Named("meat"));
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .Named("sauce"));
```

Каждой регистрации вы можете присвоить уникальное имя, которое позднее может использоваться для того, чтобы отличить этот компонент от других компонентов.

Имена представленные в листинге 10-5 именованные компоненты, вы можете разрешить и `Steak`, и `SauceBéarnaise` следующим образом:

```
var meat = container.Resolve<IIngredient>("meat");
var sauce = container.Resolve<IIngredient>("sauce");
```

Заметьте, что вы применяете тот же самый идентификатор, который использовали для присваивания имени компоненту во время регистрации.

Если вы примете как должное то, что вам следует всегда разрешать сервисы в единственном `Composition Root`, вы уже не будете сталкиваться с такой неопределенностью на данном уровне.

Подсказка

Если вы обнаружили, что вызываете метод `Resolve` со специальным идентификатором, подумайте, можете ли вы сделать свою технологию менее неопределенной.

Тем не менее, вы можете использовать именованные компоненты для того, чтобы выбрать между составными альтернативами при конфигурировании зависимостей данного сервиса.

Регистрация именованных зависимостей

Насколько бы полезной ни была автоматическая интеграция, иногда вам необходимо переопределять обычное поведение таким образом, чтобы обеспечить более разветвленное управление зависимостями. Но может случиться и то, что вам нужно будет обратиться к неопределенному API. В качестве примера рассмотрите приведенный ниже конструктор:

```
public ThreeCourseMeal(ICourse entrée,  
    ICourse mainCourse, ICourse dessert)
```

В данном примере у вас имеется три одинаково типизированных зависимости, каждая из которых представляет отличную от других сущность. В большинстве случаев вы хотите преобразовать каждую зависимость к отдельному типу. Следующий листинг демонстрирует, каким образом вы могли бы зарегистрировать преобразования `ICourse`.

Листинг 10-6: Регистрация именованных `course`'ов

```
container.Register(Component  
    .For<ICourse>()  
    .ImplementedBy<Rillettes>()  
    .Named("entrée"));  
container.Register(Component  
    .For<ICourse>()  
    .ImplementedBy<CordonBleu>()  
    .Named("mainCourse"));  
container.Register(Component  
    .For<ICourse>()  
    .ImplementedBy<MousseAuChocolat>()  
    .Named("dessert"));
```

Согласно листингу 10-6 вы регистрируете три именованных компонента, преобразуя `Rillettes` в регистрацию с именем `"entrée"` (напоминаю американским читателям, что это то же самое, что и *starter* или *appetizer* – то, что возбуждает аппетит, придает вкус), `CordonBleu` – в регистрацию с именем `"mainCourse"`, а `MousseAuChocolat` – в регистрацию с именем `"dessert"`.

Имея такую конфигурацию, вы теперь можете зарегистрировать класс `ThreeCourseMeal`, как это продемонстрировано в следующем листинге.

Листинг 10-7: Переопределение автоматической интеграции

```
container.Register(Component  
    .For<IMeal>()  
    .ImplementedBy<ThreeCourseMeal>()  
    .ServiceOverrides(new  
    {  
        entrée = "entrée",  
        mainCourse = "mainCourse",  
        dessert = "dessert"  
    }));
```

Вы можете явным образом обеспечить переопределения для тех параметров (или свойств), к которым хотите явно обращаться. В случае класса `ThreeCourseMeal` вам нужно обращаться ко всем трем параметрам конструктора. Тем не менее, в других случаях вы можете захотеть переопределить только один из нескольких параметров; это тоже возможно. Метод `ServiceOverrides` позволяет вам применять анонимный

типизированный объект, который указывает, какие параметры необходимо переопределить. Если вы не хотите использовать безымянные типы, то другие перегрузки методов `ServiceOverrides` дают нам возможность использовать массив специализированных экземпляров `ServiceOverride` или `IDictionary`.

Используя анонимный тип, вы сравниваете параметры, которые хотите переопределить, с именованной регистрацией. В первом случае вы сравниваете имя параметра `entrée` с регистрацией, имеющей имя "entrée". В данном случае имена параметров идентичны именам регистраций, но это не обязательно. Остальные параметры преобразуются похожим способом.

Предупреждение

Несмотря на то, что анонимный тип может выглядеть строго типизированным, если он используется указанным образом, то представляет собой еще одну группу "волшебных строк". В конце концов, он конвертируется в словарь имен и значений. Имена свойств анонимный типа должны совпадать с именами параметров соответствующего конструктора. Если в конструкторе вы измените имя параметра, то `ServiceOverride` перестанет функционировать до того момента, пока вы не исправите имя параметра, поэтому не полагайтесь на эту возможность чаще, чем это требуется.

Поскольку метод `ServiceOverrides` зависит от сравнения имен параметров и сконфигурированных переопределений по текстовому принципу, лучше слишком сильно на него не полагаться. Если вы чувствуете необходимость использовать этот метод только для того, чтобы справиться с неопределенностью, то для избавления от этой неопределенности лучше всего сконструировать API. Это чаще всего приводит к более полноценному дизайну.

В следующем разделе вы увидите, как выполнить рефакторинг текущего класса `ThreeCourseMeal` для получения более общей реализации и в то же время для избавления от присущей неопределенности. Сделать это можно, разрешив в обеде произвольное количество блюд, — но это принуждает вас к осознанию того, как `Castle Windsor` подключает списки и последовательности.

Интеграция последовательностей

В разделе 6.4.1 мы обсуждали, как `Constructor Injection` выступает в роли системы оповещения при нарушениях принципа единственной ответственности. Урок, извлеченный из того раздела, — вместо того, чтобы рассматривать *constructor over-injection* как слабую сторону паттерна `Constructor Injection`, мы должны скорее порадоваться тому, что *constructor over-injection* делает сомнительную композицию столь очевидной.

Когда дело доходит до DI-контейнеров и неопределенности, мы видим аналогичную взаимосвязь. DI-контейнеры, в общем случае, не справляются с неопределенностью столь грациозно. Несмотря на то, что мы можем заставить DI-контейнер, например, `Castle Windsor`, справляться с неопределенностью, он чаще всего делает это с трудом. Чаще всего это говорит о том, что мы могли бы улучшить конструкцию нашего собственного кода.

Подсказка

Если конфигурировать определенную составляющую вашего API в Castle Windsor сложно, подумайте, можете ли вы сделать ваше API более явным и четким. Это не только облегчит его конфигурацию в рамках Castle Windsor, но также, вероятнее всего, улучшит общую конструкцию вашего кода.

Вместо того чтобы чувствовать себя стесненными Castle Windsor, мы должны принять его соглашения и позволить ему провести нас к улучшенной и более последовательной конструкции. В данном разделе мы рассмотрим пример, который продемонстрирует, как можно выполнить рефакторинг таким образом, чтобы избавиться от неопределенности, а также покажем, как Castle Windsor работает с последовательностями, массивами и списками.

Рефакторинг с целью получения лучшего course

В разделе 10.3.1 "Выбор из составных кандидатов" вы увидели, как ThreeCourseMeal и присущая ему неопределенность вынудили нас отказаться от автоматической интеграции и вместо нее использовать явный ServiceOverride. Это должно было дать вам намек на пересмотр конструкции API.

Простое обобщение приводит нас к реализации IMeal, которая принимает в качестве параметров произвольное количество экземпляров ICourse вместо явных трех, как было в случае с классом ThreeCourseMeal:

```
public Meal(IEnumerable<ICourse> courses)
```

Обратите внимание на то, что вместо требования о наличии в конструкторе трех отдельных экземпляров ICourse, единичная зависимость экземпляра IEnumerable<ICourse> позволяет вам обеспечить любое количество блюд в классе Meal – от 0 до ... бесконечности! Это решает проблему неопределенности, поскольку теперь существует только одна зависимость. Кроме того, это улучшает API и реализацию, обеспечивая единственный универсальный класс, который может моделировать множество различных видов обедов, от простого обеда с единственным блюдом до сложного, состоящего из 12 блюд.

Имея регистрацию course'ов, продемонстрированную в листинге 10-6, вы сможете автоматически разрешать IMeal, если зарегистрируете его следующим образом:

```
container.Register(Component
    .For<IMeal>()
    .ImplementedBy<Meal>());
```

Тем не менее, когда вы пытаетесь разрешить IMeal, контейнер выдает исключение. Несмотря на то, что исключение совсем ни о чем нам не говорит, причина в том, что вы не сообщили контейнеру, как он должен разрешать IEnumerable<ICourse>. Давайте сделаем обзор некоторых различных доступных вариантов.

Конфигурирование массивов

Castle Windsor отлично понимает массивы. Поскольку массивы реализуют IEnumerable<T>, вы можете явным образом сконфигурировать массив для параметра конструктора courses. Это можно сделать способом, похожим на синтаксис, который вы

видели в листинге 10-7. В следующем листинге вы увидите те же самые `courses`, определенные в виде сервисов.

Листинг 10-8: Явное определение массива сервисов

```
1. container.Register(Component
2.     .For<IMeal>()
3.     .ImplementedBy<Meal>()
4.     .ServiceOverrides(new
5.     {
6.         courses = new[]
7.         {
8.             "entrée",
9.             "mainCourse",
10.            "dessert"
11.        }
12.    }));
```

Строка 6-10: Переопределение параметра `courses`

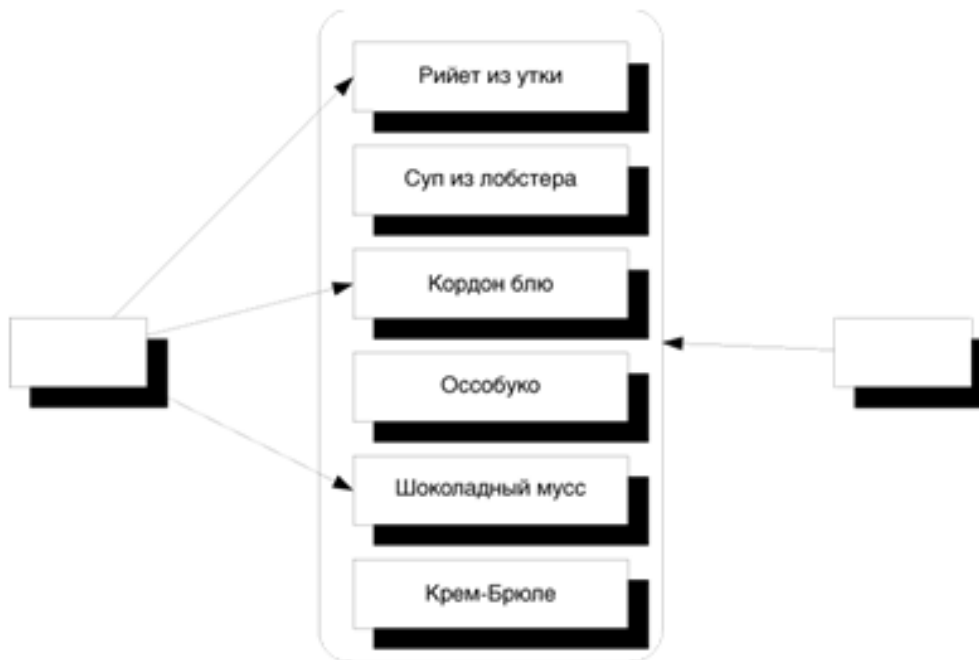
Аналогично листингу 10-7 вы используете метод `ServiceOverrides`, когда хотите переопределить автоматическую интеграцию для конкретных параметров. В данном примере вы хотите явным образом сконфигурировать параметр конструктора `courses` для класса `Meal`. Поскольку этот параметр является `IEnumerable<ICourse>`, вы должны теперь задать последовательность сервисов `ICourse`.

Поскольку массивы реализуют `IEnumerable<T>`, вы можете определить массив именованных сервисов. Осуществляете вы это путем создания массива имен сервисов. Эти имена идентичны именам, присвоенным каждой регистрации в листинге 10-6, а `Castle Windsor` настолько добр, что преобразует этот массив имен сервисов в массив экземпляров `ICourse` в рабочей среде. Все это аналогично листингу 10-7 за одним лишь исключением — `fluent registration API` по своей природе понимает и преобразует массивы имен сервисов в массивы сервисов.

Несмотря на то, что рефакторинг от `ThreeCourseMeal` к `Meal` казался шагом в правильном направлении, кажется, вы ничего не сделали касательно громоздкости конфигурации. Можно ли сделать это лучшим способом?

Наверняка упростить конфигурацию можно, но это приведет к меньшему контролю. Как иллюстрирует рисунок 10-6, иногда нам нужно уметь делать выбор из списка всех сконфигурированных сервисов данного типа, но в других ситуациях нам нужны они все.

Рисунок 10-6: Существует несколько способов работы с исчисляемыми зависимостями. В ситуации, продемонстрированной справа, необходимо разрешить все сервисы, сконфигурированные в контейнере. В ситуации слева – только некоторую подгруппу.



Пример, который вы видели, соответствует ситуации слева, при которой мы тщательно подбираем явный список именованных сервисов из концептуально громадного списка всех сконфигурированных сервисов данного типа. В остальных случаях мы, возможно, предпочтем более простое соглашение, согласно которому мы используем все доступные сервисы запрашиваемого типа. Давайте посмотрим, как это может выполняться.

Разрешение последовательностей

Castle Windsor по умолчанию не разрешает массивы или `IEnumerable<T>`. Это может показаться довольно неожиданным, поскольку в результате вызова `ResolveAll` возвращается массив:

```
IIngredient[] ingredients = container.ResolveAll<IIngredient>();
```

Тем не менее, если вы попытаетесь позволить контейнеру разрешить компонент, который зависит от массива сервисов, вы получите исключение. Подходящий способ решения данной проблемы – зарегистрировать встроенный `CollectionResolver` с помощью контейнера, как это показано ниже:

```
container.Kernel.Resolver.AddSubResolver(
    new CollectionResolver(container.Kernel));
```

`CollectionResolver` даст контейнеру возможность разрешать такие последовательности зависимостей, как `IEnumerable<T>`. Благодаря этому теперь вы можете разрешать класс `Meal`, не используя явный `ServiceOverrides`. Имея данную регистрацию

```
container.Register(Component
    .For<IMeal>()
    .ImplementedBy<Meal>());
```

вы можете разрешать `IMeal` с помощью `CollectionResolver`:

```
var meal = container.Resolve<IMeal>();
```

Будет создан экземпляр `Meal` со всеми сервисами `ICourse` контейнера.

Возможно, потребители, которые полагаются на списки зависимостей, являются самым понятным случаем использования составных регистраций одной и той же абстракции, но перед тем, как окончательно завершить обсуждение этой темы, необходимо рассмотреть последний и, вероятно, слегка неожиданный случай: когда начинают действовать составные регистрации.

Интеграция Decorator'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования `Decorator` полезен при реализации сквозных сущностей. По определению `Decorator`'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам `Decorator` и обернутый тип. Если бы мы помещали `Decorator`'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных регистраций одного и того же сервиса. В отличие от предыдущих разделов эти регистрации не являются концептуально равносильными, а зависят друг от друга. В данном разделе вы увидите, как сконфигурировать `Castle Windsor` таким образом, чтобы он мог работать с этим паттерном.

Явное подключение Decorator'ов

`Castle Windsor` требует от нас регистрации всех компонентов, которые мы собираемся использовать. Когда дело касается `Decorator`'ов, мы должны регистрировать и `Decorator`, и обернутые типы. Поскольку оба типа реализуют один и тот же интерфейс, мы тем самым, вводим неопределенность, которую можем побороть. Как показывает следующий листинг, мы можем сделать это явным образом.

Листинг 10-9: Явная конфигурация `Decorator`'а

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Breeding>()
    .ServiceOverrides(new
    {
        ingredient = "cutlet"
    }));
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<VealCutlet>()
    .Named("cutlet"));
```

Как мы уже обсуждали во введении к главе 9, панировку можно рассматривать в качестве обертки для говяжьей отбивной – регистрация `Cotoletta`. При разрешении `Cotoletta` вам нужна ссылка на `breeding`, которая должна содержать в себе `veal cutlet`. Сначала вы регистрируете `Breeding`. Вспомните, что в `Castle Windsor` всегда выигрывает первая регистрация. Вы явным образом используете метод `ServiceOverrides` для того, чтобы сконфигурировать, какой именованный сервис должен использоваться для параметра конструктора `ingredient`. Обратите внимание на то, что вы ссылаетесь на компонент под

названием `cutlet`, несмотря на то, что данный компонент на данном этапе еще не был зарегистрирован. Такое возможно, поскольку порядок регистрации мало что значит. Можно регистрировать компоненты до того, как вы зарегистрируете их зависимости, и при этом все будет работать, поскольку при попытке разрешения сервисов все регистрируется должным образом.

Это означает, что перед разрешением `IIngredient` вы все равно должны зарегистрировать `veal cutlet`. Впоследствии вы регистрируете ее под именем `cutlet`. Это имя совпадает с именем сервиса, переданного в параметр конструктора `ingredient` в предыдущей конфигурации.

Несмотря на то, что такая явная конфигурация Decorator'ов возможна и иногда даже необходима, Castle Windsor по своей природе понимает паттерн Decorator и предоставляет более неявный способ выполнения тех же самых действий.

Неявное подключение Decorator'ов

Castle Windsor дает нам возможность конфигурировать Decorator'ы неявным образом, регистрируя их в подходящем порядке. Вспомните, что выигрывает первая регистрация, в том смысле, что это тип, который возвращается в результате вызова метода `Resolve`. Мы должны сначала зарегистрировать наиболее удаленный Decorator.

По определению Decorator имеет зависимость от другого экземпляра одного и того же типа. Если мы явным образом не определим, какую регистрацию использовать, мы можем ожидать появления циклических ссылок. Тем не менее, Castle Windsor умнее всего этого. Он выбирает следующую регистрацию соответствующего типа. Это означает, что вместо листинга 10-9 вы можете записать

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Breeding>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<VealCutlet>());
```

Нет необходимости явным образом присваивать компонентам имена или использовать метод `ServiceOverrides` для конфигурирования зависимостей. При разрешении `IIngredient` Castle Windsor будет автоматически интегрировать класс `Breeding` со следующим доступным сервисом `IIngredient`, то есть классом `VealCutlet`.

Примечание

Следующий логический шаг вперед от Decorator'a – механизм перехвата. Castle Windsor обладает великолепными возможностями для осуществления перехвата. В разделе 9.3.3 "Пример: перехват с помощью Windsor" уже приводился исчерпывающий пример. Поэтому вместо того, чтобы заново повторять его в данном разделе я сошлюсь на раздел 9.3.3.

Castle Windsor позволяет нам работать с составными компонентами несколькими различными способами. Мы можем регистрировать компоненты в виде альтернатив друг другу, в виде пиров, которые разрешаются в виде последовательностей, или в виде иерархических Decorator'ов. В большинстве случаев Castle Windsor будет понимать, что

ему делать, но если нам нужен более явный контроль, мы всегда можем использовать метод `ServiceOverrides`, чтобы явно определить, каким образом komponуются сервисы

Кроме того, это может происходить в ситуации, когда нам нужно иметь дело с API, отклоняющимися от `Constructor Injection`. До настоящего момента вы наблюдали за тем, как конфигурировать компоненты в `WindsorContainer`, включая то, как определять стили существования, и как работать с составными компонентами. Но до этого момента мы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют `Constructor Injection`. Поскольку это не всегда происходит именно так, в следующем разделе мы сделаем краткий обзор того, как работать с классами, экземпляры которых должны создаваться особым образом.

10.4. Конфигурирование сложных API

До настоящего момента мы рассматривали то, как можно конфигурировать компоненты, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры, например, Castle Windsor, могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей.

Все становится менее понятным, когда API не столь хорошо функционируют. В данном разделе вы увидите, как работать с простыми аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Давайте начнем с рассмотрения классов, которые принимают в качестве параметров простые типы, например, строки и целые числа.

Конфигурирование простыми зависимостями

Пока мы внедряем абстракции в потребителей, все в порядке. Но данный процесс усложняется, если конструктор зависит от простого типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве компонента контейнера не имеет особого смысла, а в Castle Windsor это и вовсе не работает. Если мы попытаемся разрешить компонент с простой зависимостью, мы получим исключение, даже если простой тип был до этого зарегистрирован.

Рассмотрите в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются "запахами" и их нужно преобразовать в полиморфные классы (имеющие разное состояние). Тем не менее, для данного примера они вполне нам подходят.

Необходимо явным образом сообщить Castle Windsor о том, как разрешать параметр конструктора `spiciness`. Следующий листинг демонстрирует, как это можно сделать, используя синтаксис, очень похожий на метод `ServiceOverrides`, но представляющий собой другой метод.

Листинг 10-10: Применение простого значения для аргумента конструктора

```
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<ChiliConCarne>()
    .DependsOn(new
    {
        spiciness = Spiciness.Hot
    }));
```

Вместо метода `ServiceOverrides`, который переопределяет автоматическую интеграцию, можно использовать метод `DependsOn`, который дает возможность применять экземпляры конкретных зависимостей. В данном примере для параметра конструктора `spiciness` вы используете значение `Spiciness.Hot`.

Примечание

Разница между `ServiceOverrides` и `DependsOn` заключается в том, что в рамках `DependsOn` мы применяем фактические экземпляры, которые используются для данного параметра или свойства, тогда как в рамках `ServiceOverrides` мы используем имена и типы сервисов, которые будут разрешаться для данного параметра или свойства.

Предупреждение

Как и в случае с `ServiceOverrides`, метод `DependsOn` полагается на соответствие между именем параметра и именем безымянного свойства, применяемого в `DependsOn`. Если мы переименуем параметр, то мы должны отредактировать также и вызов `DependsOn`.

Всякий раз, когда нам нужно использовать простое значение, например, строку соединения, мы можем явным образом определить значение в коде (или взять его из конфигурации приложения) и присвоить его при помощи метода `DependsOn`. Что хорошо при использовании `DependsOn`, так это то, что нам не нужно явным образом вызывать конструктор или применять какие-либо другие зависимости, в которых автоматическая интеграция была бы более подходящей. Но недостаток использования `DependsOn` — он более хрупок для выполнения рефакторинга.

Существует более мощная альтернатива, которая позволяет явным образом вызывать конструктор. Она также может использоваться для работы с классами, которые не имеют традиционных конструкторов.

Регистрация компонентов с помощью блоков кода

Экземпляры некоторых классов не могут быть созданы посредством открытого конструктора. Вместо него для создания экземпляров типов вы должны использовать некоторого рода фабрику. Это всегда проблематично для DI-контейнеров, поскольку по умолчанию они следят за наличием открытых конструкторов.

Рассмотрите приведенный ниже пример конструктора класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже если класс `JunkFood` является открытым, конструктор расположен внутри него. Очевидно, экземпляры `JunkFood` должны создаваться посредством статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С точки зрения `Castle Windsor` это проблемное API, поскольку в нем отсутствуют точно выраженные изданные соглашения касательно статических фабрик. Тут требуется помощь — и мы можем предоставить ее посредством блока кода, что продемонстрировано в следующем листинге.

Листинг 10-11: Конфигурирование метода фабрики

```
container.Register(Component
    .For<IMeal>()
    .UsingFactoryMethod(() =>
        JunkFoodFactory.Create("chicken meal")));
```

Для определения блока кода, который создает соответствующий экземпляр, можно использовать метод `UsingFactoryMethod` — в данном примере путем вызова метода `Create` с необходимым параметром для `JunkFoodFactory`.

Данный блок кода будет вызываться в соответствующее время согласно сконфигурированному стилю существования компонента. В данном примере, поскольку мы явным образом не определили стиль существования, по умолчанию используется `Singleton`, а метод `Create` будет вызываться всего единожды, вне зависимости от того, сколько раз вы разрешаете `IMeal`. Если бы вы сконфигурировали компонент таким образом, чтобы для него использовался стиль существования `Transient`, то метод `Create` вызывался бы всякий раз, когда вы разрешали бы `IMeal`.

Использование блока кода предоставляет возможность более экзотичной инициализации объектов, нежели обычные открытые конструкторы. Кроме того, используя блок кода, мы получаем обеспечивающую наибольшую типовую безопасность альтернативу применению простых типов, нежели та, которую предоставляет метод `DependsOn`, который вы наблюдали в разделе 10.4.1 "Конфигурирование простых зависимостей":

```
container.Register(Component
    .For<ICourse>()
    .UsingFactoryMethod(() =>
        new ChiliConCarne(Spiciness.Hot)));
```

В данном примере вы используете блок кода, чтобы явным образом создать новый экземпляр класса `ChiliConCarne` с необходимым `Spiciness`. Это обеспечивает большую типовую безопасность, но полностью устраняет возможность автоматической интеграции для рассматриваемого типа.

Подсказка

Существуют более продвинутые перегрузки `UsingFactoryMethod`, которые позволяют разрешать зависимости из контейнера. Это полезно в ситуации, когда нам нужно использовать `UsingFactoryMethod` для того, чтобы явным образом присвоить только один из нескольких параметров, но при этом для выполнения компиляции мы должны передавать все остальные параметры.

`UsingFactoryMethod` — хороший инструмент для работы с классами, которые не могут быть созданы посредством открытого конструктора. Пока у вас есть некое открытое API, которое вы можете вызвать для создания необходимого экземпляра класса, вы можете использовать метод `UsingFactoryMethod` для того, чтобы явным образом определить блок кода, который будет создавать запрашиваемый экземпляр.

Последним общепринятым отклонением от `Constructor Injection`, которое мы будем здесь наблюдать, является `Property Injection`.

Подключение с помощью `Property Injection`

`Property Injection` является не столь четко определенной формой механизма внедрения зависимостей, поскольку компилятор не принуждает вас присваивать значение свойству, доступному для записи. И все-таки `Castle Windsor`, по своей природе, понимает `Property Injection` и по возможности присваивает значения доступным для записи свойствам.

Рассмотрите приведенный ниже класс `CaesarSalad`:

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

Согласно общепринятому заблуждению в салат "Цезарь" входит курица. По существу салат "Цезарь" является салатом, но, поскольку с курицей он вкуснее, многие рестораны предлагают возможность добавления в него курицы в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Если вы зарегистрируете только класс `CaesarSalad` без какого-либо `Chicken`, то свойству `Extra` не будет присвоено значение:

```
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<CaesarSalad>()) ;
```

Благодаря такой регистрации в результате разрешения `ICourse` будет возвращаться экземпляр `CaesarSalad` без какого-либо ингредиента `Extra`. Тем не менее, вы можете изменить выходной результат, добавив в контейнер `Chicken`:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Chicken>()) ;
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<CaesarSalad>()) ;
```

Теперь при разрешении `ICourse` свойство `Extra` возвращаемого экземпляра `CaesarSalad` будет представлять собой экземпляр класса `Chicken`. То есть, `Castle Windsor` просматривает новый экземпляр на наличие доступных для записи свойств и присваивает им значения, если может предоставить компонент, который по типу совпадает с типом свойства.

Подсказка

В ситуациях, когда вам необходимо явным образом управлять тем, как присваиваются значения свойствам, вы можете использовать метод `ServiceOverrides`.

В данном разделе вы увидели, как работать с API, которые отклоняются от `Constructor Injection`. Вы можете обратиться к простым аргументам конструктора с помощью метода `DependsOn` или `UsingFactoryMethod`, который также поддерживает методы фабрики и другие альтернативы открытых конструкторов. `Castle Windsor`, по своей природе, поддерживает `Property Injection`.

10.5. Резюме

Обзор Castle Windsor, предоставленный в данной главе, только обрисовывает то, что возможно выполнить с помощью одного из самых развитых и исчерпывающих, доступных DI-контейнеров. Поскольку Seam'ы находятся повсюду, мы можем настроить контейнер для наших собственных нужд, и при этом станут доступными многие дополнительные возможности.

В данной главе мы сконцентрировались на наиболее универсальной части API Windsor. Представленный в ней материал охватывает основное применение контейнера, а также советы по более продвинутому составляющему его API. Это все, что вам нужно знать, если ваша база кода руководствуется DI паттернами и соглашениями. Вооружившись данным знанием, вы должны уметь подключать с помощью Castle Windsor целые приложения.

Даже будучи пустым DI-контейнером, Castle Windsor является довольно значительным. Он поддерживает практически любую возможность, которую мы могли бы запросить. Являясь одним из самых старейших .NET DI-контейнеров, он значительно выигрывает благодаря длительным годам своего развития. Однако это не демонстрирует его возраст; напротив, он поддерживает множество новых идей и современных конструкций языка. Все же, возможно, самый значительный недостаток Castle Windsor заключается в том, что огромный набор возможностей реализуется за счет утраты некоторого неоднородного API. Несмотря на то, что начать работать с классом WindsorContainer довольно легко, более продвинутые сценарии реализовывать будет сложно до тех пор, пока вы точно не овладеете всем API. К счастью, поскольку форум поддержки Castle Windsor активен, и опытные разработчики выполняют его мониторинг, если у вас появляется вопрос, вы, скорее всего, быстро получите ответ на него.

Несмотря на то, что продвинутое API может казаться устрашающим, начать работать с Castle Windsor намного легче, чем с любым другим DI-контейнером: создайте экземпляр WindsorContainer, сконфигурируйте его и разрешайте с помощью него компоненты.

Существует несколько способов конфигурирования контейнера: возможно и использование кода в качестве конфигурации (code as configuration), и XML, и конфигурация на основании соглашений; мы даже можем сочетать и сопоставлять все эти три способа, чтобы достичь оптимального решения.

В Castle Windsor доступен широкий набор стилей существования, включая Singleton, Transient и Web Request Context. Если не подходят встроенные стили существования, вы можете реализовать пользовательские стили существования — но такое довольно редко случается.

Кажется, что поддержка составных компонентов одной и той же абстракции является одной из слабых сторон Castle Windsor. Он, кажется, лучше понимает массивы, чем другие типы последовательностей или списков, но мы можем справиться с этим недостатком довольно легко. Точный способ решения данной проблемы зависит от того, хотим мы разрешить все компоненты или только подмножество компонентов одного и того же сервиса.

Несмотря на то, что нам следует полагаться на автоматическую интеграцию, метод ServiceOverrides дает нам возможность явным образом сконфигурировать то, как зависимости присваиваются компонентам.

Иногда компоненты не используют Constructor Injection, а вместо него могут использовать Property Injection или требовать использования отдельных классов фабрики. Такие сценарии также поддерживаются посредством различных методов.

Поскольку Castle Windsor является одним из самых универсальных среди доступных DI-контейнеров, нет причины не использовать его, но он не исключает альтернатив, которые столь же хороши. В следующей главе мы рассмотрим другой развитый и продвинутый DI-контейнер: StructureMap.

11. StructureMap

Меню:

- Знакомство со StructureMap
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В предыдущей главе мы рассматривали DI-контейнер Castle Windsor с целью увидеть, как к нему применяются принципы и паттерны, описанные в частях 1-3. В данной главе мы практически то же самое сделаем и для другого DI-контейнера – StructureMap.

StructureMap – старейший DI-контейнер из всех DI-контейнеров для .NET. Он существует дольше, чем все остальные контейнеры. Несмотря на его возраст, он все еще продолжает активно развиваться и обладает множеством современных возможностей, поэтому мы должны рассматривать его возраст, главным образом, как доказательство его развитости. Кроме того, StructureMap является одним из наиболее универсально используемых DI-контейнеров.

В данной главе мы рассмотрим то, как можно использовать StructureMap для практического применения принципов и паттернов, изложенных ранее в частях 1-3. Рисунок 11-1 демонстрирует структуру данной главы.

Рисунок 11-1: Данная глава разделена на 4 раздела. Первый раздел знакомит нас с API StructureMap и должен рассматриваться в качестве предпосылки следующих трех разделов. Каждый из четырех разделов данной главы может быть прочитан независимо от остальных разделов, несмотря на то, что в четвертом разделе используются методы которые вводятся в третьем разделе. Данные методы имеют относительно понятные имена, поэтому вы можете прочесть четвертый раздел, не читая при этом третий. Но, с другой стороны, вы можете обнаружить, что вам время от времени придется обращаться к этому разделу.



Первый раздел обеспечивает полноценное знакомство со StructureMap и демонстрирует, как конфигурировать и разрешать компоненты. Каждая из трех последующих глав имеет дело с паттернами применения, которые требуют к себе дополнительного внимания. Вы можете читать их по порядку или пропустить некоторые и прочитать только те, которые вам интересны.

Данная глава должна дать вам возможность приступить к работе, а также справиться с большинством общепринятых проблем, которые могут возникнуть при ежедневном

использовании StructureMap. Это не окончательная трактовка StructureMap, поскольку всецелое его рассмотрение заняло бы несколько глав или, может быть, даже целую книгу.

Вы можете прочитать эту главу, не читая при этом остальные главы части 4, специально для того, чтобы познакомиться со StructureMap, или можете прочитать ее совместно с остальными главами части 4, чтобы сравнить DI-контейнеры. Цель данной главы – показать, как StructureMap связан с паттернами и принципами, описанными в частях 1-3, и как он их реализует.

[11.1. Знакомство с StructureMap](#)

[11.2. Управление жизненным циклом](#)

[11.3. Работа с составными компонентами](#)

[11.4. Конфигурирование сложных API](#)

[11.5. Резюме](#)

11.1. Знакомство с StructureMap

Из этого раздела вы узнаете, где можно взять StructureMap, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим универсальные варианты конфигурирования, а также то, как упаковать настройки конфигурации в повторно используемые компоненты. Таблица 11-1 предоставляет основополагающую информацию, которая, скорее всего, понадобится вам для того, чтобы приступить к работе со StructureMap.

Таблица 11-1: Краткая информация о StructureMap

Вопрос	Ответ
Откуда мне его получить?	Перейти на сайт http://structuremap.github.com/structuremap/index.html и нажать на ссылку Download the Latest Release. Из Visual Studio 2010 можно получить его посредством NuGet. Имя пакета – <i>structuremap</i> .
Что находится в загруженном файле?	Можно загрузить zip-файл, содержащий предварительно скомпилированные бинарные файлы. Кроме того, можно получить текущий исходный код и скомпилировать его самостоятельно. <i>Бинарные файлы</i> – это dll-файлы, которые можно размещать там, где захочется, и ссылаться на них из собственного кода.
Какие платформы поддерживаются?	.NET 3.5 SP1, .NET 4
Сколько он стоит?	Нисколько. Это программное обеспечение с открытым исходным кодом.
Откуда мне получить помощь?	Гарантированная поддержка отсутствует, но получить помощь можно на официальном форуме http://groups.google.com/group/structuremapusers .
На какой версии StructureMap основана данная глава?	2.6.1

Как и в случае с Castle Windsor, при использовании StructureMap соблюдается простой ритм, проиллюстрированный на рисунке 11-2.

Рисунок 11-2: Полноценный паттерн применения StructureMap довольно прост: сначала мы конфигурируем контейнер, а затем разрешаем компоненты из этого контейнера. В большинстве случаев мы создаем экземпляр класса `Container` и полностью конфигурируем его перед тем, как начать разрешать компоненты из него. Мы разрешаем компоненты того же экземпляра, который и конфигурировали.



Контейнер или ObjectFactory?

В более ранних версиях StructureMap статический класс `ObjectFactory` использовался в качестве одиночного контейнера приложений. Он использовался следующим образом:

```
SauceBéarnaise sauce = ObjectFactory.GetInstance<SauceBéarnaise>();
```

Среди нескольких проблем, возникающих при использовании статической фабрики, можно выделить тот факт, что статическая фабрика способствует неправильному ее использованию в качестве `Service Locator`. На сегодняшний момент использование класса `ObjectFactory` не приветствуется, при этом предпочтение отдается экземплярам контейнеров. На сайте StructureMap (и не только на нем) представлено множество примеров, в которых образцы кода `ObjectFactory` используются для демонстрации различных возможностей StructureMap. Но мы должны рассматривать их как рудименты более ранних лет.

В остальных частях данной главы мы будем игнорировать тот факт, что `ObjectFactory` существует и концентрируется исключительно на экземплярах контейнера.

Предупреждение

API StructureMap за последнее время значительно изменился. Очень часто мы находим в интернете пример кода, в котором используется метод или класс, недоступный в текущей версии. Скорее всего, он был переименован или, в противном случае, был выполнен его рефакторинг. Несмотря на то, что при написании данной главы все примеры кода компилировались и работали, возможно, некоторые составляющие API изменились за период, начиная с момента написания главы и до момента ее прочтения вами.

После прочтения этого раздела вы приобретете полноценное понимание всей сущности паттерна применения StructureMap и сможете начать использовать его в сценариях, в которых все компоненты руководствуются должным DI-паттерном, например, `Constructor Injection`. Давайте начнем с простейшего сценария и посмотрим, как можно разрешать объекты с помощью контейнера StructureMap.

Разрешение объектов

Ключевая обязанность любого DI-контейнера – разрешение компонентов. В данном разделе мы рассмотрим API, позволяющее нам разрешать компоненты с помощью StructureMap.

Если вы помните, как мы обсуждали процесс разрешения компонентов с помощью Castle Windsor, то, возможно, вспомните, что Windsor требует, чтобы вы регистрировали все соответствующие компоненты перед тем, как сможете их разрешить. StructureMap этого не требует. Если вы запрашиваете конкретный тип с конструктором по умолчанию, то никакой конфигурации не нужно. Самый простой вариант использования StructureMap представлен ниже:

```
var container = new Container();  
SauceBéarnaise sauce = container.GetInstance<SauceBéarnaise>();
```

Если у вас есть экземпляр StructureMap.Container, то для получения экземпляра конкретного класса SauceBéarnaise, вы можете использовать generic-метод GetInstance. Поскольку у этого класса есть конструктор по умолчанию, StructureMap автоматически поймет, как создать его экземпляр. При этом никакой явной конфигурации контейнера не требуется.

Примечание

Метод GetInstance<T> аналогичен методу Resolve<T> контейнера Windsor.

Поскольку StructureMap поддерживает возможность автоматической интеграции, то даже при отсутствии конструктора по умолчанию он сможет создавать экземпляры без предварительной конфигурации. Делать он это сможет, пока все рассматриваемые параметры конструктора относятся к конкретным типам, а полноценное дерево параметров имеет такие типы листьев, которые обладают конструкторами по умолчанию.

В качестве примера рассмотрите приведенный ниже конструктор Mayonnaise:

```
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

Между тем, как рецепт майонеза достаточно прост, и EggYolk, и OliveOil являются конкретными классами, обладающими конструкторами по умолчанию. Несмотря на то, что сам Mayonnaise не имеет конструктора по умолчанию, StructureMap все равно может создать его, не выполняя при этом никакой конфигурации:

```
var container = new Container();  
var mayo = container.GetInstance<Mayonnaise>();
```

Это происходит, потому что StructureMap может определять, каким образом создавать все необходимые параметры конструктора. Тем не менее, как только мы введем слабое связывание, нам нужно будет сконфигурировать StructureMap таким образом, чтобы абстракции преобразовывались в конкретные типы.

Преобразование абстракций в конкретные типы

Несмотря на то, что способность StructureMap автоматически интегрировать конкретные типы, наверняка, время от времени может искусно появиться, при слабом связывании обычно необходимо преобразовывать абстракции в конкретные типы. Создание экземпляров на основании таких преобразований – ключевая услуга, предоставляемая любым DI-контейнером, но вам все равно нужно задать соответствие.

В данном примере вы преобразуете интерфейс `IIngredient` в конкретный класс `SauceBéarnaise`, который позволяет успешно разрешать `IIngredient`:

```
var container = new Container();
container.Configure(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
IIngredient ingredient = container.GetInstance<IIngredient>();
```

Метод `Configure` предоставляет возможность сконфигурировать `ConfigurationExpression` с помощью блока кода (объяснение смотри в приведенном ниже блоке "Nested Closures" (вложенные замыкания)). Оператор конфигурации читается почти как предложение (или как инструкция из кулинарной книги): *для `IIngredient` используйте `SauceBéarnaise`*. Метод `For` позволяет определить абстракцию, а метод `Use` позволяет определить конкретный тип, реализующий абстракцию.

Строго типизированное API, предоставляемое классом `ConfigurationExpression`, помогает предотвратить ошибки конфигурации, поскольку метод `Use` имеет generic-ограничение, настаивающее на том, чтобы тип, указанный в аргументе типа, наследовался от аргумента типа абстракции, указанного в методе `For`. Предыдущий пример кода компилируется, поскольку `SauceBéarnaise` реализует `IIngredient`.

В большинстве случаев строго типизированное API – это все, что нам нужно, так как оно предоставляет желаемую проверку во время компиляции. Мы должны использовать его там, где это возможно. Однако существуют ситуации, когда нам нужен более слабо типизированный способ разрешения сервисов. Это тоже возможно.

Разрешение слабо типизированных сервисов

Иногда мы не можем использовать generic API, так как во время проектирования мы не знаем, какой тип нам понадобится. Все, что у нас есть – это экземпляр `Type`, но нам все равно хотелось бы получить экземпляр этого типа. Пример этого вы видели в разделе 7-2 "Построение ASP.NET MVC приложений", где мы обсуждали ASP.NET MVC класс `DefaultControllerFactory`. Соответствующий метод представлен ниже:

```
protected internal virtual IController GetControllerInstance(
    RequestContext requestContext, Type controllerType);
```

Поскольку у вас есть только экземпляр `Type`, вы не можете воспользоваться generic'ами и вместо них должны обратиться к слабо типизированному API. К счастью, StructureMap предлагает слабо типизированную перегрузку метода `GetInstance`, которая позволяет реализовать метод `GetControllerInstance` приведенным ниже способом:

```
return (IController) this.container.GetInstance(controllerType);
```

Слабо типизированная перегрузка `GetInstance` позволяет передавать аргумент `controllerType` прямо в `StructureMap`, но также требует, чтобы вы явным образом привели возвращаемое значение к `IController`.

Не важно, какую перегрузку `GetInstance` вы используете, `StructureMap` гарантирует, что эта перегрузка вернет экземпляр требуемого типа или выдаст исключение, если есть зависимости, которые не могут соответствовать заданным требованиям. После того, как все необходимые зависимости сконфигурированы должным образом, `StructureMap` может автоматически интегрировать необходимый тип.

В предыдущем примере `this.container` — это экземпляр `StructureMap.IContainer`. Для обеспечения возможности разрешать необходимый тип все слабо связанные зависимости должны быть заранее сконфигурированы. Существует множество способов конфигурации `StructureMap`. В следующем разделе приводится обзор самых универсальных способов.

Конфигурирование контейнера

Как обсуждалось в разделе 3-2 "Конфигурирование DI-контейнеров", существует несколько концептуально разных способов конфигурирования DI-контейнера. На рисунке 11-3 представлен обзор возможных вариантов.

Рисунок 11-3: Концептуально разные варианты конфигурирования. Использование кода в качестве конфигурации подразумевает строгую типизированность и имеет тенденцию к явному определению. XML, с другой стороны, предполагает позднее связывание, но тоже склонно к явному определению. Автоматическая регистрация, напротив, полагается на соглашения, которые могут быть и строго типизированными, и более слабо определенными.



Как и другие, имеющие длительную историю DI-контейнеры, `StructureMap` сначала использовал XML в качестве основного источника конфигурации. Тем не менее, многие команды разработчиков вскоре узнали, что определение регистраций типа в XML чрезвычайно хрупко, поэтому на сегодняшний день мы предпочитаем строго типизированную конфигурацию. Строго типизированную конфигурацию можно

выполнить, используя код в качестве конфигурации, но чаще всего наиболее эффективной является основанная на соглашениях автоматическая регистрация.

StructureMap поддерживает все три подхода и даже позволяет нам сочетать их в пределах одного и того же контейнера. В этом отношении StructureMap предоставляет нам все, что мы только можем попросить. В данном разделе вы увидите, как можно использовать каждый из этих трех видов источников конфигурации.

Конфигурация в коде

В разделе 11.1.1 "Разрешение объектов" вы уже видели небольшой намек на строго типизированное API для конфигурации StructureMap. В данном разделе мы рассмотрим его подробнее.

Существует несколько точек входа в конфигурационное API. Вы уже видели, что оно вызывается посредством явного использования метода `Configure`:

```
var container = new Container();
container.Configure(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
```

Еще один вариант — определение точно такого же блока кода прямо при создании экземпляра `Container`:

```
var container = new Container(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
```

Результат тот же, однако, в данной главе я руководствуюсь соответствующим соглашением и предпочитаю использовать метод `Configure`, а не конструктор.

Вложенные замыкания

StructureMap широко использует паттерн *Вложенное замыкание* (Nested Closure), в котором конфигурация определяется блоками кода (известными как лямбда-выражения). В качестве примера ниже приведена сигнатура метода `Configure`:

```
public void Configure(Action<ConfigurationExpression> configure);
```

Параметр `configure` — делегат, который принимает `ConfigurationExpression` в качестве входного параметра. В примерах кода, приведенных в этой главе, этот параметр обычно обозначается как `r`, и обычно я передаю делегат в виде блока кода, выраженного посредством параметра `r`.

При просмотре примеров кода, встречающихся на сайте StructureMap или в блоге Джереми Миллера, можно обнаружить, что иногда имя параметра, используемого в блоке кода, задается как `x`, а иногда — как `registry`. Поскольку подходящие прецеденты отсутствуют, я решил использовать `r` (соответствующее `registry`) в качестве условного обозначения, которое будет применяться в данной главе. Несмотря на то, что `r` не является достаточно понятным именем для переменной, рассматриваемые здесь небольшие по объему блоки кода делают `r` более подходящим для этих целей именем, нежели более длинное и менее краткое имя.

Класс `ConfigurationExpression` содержит множество методов, которые можно использовать для конфигурирования `StructureMap`. Один из этих методов, `For`, мы уже видели ранее. Как вы увидите в данном разделе позднее, еще одним таким методом является метод `Scan` с приведенной ниже сигнатурой:

```
public void Scan(Action<IAssemblyScanner> action);
```

Обратите внимание на то, что сам метод `Scan` принимает делегат в качестве входного параметра. Когда вы передаете блок кода метода `Scan`, то получается, что у вас имеется один блок кода внутри другого блока кода – отсюда и название *Nested Closure* (вложенное замыкание).

В отличие от `Castle Windsor` преобразование `IIngredient` в `SauceBéarnaise` продемонстрированным ранее способом не исключает разрешения самого `SauceBéarnaise`. То есть, и `sauce`, и `ingredient` будут разрешены должным образом:

```
container.Configure(r =>
    r.For<IIngredient>().Use<SauceBéarnaise>());
var sauce = container.GetInstance<SauceBéarnaise>();
var ingredient = container.GetInstance<IIngredient>();
```

Если вы вспомните обсуждение, проводимое в разделе 10.1.2 "Конфигурирование контейнера", то вспомните и то, что преобразование `IIngredient` в `SauceBéarnaise` с помощью `Castle Windsor` приводит к тому, что "исчезает" конкретный класс (`SauceBéarnaise`), и вам приходится использовать *перенаправление типов* (*Type Forwarding*), чтобы суметь разрешить и `IIngredient`, и `SauceBéarnaise`. При использовании `StructureMap` такие дополнительные шаги выполнять не нужно, поскольку `StructureMap` умеет преобразовывать и `IIngredient`, и `SauceBéarnaise`. В обоих случаях возвращаемые объекты являются экземплярами `SauceBéarnaise`.

В реальных приложениях нам всегда нужно преобразовывать более одной абстракции, поэтому нам необходимо сконфигурировать составные преобразования. Сделать это можно при помощи единичного вызова метода `Configure` или посредством составных последовательных вызовов. Приведенные ниже примеры аналогичны:

```
container.Configure(r =>
{
    r.For<IIngredient>()
        .Use<SauceBéarnaise>();
    r.For<ICourse>()
        .Use<Course>();
});
```

и

```
container.Configure(r => r
    .For<IIngredient>()
        .Use<SauceBéarnaise>());
container.Configure(r => r
    .For<ICourse>()
        .Use<Course>());
```

Несмотря на то, что во втором примере для конфигурации используются два последовательных вызова, в первом примере блок кода с большим количеством операторов передается в единичный вызов метода `Configure`. Оба примера кода

завершаются регистрацией корректных преобразований как для интерфейса `ICourse`, так и для интерфейса `IIngredient`. Тем не менее, конфигурирование одной и той же абстракции несколько раз приводит к интересным результатам:

```
container.Configure(r =>
    r.For<IIngredient>().Use<SauceBéarnaise>());
container.Configure(r =>
    r.For<IIngredient>().Use<Steak>());
```

В этом примере вы регистрируете `IIngredient` дважды. Если вы разрешаете `IIngredient`, то получаете экземпляр `Steak`. Выигрывает последняя конфигурация, но предыдущие конфигурации не забыты. `StructureMap` отлично управляет составными конфигурациями одной и той же абстракции, но к этой теме мы вернемся в разделе 11.3.1 "Выбор из составных кандидатов".

Возможны более продвинутые варианты конфигурирования `StructureMap`, но можно сконфигурировать все приложение при помощи продемонстрированных ниже методов. Для того чтобы избежать излишнего явного сопровождения конфигурации контейнера, мы могли бы рассмотреть подход, в большей мере основанный на соглашениях, — автоматическую регистрацию.

Автоматическая регистрация

В большинстве случаев регистрации будут аналогичными. Такие регистрации трудно сопровождать, а явная регистрация каждого компонента, возможно, не самый продуктивный подход.

Рассмотрим библиотеку, которая состоит из множества реализаций `IIngredient`. Мы можем сконфигурировать каждый класс индивидуальным образом, но это приведет к многочисленным, схожим по внешнему виду вызовам метода `Configure`. Что еще хуже, всякий раз при добавлении новой реализации `IIngredient`, мы должны также явным образом конфигурировать эту реализацию в контейнере, если хотим, чтобы она была доступной. Было бы более продуктивно установить, что все реализации `IIngredient`, находящиеся в данной сборке, необходимо зарегистрировать.

Это возможно благодаря методу `Scan`, который является еще одним примером обширного применения `StructureMap` делегатов. Метод `Scan` доступен в классе `ConfigurationExpression`, который уже доступен через блок кода. Именно здесь мы и видим паттерн `Nested Closure` в действии. Приведенный ниже пример кода конфигурирует все реализации `IIngredient` одним махом:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
    }));
```

Метод `Scan` расположен в пределах блока кода `Configure`. Переменная `s` представляет собой экземпляр `IAssemblyScanner`, который можно использовать для определения того, каким образом необходимо просматривать сборку и как должны конфигурироваться типы.

Экземпляр `IAssemblyScanner` предоставляет несколько методов, которые можно использовать для того, чтобы определить, какие сборки необходимо просматривать и как конфигурировать типы из этой сборки. Мы можем использовать генетис-метод `AssemblyContainingType` для идентификации сборки по типу представителя, но существует несколько других методов, позволяющих нам предоставить экземпляр `Assembly` или даже добавить все сборки, находящиеся по данному пути.

Другой набор методов дает нам возможность определить, какие типы добавлять и как их преобразовывать. Метод `AddAllTypesOf` обеспечивает самый быстрый способ добавления всех типов, реализующих данный интерфейс, но существует и несколько других методов, которые позволяют нам тщательно контролировать то, как конфигурируются типы.

Предыдущий пример, безусловно, конфигурирует все реализации интерфейса `IIngredient`, но мы можем предоставить фильтры, которые позволят нам выбирать только подмножества. Ниже приведен пример сканирования на основании соглашений, при котором вы добавляете только те классы, имена которых начинаются с `Sauce`:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
        s.Include(t => t.Name.StartsWith("Sauce"));
    }));
```

Единственное отличие от предыдущего примера заключается в добавлении вызова метода `Include`, который вводит третий уровень `Nested Closure`. Метод `Include` принимает в качестве параметра предикат, который используется для определения того, нужно ли включать данный `Type` или нет. В этом примере ответ — `true`, поскольку `Name` для `Type` начинается с `Sauce`.

Если мы хотим полностью контролировать процесс конфигурирования на основании соглашений, мы можем определить пользовательское соглашение путем реализации интерфейса `IRegistrationConvention`. В следующем листинге демонстрируется соглашение `Sauce`, реализованное в виде пользовательского соглашения.

Листинг 11-1: Реализация пользовательского соглашения

```
public class SauceConvention : IRegistrationConvention
{
    public void Process(Type type, Registry registry)
    {
        var interfaceType = typeof(IIngredient);
        if (!interfaceType.IsAssignableFrom(type))
        {
            return;
        }
        if (!type.Name.StartsWith("Sauce"))
        {
            return;
        }
        registry.For(interfaceType).Use(type);
    }
}
```

Класс `SauceConvention` реализует `IRegistrationConvention`, который определяет единственный член. Метод `Process` будет вызываться `StructureMap` для каждого типа сборки, определенного в методе `Scan`, поэтому вы должны явным образом предоставить набор граничных операторов, которые отфильтровывают все не нужные вам типы.

Граничные операторы гарантируют, что любой тип, проходящий через них, – это `IIIngredient`, чье имя начинается с `Sauce`, поэтому теперь вы можете зарегистрировать этот тип с помощью `registry`. Обратите внимание на то, что `Registry`, между прочим, предоставляется посредством `Method Injection`, который имеет огромный смысл, поскольку `IRegistrationConvention` определяет встраиваемый элемент для `StructureMap`.

Можно использовать класс `SauceConvention` в методе `Scan` следующим образом:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.Convention<SauceConvention>();
    }));
```

Обратите внимание на то, что вы все равно определяете сборку за рамками соглашения. Это позволяет вам менять источники типов, которые обрабатываются независимо от самого соглашения. `SauceConvention` определяется с помощью метода `Convention`. Данный метод требует, чтобы `IRegistrationConvention`, указанный в качестве аргумента типа, имел конструктор по умолчанию. Но существует также метод `With`, принимающий в качестве входного параметра экземпляр `IRegistrationConvention`, который можно создать вручную любым необходимым способом.

Поскольку вы можете использовать метод `Scan` для того, чтобы просматривать все сборки в указанной папке, вы также можете применять его для реализации дополнительной функциональности, в которой дополнения могут добавляться без повторной компиляции основного приложения. Это один из способов реализации позднего связывания. Еще один вариант – использовать конфигурационное API, основанное на XML.

XML конфигурация

XML конфигурация – отличный вариант в тех ситуациях, когда нам необходимо уметь изменять конфигурацию без повторной компиляции приложения.

Подсказка

Используйте XML конфигурацию только для тех типов, изменить которые вам нужно, не выполняя при этом повторной компиляции приложения. В остальных случаях применяйте автоматическую регистрацию или технологию использования кода в качестве конфигурации.

Мы можем использовать специализированные XML-файлы для того, чтобы сконфигурировать `StructureMap`, или даже вставить конфигурацию в стандартный конфигурационный файл приложения. Тем не менее, удивительно, но возможность вставки конфигурации в конфигурационный файл приложения напрямую не поддерживается в `StructureMap`, поэтому давайте сначала рассмотрим использование специализированных XML-файлов.

Конфигурацию можно определить в XML и прочитать при помощи метода `AddConfigurationFromXmlFile`:

```
container.Configure(r =>
    r.AddConfigurationFromXmlFile(configName));
```

В данном примере `configName` — это строка, которая содержит имя соответствующего XML-файла. Если вы захотите использовать стандартный конфигурационный файл приложения, то вам нужно будет использовать `AppDomain API` для того, чтобы определить путь к текущему конфигурационному файлу:

```
var configName =
    AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
```

Примечание

Несмотря на то, что статический класс `ObjectFactory` напрямую поддерживает чтение конфигурации из `App.config`, данная возможность не поддерживается для экземпляров контейнера. Использование `AppDomain API` для получения имени файла — рекомендуемая технология работы.

Помимо направления `StructureMap` к соответствующему файлу XML конфигурацию можно передать в виде `XmlNode`:

```
container.Configure(r =>
    r.AddConfigurationFromNode(xmlNode));
```

Данный код позволяет вам получать XML конфигурацию не только из файлов, но и из таких произвольных мест, как базы данных или вложенные ресурсы.

Не важно, каков источник XML, схема остается той же. Ниже приведена простая конфигурация, которая преобразует `IIngredient` в `Steak`:

```
<StructureMap MementoStyle="Attribute">
  <DefaultInstance PluginType="Ploeh.Samples.MenuModel.IIngredient,
    ➡ Ploeh.Samples.MenuModel"
    PluggedType="Ploeh.Samples.MenuModel.Steak,
    ➡ Ploeh.Samples.MenuModel" />
</StructureMap>
```

Обратите внимание на то, что вы должны передавать квалифицированное имя типа сборки как для абстракции, так и для реализации — `StructureMap` называет их *Плагинами* (Plugins) или *Подключаемыми типами* (Plugged types).

Если вы хотите вставить данный код XML в конфигурационный файл приложения, то вы должны зарегистрировать элемент `StructureMap` в виде раздела конфигурации:

```
<configSections>
  <section name="StructureMap"
    type="StructureMap.Configuration.
    ➡ StructureMapConfigurationSection, StructureMap"/>
</configSections>
```

XML конфигурация – отличный вариант в тех ситуациях, когда вам необходимо изменить конфигурацию одного или более одного компонента, не выполняя при этом повторной компиляции приложения. Тем не менее, поскольку этот вариант довольно хрупок, поберегите его для таких конкретных ситуаций, а для основной части конфигурации контейнера используйте либо автоматическую регистрацию, либо код в качестве конфигурации.

Подсказка

Помните, что выигрывает последняя конфигурация типа? Данное поведение вы можете использовать для того, чтобы перезаписать жестко-закодированную конфигурацию XML конфигурацией. Для этого вы должны не забывать считывать XML конфигурацию после того, как будут сконфигурированы любые другие компоненты.

В данном разделе мы, главным образом, рассматривали различные API для конфигурации StructureMap. Несмотря на то, что, наверняка, можно записать один большой блок неструктурированного кода конфигурации, лучше всего разделить конфигурацию на модули. StructureMap поддерживает эту возможность посредством *регистров* (Registries).

Пакетирование конфигурации

Иногда существует необходимость упаковать логику конфигурации в повторно используемые группы. И даже в тех случаях, когда само по себе повторное использование не является для вас высшим приоритетом, вы можете захотеть предоставить некоторого рода структуру, когда вам нужно конфигурировать большое и сложное приложение.

В рамках StructureMap мы можем упаковывать конфигурацию в *регистры* (Registries), которые представляют собой классы, унаследованные от конкретного класса Registry. Рисунок 11-4 демонстрирует взаимосвязь между классом Registry и методом Configure, который использовался в разделе 11.1.2 "Конфигурирование контейнера".

Рисунок 11-4: Метод Configure класса Container принимает в качестве входного параметра делегат, который действует для ConfigurationExpression – в данной главе мы обозначаем экземпляр этого ConfigurationExpression с помощью имени переменной r. Класс ConfigurationExpression – дочерний класс конкретного класса Registry.



Каждый раз при использовании метода Configure в этой главе вы представляете экземпляр ConfigurationExpression с помощью имени переменной r. Большинство методов, выываемых для r (например, методы For и Scan), определены в классе Registry.

Для того чтобы реализовать Registry, мы реализуем класс, который наследуется от Registry. Приведенный ниже листинг демонстрирует пример, который конфигурирует используемый по умолчанию ICourse, а также добавляет типы IIngredient из сборки. В нем используется то же самое API, которое мы ранее использовали в разделе 11.1.2

"Конфигурирование контейнера", но в настоящее время это API упаковано в отдельный класс.

Листинг 11-2: Реализация регистра

```
public class MenuRegistry : Registry
{
    public MenuRegistry()
    {
        this.For<ICourse>().Use<Course>();
        this.Scan(s =>
        {
            s.AssemblyContainingType<Steak>();
            s.AddAllTypesOf<Ingredient>();
        });
    }
}
```

Registry или ConfigurationExpression?

Несмотря на то, что большинство конфигурационных API (например, методы `For` и `Scan`) все равно доступны при прямом наследовании от `Registry`, мы не можем использовать методы, определенные непосредственно для класса `ConfigurationExpression`. Какую функциональность мы теряем?

Существует только пять методов, которые определены непосредственно для `ConfigurationExpression`, и разделяются они на две категории:

- Методы, которые считывают конфигурацию из XML
- Методы, добавляющие регистры

Скорее всего, нам не понадобится добавлять `Registry` внутри `Registry`, поэтому данная возможность не столь важна.

Конфигурация, определенная в XML, – совершенно другой способ выражения конфигурации. Мы выражаем определенную часть конфигурации приложения либо в XML, либо при помощи конфигурационного API, но использовать эти два способа одновременно нельзя. В этом свете невозможность определения источников XML из `Registry` не является большим недостатком.

И все-таки, можно ли унаследовать регистр от `ConfigurationExpression` вместо того, чтобы наследовать его напрямую от `Registry`? К несчастью, мы не можем это сделать, поскольку конструктор `ConfigurationExpression` – внутренний.

Основной момент – это то, что регистр не может быть унаследован от `ConfigurationExpression`, а должен наследоваться от самого `Registry`.

Класс `MenuRegistry` наследуется от `Registry` и определяет всю конфигурацию в конструкторе. Внутри класса вы можете получить доступ ко всему открытому API класса `Registry`, поэтому использовать методы `For` и `Scan` вы можете тем же самым способом, что и в разделе 11.1.2 "Конфигурирование контейнера". Единственное отличие заключается в том, что в данном случае вы реализуете не безымянный делегат, а конструктор. Вместо блока кода и вездесущей переменной `r`, к которым вы, возможно, на данный момент уже привыкли, вы обращаетесь к API посредством переменной `this`.

После получения `MenuRegistry` вы теперь можете добавить его в контейнер с помощью метода `Configure`:

```
container.Configure(r =>
    r.AddRegistry<MenuRegistry>());
```

Эта generic-версия метода `AddRegistry` требует, чтобы реализация `Registry` имела конструктор по умолчанию, но помимо этого доступна и не generic-перегрузка, которая принимает экземпляр `Registry` в качестве входного параметра, предоставляя нам полный контроль над тем, как он создается.

Примечание

Методы `AddRegistry` – два из пяти методов, определенных непосредственно для `ConfigurationExpression` и недоступных внутри `Registry`.

Вы также можете передать `Registry` непосредственно через конструктор контейнера:

```
var container = new Container(new MenuRegistry());
```

Я предпочитаю использовать метод `Configure`, поскольку он позволяет мне добавлять в последовательность более одного регистра.

Подсказка

Регистры позволяют вам упаковать и структурировать код конфигурации вашего контейнера. Используйте их вместо однострочной конфигурации, так как это делает вашу `Composition Root` более читабельной.

Благодаря регистрам мы можем конфигурировать `StructureMap`, используя в качестве конфигурации код или автоматическую регистрацию, тогда как XML конфигурация должна импортироваться непосредственно через метод `Configure`. Кроме того, мы можем сочетать оба подхода, получая некоторую часть конфигурации из XML, а остальную – из одного или более чем одного регистра:

```
container.Configure(r =>
{
    r.AddConfigurationFromXmlFile(configName);
    r.AddRegistry<MenuRegistry>();
});
```

После того, как контейнер сконфигурирован, вы можете начать разрешать с помощью него сервисы, как это описано в разделе 11.1.1 "Разрешение объектов".

Данная глава познакомила вас с DI-контейнером `StructureMap` и продемонстрировала фундаментальные механизмы: как конфигурировать контейнер и впоследствии использовать его для разрешения сервисов. Разрешение сервисов с легкостью выполняется при помощи единичного вызова метода `GetInstance`, поэтому вся сложность заключается в конфигурировании контейнера. Это можно сделать несколькими различными способами, включая императивный код и XML. До настоящего момента мы рассматривали только самое основное API, поэтому нам придется рассмотреть еще и более продвинутое API. Один из самых важных вопросов – как управлять жизненным циклом компонентов.

11.2. Управление жизненным циклом

В главе 8 "Жизненный цикл объектов" мы обсуждали процесс управления жизненным циклом, включая самые универсальные, концептуальные стили существования, например, Singleton и Transient. StructureMap поддерживает множество различных стилей существования и позволяет нам конфигурировать жизненные циклы всех сервисов. Стили существования, продемонстрированные в таблице 11-2, доступны в виде составляющей API StructureMap.

Таблица 11-2: Стили существования StructureMap

1. Название	2. Комментарии
3. PerRequest	4. Название стиля существования Per Graph, используемое в рамках StructureMap. Этот стиль используется им по умолчанию. Экземпляры контейнером не отслеживаются.
5. Singleton	6. Стандартный Singleton
7. HttpContext	8. Название стиля существования Web Request Context, используемое в рамках StructureMap.
9. ThreadLocal	10. На один поток создается один экземпляр.
11. Hybrid	12. Комбинация HttpContext и ThreadLocal. HttpContext используется тогда, когда он доступен (например, когда контейнер размещается в веб-приложении), а ThreadLocal используется в качестве резерва.
13. HttpSession	14. На одну HTTP-сессию создается один экземпляр. Используйте его с осторожностью.
15. HybridHttpSession	16. Комбинация HttpSession и ThreadLocal. HttpSession используется тогда, когда он доступен (например, когда контейнер размещается в веб-приложении), а ThreadLocal используется в качестве резерва.
17. Unique	18. Название стиля существования Transient, используемое в рамках StructureMap.

Реализации различных стилей существования в StructureMap эквивалентны основным паттернам стилей существования, описанным в главе 8, поэтому в этой главе я не буду много времени тратить на их обсуждение.

Подсказка

Стиль существования, используемый в StructureMap по умолчанию, – это Per Graph. Как мы уже обсуждали в разделе 8.3.3 "Per Graph", данный стиль предлагает наилучший баланс между эффективностью и безопасностью. Кроме того, если ваши сервисы потоко-безопасны, то наиболее эффективный стиль существования в этом случае – Singleton, но при этом вы должны не забывать конфигурировать такие сервисы.

В данном разделе вы увидите, как определять стили существования для компонентов и в коде, и в XML. Помимо этого в качестве более продвинутого сценария вы рассмотрите то, как реализовать пользовательский стиль существования с целью продемонстрировать, что ваши возможности не ограничены встроенными в StructureMap стилями существования.

После прочтения этого раздела вы должны будете уметь использовать стили существования StructureMap в своем собственном приложении.

Давайте начнем с обзора того, как конфигурировать стили существования для компонентов.

Конфигурирование стилей существования

В данном разделе мы приведем обзор того, как управлять жизненными стилями компонентов с помощью StructureMap. Стили существования конфигурируются в виде составляющей части конфигурируемых компонентов, и вы можете определить их как в коде, так и в XML. Мы поочередно рассмотрим каждый из этих стилей существования.

Конфигурирование стилей существования с помощью кода

Стили существования конфигурируются в виде составляющей части Configure API, которое вы используете для конфигурирования компонентов в целом. Это столь же просто, как и приведенный ниже код:

```
container.Configure(r =>
    r.For<SauceBéarnaise>().Singleton());
```

Данный код конфигурирует конкретный класс `SauceBéarnaise` в виде `Singleton` таким образом, что каждый раз, когда запрашивается `SauceBéarnaise`, создается один и тот же экземпляр. Если вы хотите преобразовать абстракцию в конкретный класс с заданным жизненным циклом, то объявление стиля существования помещается между вызовами методов `For` и `Use`:

```
container.Configure(r =>
    r.For<IIngredient>().Singleton().Use<SauceBéarnaise>());
```

Данный код преобразует `IIngredient` в `SauceBéarnaise`, а также конфигурирует его в виде `Singleton`. Существуют и другие методы, аналогичные методу `Singleton`, которые позволяют нам объявить множество других стилей существования. Но не все стили существования обладают таким методом. Все стили существования могут конфигурироваться при помощи универсального метода `LifecycleIs`. К примеру, стиль существования `Unique` не имеет такого метода, но может быть сконфигурирован следующим образом:

```
container.Configure(r => r
    .For<SauceBéarnaise>()
    .LifecycleIs(new UniquePerRequestLifecycle()));
```

Метод `LifecycleIs` принимает в качестве параметра экземпляр `ILifecycle`, поэтому вы можете передать его в любой класс, реализующий этот интерфейс. Как вы увидите в разделе 11.2.2 "Разработка пользовательского стиля существования", таким же способом мы конфигурируем компонент, имеющий пользовательский жизненный цикл.

Все встроенные в StructureMap стили существования обладают соответствующей реализацией `ILifecycle`, за исключением используемого по умолчанию стиля существования `Per Graph`. Этот стиль существования обычно неявным образом конфигурируется за счет опускания явного стиля существования. Во всех конфигурациях,

которые вы видели в разделе 11.1 "Знакомство со StructureMap", использовался стиль существования `Per Graph`.

Подсказка

Описание объявления стиля существования подразумевает `Per Graph`, который используется в `StructureMap` по умолчанию. Но `null` на месте экземпляра `ILifecycle` также подразумевает `Per Graph`.

Если мы создаем некоторого рода универсальный код, который принимает в качестве входной информации экземпляр `ILifecycle` и передает его в метод `LifecycleIs`, то можем использовать его для конфигурирования компонента с помощью стиля существования `Per Graph`. Наличие `null` подразумевает `Per Graph`, поэтому два приведенных ниже примера, эквивалентны:

```
container.Configure(r => r
    .For<IIngredient>()
    .LifecycleIs(null)
    .Use<SauceBéarnaise>());
```

и

```
container.Configure(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
```

Подсказка

Несмотря на то, что вы можете использовать `null` для того, чтобы дать намек на `Per Graph`, лучше всего полностью опустить объявление стиля существования.

В то время как API, раскрывающееся при помощи метода `Configure` и `ConfigurationExpression`, позволяет нам явным образом объявлять стиль существования, `Scan API`, основанное на соглашениях, не позволяет нам сделать это. В интерфейсе `IAssemblyScanner` нет ни одного метода, который дал бы нам возможность одним махом явно объявить стиль существования для набора компонентов.

Тем не менее, мы можем реализовать простой `IRegistrationConvention`, который может объявить стиль существования для набора компонентов одним махом. Ниже приведен пример использования экземпляра `IRegistrationConvention` под названием `SingletonConvention`:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
        s.Convention<SingletonConvention>();
    }));
```

Обратите внимание на то, что здесь приведена такая же конфигурация, как и в первом примере автоматической регистрации из раздела 11.1.2 "Конфигурирование контейнера". Вы добавили только одну строку кода, которая добавляет `SingletonConvention`, продемонстрированный в приведенном ниже листинге.

Листинг 11-3: Реализация соглашения по объявлению стиля существования

```
public class SingletonConvention : IRegistrationConvention
{
    public void Process(Type type, Registry registry)
    {
        registry.For(type).Singleton();
    }
}
```

Если вы помните предыдущее обсуждение `IRegistrationConvention` из листинга 11-1, то помните и то, что метод `Process` вызывается в сборке операции `Scan` для каждого включенного типа. В этом случае единственное, что вам необходимо сделать – объявить стиль существования для каждого типа, используя метод `Singleton`. Таким образом, мы сконфигурируем каждый тип в виде `Singleton`.

Используя код в качестве конфигурации, мы можем конфигурировать компоненты, имеющие различные стили существования, тем способом, которым только захотим. Несмотря на то, что это намного более гибкий способ конфигурирования компонентов, нам иногда нужно обращаться к XML с целью получения позднего связывания. В этом случае мы также можем объявлять стили существования.

Конфигурирование стилей существования с помощью XML

В ситуациях, когда нам нужно определять компоненты в XML, мы еще хотим в этом же самом месте уметь конфигурировать их стили существования. Это легко выполняется в виде составляющей части XML-схемы, введенной в разделе 11.1.2 "Конфигурирование контейнера". Для объявления стиля существования можно использовать необязательный атрибут `Scope`:

```
<DefaultInstance PluginType="Ploeh.Samples.MenuModel.IIngredient,
    ➤ Ploeh.Samples.MenuModel"
    PluggedType="Ploeh.Samples.MenuModel.Steak,
    ➤ Ploeh.Samples.MenuModel"
    Scope="Singleton" />
```

Единственное отличие этого примера от примера из раздела 11.1.2 – добавленный атрибут, который конфигурирует экземпляр в виде `Singleton`. Когда вы ранее опускали атрибут `Scope`, автоматически применялся `Per Graph`, используемый в `StructureMap` по умолчанию.

И в коде, и в XML конфигурировать стили существования для компонентов легко. Во всех случаях это выполняется в довольно декларативной манере. Несмотря на то, что выполнять конфигурацию легко, вы должны не забывать, что в некоторые стили существования включаются объекты-долгожители, которые, пока существуют, используют память.

Предотвращение утечек памяти

Как и любой другой DI-контейнер `StructureMap` создает для нас диаграмму объектов, но не отслеживает созданные объекты. Он может отслеживать эти объекты в своих собственных целях, но зависит это от жизненного цикла объекта. К примеру, для того чтобы реализовать область применения `Singleton`, `StructureMap` должен сохранять ссылку на созданный экземпляр. Это справедливо и для стиля существования `HttpContext`, в котором

все экземпляры хранятся в `HttpContext.Current.Items`. Тем не менее, после завершения HTTP-запроса все эти экземпляры выходят за пределы области применения и могут быть уничтожены сборщиком мусора.

С другой стороны, стили существования `Per Graph` и `Transient` не отслеживают созданные `StructureMap` объекты. Как вы видели в листингах 8-7 и 8-8, экземпляры объектов создаются и возвращаются без внутреннего сопровождения. Это имеет некоторые преимущества и недостатки.

Поскольку `StructureMap` особо не держится за экземпляры, риск появления неумышленных утечек памяти в этом случае намного меньше. Для такого контейнера, как `Castle Windsor`, утечки памяти будут гарантированно возникать, если мы забудем вызывать метод `Release` для всех разрешенных диаграмм объектов. Такого не происходит со `StructureMap`, поскольку, как только объекты выйдут за рамки области применения, они будут уничтожены сборщиком мусора.

Недостаток заключается в том, что устраняемые объекты не могут детерминированно уничтожаться. Так как мы не можем явным образом высвобождать диаграмму объектов, мы не можем уничтожать какие-либо устраняемые объекты. Это означает, что обертывание устраняемых API в неустраняемые сервисы, которое обсуждалось в разделе 6.2.1, становится еще более значимым.

Короче говоря, `StructureMap` позволяет сборщику мусора уничтожать объекты после того, как в вашем коде они выйдут за рамки области применения, но наши собственные классы должны вести себя точно также. Мы не можем полагаться на то, что контейнер или код, осуществляющий вызов, будут уничтожать любые сервисы, поэтому мы должны продолжать использовать устраняемые объекты в рамках единичных методов.

Встроенные в `StructureMap` стили существования представляют собой довольно исчерпывающий набор, который должен удовлетворять самым повседневным нуждам. Помимо этого в редких случаях, когда нам нужен специализированный стиль существования, у нас есть возможность создать свой собственный стиль существования.

Разработка пользовательского стиля существования

В большинстве случаев мы должны уметь выходить из разных ситуаций с помощью исчерпывающего набора стилей существования, уже предоставляемых `StructureMap`, но если у нас имеются особые нужды, можно реализовать пользовательский стиль существования. В данном разделе вы увидите, как это сделать. После краткого обзора `Seam`, который делает возможным создание пользовательского стиля существования, большую часть времени мы потратим на пример.

Понимание API стиля существования

В разделе 11.1.2 "Конфигурирование контейнера" вы уже получили некоторое представление об API стилей существования `StructureMap`. Метод `LifecycleIs` принимает в качестве параметра экземпляр интерфейса `ILifecycle`, который моделирует то, как стили существования взаимодействуют с остальной частью контейнера `StructureMap`:

```

public interface ILifecycle
{
    string Scope { get; }
    void EjectAll();
    IObjectCache FindCache();
}

```

Среди этих трех методов центральным методом является `FindCache`. Он возвращает кэш, который `StructureMap` использует для поиска и вставки объектов, имеющих конкретный стиль существования. Интерфейс `ILifecycle`, главным образом, выступает в роли абстрактной фабрики для экземпляров `IObjectCache`, в которых содержится реализация стиля существования. Этот интерфейс довольно-таки сложен, но его не столь сложно реализовать:

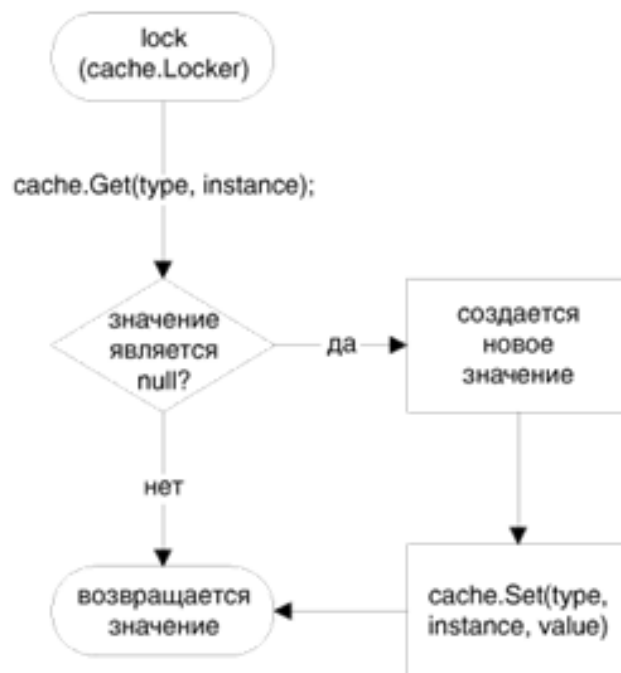
```

public interface IObjectCache
{
    object Locker { get; }
    int Count { get; }
    bool Has(Type pluginType, Instance instance);
    void Eject(Type pluginType, Instance instance);
    object Get(Type pluginType, Instance instance);
    void Set(Type pluginType, Instance instance, object value);
    void DisposeAndClear();
}

```

Большинство методов данного интерфейса имеют дело с поиском, передачей или удалением экземпляра на основании `Type` и `Instance`. Рисунок 11-5 иллюстрирует, каким образом `StructureMap` взаимодействует с реализацией `IObjectCache`.

Рисунок 11-5: `StructureMap` взаимодействует с интерфейсом `IObjectCache`, в первую очередь, вызывая метод `Get` для объекта-кэша. Если кэш возвращает значение, то данное значение используется незамедлительно. В противном случае `StructureMap` создает новое значение и добавляет это значение в кэш перед тем, как его вернуть.



Примечание

Механизм, проиллюстрированный на рисунке 11-5, похож на взаимодействие между Unity и `ILifetimePolicy`, продемонстрированное на рисунке 14-6.

Сначала `StructureMap` пытается получить запрашиваемый экземпляр из метода `Get`. Если этот метод возвращает значение `null` для предоставленных `Type` и `Instance`, то `StructureMap` создаст запрашиваемый экземпляр и перед тем, как его вернуть, добавляет этот экземпляр в кэш посредством метода `Set`.

Давайте на примере рассмотрим, как это работает.

Разработка стиля существования Caching

В данном примере вы будете разрабатывать такой же стиль существования `Caching`, как вы создавали для `Castle Windsor` в разделе 10.2.3 "Разработка пользовательского стиля существования". Короче говоря, этот стиль существования кэширует и повторно использует экземпляры в течение некоторого времени перед тем, как их высвободить.

Предупреждение

Этот шаблонный код не является потоко-безопасным, но соответствующая рабочая реализация должна быть потоко-безопасной, поскольку, скорее всего, несколько потоков будут одновременно пытаться разрешать объекты из контейнера.

Давайте начнем с самой простой составляющей. Следующий листинг демонстрирует реализацию интерфейса `ILifecycle`.

Листинг 11-4: Реализация `ILifecycle`

```
1. public partial class CacheLifecycle : ILifecycle
2. {
3.     private readonly LeasedObjectCache cache;
4.     public CacheLifecycle(ILease lease)
5.     {
6.         if (lease == null)
7.         {
8.             throw new ArgumentNullException("lease");
9.         }
10.        this.cache = new LeasedObjectCache(lease);
11.    }
12.    public void EjectAll()
13.    {
14.        this.FindCache().DisposeAndClear();
15.    }
16.    public IOBJECTCache FindCache()
17.    {
18.        return this.cache;
19.    }
20.    public string Scope
21.    {
22.        get { return "Cache"; }
23.    }
24. }
```

Строка 10: Сохраняет `lease` в пользовательском `cache`

Строка 12, 16, 20: Члены `ILifecycle`

Строка 18: Возвращает пользовательский `cache`

Класс `CacheLifecycle`, как и требуется, реализует интерфейс `ILifecycle`. Для получения экземпляра `ILease` он использует паттерн `Constructor Injection`. Интерфейс `ILease` – локальный вспомогательный (*helper*) интерфейс, который вводится для реализации `CacheLifecycle`. Первоначально этот интерфейс был введен в разделе 10.2.3 "Разработка пользовательского стиля существования" и не имеет никакого отношения ни к `StructureMap`, ни к любому другому DI-контейнеру.

Примечание

Чтобы увидеть пример реализации `ILease`, взгляните на раздел 10.2.3.

Вместо того чтобы сохранять экземпляр `ILease` напрямую в приватном поле, вы незамедлительно оборачиваете его в пользовательскую реализацию интерфейса `IObjectCache` под названием `LeasedObjectCache`. Это тот юж, который возвращается методом `FindCache`.

Примечание

Сравните конструктор из листинга 11-4 с намного более сложным кодом листинга 10-2. Данное сравнение ясно иллюстрирует превосходство `Constructor Injection` над `Method Injection`.

Несмотря на то, что `CacheLifecycle` предоставляет исходный интерфейс `ILifecycle`, истинная реализация обеспечивается при помощи пользовательского класса `LeasedObjectCache`, который реализует интерфейс `IObjectCache`.

`StructureMap` уже предоставляет реализацию `IObjectCache` под названием `MainObjectCache`. К несчастью, `MainObjectCache` не имеет ни одного виртуального члена, который мы могли бы переопределить для того, чтобы реализовать стиль существования `Caching`. Вместо этого мы можем обернуть `MainObjectCache` пользовательским `LeasedObjectCache`. В следующем листинге продемонстрирован конструктор.

Листинг 11-5: Конструирование `LeasedObjectCache`

```
private readonly IObjectCache objectCache;
private readonly ILease lease;
public LeasedObjectCache(ILease lease)
{
    if (lease == null)
    {
        throw new ArgumentNullException("lease");
    }
    this.lease = lease;
    this.objectCache = new MainObjectCache();
}
```

В конструкторе `LeasedObjectCache` вы используете стандартный `Constructor Injection` для того чтобы внедрить экземпляр `ILease`. `LeasedObjectCache` – это `Decorator` для `MainObjectCache`, поэтому вы создаете экземпляр и присваиваете его приватному полю. Обратите внимание на то, что поле `objectCache` объявлено как `IObjectCache`, поэтому вы

могли бы просто расширить класс `LeasedObjectCache` перегруженным конструктором, который позволял бы вам внедрять любую реализацию `IOBJECTCache` из вне.

Комбинация обернутого `IOBJECTCache` и члена `ILease` приближает реализацию класса `LeasedObjectCache` к тривиальной. Следующий листинг демонстрирует реализацию важных методов `Get` и `Set`, а остальная реализация руководствуется тем же самым проектом.

Листинг 11-6: Реализация методов `Get` и `Set`

```
public object Get(Type pluginType, Instance instance)
{
    this.CheckLease();
    return this.objectCache.Get(pluginType, instance);
}
public void Set(Type pluginType, Instance instance, object value)
{
    this.objectCache.Set(pluginType, instance, value);
    this.lease.Renew();
}
private void CheckLease()
{
    if (this.lease.IsExpired)
    {
        this.objectCache.DisposeAndClear();
    }
}
```

Когда `StructureMap` вызывает метод `Get`, вы, в первую очередь, удостоверяетесь в том, что кэш не содержит какие-либо просроченные экземпляры. При возврате этим методом результата вы можете быть уверены в том, что, если обернутый кэш содержит запрашиваемый экземпляр, вы сможете безопасно его вернуть.

Наоборот, при вызове метода `Set` вы тотчас же делегируете этот метод обернутому объекту-кэшу. Поскольку вы понимаете, что `StructureMap` использует `IOBJECTCache` так как показано на рисунке 11-5, вы знаете, что метод `Set` вызывается только тогда, когда контейнер создает новый экземпляр, поскольку ни один экземпляр кэша недоступен. Это означает, что экземпляр, переданный при помощи параметра `value`, представляет собой только что созданный экземпляр, поэтому вы можете безопасно обновить срок аренды.

Вспомогательный метод `CheckLease` вызывается многими реализациями члена `IOBJECTCache` способами, аналогичными методу `Get`. Он игнорирует обернутый кэш, если срок его аренды истек.

Теперь, когда вы знаете, как реализовать пользовательский стиль существования и любые пользовательские зависимости, которыми он может обладать, вам остается только узнать, как его применять.

Конфигурирование компонентов с помощью пользовательского стиля существования

Использовать `CacheLifecycle` при конфигурировании компонента легко, а выполняется это таким же самым способом, которым бы вы конфигурировали любой другой стиль существования:

```
var lease = new SlidingLease(TimeSpan.FromMinutes(1));
var cache = new CacheLifecycle(lease);
container.Configure(r => r
    .For<IIngredient>()
    .LifecycleIs(cache)
    .Use<SauceBéarnaise>());
```

При помощи данного кода контейнер конфигурируется таким образом, что `CacheLifecycle` используется для интерфейса `IIngredient` с одноминутным промежутком. В пределах одной минуты вы можете запросить столько диаграмм объектов, сколько только пожелаете, и при этом будете получать обратно тот же самый `SauceBéarnaise` всякий раз, когда в диаграмме будет содержаться экземпляр `IIngredient`. По окончании одноминутного промежутка последующие запросы будут получать новый экземпляр `SauceBéarnaise`.

Следует отметить, что способ, при помощи которого реализуется `CacheLifecycle`, может следующим образом использоваться для связывания вместе нескольких экземпляров, имеющих один и тот же срок аренды:

```
container.Configure(r =>
{
    r.For<IIngredient>().LifecycleIs(cache).Use<Steak>();
    r.For<ICourse>().LifecycleIs(cache).Use<Course>();
});
```

Это будет приводить к тому, что экземпляры `ICourse` и `IIngredient` будут просрочены и вновь созданы в одно и то же время. Иногда это может быть нужно, а иногда и нет. Альтернативный вариант – использовать два отдельных экземпляра `CacheLifecycle`. Как показывает следующий листинг, такая возможность также позволяет вам использовать два различных таймаута.

Листинг 11-7: Использование разных стилей существования `Cache` для каждого `Instance`

```
container.Configure(r => r
    .For<IIngredient>()
    .LifecycleIs(
        new CacheLifecycle(
            new SlidingLease(
                TimeSpan.FromHours(1)))
    .Use<Steak>());
container.Configure(r => r
    .For<ICourse>()
    .LifecycleIs(
        new CacheLifecycle(
            new SlidingLease(
                TimeSpan.FromMinutes(15)))
    .Use<Course>());
```

Первый `cache` задается с одночасовым таймаутом. Не важно, на сколь долгий или сколь малый промежуток времени вам необходим `IIngredient`, в течение одночасового периода вы будете получать один и тот же экземпляр. По истечении одного часа старый экземпляр перестает учитываться, а новый экземпляр беспорядочно используется в течение следующего часа.

`Cache` для `ICourse` – это другой экземпляр, сконфигурированный с 15-минутным таймаутом. В течение этих 15 минут вы будете получать один и тот же экземпляр, но когда они истекут, будет использоваться новый экземпляр. Стоит отметить, что даже

когда время `ICourse` истекает, `IIngredient` продолжает существовать благодаря своему более длительному сроку аренды. Несмотря на то, что `ICourse` и `IIngredient` используют один и тот же стиль существования `type`, они имеют разные расписания.

В листинге 11-7 вы использовали разные таймауты, но при этом в обоих случаях применяли тип `SlidingLease`. Это не является обязательным условием, вы могли бы использовать две совершенно разные реализации `ILease` для каждого экземпляра.

Реализация пользовательского стиля существования для `StructureMap` не столь сложна. В теории она может казаться довольно сложной, но если бы вы рассмотрели этот процесс в интегрированной среде разработки (IDE), то быстро поняли бы, что реализация пользовательского стиля существования состоит всего лишь из двух классов, в которых самый сложный метод (`CheckLease`) имеет только один оператор `if` и состоит из двух строк кода.

Все-таки необходимость реализации пользовательского стиля существования для `StructureMap` должна встречаться довольно редко. Искрывающий набор встроенных стилей существования должен удовлетворять нашим повседневным нуждам.

На этом мы завершаем наше путешествие по управлению жизненным циклом в рамках `StructureMap`. Компоненты должны конфигурироваться при помощи сочетания стилей существования, и это справедливо, даже когда мы регистрируем составные реализации одной и той же абстракции. Нам еще нужно будет рассмотреть то, как работать с составными компонентами, поэтому давайте направим наше внимание в этом направлении.

11.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями.
- Используются Decorator'ы.

В данном разделе мы рассмотрим каждую из этих ситуаций и увидим, как StructureMap поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь регистрировать и разрешать компоненты даже тогда, когда в дело вступают составные реализации одной и той же абстракции.

Давайте сначала рассмотрим то, как можно обеспечить более разветвленное управление, нежели то, которое предоставляет нам автоматическая интеграция (Auto-Wiring).

Выбор из составных кандидатов

Автоматическая интеграция – удобный и мощный инструмент, но предоставляет нам меньшие возможности контроля. Пока все абстракции преобразуются в конкретные типы отдельно друг от друга, никаких трудностей не возникает, но как только мы вводим большее количество реализаций для одного и того же интерфейса, возникает неопределенность.

Давайте для начала повторим, как StructureMap работает с составными регистрациями одной и той же абстракции.

Конфигурирование составных реализаций одного и того же плагина

Как вы уже видели в разделе 11.1.2 "Конфигурирование контейнера", вы можете конфигурировать составные плагины одного и того же сервиса:

```
container.Configure(r =>
{
    r.For<IIIngredient>().Use<SauceBéarnaise>();
    r.For<IIIngredient>().Use<Steak>();
});
```

Данный пример кода регистрирует как класс `Steak`, так и класс `SauceBéarnaise` вместе с плагином `IIIngredient`. Выигрывает последняя реализация, поэтому, если вы разрешаете `IIIngredient` посредством `container.GetInstance<IIIngredient>()`, вы получите

экземпляр `Steak`. Тем не менее, вызов `container.GetAllInstances<IIIngredient>()` вернет `IList<IIIngredient>`, который содержит как `Steak`, так и `SauceBéarnaise`. То есть, последующие конфигурации не позабыты, но их трудно получить.

Подсказка

Выигрывает последняя конфигурация данного типа. Она задает экземпляр по умолчанию для этого типа.

Если есть сконфигурированные экземпляры плагина, которые не могут быть разрешены при вызове `GetAllInstances`, `StructureMap` выдает исключение, объясняющее, что есть зависимости, которые не удовлетворяют заданным условиям. Это сообразно поведению метода `GetInstance`, но отличается от поведения `Castle Windsor` или `MEF`.

Следующий листинг демонстрирует, как можно обеспечить появление подсказок, которые впоследствии могут использоваться для осуществления выбора между различными сконфигурированными экземплярами.

Листинг 11-8: Присваивание имен экземплярам

```
container.Configure(r =>
{
    r.For<IIIngredient>()
        .Use<SauceBéarnaise>()
        .Named("sauce");
    r.For<IIIngredient>()
        .Use<Steak>()
        .Named("meat");
});
```

Вы можете присвоить каждому сконфигурированному экземпляру уникальное имя, которое в дальнейшем можете применять для того, чтобы отличать друг от друга схожие экземпляры.

При наличии именованных экземпляров из листинга 11-8, вы можете разрешать и `Steak`, и `SauceBéarnaise` следующим образом:

```
var meat = container.GetInstance<IIIngredient>("meat");
var sauce = container.GetInstance<IIIngredient>("sauce");
```

Обратите внимание на то, что вы передаете тот же самый идентификатор, который использовали, когда присваивали имя экземпляру при конфигурировании.

Предполагая, что вам всегда следует разрешать сервисы в единственном `Composition Root`, вы не должны ожидать, что на данном уровне столкнетесь с такой неопределенностью.

Подсказка

Если вы обнаружили, что вызываете метод `GetInstance` с конкретным идентификатором, подумайте, можете ли вы изменить свой подход так, чтобы он стал менее неопределенным.

Можно использовать именованные экземпляры для того, чтобы сделать выбор между составными альтернативами при конфигурировании зависимости для данного плагина.

Конфигурирование именованных зависимостей

Для тех же целей, которые обеспечивает автоматическая интеграция, нам иногда нужно переопределять обычное поведение, чтобы обеспечить более расширенный контроль над тем, какие зависимости, куда направляются. Иногда возникают ситуации, когда нам нужно обращаться к неопределенному API. В качестве примера рассмотрите приведенный ниже конструктор:

```
public ThreeCourseMeal(ICourse entrée,  
    ICourse mainCourse, ICourse dessert)
```

В данном примере у вас имеется три идентично типизированных зависимости, каждая из которых представляет сущность, отличную от остальных. В большинстве случаев вам необходимо преобразовать каждую зависимость к отдельному типу. Следующий листинг демонстрирует, как вы могли бы решить конфигурировать преобразования `ICourse`.

Листинг 11-9: Конфигурирование именованных `course`'ов

```
container.Configure(r => r  
    .For<ICourse>()  
    .Use<Rillettes>()  
    .Named("entrée"));  
container.Configure(r => r  
    .For<ICourse>()  
    .Use<CordonBleu>()  
    .Named("mainCourse"));  
container.Configure(r => r  
    .For<ICourse>()  
    .Use<MousseAuChocolat>()  
    .Named("dessert"));
```

Как и в листинге 11-8 вы регистрируете три именованных компонента, преобразуя `Rillettes` в экземпляр под названием `"entrée"`, `CordonBleu` – в экземпляр с именем `"mainCourse"`, а `MousseAuChocolat` – в экземпляр под названием `"dessert"`.

При такой конфигурации вы теперь можете регистрировать класс `ThreeCourseMeal` так, как показано в следующем листинге.

Листинг 11-10: Переопределение автоматической интеграции

```
container.Configure(r => r  
    .For<IMeal>()  
    .Use<ThreeCourseMeal>()  
    .Ctor<ICourse>("entrée").Is(i =>  
        i.TheInstanceNamed("entrée"))  
    .Ctor<ICourse>("mainCourse").Is(i =>  
        i.TheInstanceNamed("mainCourse"))  
    .Ctor<ICourse>("dessert").Is(i =>  
        i.TheInstanceNamed("dessert")));
```

Как обычно, выражение конфигурирования вы начинаете с преобразования интерфейса `IMeal` к конкретному `ThreeCourseMeal`. Но в дальнейшем вы расширяете выражение при помощи метода `Ctor`. Метод `Ctor` (сокращение от `constructor`) позволяет вам выражать то, как должен преобразовываться параметр конструктора данного типа. В случае, когда для данного типа существует только один параметр, можно использовать перегрузку, в которой вам не приходится передавать имя параметра. Тем не менее, поскольку

`ThreeCourseMeal` имеет три параметра `ICourse`, вам необходимо идентифицировать параметр по его имени, `"entrée"`.

Метод `Ctor` возвращает объект, который позволяет вам задать то, как будет заполняться параметр конструктора. Метод `Is` позволяет вам использовать `IInstanceExpression<ICourse>` для отбора именованного экземпляра, что является еще одним примером паттерна `Nested Closure`. Эти же выражения вы в дальнейшем можете повторить и для следующих двух параметров.

Примечание

В этом примере я присвоил экземплярам конфигурации такие же имена, как и параметрам, но это не является необходимым. Я мог назвать экземпляры так, как мне пожелается, несмотря на то, что имена параметров, очевидно, привязаны к именам реальных параметров конструктора.

Предупреждение

Идентифицировать параметры по их именам удобно, но это не приводит к безопасному рефакторингу. Если мы переименуем параметр, мы можем разрушить конфигурацию (это зависит от вашего инструмента рефакторинга).

Переопределение автоматической интеграции путем явного преобразования параметров в именованные экземпляры – общепринятый вариант работы. Сделать это мы можем, даже если мы конфигурируем именованные экземпляры с помощью одного выражения, а конструктор – с помощью совершенно другого выражения, поскольку единственная идентификация, связывающая именованный экземпляр с параметром, – это имя. Так сделать можно всегда, но, когда нам нужно управлять большим количеством имен, такой вариант может оказаться хрупким.

Интеграция ссылок на экземпляры

Всякий раз, когда у нас есть возможность определить экземпляры и конструктор в одном выражении, мы можем сделать это более изящно. Ниже приведенный листинг показывает, как это можно сделать.

Листинг 11-11: Использование ссылок на экземпляры для переопределения автоматической интеграции

```
1. container.Configure(r =>
2. {
3.     var entrée =
4.         r.For<ICourse>().Use<Rillettes>();
5.     var mainCourse =
6.         r.For<ICourse>().Use<CordonBleu>();
7.     var dessert =
8.         r.For<ICourse>().Use<MousseAuChocolat>();
9.     r.For<IMeal>()
10.        .Use<ThreeCourseMeal>()
11.        .Ctor<ICourse>("entrée").Is(entree)
12.        .Ctor<ICourse>("mainCourse").Is(mainCourse)
13.        .Ctor<ICourse>("dessert").Is(dessert);
14. });
```

Строка 3,5,7: Ссылки на экземпляры

Строка 11-13: Использование ссылок на экземпляры

До настоящего момента мы игнорировали тот факт, что типичная цепочка методов `For/Use` возвращает какой-то результат, поскольку для нас это было бесполезно. Но возвращаемые значения являются экземплярами `SmartInstance<T>`, которые можно использовать в качестве ссылок на конфигурации, которые вы делали до этого. Вместо имен экземпляров, которые вам пришлось использовать в листинге 11-10, вы можете использовать эти ссылки прямо с одной из множества перегрузок метода `Is`, сравнивая каждую локальную переменную с соответствующим именованным параметром конструктора.

Несмотря на то, что данная возможность позволяет нам избавиться от имен экземпляров, у нас все еще остаются "волшебные" строки, которые идентифицируют параметры конструктора. Данное API зависит от текстового совпадения между конфигурацией и именами параметров, поэтому оно является недолговечным, и такого API стоит по возможности избегать. Если мы чувствуем, что это нужно использовать только для того, чтобы справиться с неопределенностью, наилучшее решение – разработать API, которое позволит нам избавиться от этой неопределенности. Чаще всего это приводит еще и к лучшему дизайну в целом.

В следующем разделе вы увидите, как можно использовать менее неопределенный и более гибкий подход, при котором разрешается любое количество блюд в обеде. Для этого вы должны узнать, как `StructureMap` работает со списками и последовательностями.

Интеграция последовательностей

В разделе 10.3.2 "Разработка пользовательского стиля существования" мы обсуждали, как выполнить рефакторинг явного класса `ThreeCourseMeal` к более универсальному классу `Meal`, который обладает приведенным ниже конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

В данном разделе мы будем рассматривать то, как можно сконфигурировать `StructureMap` таким образом, чтобы интегрировать экземпляры `Meal` с соответствующими зависимостями `ICourse`. После рассмотрения этого вопроса вы должны будете приобрести хорошее понимание тех возможностей, которые доступны в случае необходимости конфигурирования экземпляров, имеющих последовательности зависимостей.

Автоматическое интегрирование последовательностей

`StructureMap` довольно хорошо разбирается в последовательностях. Если мы хотим использовать все сконфигурированные экземпляры данного плагина, то в этом случае нам нужна именно автоматическая интеграция. К примеру, при наличии сконфигурированных экземпляров `ICourse` из листинга 11-9 можно сконфигурировать плагин `IMeal` следующим образом:

```
container.Configure(r => r.For<IMeal>().Use<Meal>());
```

Обратите внимание на то, что в данном примере приведено совершенно стандартное преобразование абстракции к конкретному типу. `StructureMap` будет автоматически понимать конструктор `Meal` и определять, что подходящее направление действий – разрешение всех экземпляров `ICourse`. При разрешении `IMeal` вы получаете экземпляр

Meal наряду с экземплярами `ICourse`, описанными в листинге 11-9: `Rillettes`, `CordonBleu` и `MousseAuChocolat`.

Примечание

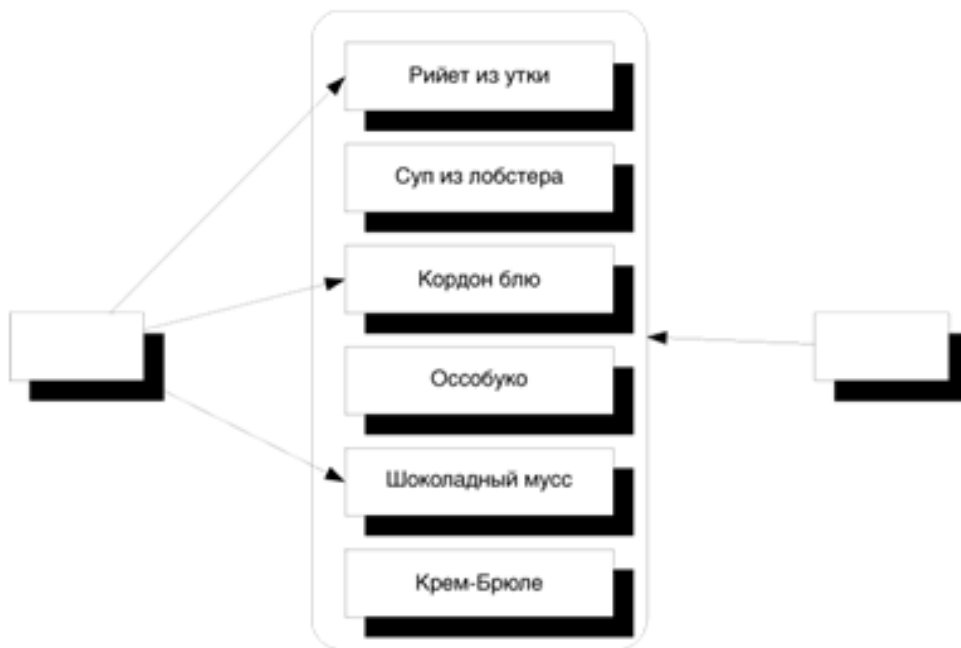
Сравните удобство автоматической интеграции последовательностей в `StructureMap` и раздел 10.3.2 "Разработка пользовательского стиля существования", который демонстрирует, как сложно обеспечить такую же функциональность в `Castle Windsor`.

`StructureMap` автоматически обрабатывает последовательности и, пока мы не укажем обратное, он делает то, что мы и предполагали: разрешает последовательность зависимостей всех зарегистрированных экземпляров данного типа. Только в тех случаях, когда нам необходимо отобразить всего лишь несколько экземпляров из огромного набора, нам нужно выполнить большее количество действий. Давайте посмотрим, как это сделать.

Отбор нескольких экземпляров из большого набора

Чаще всего используемая `StructureMap` по умолчанию стратегия внедрения всех экземпляров является корректной методикой, но, как показывает рисунок 11-6, могут возникать ситуации, когда нам нужно отобразить несколько сконфигурированных экземпляров из большого набора всех сконфигурированных экземпляров.

Рисунок 11-6: В ситуации, приведенной слева, мы хотим явно выбрать только конкретные зависимости из большого перечня всех сконфигурированных экземпляров. Данная ситуация отличается от той, которая приведена справа – здесь мы без разбора отбираем все экземпляры.



Когда мы ранее позволили `StructureMap` автоматически интегрировать все сконфигурированные экземпляры, это привело к ситуации, изображенной в правой части рисунка 11-6. Если мы хотим сконфигурировать экземпляр так, как это показано в левой части рисунка, мы должны явным образом указать то, какие экземпляры должны использоваться.

Когда у нас есть возможность конфигурировать зависимости и потребителя в одном вызове метода `Configure`, мы можем использовать ссылочные экземпляры так, как вы это видели в листинге 11-11. Следующий листинг демонстрирует эквивалентную конфигурацию сценария, при котором в конструкторе предполагается наличие последовательности зависимостей.

Листинг 11-12: Использование ссылок на экземпляры для внедрения последовательностей

```
container.Configure(r =>
{
    var entrée = r.For<ICourse>().Use<Rillettes>();
    var entrée1 = r.For<ICourse>().Use<LobsterBisque>();
    var mainCourse = r.For<ICourse>().Use<CordonBleu>();
    var dessert = r.For<ICourse>().Use<MousseAuChocolat>();
    r.For<IMeal>().Use<Meal>()
        .EnumerableOf<ICourse>()
        .Contains(entrée, mainCourse, dessert);
});
```

Аналогично коду из листинга 11-11 вы присваиваете переменную каждому экземпляру, который возвращается методом `Use`. Обратите внимание на то, что вы конфигурируете *четыре* экземпляра `ICourse`, даже если вы используете только три из них для экземпляра `IMeal`. Тем не менее, вам может понадобиться преобразовать `ICourse` к `LobsterBisque` для некоторых других целей, непродemonстрированных здесь. Поскольку вы не используете результирующую переменную `entrée1`, вы могли бы ее полностью опустить, но я решил ее включить для того, чтобы код был последовательным.

Поскольку конструктор `Meal` принимает в качестве входного параметра `IEnumerable<ICourse>`, вы можете использовать метод `EnumerableOf` для обозначения последовательности экземпляров `ICourse`, явно определенных в методе `Contains`, где вы передаете три ссылки на экземпляры, которые собираетесь использовать.

Этот подход хорошо использовать тогда, когда есть возможность сконфигурировать все соответствующие экземпляры `ICourse` в том же блоке кода, в котором выполняется конфигурация `IMeal`. Это не всегда возможно. Может случаться, что конфигурация экземпляров `ICourse` разделяется на несколько различных регистров из различных сборок. В этом случае мы можем прибегнуть к способу обращения к ним по имени. Следующий листинг демонстрирует один из таких способов.

Листинг 11-13: Внедрение именованных экземпляров в последовательности

```
container.Configure(r => r
    .For<IMeal>()
    .Use<Meal>()
    .EnumerableOf<ICourse>().Contains(i =>
    {
        i.TheInstanceNamed("entrée");
        i.TheInstanceNamed("mainCourse");
        i.TheInstanceNamed("dessert");
    }));
```

При наличии набора именованных экземпляров, аналогичных тем, которые были созданы в листинге 11-9, вы можете ссылаться на каждый именованный экземпляр при конфигурировании экземпляра `IMeal`. Как и в листинге 11-12, вы используете цепочку методов `EnumerableOf/Contains` для обозначения последовательности зависимостей. На

этот момент у вас нет переменных `Instance`, поэтому вы должны искать их по имени. Перегрузка метода `Contains` дает вам возможность использовать `Nested Closure`, который объявляет то, какие именованные экземпляры вы хотите внедрить в экземпляр `Meal`.

`StructureMap` понимает последовательности. До тех пор пока не появляется необходимость отбирать только некоторые экземпляры из всех плагинов данного типа, `StructureMap` автоматически все делает правильно. Автоматическая интеграция применяется не только для единичных экземпляров, но также и для последовательностей. А контейнер преобразует последовательность во все сконфигурированные экземпляры соответствующего типа.

Потребители, которые полагаются на последовательности зависимостей, могут быть самыми интуитивно понятными пользователю составными экземплярами одной и той же зависимости. Но перед тем как мы полностью отойдем от данной темы, нам необходимо рассмотреть последний (и, возможно, слегка неожиданный) случай, когда в дело вступают составные экземпляры.

Интеграция `Decorator`'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования `Decorator` полезен при реализации сквозных сущностей. По определению `Decorator`'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам `Decorator` и обернутый тип. Если бы мы помещали `Decorator`'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных регистраций одного и того же сервиса. В отличие от предыдущих разделов эти регистрации не являются концептуально равносильными, а зависят друг от друга. В данном разделе вы увидите, как сконфигурировать `StructureMap` таким образом, чтобы он мог работать с этим паттерном. Существует множество способов конфигурирования `Decorator`'а, и мы рассмотрим три различных способа достижения одного и того же результата. Каждый из них имеет свои собственные достоинства и недостатки.

Создание обертки при помощи ссылок на экземпляры

Давайте посмотрим, как можно сконфигурировать класс `Breading`, который является `Decorator`'ом `IIngredient`. Этот класс использует `Constructor Injection` для получения экземпляра, который необходимо обернуть:

```
public Breading(IIngredient ingredient)
```

Для того чтобы сделать панированную телячью котлету, вам хотелось бы обернуть `VealCutlet` (еще один `IIngredient`) в класс `Breading`. Один из способов это сделать — использовать ссылки на экземпляры в рамках единичного метода `Configure`:

```
container.Configure(r =>
{
    var cutlet = r.For<IIngredient>().Use<VealCutlet>();
    r.For<IIngredient>().Use<Breading>()
        .Ctor<IIngredient>().Is(cutlet);
});
```

Как вы уже видели в листингах 11-11 и 11-12, можно использовать возвращаемое методом `Use` значение для того, чтобы перехватить ссылку на экземпляры. Переменная `cutlet` представляет собой сконфигурированное преобразование `IIngredient` в `VealCutlet`. И вы можете использовать этот факт для того, чтобы объявить, что эта переменная и есть `Instance`, который можно использовать в параметре `IIngredient` конструктора класса `Breeding`. Поскольку выигрывает последняя конфигурация, `Breeding Instance` на данный момент является используемым по умолчанию `Instance`.

Когда вы попросите контейнер разрешить `IIngredient`, он вернет объект, основанный на используемом по умолчанию `Instance`. Это и есть `Breeding Instance`, в котором вы предоставили дополнительный намек на то, что он должен разрешить `cutlet Instance` для параметра `IIngredient` класса `Breeding`. В результате мы получаем экземпляр `Breeding`, содержащий экземпляр `Cutlet`.

Передача объектов безопаснее передачи строк, поэтому нам следует отдавать предпочтение использованию данной методики всякий раз, когда у нас есть возможность конфигурировать `Decorator` и обернутый тип в одном выводе метода. Тем не менее, это не всегда возможно.

Создание обертки при помощи именованных экземпляров

Иногда нам приходится прибегать к именам `Instance`, потому что мы конфигурируем вложенные соучастники в разных вызовах метода, возможно, даже в разных регистрах, реализованных в отдельных библиотеках. В таких случаях мы не можем передавать объекты, а должны полагаться на строки, даже если их легко испортить.

Давайте предположим, что вы уже сконфигурировали `VealCutlet` следующим образом:

```
container.Configure(r => r
    .For<IIngredient>()
    .Use<VealCutlet>()
    .Named("cutlet"));
```

Поскольку вы знаете, что имя экземпляра — `cutlet`, вы можете использовать его для конфигурирования класса `Breeding`:

```
container.Configure(r => r
    .For<IIngredient>()
    .Use<Breeding>()
    .Ctor<IIngredient>()
    .Is(i => i.TheInstanceNamed("cutlet")));
```

Как и в листингах 11-10 и 11-13 вы используете перегрузку метода `Is`, которая дает вам возможность предоставить блок кода, идентифицирующий именованный экземпляр. И снова вы видите паттерн `Nested Closure` в действии.

Если вы сравните два предыдущих примера, то не заметите, что они похожи. В обоих случаях вы использовали метод `Ctor<T>`, олицетворяющий параметр конструктора. Единственное отличие заключается в том, как вы идентифицируете параметр с методом `Is`.

Цепочка методов `Ctor/Is` имеет преимущество, заключающееся в том, что мы можем использовать ее для указания единственного параметра конструктора, даже если

рассматриваемый конструктор имеет более одного параметра. Все параметры, которые мы не сможем сконфигурировать при помощи метода `Ctor`, будут автоматически интегрированы на основании алгоритмов, используемых `StructureMap` по умолчанию. Это полезно, если мы хотим сконфигурировать только один из нескольких параметров.

Тем не менее, такой вариант не является строго типизированным. Нет ни какой гарантии, что рассматриваемый конструктор имеет параметр указанного типа. Он мог его иметь, но потом мы изменили конструкцию, и теперь он принимает параметры другого типа. Компилятор об этом не знает, поскольку, когда мы вызываем метод `Ctor` с конкретным аргументом типа, он доверяет нам.

Другой вариант предлагает более строго типизированный подход.

Создание обертки при помощи делегатов

Вместо того чтобы ссылаться на параметр конструктора по типу или имени, мы можем записать строго типизированный блок кода, в котором используется конструктор. Несмотря на то, что такой подход также имеет недостатки, к которым мы позднее вернемся, его достоинство заключается в том, что он строго типизирован и поэтому более безопасен с точки зрения момента конструирования.

Это выглядит слегка абстрактным, поэтому давайте рассмотрим пример, демонстрирующий, как сконфигурировать `Cotoletta` следующим образом:

```
container.Configure(r => r
    .For<IIIngredient>().Use<VealCutlet>()
    .EnrichWith(i => new Breeding(i)));
```

Метод `EnrichWith` — член `generic`-класса `SmartInstance<T>`, который возвращается методом `Use`. В данном случае вы вызываете метод `Use` с аргументом типа `VealCutlet`. Этот метод возвращает экземпляр `SmartInstance<VealCutlet>`. Метод `EnrichWith` принимает в качестве параметра делегат, который, в свою очередь, принимает в качестве входного параметра `VealCutlet` и возвращает объект.

Вы можете сравнить этот делегат с блоком кода, который принимает в качестве входного параметра `VealCutlet`. Компилятор делает вывод, что переменная `i` — это экземпляр `VealCutlet`, поэтому вы теперь можете реализовать блок кода посредством вызова конструктора `Breeding` с переменной `VealCutlet`.

Когда вы попросите контейнер разрешить `IIIngredient`, он сначала создаст экземпляр `VealCutlet`, а затем передаст этот экземпляр в качестве входной информации в блок кода, который вы определили с помощью метода `EnrichWith`. При выполнении блока кода экземпляр `VealCutlet` передается в конструктор `Breeding`, и возвращается экземпляр `Breeding`.

Достоинство данного подхода заключается в том, что в блоке кода вы записываете код, в котором используется конструктор `Breeding`. Этот код является такой же строкой кода, как и любая другая строка кода, поэтому она проверяется компилятором. Это обеспечивает вас уверенностью в том, что если метод `Configure` компилируется, то `VealCutlet` будет корректно обернута.

Несмотря на то, что строгая типизированность безопаснее, ее сложнее поддерживать. Если вы впоследствии решите добавить еще один параметр в конструктор `Breading`, блок кода больше не будет компилироваться, и вы должны будете вручную справиться с данной проблемой. Это было бы не нужно, если бы вы использовали метод `ctor<T>`, поскольку `StructureMap` смог бы отсортировать новый параметр благодаря автоматической интеграции.

Как вы уже видели, существует несколько способов конфигурирования `Decorator`'ов. Строго типизированный подход более безопасен, но для него может потребоваться более сложное сопровождение. Более слабо типизированное API – более гибкое, и дает `StructureMap` возможность справиться с изменениями нашего API, но ценой менее слабой типовой безопасности.

Примечание

В данном разделе мы не обсуждали механизм перехвата во время выполнения. Несмотря на то, что `StructureMap` имеет `Seam`'ы, которые разрешают механизм перехвата, он не обладает встроенной поддержкой динамически создаваемых прокси. Можно использовать эти `Seam`'ы для того, чтобы использовать другую библиотеку (например, `Castle Dynamic Proxy`) для создания таких классов. Но поскольку они не являются частью `StructureMap`, их обсуждение выходит за рамки данной главы.

`StructureMap` позволяет нам работать с составными экземплярами несколькими различными способами. Мы можем конфигурировать экземпляры как альтернативы друг другу, как пиры, разрешенные в виде последовательностей, или как иерархические `Decorator`'ы. В большинстве случаев `StructureMap` поймет, что делать, но мы всегда можем явно определить, каким образом скомпоновать сервисы, если нам нужен более явный контроль.

Кроме того, это может быть необходимо в тех случаях, когда нам приходится иметь дело с API, которые отклоняются от `Constructor Injection`. До настоящего момента вы наблюдали, как конфигурировать экземпляры, включая то, как задавать стили существования и как работать с составными компонентами. Но до настоящего момента вы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют `Constructor Injection`. Это не всегда так. В следующем разделе мы сделаем обзор того, как работать с классами, экземпляры которых должны создаваться особыми способами.

11.4. Конфигурирование сложных API

До настоящего момента мы рассматривали то, как можно конфигурировать компоненты, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры, например, StructureMap, могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей.

Все становится менее понятным, когда API не столь хорошо функционируют. В данном разделе вы увидите, как работать с простейшими аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Давайте начнем с рассмотрения классов, которые принимают в качестве параметров простейшие типы, например, строки и целые числа.

Конфигурирование простейших зависимостей

Пока мы внедряем абстракции в потребителей, все в порядке. Но данный процесс усложняется, если конструктор зависит от простейшего типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве компонента контейнера не имеет особого смысла, а в StructureMap это и вовсе не работает. Если мы попытаемся разрешить компонент с простейшей зависимостью, мы получим исключение, даже если простейший тип был до этого зарегистрирован.

Рассмотрите в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются `code smell`'ами и их нужно преобразовывать в полиморфные классы (имеющие разное состояние). Тем не менее, для данного примера они вполне нам подходят.

Необходимо явным образом сообщить StructureMap о том, как разрешать параметр конструктора `spiciness`. Приведенный ниже пример демонстрирует, как можно использовать метод `Ctor<T>` для того, чтобы явным образом предоставить значение для параметра конструктора:

```

container.Configure(r => r
    .For<ICourse>()
    .Use<ChiliConCarne>()
    .Ctor<Spiciness>()
    .Is (Spiciness.Hot) );

```

В разделе 11.3 "Работа с составными компонентами" вы не раз видели, каким образом можно использовать метод `Ctor<T>` для того, чтобы переопределить автоматическую интеграцию для конкретного параметра конструктора. В данном разделе вы косвенным образом устанавливаете, что, подразумевается, что конструктор `ChiliConCarne` имеет только один параметр `Spiciness`. В противном случае вызов метода `Ctor<Spiciness>()` будет неоднозначным, и вам придется передавать также и имя параметра.

Метод `Ctor<T>` возвращает `SmartInstance<T>`, который имеет разнообразные методы. Существует 5 перегрузок метода `Is`, а одна из них дает возможность предоставить экземпляр соответствующего типа. Аргументом типа `T` в данном случае является `Spiciness`, поэтому вы предоставляете `Spiciness.Hot` в качестве конкретного значения.

Как мы уже обсуждали в разделе 11.3 "Работа с составными компонентами", использование метода `Ctor<T>` имеет свои преимущества и недостатки. Если нам нужна более строго типизированная конфигурация, которая вызывает конструктор или статическую фабрику, мы также можем это сделать.

Создание объектов с помощью блока кода

Экземпляры некоторых классов не могут создаваться посредством открытых конструкторов. Поэтому в таких случаях для создания экземпляров типов мы должны использовать некоторого рода фабрику. Это всегда проблематично для DI-контейнеров, поскольку по умолчанию они присматривают открытые конструкторы.

Рассмотрим приведенный ниже пример конструктора для открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Несмотря на то, что класс `JunkFood` является открытым классом, его конструктор является внутренним. Очевидно, экземпляры `JunkFood` должны создаваться посредством статического класса `JunkFoodFactory`:

```

public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}

```

С точки зрения `StructureMap`, это API проблематично, поскольку не существует каких-либо определенных и опубликованных соглашений, касающихся статических фабрик. Поэтому ему необходима помощь, которую вы можете предоставить посредством блока кода, который `StructureMap` может исполнить для того, чтобы создать экземпляр:


```

container.Configure(r => r
    .For<IMeal>()
    .Use(() =>
        JunkFoodFactory.Create("chicken meal")));

```

К этому времени цепочка методов `For/Use` должна быть вам уже знакома. Тем не менее, в данном случае вы используете перегрузку метода `Use`, отличную от той, которую вы использовали ранее. Эта перегрузка позволяет вам передавать `Func<IMeal>`, что вы делаете посредством блока кода, который вызывает статический метод `Create` класса `JunkFoodFactory`.

Подсказка

Если вы хотите разрешить класс `ChiliConCarne` из раздела 11.4.1 "Конфигурирование простейших зависимостей" строго типизированным способом, то можете использовать данную перегрузку `Use` для непосредственного вызова конструктора.

После завершения написания кода, который создает экземпляр, можете ли вы ответить, почему такой подход в любом случае лучше непосредственного вьюа кода? Используя блок кода внутри оператора `For/Use`, вы кое-что, таким образом, приобретаете:

- Преобразуете `IMeal` к `JunkFood`.
- Стиль существования все еще можно конфигурировать. Несмотря на то, что для создания экземпляра будет вызываться блок кода, он может и не вызываться всякий раз, когда запрашивается данный экземпляр. За исключением тех случаев, когда вы используете стиль существования `Unique`, иногда вместо блока кода может использоваться кэшированный экземпляр.

В общем, существует пять различных перегрузок метода `Use`. Для задания конкретного типа можно использовать `generic`-версию, но остальные перегрузки позволяют нам передавать конкретный экземпляр или блоки кода, создающие конкретный экземпляр.

Последним общепринятым отклонением от `Constructor Injection`, которое мы рассмотрим в данном разделе, является `Property Injection`.

Интеграция с помощью Property Injection

`Property Injection` является не столь четко определенной формой механизма внедрения зависимостей, поскольку компилятор не принуждает вас присваивать значение свойству, доступному для записи. Это справедливо и для `StructureMap`, который будет пропускать доступные для записи свойства до тех пор, пока мы явно не попросим его что-нибудь с ними сделать.

Рассмотрите приведенный ниже класс `CaesarSalad`:

```

public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}

```

Согласно общепринятому заблуждению в салат "Цезарь" входит курица. По существу салат "Цезарь" является салатом, но, поскольку с курицей он вкуснее, многие рестораны предлагают возможность добавления в него курицы в качестве дополнительного

ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Если вы конфигурируете только класс `CaesarSalad`, явно не обращаясь к свойству `Extra`, то этому свойству не будет присвоено значение. Вы все равно можете разрешать экземпляр, но свойство `Extra` будет иметь значение по умолчанию, которое ему присвоил конструктор (если только это имеет место).

Существует несколько способов, с помощью которых можно сконфигурировать `CaesarSalad` таким образом, чтобы свойство `Extra` заполнялось соответствующим образом. Один из таких способов – использование ссылок на экземпляры. Этот способ вы уже ранее несколько раз видели в данной главе:

```
container.Configure(r =>
{
    var chicken = r.For<IIngredient>().Use<Chicken>();
    r.For<ICourse>().Use<CaesarSalad>()
        .Setter<IIngredient>().Is(chicken);
});
```

Из нескольких предыдущих примеров вы можете вспомнить, что метод `Use` возвращает `Instance`, который вы можете помнить как переменную. В листинге 11-10 и во многих последующих примерах вы использовали метод `Ctor<T>` для того, чтобы обозначить параметр конструктора определенного типа. Метод `Setter<T>` работает аналогичным образом, но только для свойств. Вы передаете экземпляр `chicken` в метод `Is`, чтобы заставить `StructureMap` присвоить значение свойству при построении экземпляра.

Когда вы будете на основании этой конфигурации разрешать `ICourse`, вы получите обратно экземпляр `CaesarSalad`, свойству `Extra` которого будет присвоен экземпляр `Chicken`. Это предоставляет вам возможность дифференцированного управления конкретными свойствами конкретных типов. API, которое в большей степени основано на соглашениях, предоставляет нам возможность утверждать, что мы хотим, чтобы `StructureMap` использовал все свойства данного типа для `PropertyInjection`. К примеру, мы могли бы установить, что все заданные свойства `IIngredient` должны внедряться вместе с соответствующим экземпляром.

В случае `CaesarSalad` вы можете выразить это следующим образом:

```
container.Configure(r =>
    r.For<IIngredient>().Use<Chicken>());
container.Configure(r =>
    r.For<ICourse>().Use<CaesarSalad>());
container.Configure(r =>
    r.FillAllPropertiesOfType<IIngredient>());
```

Благодаря методу `FillAllPropertiesOfType` вы можете установить, что всем доступным для записи свойствам типа `IIngredient` должно быть присвоено значение. `StructureMap` будет использовать экземпляр по умолчанию, сконфигурированный для `IIngredient`, поэтому при разрешении `ICourse` вы получите экземпляр `CaesarSalad` со свойством `Extra` равным `Chicken`.

`FillAllPropertiesOfType` будет заполнять любое доступное для записи свойство указанного типа, поэтому, если другие конкретные классы также обладают доступными

для записи свойствами того же типа, в них также будут внедрены сконфигурированные экземпляры. Это может быть целесообразным, если мы руководствуемся соглашением, которое использует Property Injection для определенных типов.

В данном разделе вы увидели, как можно использовать StructureMap для работы с более сложными API создания экземпляров. Для того чтобы задать конкретные экземпляры или блоки кода, которые будут применяться для создания экземпляров, можно использовать множество перегрузок методов Use и Is. Вы также видели, что Property Injection можно конфигурировать непосредственно при конфигурировании экземпляров или в виде соглашения для конкретного типа.

11.5. Резюме

Данная глава представляет собой дегустационное меню StructureMap и его возможностей. Мы соотносим принципы и паттерны остальной части книги с API контейнера StructureMap. StructureMap – это старейший из доступных в .NET DI-контейнеров, но этот факт ничего не говорит ни о его возрасте, ни о доминирующем использовании *вложенных замыканий* (Nested Closures), ни о его конфигурационном API, которое обладает свойством типовой безопасности, ни о возможности поиска типов в нем на основании соглашений.

Использование паттерна Nested Closure, возможно, является одной из самых отличительных особенностей StructureMap. Для его использования необходимо хорошо разбираться в делегатах и блоках кода.

Начать работать со StructureMap довольно легко. Он поддерживает автоматическую интеграцию и автоматически определяет, каким образом создавать конкретные типы, даже если они не были явным образом сконфигурированы. Это означает, что вы можете сконцентрироваться на преобразовании абстракций в конкретные типы, и, когда вы закончите это преобразование, вы сможете разрешать диаграммы объектов. API, используемое для поиска типов, даже дает вам возможность сконфигурировать множество сервисов посредством всего нескольких строк кода, используя при этом для конфигурирования подход, основанный на соглашениях.

Несмотря на то, что нам не нужно конфигурировать конкретные сервисы, мы можем захотеть сделать это в тех ситуациях, когда нам нужно изменить стиль существования. По умолчанию используется стиль существования Per Graph, поэтому всякий раз, когда у нас имеются потоко-безопасные сервисы, мы можем потенциально увеличить эффективность путем конфигурирования их в виде Singleton'ов. Для этого нужен определенный шаг, несмотря на то, что можно было бы выразить это во время поиска типа посредством использования пользовательского соглашения о регистрации.

Не гарантируется, что экземпляры будут отслеживаться контейнером, поэтому StructureMap не предлагает никакого API для высвобождения конкретной диаграммы объектов. Это эффективно предотвращает утечки памяти для обычных классов, но, с другой стороны, почти гарантирует утечки памяти для устраняемых зависимостей. Поэтому важно реализовывать все зависимости таким образом, чтобы они самостоятельно управляли всеми внутренними использованиями устраняемых типов.

StructureMap свободно обрабатывает последовательности зависимостей. Когда класс зависит от последовательности экземпляров одного и того же типа, StructureMap будет автоматически интегрировать экземпляр со всеми экземплярами того типа, который имеет зависимость. Поведение StructureMap интуитивно понятно, поэтому нам нужно всего лишь явно выполнить определенное действие в тех ситуациях, когда нам требуется отобрать только некоторое подмножество экземпляров из всех доступных экземпляров.

Несмотря на то, что мы можем явным образом сконфигурировать Decorator'ы, StructureMap не имеет никаких соглашений по интеграции Decorator'ов и не обладает возможностями динамического перехвата. Существуют Seam'ы, которые могут использоваться для того, чтобы интегрировать в StructureMap динамическое прокси API, если нам нужно интегрировать Decorator'ы.

Поскольку StructureMap столь глубоко полагается на вложенные замыкания, неудивительно, что многие методы конфигурации обладают перегрузками, дающими нам возможность использовать блок кода, который будет вызываться при создании экземпляров. Несмотря на то, что эти перегрузки не являются необходимыми в тех ситуациях, когда регистрируемые нами классы используют Constructor Injection, мы можем использовать их, если один или более одного из наших классов должны создаваться особым образом.

StructureMap – исчерпывающий DI-контейнер, который предлагает широкий набор продвинутых возможностей. Он имеет отличное поведение по умолчанию и может с легкостью использоваться, особенно в тех случаях, когда дело касается автоматической интеграции конкретных типов или последовательностей. С другой стороны, он не обладает возможностями динамического перехвата и не может уничтожать устраняемые зависимости. Эти очевидные недостатки являются, главным образом, результатом философии проектирования. Если мы никогда не будем реализовывать устраняемые сервисы и вместо динамического перехвата решим использовать явные Decorator'ы, то в этом случае StructureMap является отличным вариантом, поскольку он использует эти ограничения для того, чтобы упростить для пользователей все остальное.

12. Spring.NET

Меню:

- Знакомство со Spring.NET
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В предыдущих главах вы наблюдали за тем, как применять к Castle Windsor и StructureMap принципы и паттерны, описанные в частях 1-3. В данной главе мы то же самое будем делать для DI-контейнера Spring.NET.

Наряду с Castle Windsor и StructureMap контейнер Spring.NET принадлежит к числу DI-контейнеров .NET "первого поколения". Появился он в 2006 году и даже на сегодняшний момент все еще поддерживает .NET 1.1. Это порт Java-версии Spring Framework, являющегося большим и исчерпывающим фреймворком приложений, который имеет дело со множеством различных аспектов программного обеспечения – не только Object Composition. DI-контейнер – один из множества доступных во фреймворке компонентов, но его можно отлично использовать и как самостоятельный компонент, не обращаясь при этом к другим компонентам Spring.NET.

В этой главе мы сконцентрируемся на рассмотрении DI-контейнера Spring.NET и опустим остальные компоненты фреймворка. Как в предыдущих и последующих главах, мы будем изучать то, как можно использовать Spring.NET для применения принципов и паттернов, описанных в частях 1-3. Рисунок 12-1 демонстрирует структуру данной главы.

Рисунок 12-1: Данная глава разделена на 4 раздела. Первый раздел знакомит нас с API Spring.NET и должен рассматриваться в качестве предпосылки следующих трех разделов. Каждый из четырех разделов данной главы может быть прочитан независимо от остальных разделов.



Первый раздел обеспечивает полноценное знакомство со Spring.NET и демонстрирует, как конфигурировать и разрешать объекты. Каждая из трех последующих глав затрагивает паттерны применения, которые требуют к себе дополнительного внимания. Вы можете читать их по порядку или пропустить некоторые и прочитать только те, которые вам интересны.

Данная глава должна дать вам возможность приступить к работе, а также справиться с большинством общепринятых проблем, которые могут возникнуть при ежедневном использовании Spring.NET. Это не окончательная трактовка Spring.NET, поскольку

всцелое его рассмотрение заняло бы несколько глав или, может быть, даже целую книгу. В любом случае весь фреймворк Spring.NET в данной книге не рассматривается. К рассматриваемому вопросу относится только DI-контейнер Spring.NET, поэтому, когда я использую название *Spring.NET*, я ссылаюсь конкретно на контейнер.

Вы можете прочитать эту главу, не читая при этом остальные главы части 4, специально для того, чтобы познакомиться со Spring.NET, или можете прочитать ее совместно с остальными главами части 4, чтобы сравнить DI-контейнеры. Цель данной главы – показать, как Spring.NET связан с паттернами и принципами, описанными в частях 1-3, и как он их реализует.

[12.1. Знакомство с Spring.NET](#)

[12.2. Управление жизненным циклом](#)

[12.3. Работа с составными компонентами](#)

[12.4. Конфигурирование сложных API](#)

[12.5. Резюме](#)

12.1. Знакомство с Spring.NET

Из этого раздела вы узнаете, где можно взять Spring.NET, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим универсальные варианты конфигурирования. Таблица 12-1 предоставляет основополагающую информацию, которая, скорее всего, понадобится вам для того, чтобы приступить к работе со Spring.NET.

Таблица 12-1: Краткая информация о Spring.NET

Вопрос	Ответ
Откуда мне его получить?	Перейти на страницу www.springframework.net/download.html и загрузить последний релиз Из Visual Studio 2010 можно получить его посредством NuGet. Имя пакета – <i>Spring.Core</i> . Но если вам нужны возможности механизма перехвата, то в этом случае вам нужен пакет <i>Spring.Aop</i> .
Что находится в загруженном файле?	Zip-файл, который содержит все вам необходимое: скомпилированные бинарные файлы, исходный код и документацию.
Какие платформы поддерживаются?	Поддерживаются все версии ASP.NET, начиная с .NET 1.1, хотя последующие версии будут поддерживать только версии .NET 2.0 и выше.
Сколько он стоит?	Нисколько. Это программное обеспечение с открытым исходным кодом.
Откуда мне получить помощь?	Коммерческую поддержку можно получить от SpringSource – организация, которая занимается разработкой Spring.NET. Невзирая на коммерческую поддержку, Spring.NET все еще остается программным обеспечением с открытым исходным кодом, которое обладает процветающей экосистемой, поэтому, скорее всего (но не гарантированно), помощь можно получить на официальном форуме http://forum.springframework.net .
На какой версии Spring.NET основана данная глава?	1.3.1

Использование DI-контейнера Spring.NET предполагает выполнение трех шагов, продемонстрированных на рисунке 12-2.

Рисунок 12-2: Полноценный паттерн применения Spring.NET включает в себя три шага: сначала мы задаем то, как объекты конфигурируются и компонуются в XML-файле. Затем мы загружаем XML-конфигурацию в экземпляр контейнера. На последнем и финальном шаге мы можем разрешать объекты из экземпляра контейнера.



После прочтения этого раздела вы приобретете полноценное понимание всей сущности паттерна применения Spring.NET и сможете начать использовать его в сценариях, в которых все компоненты руководствуются должным DI-паттерном, например, Constructor Injection. Давайте начнем с простейшего сценария и посмотрим, как можно разрешать объекты с помощью контейнера Spring.NET.

Разрешение объектов

Основная услуга, предоставляемая любым DI-контейнером – компоновка диаграмм объектов, и Spring.NET не является исключением. Поскольку это основная возможность контейнера, именно сейчас и стоит приступить к знакомству с соответствующим API, что я и буду делать в этом разделе.

В предыдущих главах, посвященных Castle Windsor и StructureMap, вы видели, как эти DI-контейнеры по-разному подходят к тому, нужно ли конфигурировать компоненты до того, как можно будет их разрешать. Castle Windsor требует, чтобы мы явно конфигурировали каждый отдельный компонент, а StructureMap может работать с конкретными типами без предварительной конфигурации. Но оба эти DI-контейнера, как и все DI-контейнеры, рассматриваемые в следующих главах, оперируют типами: мы отправляем в контейнер запрос на разрешение соответствующего типа.

Spring.NET отличается от остальных DI-контейнеров, поскольку его ключевой механизм запросов основывается не на типах, а на именах. Вместо того чтобы запрашивать конкретный тип, мы запрашиваем у Spring.NET именованный объект. По аналогии с Castle Windsor все объекты должны быть сконфигурированы до того, как мы сможем их использовать.

Конфигурирование Spring.NET выполняется в XML, поэтому даже самый простой сценарий включает в себя фрагмент XML, а также некоторый .NET код. К примеру, для того чтобы разрешить конкретный класс `SauceBéarnaise`, вы должны сначала задать объект в XML-конфигурации:

```

<objects xmlns="http://www.springframework.net">
  <object id="Sauce"
    type="Ploeh.Samples.MenuModel.SauceBéarnaise,
    ➤ Ploeh.Samples.MenuModel" />
</objects>
  
```

В Spring.NET каждый сконфигурированный объект должен отображаться в элементе `object`. Данный элемент может иметь атрибут `id`, который присваивает имя объекту, а

также атрибут `type`, который определяет .NET тип объекта. Имя используется в тех случаях, когда вы собираетесь разрешить объект.

Для разрешения экземпляра `SauceBéarnaise` вы должны загрузить XML-конфигурацию в экземпляр контейнера. При помощи `XmlApplicationContext` вы можете загрузить XML из нескольких различных источников, включая вложенные ресурсы и конфигурационный файл приложения. Но в примере ниже используется самостоятельный XML-файл под названием `sauce.xml`:

```
var context = new XmlApplicationContext("sauce.xml");
SauceBéarnaise sauce = (SauceBéarnaise)context.GetObject("Sauce");
```

Для того чтобы разрешить экземпляр `SauceBéarnaise` вы вызываете метод `GetObject` с ID, равным `Sauce`, который вы задали для объекта в XML-конфигурации. ID может быть любой строкой, но Spring.NET рекомендует использовать Pascal нотацию в качестве соглашения по именованию.

Поскольку метод `GetObject` возвращает слабо типизированный экземпляр `System.Object`, чтобы начать его использовать, вам необходимо привести возвращаемое значение к соответствующему типу

Обратите внимание на то, что Spring.NET явным образом не различает конкретные и абстрактные типы. В то время как другие DI-контейнеры требуют от нас преобразования абстрактных типов в конкретные, Spring.NET основан только на преобразованиях *имен* в конкретные типы. Как продемонстрирует данная глава, Spring.NET все равно может разрешать запросы преобразования абстракций в конкретные экземпляры, но используемый механизм более явный, чем тот, который применяют другие DI-контейнеры.

Метод `GetObject` определяется интерфейсом `IObjectFactory`, который является одним из фундаментальных интерфейсов, задаваемых Spring.NET. Как и подразумевает его имя, интерфейс `IObjectFactory` сконцентрирован на *создании* объектов и не содержит методы, позволяющие конфигурировать контейнер. Скорее за это отвечают типы более высокого уровня, например, `XmlApplicationContext`.

Метод `GetObject` является одним из нескольких методов, которые можно использовать для разрешения объектов. Тем не менее, поскольку все они слабо типизированы, мы должны всегда явным образом приводить возвращаемое значение к типу, с которым мы собираемся работать. С помощью метода `GetObject` мы можем запрашивать объекты только по имени, а не по типу, поэтому, каким образом мы должны выходить из ситуаций, когда все, что у нас есть – это тип?

Разрешение запросов типа

Иногда у нас имеется не имя, а экземпляр `Type`, который мы должны разрешить в экземпляр этого типа. Пример этого вы видели в разделе 7.2 "Построение ASP.NET MVC приложений", где мы обсуждали ASP.NET MVC класс `DefaultControllerFactory`. Соответствующий метод приведен ниже:

```
protected internal virtual IController GetControllerInstance(
    RequestContext requestContext, Type controllerType);
```

При наличии типа вместо имени мы могли бы соблазниться заданием и сопровождением явного преобразования типов в имени, но это было бы излишним. Наиболее подходящим вариантом было бы использование соглашения по именованию, которое позволяло бы нам детерминированно наследовать имя от экземпляра `Type`. Но интерфейс `IListableObjectFactory`, который наследуется напрямую от `IObjectFactory`, задает метод под названием `GetObjectsOfType`, который можно использовать для получения всех объектов, соответствующих данному типу. Полагая, что запрашиваемый `controllerType` – уникален в конфигурации Spring.NET, вы можете реализовать метод `GetControllerInstance` следующим образом:

```
IDictionary controllers =
    this.context.GetObjectsOfType(controllerType);
return controllers.Values.OfType<IController>().Single();
```

Поле `context` – экземпляр `IListableObjectFactory`, который можно запросить для всех объектов, соответствующих `controllerType`. Несмотря на то, что вам возвращается словарь, вам интересны только значения, и при этом вы предполагаете, что каждый запрашиваемый контроллер будет уникальным в пределах упомянутой выше XML-конфигурации.

Хотя Spring.NET не предоставляет никакого generic API, вы можете легко инкапсулировать предыдущий запрос в метод расширения:

```
public static T Resolve<T>(this IListableObjectFactory factory)
{
    return factory.GetObjectsOfType(typeof(T))
        .Values.OfType<T>().Single();
}
```

Это позволит вам разрешать тип следующим образом:

```
SauceBéarnaise sauce = context.Resolve<SauceBéarnaise>();
```

Метод `GetObjectsOfType` возвращает все сконфигурированные объекты, которые соответствуют запрашиваемому типу. Поскольку `SauceBéarnaise` реализует интерфейс `IIngredient`, вы можете также разрешить `IIngredient` из контекста:

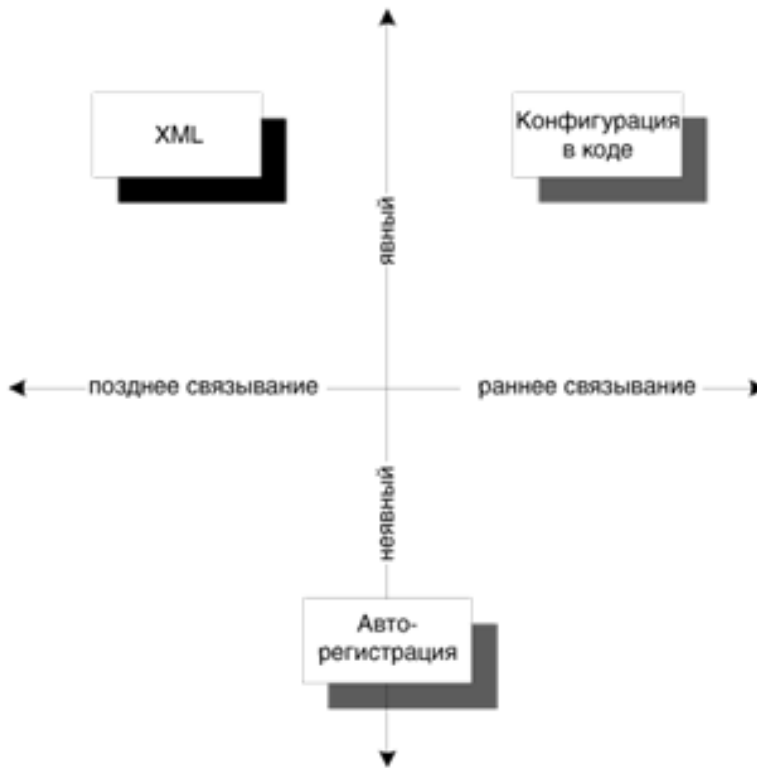
```
IIngredient ingredient = context.Resolve<IIngredient>();
```

Типичный ASP.NET Controller и любой другой код приложения, который мы, скорее всего, напишем, будет иметь сложную иерархию зависимостей. Для того чтобы позволить Spring.NET компоновать объекты из слабо связанных сервисов, мы должны обеспечить должную конфигурацию.

Конфигурирование контейнера

Как мы уже обсуждали в разделе 3.2 "Конфигурирование DI-контейнеров", существует несколько концептуально разных способа конфигурирования DI-контейнера. На рисунке 12-3 представлен обзор возможных вариантов и то, какие варианты подходят для Spring.NET.

Рисунок 12-3: Spring.NET из трех возможных вариантов, перечисленных в главе 3, поддерживает, главным образом, XML-конфигурацию. Технология использования кода в качестве конфигурации поддерживается в минимальной степени, а автоматическая регистрация вообще недоступна. Поэтому данные варианты отмечены серым цветом.



Как и другие, имеющие длительную историю DI-контейнеры, Spring.NET сначала использовал XML в качестве основного источника конфигурации. Но в отличие от Castle Windsor и StructureMap контейнер Spring.NET продолжает концентрироваться на XML, возможно, из-за его сильных связей с фреймворком Java Spring.

Технология конфигурирования в коде в Spring.NET

В то время, когда я писал эту главу, Spring.NET не поддерживал технологию конфигурирования в коде. Тем не менее, буквально за несколько дней до того, как я приступил к финальной части книги, SpringSource выпустил `Spring CodeConfig`, который обеспечивает поддержку технологии конфигурирования в коде в Spring.NET.

К несчастью, это случилось слишком поздно, и у меня не оставалось времени на то, чтобы переписать эту главу.

В данной главе мы будем рассматривать только XML-конфигурацию.

Работа с .NET типами в XML

Spring.NET использует конфигурацию, основанную на XML. Этот XML может браться из разнообразных источников. В большинстве случаев мы будем загружать XML из файла.

В разделе 12.1.1 "Разрешение объектов" вы уже видели простой пример XML-конфигурации Spring.NET:

```
<objects xmlns="http://www.springframework.net">
  <object id="Sauce"
    type="Ploeh.Samples.MenuModel.SauceBéarnaise,
      ➡Ploeh.Samples.MenuModel" />
</objects>
```

Не секрет, что XML сам по себе является многословным языком, но когда дело доходит до идентификации типов, он становится чересчур многословным. Для того чтобы должным образом идентифицировать .NET тип с помощью строки, мы должны прибегнуть к квалифицированным именам сборки. Даже в случаях, когда мы можем опустить *Culture*, *Version* и *PublicKeyToken*, тип будет чаще всего идентифицироваться при помощи длинной строки, содержащей множество повторяющейся информации. Это наносит вред как читабельности, так и удобству сопровождения.

На читабельность оказывается влияние, поскольку соответствующая часть имени типа (*SauceBéarnaise*) размещена между пространством имен и именем сборки. Удобство сопровождения подвергается влиянию, поскольку становится сложнее переименовывать пространства имен и сборки. Всякий раз, когда мы что-либо переименовываем, нам приходится редактировать потенциально большой набор определений типов.

Допустим, эти проблемы применяются ко всем фреймворкам, в которых типы должны задаваться в виде XML, но тогда это еще одна причина того, почему все остальные DI-контейнеры используют для конфигурирования контейнера другие варианты. По этой причине я также чаще всего не рекомендую использовать XML-конфигурацию до тех пор, пока она не будет предписана сценарием применения. Тем не менее, когда дело касается Spring.NET, XML – самый распространенный вариант конфигурирования.

Для облегчения работы с .NET типами в XML SpringSource предоставляет такие средства, как XML-схемы и дополнение для Visual Studio с завершением типов и свойств. Сам фреймворк также позволяет нам задавать набор псевдонимов типов, являющихся сокращенными именами, которые можно использовать для определения типов. Это также выполняется в XML. Псевдоним типа для класса *SauceBéarnaise* может выглядеть следующим образом:

```
<alias name="SauceBéarnaise"
  type="Ploeh.Samples.MenuModel.SauceBéarnaise,
    ➡Ploeh.Samples.MenuModel" />
```

Имя может быть любым, но мне кажется, чтобы оно было наиболее понятно и его можно было бы легко запоминать, для имени типа следует использовать его псевдоним.

Указанный псевдоним типа позволяет нам переписать предыдущий пример следующим образом:

```
<object id="Sauce" type="SauceBéarnaise" />
```

Данная возможность может быть полезна, когда нам нужно несколько раз обратиться к одному и тому же типу в одном и том же XML-файле. В любом случае это делает конфигурацию более читабельной. В оставшейся части данной главы я буду использовать псевдонимы типов. Благодаря такому обозначению элемент *object* начинает походить на то, чем он является: преобразование имени в конкретный тип.

В такой простой форме это и есть элемент `object`. При наличии у объектов зависимостей вы должны сообщить Spring.NET, как их необходимо разрешать.

Явное конфигурирование зависимостей

Создать класс `SauceBéarnaise` довольно легко, поскольку он имеет конструктор по умолчанию. Ни одному DI-контейнеру не нужно никакой особой помощи для создания таких типов. Все меняется, когда конструктор по умолчанию отсутствует. К примеру, рассмотрим конструктор `Mayonnaise`:

```
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

Несмотря на то, что рецепт майонеза слегка упрощен, и `EggYolk`, и `OliveOil` — это конкретные классы, имеющие конструкторы по умолчанию. Однако, поскольку класс `Mayonnaise` не имеет конструктора по умолчанию, вы должны сообщить Spring.NET, как его разрешить. Один из вариантов — явным образом одновременно интегрировать типы

```
<object id="EggYolk" type="EggYolk" />
<object id="OliveOil" type="OliveOil" />
<object id="Mayonnaise" type="Mayonnaise">
  <constructor-arg ref="EggYolk" />
  <constructor-arg ref="OliveOil" />
</object>
```

Типы `EggYolk` и `OliveOil` конфигурируются таким же образом, как это делалось ранее, но элемент `Mayonnaise` теперь содержит два элемента `constructor-arg`. Каждый из этих элементов ссылается на именованный объект для того, чтобы определить параметры конструктора `Mayonnaise`. Атрибут `ref` идентифицирует еще один сконфигурированный объект по имени, таким образом, `EggYolk` ссылается на имя `EggYolk`, а не явно на тип `EggYolk`.

В предыдущем примере важен порядок элементов `constructor-arg`, поскольку вы не обращаетесь явным образом к именам параметров, что также возможно.

Несмотря на то, что мы всегда можем сконфигурировать зависимости таким способом, мы не получаем никаких преимуществ, которые предлагает нам механизм автоматической интеграции. В противоположность другим DI-контейнерам при работе с Spring.NET мы должны явным образом просить его использовать механизм автоматической интеграции, и все равно этот механизм работает только в конкретных случаях.

Автоматическая интеграция зависимостей

Возможности механизма автоматической интеграции в Spring.NET ограничены, но мы должны явным образом переключить его на использование XML-атрибута. Вместо явной интеграции `Mayonnaise` указанным ранее способом вы могли бы сконфигурировать его следующим образом:

```
<object id="EggYolk" type="EggYolk" />
<object id="OliveOil" type="OliveOil" />
<object id="Mayonnaise" type="Mayonnaise"
  autowire="autodetect" />
```

Необязательный атрибут `autowire` можно использовать для включения для этого объекта механизма автоматической интеграции. В данном примере мы используем значение `autodetect`, которое сообщает Spring.NET о том, что необходимо точно вычислить, каким образом выполняется поиск соответствующих объектов. Остальные доступные параметры позволяют нам указывать, что поиск соответствующих объектов необходимо выполнять по именам, типам или другими средствами.

Если мы планируем использовать автоматическую интеграцию для всех объектов, то мы можем разрешить ее для целого блока сконфигурированных объектов вместо того, чтобы приписывать атрибут `autowire` для каждого элемента `object`:

```
<objects xmlns="http://www.springframework.net"
  default-autowire="autodetect">
  <object id="EggYolk" type="EggYolk" />
  <object id="OliveOil" type="OliveOil" />
  <object id="Mayonnaise" type="Mayonnaise" />
</objects>
```

Атрибут `default-autowire` задает стратегию автоматической интеграции по умолчанию для всех объектов в рамках элемента `objects`. Это самый простой способ включения автоматической интеграции сразу для всех объектов, но вы должны иметь в виду, что этот механизм работает не всегда.

Поддержка автоматической интеграции в Spring.NET основывается на однозначности. Spring.NET исследует конструктор класса `Mayonnaise` и определяет, что ему нужны экземпляры `EggYolk` и `OliveOil`. Для выделения зависимости `EggYolk` он выполняет поиск среди всех остальных сконфигурированных элементов с целью обнаружения элемента, который может удовлетворять данному условию (то же самое он делает и для зависимости `OliveOil`).

В предыдущем примере есть только один объект, удовлетворяющий зависимости `EggYolk`, поэтому неоднозначность отсутствует. Однако если бы данному условию удовлетворяло более одного объекта, то возникло бы исключение. Это справедливо не только для конкретных элементов `EggYolk`, но также и для унаследованных типов.

Примечание

В Spring.NET автоматическая интеграция работает только в тех ситуациях, когда зависимости могут разрешаться однозначно. В этом заключается отличие Spring.NET от остальных DI-контейнеров.

Условие индивидуальности, обеспечивающее поддержку автоматической интеграции, дает нам преимущество, которое заключается в том, что контракт в рамках контейнера является понятным и явным. Автоматическая интеграция возможна только при отсутствии неопределенности, связанной с разрешенными типами, поэтому риск неправильного конфигурирования становится намного меньше. С другой стороны, такое конструирование усложняет работу с составными объектами, реализующими одну и ту же абстракцию.

Мы рассмотрели некоторые основные опции конфигурации Spring.NET. Несмотря на то, что можно, конечно, написать один большой блок неструктурированного XML кода, лучше всего разделить конфигурацию на модули. Spring.NET поддерживает такую возможность, позволяя XML загружаться более чем из одного источника.

Загрузка XML

Иногда нам может потребоваться упаковать конфигурацию в повторно используемые группы, и даже когда повторное использование само по себе не является нашим высшим приоритетом, нам может понадобиться некоторого рода структура в случае необходимости конфигурирования большого и сложного приложения.

В рамках Spring.NET мы можем упаковать конфигурацию в отдельные XML элементы, которые определены в разных ресурсах. В таблице 12-2 перечисляются поддерживаемые типы ресурсов. Их легко использовать, но я кратко расскажу о каждом из них, чтобы у вас осталось это в памяти.

Таблица 12-2: Типы ресурсов XML

Тип ресурса	Синтаксис URI	Описание
FileSystemResource	file://<filename> Моникер <i>file:// moniker</i> не обязателен.	XML-конфигурация задается в файлах.
ConfigSectionResource	config//<path to section>	XML-конфигурация задается в конфигурационном файле приложения.
UriResource	Поддерживается стандартный синтаксис .NET URI.	XML-конфигурация читается из таких стандартных протоколов System.Uri, как HTTP и HTTPS.
AssemblyResource	assembly://<AssemblyName>/<NameSpace>/<ResourceName>	XML-конфигурация вложена в сборку.
InputStreamResource	Не поддерживается	XML-конфигурация читается из System.IO.Stream.

Некоторые типы ресурсов поддерживают синтаксис URI, в котором можно использовать моникер для обозначения типа ресурса в качестве составляющей части адреса, закодированного строкой. Если моникер не применяется, то предполагается, что ресурс является файлом.

Использование XML-файлов

До настоящего момента вы видели примеры загрузки XML-конфигурации только из одного файла. В примере ниже загрузка всей конфигурации выполняется из файла *sauce.xml*:

```
var context = new XmlApplicationContext("sauce.xml");
```

Поскольку о пути не предоставлено никакой явной информации, предполагается, что файл *sauce.xml* располагается в рабочей папке запущенного процесса. Кроме того, можно использовать и полный путь.

В этом примере моникер не использовался, поэтому Spring.NET принимает в качестве значения по умолчанию *FileSystemResource*. В противном случае вы могли бы явно использовать моникер *file://*, как это продемонстрировано ниже:

```
var context = new XmlApplicationContext("file://sauce.xml");
```

Этот пример аналогичен предыдущему примеру. Чаще всего работа с XML в виде файлов интуитивно понятна, поэтому в большинстве случаев имеет смысл обойтись без моникера *file://* и вместо этого явно прописывать путь к файлу.

Помимо определения XML-конфигурации в текстовых файлах можно также интегрировать ее в стандартный конфигурационный файл приложения.

Использование конфигурационных файлов приложения

Если мы предпочитаем интегрировать конфигурацию Spring.NET с остальной частью конфигурации приложения, то можем использовать стандартный *.config* файл .NET приложения.

Поскольку система конфигурации .NET приложения предполагает, что разделы пользовательской конфигурации должны регистрироваться явным образом, мы также должны зарегистрировать разделы конфигурации Spring.NET, в которых собираемся использовать файл *.config*. Можно регистрировать разнообразные разделы конфигурации, но для того чтобы использовать элемент *objects*, который вы применяли до настоящего момента, необходимо зарегистрировать используемый по умолчанию обработчик раздела Spring.NET:

```
<configSections>
  <sectionGroup name="spring">
    <section name="objects"
      type="Spring.Context.Support.DefaultSectionHandler,
        Spring.Core" />
  </sectionGroup>
</configSections>
```

Это позволяет определять объекты напрямую в файле *.config*, как вы это делали ранее в автономных XML-файлах:

```
<spring>
  <objects xmlns="http://www.springframework.net">
    <object id="Sauce" type="SauceBéarnaise" />
  </objects>
</spring>
```

Используя моникер *config://*, теперь можно загрузить конфигурацию Spring.NET из файла *.config* в экземпляре `XmlApplicationContext` следующим образом:

```
var context = new XmlApplicationContext("config://spring/objects");
```

Теперь экземпляр `context` может безопасно разрешать имя *Sauce*.

Во многих отношениях интеграция конфигурации Spring.NET в формат стандартного конфигурационного файла .NET приложения является особым случаем использования XML-файла, поскольку файлы *.config* также являются XML-файлами. Но помимо этого мы можем рассматривать файлы как особый случай загруженного XML из любого URI.

Загрузка XML из URI

При загрузке XML из файлов мы используем моникер в качестве разграничителя URI схемы. Spring.NET может загружать XML-файлы не из файлов, а из других URI, например, HTTP, HTTPS и FTP. Это столь же просто, как и приведенный ниже код:

```
var context = new XmlApplicationContext("http://localhost/sauce.xml");
```

В этом примере файл *sauce.xml* размещен на веб-сервере локального компьютера, но использовать можно любой публично доступный ресурс.

Во всех предыдущих случаях мы могли изменять конфигурацию, заново не компилируя приложение. Это невозможно для следующей опции.

Использование вложенных ресурсов

В .NET мы можем скомпилировать ресурсы в сборку. Если мы внедрим XML-конфигурацию в сборку, то Spring.NET сможет ее загрузить.

Преимущество вложенных XML-файлов заключается в том, что операторы не могут случайно изменить конфигурационные значения, когда они компилируются в сборку. Только файлы, имеющие параметры, значения которых, возможно, должны устанавливаться операторами, должны передаваться вне сборки в файловую систему, где их можно редактировать.

При внедрении файла *sauce.xml* в сборку можно загрузить его в `XmlApplicationContext` следующим образом:

```
var context = new XmlApplicationContext(  
    "assembly://Ploeh.Samples.Menu.SpringNet/  
    ➡Ploeh.Samples.Menu.SpringNet/sauce.xml");
```

Для того чтобы выполнить загрузку из вложенного ресурса, мы можем сконструировать строку ресурса из моникера *assembly://*, за которым следует название сборки, пространство имен и название самого вложенного ресурса. В таблице 12-2 показан требуемый формат, используемый для обращения к вложенному ресурсу.

`AssemblyResources` позволяет нам загружать конфигурацию Spring.NET не только из внешне заданных XML-файлов и URI, но и из вложенных ресурсов. Если мы храним

XML-конфигурацию в других местах, то нам может понадобиться такая возможность, как чтение из потоков.

Использование потоков

До настоящего момента мы рассматривали загрузку XML из статических ресурсов, например, из файлов, конфигурации приложения, веб-ресурсов и вложенных ресурсов. Иногда нам нужно загружать XML из других источников или, возможно, необходимо динамически создавать XML-конфигурацию. Один из способов достижения этой цели – загружать конфигурацию напрямую из потока.

Поскольку поток не является статическим ресурсом, Spring.NET не поддерживает возможность идентификации его по строке. У нас нет возможности использовать класс `XmlApplicationContext`, и вместо него мы должны прибегнуть к одному из многочисленных контекстных классов Spring.NET:

```
var resource = new InputStreamResource(stream, "");  
var context = new XmlObjectFactory(resource);
```

`InputStreamResource` выступает в роли адаптера для объекта `System.IO.Stream`. Объект `stream` содержит XML-конфигурацию, которую вы собираетесь загрузить. Мы можем загрузить XML в поток из множества различных источников, включая строку, или посредством построения XML модели с помощью технологии LINQ to XML. Пустая строка, используемая в конструкторе `InputStreamResource` – это описание. Мы можем передать соответствующее описание, но это необязательно.

Благодаря `resource` (или любой реализации `IResource`), мы теперь можем создать экземпляр `XmlObjectFactory`. Этот класс предлагает функциональность, аналогичную `XmlApplicationContext`, но загружает конфигурацию напрямую из экземпляра `IResource` вместо того, чтобы загружать ее из строк, представляющих собой статические ресурсы.

Интерфейс `IResource` – это универсальный интерфейс для всех рассматриваемых нами XML ресурсов. Мы можем предоставить пользовательскую реализацию `IResource` и использовать ее тем же самым способом, что и `InputStreamResource`, или можем зарегистрировать ее с помощью собственного моникера. Но рассмотрение этого варианта уже выходит за рамки этой главы.

До настоящего момента вы наблюдали, как загружать одну XML-конфигурацию из одного ресурса, но для того чтобы достичь модульности, мы чаще всего будем организовывать части конфигурации больших приложений в разных модулях.

Комбинирование XML ресурсов

Для большого приложения потребуется большое количество кода XML-конфигурации. Чтобы обеспечить наилучшее сопровождение конфигурации, мы, возможно, захотим разделить конфигурацию на несколько небольших документов. Может быть, мы даже захотим хранить их в отдельных местах: некоторые в XML-файлах, некоторые в файле `.config`, а некоторые – в виде вложенных ресурсов.

`XmlApplicationContext` позволяет комбинировать несколько разных ресурсов, поскольку он получает каждую строку ресурса как часть массива параметров.

Ниже приведена сигнатура конструктора, который вы использовали на протяжении всего этого времени:

```
public XmlApplicationContext(params string[] configurationLocations)
```

Обратите внимание на то, что параметр `configurationLocations` задан в виде массива параметров. До настоящего момента вы за один раз использовали только один ресурс приведенным ниже способом:

```
var context = new XmlApplicationContext("sauce.xml");
```

Однако вы можете использовать произвольное количество строк в конструкторе для комбинирования в одном контексте нескольких ресурсов:

```
var context = new XmlApplicationContext(  
    "config://spring/objects",  
    "meat.xml",  
    "file://course.xml");
```

В этом примере комбинируются три разных ресурса, каждый из которых определяет фрагмент единого целого. Одна часть конфигурации задается и загружается из конфигурационного файла приложения, в то время как две остальные части загружаются из XML-файлов: одна – посредством использования для имен файлов неявного синтаксиса, а другая – с помощью явного использования моникера *file://*. Вместе эти части формируют полноценную систему конфигурации, которая определяет `XmlApplicationContext`.

Еще один способ комбинирования составных ресурсов – посредством элемента `import` XML-файла:

```
<objects xmlns="http://www.springframework.net">  
    <import resource="config://spring/objects" />  
    <import resource="meat.xml" />  
    <import resource="file://course.xml" />  
</objects>
```

Эта конфигурация аналогична предыдущему примеру.

Как вы уже видели, существует несколько различных способов, с помощью которых можно загрузить XML-конфигурацию в Spring.NET. Поскольку мы можем загрузить и комбинировать конфигурацию из более чем одного ресурса, это дает нам некоторую степень модульности, которая очень нужна нам для обеспечения сопровождаемости.

Но мы не должны забывать о том, что в целом XML-конфигурация не самый лучший способ конфигурирования DI-контейнера. Это очень хрупкий и слишком подробный способ, и к тому же он сложен для разрешения.

Этот раздел познакомил нас с DI-контейнером Spring.NET и продемонстрировал фундаментальные принципы, как конфигурировать контейнер с помощью XML и впоследствии использовать его для разрешения объектов. Разрешение объектов выполняется посредством единичного вызова метода `GetObject`, поэтому вся сложность заключается в конфигурировании контейнера. До настоящего момента мы рассматривали только самое основное API. Но есть и более продвинутые области, которые мы еще не

рассмотрели. Один из самых важных вопросов – как управлять жизненным циклом компонентов.

12.2. Управление жизненным циклом

В главе 8 мы обсуждали процесс управления жизненным циклом, в том числе такие самые универсальные концептуальные стили существования, как Singleton и Transient. Spring.NET поддерживает несколько других стилей существования и позволяет нам конфигурировать жизненные циклы всех объектов. Стили существования, продемонстрированные в таблице 12-3 доступны в виде составляющей пакета.

Примечание

В документации к Spring.NET стили существования называются *областями применения объектов*.

Реализации Singleton и Transient в Spring.NET аналогичны основным стилям существования, описанным в главе 8, поэтому в данной главе я не буду уделять им много внимания.

Таблица 12-3: Стили существования Spring.NET

Название	Комментарии
Singleton	Стиль существования по умолчанию.
Prototype	Название стиля существования Transient, используемое в рамках Spring.NET. Экземпляры контейнером не отслеживаются.
Request	Название стиля существования Web Request Context, используемое в рамках Spring.NET. Корректно только в контексте <code>IAApplicationContext</code> .
Session	Для одной HTTP-сессии создается один экземпляр. Используйте с осторожностью. Корректно только в контексте <code>IAApplicationContext</code> .
Application	Расширяет определение одного объекта до жизненного цикла веб-приложения. Корректно только в контексте <code>IAApplicationContext</code> .

Примечание

По умолчанию в Spring.NET используется стиль существования Singleton. Этим он отличается от большинства других контейнеров. Как уже обсуждалось в главе 8, Singleton – это самый эффективный, хотя и не всегда безопасный, из всех стилей существования объектов. В Spring.NET эффективность приоритетнее безопасности.

Три веб-стиля (Request, Session и Application) тесно связаны между собой соответствующими `IAApplicationContexts` и не работают с `XmlApplicationContext` или с `XmlObjectFactory`, которые мы рассматривали до настоящего момента. Доступные на данный момент реализации настолько сильно связаны с ASP.NET Web Forms, что заставить их работать с ASP.NET MVC довольно трудно. Откровенно говоря, эти стили довольно запутаны и не используются, поэтому в этой главе мы не будем их рассматривать. Предполагается, что в последующих версиях Spring.NET этот вопрос будет модернизирован.

В этой главе я продемонстрирую вам, как конфигурировать области применения объектов на примере стилей Singleton и Transient. Поскольку Spring.NET не поддерживает

пользовательские стили существования, эта глава будет краткой. После прочтения этой главы вы сможете использовать стили существования объектов в Spring.NET.

Конфигурирование областей применения

Области применения конфигурируются в качестве составляющей части конфигурации объектов в XML.

Для того чтобы сконфигурировать объект как `Transient`, необходимо установить атрибуту `singleton` значение `false`:

```
<object id="Sauce" type="SauceBéarnaise" singleton="false" />
```

Изменив значение атрибута на `true`, мы сконфигурируем объект как `Singleton`:

```
<object id="Sauce" type="SauceBéarnaise" singleton="true" />
```

Атрибут `singleton` является необязательным, поскольку стиль `Singleton` используется по умолчанию, за исключением тех случаев, когда он неявным образом конфигурируется как `Singleton`. Именно это вы и будете делать далее в этой главе.

Предотвращение утечек памяти

Как и любой другой DI-контейнер Spring.NET создает диаграммы объектов. Однако он не выполняет за нас отслеживание созданных объектов. Он может отслеживать созданные объекты для своих собственных целей, но все это зависит от стиля существования объекта. Например, для реализации области применения `Singleton` Spring.NET должен хранить ссылку на созданный экземпляр. С другой стороны, область применения `Transient` не отслеживает объекты, созданные Spring.NET. Как вы уже видели в листингах 8-7 и 8-8, экземпляры объектов создаются и возвращаются без внутреннего сопровождения. Все это имеет некоторые достоинства и преимущества.

Поскольку Spring.NET не держится за экземпляры, риск случайных утечек памяти не велик. При использовании таких контейнеров, как Castle Windsor, утечки памяти гарантированы, если вы забыли вызвать метод `Release` для всех разрешенных диаграмм объектов. В Spring.NET все не так, поскольку объекты будут автоматически уничтожаться "сборщиком мусора", как только они будут выходить за рамки области применения.

Недостаток заключается в том, что устранимые объекты нельзя детерминированно уничтожать. Поскольку мы не можем явно высвободить диаграмму объектов, мы не можем уничтожать любые устранимые объекты. Это означает, что наибольшую значимость приобретает заворачивание устранимых API в неустраиваемые сервисы, что обсуждалось в разделе 6.2.1.

Короче говоря, Spring.NET ведет себя подбающим образом и позволяет объектам уничтожаться "сборщиком мусора", когда они выходят за рамки области применения вашего кода. Но при этом необходимо, чтобы ваши собственные классы также вели себя подбающим образом. Поскольку в вопросе уничтожения любого сервиса мы не можем полагаться на контейнер или код, выполняющий вызов, мы должны продолжать использовать устранимые объекты в рамках единичных методов.

Эта глава предоставила удивительно краткий обзор областей применения объектов, применяемых в Spring.NET. По этому вопросу особо и нечего сказать. Единственные повсеместно доступные области применения – это Singleton и Transient, тогда как пара остальных областей применения полагается на конкретные реализации `IApplicationContext`. При конфигурировании объектов мы можем конфигурировать некоторые из них как Singleton, а некоторые – как Transient, и это справедливо даже, когда мы конфигурируем составные реализации одной и той же абстракции. Мы уже рассматривали то, как работать с составными компонентами, поэтому давайте переключим наше внимание в этом направлении.

12.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями
- Используются Decorator'ы

Мы рассмотрим каждую из этих ситуаций и увидим, как Spring.NET поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь регистрировать и разрешать компоненты даже тогда, когда в дело вступают составные реализации одной и той же абстракции.

Как вы уже видели в разделе 12.1.2 "Конфигурирование контейнера", по сравнению с большинством других DI-контейнеров автоматическая интеграция не является поведением по умолчанию в Spring.NET. Более разветвленное управление интеграцией столь же проверенная опция и может использоваться для осуществления выбора между составными кандидатами.

Выбор из составных кандидатов

В Spring.NET можно использовать механизм автоматической интеграции, но для этого необходимо, чтобы сервисы были индивидуальными. Пока у нас есть только один единственный объект, соответствующий конкретной абстракции, никаких проблем нет, но как только мы введем больше реализаций одного и того же интерфейса, возникает неопределенность.

Для того чтобы побороть эту неопределенность, мы можем использовать явную интеграцию зависимостей, которую вы уже видели в некоторых примерах.

Конфигурирование составных реализаций одной и той же абстракции

До настоящего момента вы конфигурировали именованные объекты, но у объектов необязательно должны быть имена. Вы можете конфигурировать составные объекты, не присваивая им имена:

```
<objects xmlns="http://www.springframework.net">
  <object type="SauceBéarnaise" />
  <object type="Steak" />
</objects>
```

Классы `SauceBéarnaise` и `Steak` конфигурируются без имен. Поскольку вы явным образом не задали имена для объектов `SauceBéarnaise` и `Steak`, Spring.NET присваивает каждому из них автоматически сгенерированное имя. Если бы вы знали алгоритм, который Spring.NET использует для генерирования имени, то вы могли бы запрашивать объекты посредством метода `GetObject`. Но это может стать некоторого рода хрупким решением. Вместо этого метода вы можете использовать метод `GetObjectsOfType`, который также был введен в разделе 12.1.1 "Разрешение объектов". Как только мы сконфигурируем в Spring.NET тип, мы сможем извлечь его с помощью типов, от которых он унаследован.

Чтобы получить экземпляр конкретного класса `Steak`, например, можно сочетать метод `GetObjectsOfType` с парой методов расширения LINQ:

```
var meat = context.GetObjectsOfType(typeof(Steak))
    .Values
    .OfType<Steak>()
    .FirstOrDefault();
```

Вы запрашиваете тип `Steak` в методе `GetObjectsOfType`. Spring.NET найдет все сконфигурированные объекты, соответствующие запрашиваемому типу (независимо от того, именованные они или нет), и вернет их в виде словаря. Ключевыми словами данного словаря являются имена объектов, но поскольку вы не знаете имен, вас интересуют только значения.

Свойство `Values` — это экземпляр не generic-интерфейса `ICollection`, поэтому для того чтобы использовать LINQ, мы должны каким-либо образом привести его к generic-последовательности. Один из вариантов — использовать метод `Cast<T>`, но более безопасно — использовать фильтр `OfType<T>`. Несмотря на то, что метод `Cast` мог бы выдать исключение в случае наличия элемента, который нельзя привести к желаемому типу, метод `OfType` фильтрует последовательность. Наконец, мы получаем из последовательности объект. В данном случае мы использовали `FirstOrDefault`, но более строгое ограничение вводится с помощью метода расширения `Single`.

Оба класса `SauceBéarnaise` и `Steak` реализуют интерфейс `IIngredient`. При конфигурировании объектов Spring.NET не накладывает никаких ограничений на то, сколько объектов данного интерфейса мы можем сконфигурировать, но он все равно позволяет нам разрешать их с помощью метода `GetObjectsOfType`:

```
var ingredients = context.GetObjectsOfType(typeof(IIngredient));
```

При наличии предыдущей конфигурации возвращаемый словарь `ingredients` будет содержать экземпляры как `SauceBéarnaise`, так и `Steak`, и для извлечения конкретных интересующих нас элементов мы можем использовать LINQ-запросы, как делали это в предыдущем примере.

Несмотря на то, что мы можем сконфигурировать несколько объектов `IIngredient` без имен, при необходимости мы можем присвоить им имена:

```
<objects xmlns="http://www.springframework.net">
  <object id="Sauce" type="SauceBéarnaise" />
  <object id="Meat" type="Steak" />
</objects>
```

Это позволяет нам разрешать каждый из объектов по его имени:

```
var meat = context.GetObject("Meat");  
var sauce = context.GetObject("Sauce");
```

Это не спасает нас от использования метода `GetObjectsOfType`, поэтому все предыдущие примеры также применимы.

Принимая как должное то, что мы всегда должны разрешать сервисы в единственном `Composition Root`, мы не должны ожидать возникновения неопределенности на этом уровне. Но для осуществления выбора между составными альтернативами при конфигурировании зависимостей этого потребителя мы можем использовать именованные объекты.

Конфигурирование именованных зависимостей

Интеграция объектов с именованными объектами – центральная возможность `Spring.NET`, несмотря на то, что возможности интеграции в `Spring.NET` ограничены. Даже если по возможности мы должны выбирать автоматическую интеграцию, существуют ситуации, когда нам необходимо обращаться к неопределенному API. В качестве примера рассмотрим приведенный ниже конструктор:

```
public ThreeCourseMeal(ICourse entrée,  
    ICourse mainCourse, ICourse dessert)
```

В этом примере у нас есть три идентично типизированных зависимости, каждая из которых представляет собой сущность, отличную от остальных. В большинстве случаев мы не хотим преобразовывать каждую зависимость в отдельный тип. В следующем листинге демонстрируется то, как мы могли бы сконфигурировать объекты `ICourse` и `ThreeCourseMeal`.

Листинг 12-1: Интеграция списка зависимостей

```
<object id="Entrée" type="Rillettes" />  
<object id="MainCourse" type="CordonBleu" />  
<object id="Dessert" type="MousseAuChocolat" />  
<object id="Meal" type="ThreeCourseMeal">  
    <constructor-arg ref="Entrée" />  
    <constructor-arg ref="MainCourse" />  
    <constructor-arg ref="Dessert" />  
</object>
```

Три реализации `ICourse` конфигурируются в виде именованных объектов. При конфигурировании объекта `ThreeCourseMeal` мы можем ссылаться на имена, когда интегрируем аргументы конструктора. Элемент `constructor-arg` также принимает необязательные атрибуты `name` или `index`, которые мы можем использовать, чтобы точно указать, на какой параметр мы ссылаемся. Но в этом примере мы перечисляем их все в соответствующем порядке.

Явное преобразование аргументов конструктора в именованные объекты – повсеместно применяемое решение. Сделать это мы можем, даже если конфигурируем именованные объекты в одном XML ресурсе, а конструктор – в совершенно другом ресурсе, поскольку единственное средство идентификации, которое связывает именованный объект с

аргументом, – это имя. Это всегда выполнимо, но может стать хрупким решением, если нам придется управлять большим количеством имен.

Когда основной причиной, побуждающей нас отказаться от автоматической интеграции, является неопределенность, наилучшее решение – сконструировать API, помогающее нам избавиться от этой неопределенности. Все это также приводит к лучшей конструкции в целом.

В следующем разделе вы увидите, как можно использовать менее неопределенный и более гибкий подход при котором разрешается использовать в обеде несколько блюд. Теперь вы должны изучить то, как Spring.NET работает со списками и последовательностями.

Интеграция последовательностей

В разделе 10.3.2 "Разработка пользовательского стиля существования" мы обсуждали, как выполнить рефакторинг явного класса `ThreeCourseMeal` к более универсальному классу `Meal`, который обладает приведенным ниже конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

Вы можете сконфигурировать Spring.NET так, чтобы интегрировать экземпляры `Meal` с соответствующими зависимостями `ICourse`, и я покажу вам, как это сделать. После рассмотрения этого вопроса вы должны будете приобрести хорошее понимание тех возможностей, которые доступны в случае необходимости конфигурирования экземпляров, имеющих последовательности зависимостей.

Автоматическая интеграция последовательностей

Spring.NET хорошо разбирается в массивах, но не в других типах последовательностей. Если нам необходим массив конкретных абстракций, и мы собираемся использовать все эти сконфигурированные абстракции, то в этом случае применяется автоматическая интеграция. К примеру, представьте себе, что класс `Meal` предлагает следующую перегрузку конструктора:

```
public Meal(params ICourse[] courses)
```

Если мы хотим, чтобы все сконфигурированные объекты `ICourse` были внедрены в `Meal`, то мы можем предоставить следующую конфигурацию:

```
<object id="Entrée" type="Rillettes" />
<object id="MainCourse" type="CordonBleu" />
<object id="Dessert" type="MousseAuChocolat" />
<object id="Meal" type="Meal" autowire="autodetect" />
```

Объект `Meal` конфигурируется для автоматической интеграции, и, поскольку Spring.NET по существу понимает массивы, он находит все объекты, которые реализуют интерфейс `ICourse`, и снабжает их конструктором `Meal`. Автоматически интегрировать массивы зависимостей довольно легко.

А теперь представьте себе, что перегрузки конструктора, которая принимает в качестве параметра массив `ICourse`, не существует. У вас имеется только конструктор, который принимает в качестве параметра `ICollection<ICourse>`. Хотя в этом случае

автоматическая интеграция и не работает, вы можете воспользоваться преимуществами встроенного понимания массивов, определив простой Decorator, экземпляры которого должны создаваться с помощью массива. Приведенный ниже листинг демонстрирует generic-реализацию. Не забывайте о том, что принятие IEnumerable<ICourse> в качестве параметра конструктора указывает на статически типизированный запрос данной конкретной зависимости. Все, что вам необходимо сделать, является столь же простым, как и преобразование этого запроса в запрос массива того же типа.

Листинг 12-2: Преобразование запросов последовательностей в запросы массивов

```
1. public class ArrayEnumerable<T> : IEnumerable<T>
2. {
3.     private readonly IEnumerable<T> sequence;
4.     public ArrayEnumerable(params T[] items)
5.     {
6.         if (items == null)
7.         {
8.             throw new ArgumentNullException("items");
9.         }
10.        this.sequence = items;
11.    }
12.    public IEnumerator<T> GetEnumerator()
13.    {
14.        return this.sequence.GetEnumerator();
15.    }
16. }
```

Строка 1: Определение последовательности

Строка 4: Необходим массив

ArrayEnumerable<T> реализует IEnumerable<T>, таким образом, он подходит для любого конструктора, которому необходима такая последовательность. С другой стороны, для него нужен массив такого же типа. Поскольку Spring.NET в сущности знает, как работать с массивами, ему может подойти закрытый ArrayEnumerable, снабженный всеми объектами, которые совпадают с типом элемента T.

Для того чтобы соответствующим образом интегрировать класс Meal со всеми объектами ICourse, вы можете сконфигурировать контекст следующим образом:

```
<object id="Entrée" type="Rillettes" />
<object id="MainCourse" type="CordonBleu" />
<object id="Dessert" type="MousseAuChocolat" />
<object id="Courses"
    type="ArrayEnumerable<ICourse>"
    autowire="autodetect" />
<object id="Meal" type="Meal" autowire="autodetect" />
```

Вы задаете Courses в виде ArrayEnumerable<ICourse> с включенной возможностью автоматической интеграции. Поскольку только для его конструктора необходим массив ICourse'ов, Spring.NET автоматически интегрирует его со всеми реализациями ICourse, которые только может найти: Rillettes, CordonBleu и MousseAuChocolat.

Для класса Meal нужен IEnumerable<ICourse>, и, кроме того, он конфигурируется таким образом, чтобы иметь возможность автоматически интегрироваться. Когда вы отправите Spring.NET запрос на разрешение объекта Meal, он будет искать сконфигурированный

объект, который реализует `IEnumerable<ICourse>`, и найдет объект `Courses`. Все три объекта `ICourse` будут внедрены в объект `Meal` посредством объекта `Courses`.

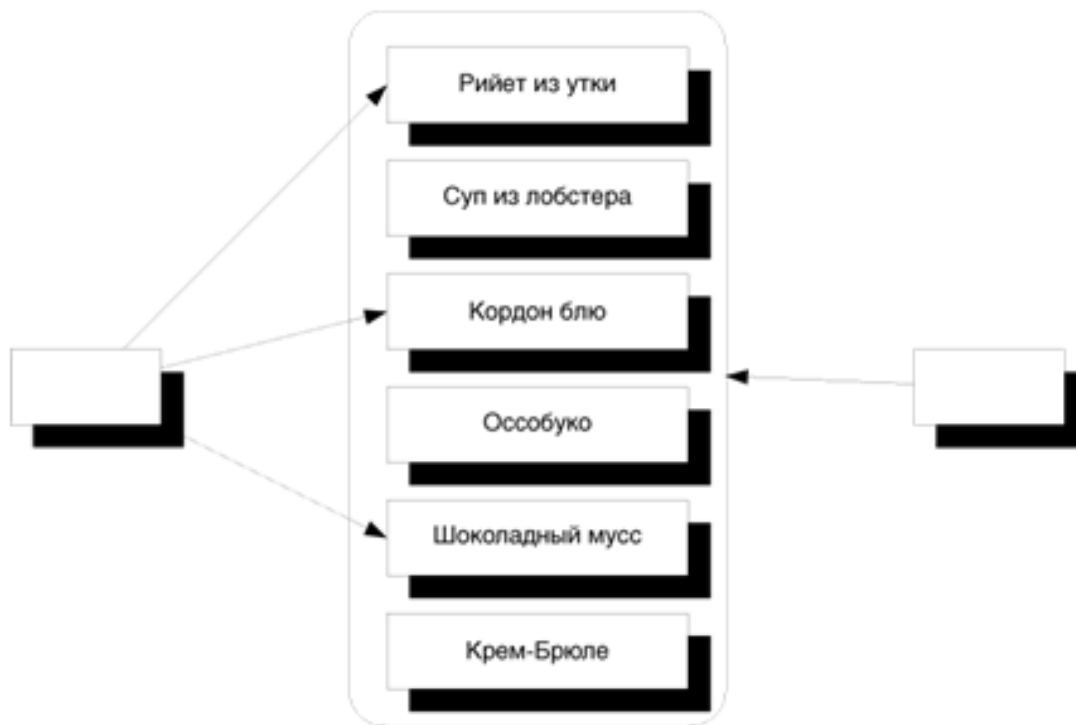
Класс `ArrayEnumerable<T>` – это небольшая забава, которая заполняет маленький пробел в Spring.NET. Это чисто инфраструктурный компонент, который можно упаковать в повторно используемую библиотеку.

Spring.NET автоматически обрабатывает массивы и, благодаря небольшой помощи `ArrayEnumerable<T>`, также обрабатывает и другие запросы последовательностей путем разрешения их в последовательности объектов, реализующих запрашиваемый тип. Единственное, что вам нужно сделать, – сконфигурировать `ArrayEnumerable` соответствующего типа элемента. Только в случае, когда вам необходимо явно отобразить несколько компонентов из большого набора, вам нужно выполнить больше действий. Это возможно благодаря более явной конфигурации.

Отбор нескольких объектов из большого набора

Когда мы используем возможность Spring.NET разрешать массивы все объекты внедряются в потребителей. Чаще всего это корректное поведение. Однако, как показано на рисунке 12-4, могут возникать ситуации, когда нам необходимо отобразить несколько компонентов из большого набора всех зарегистрированных компонентов.

Рисунок 12-4: В ситуации, продемонстрированной слева, мы хотим явным образом отобразить определенные зависимости из большого списка всех сконфигурированных объектов. Это отличается от ситуации, приведенной справа, когда мы отбираем все без разбора.



Когда мы ранее позволяли Spring.NET автоматически интегрировать все сконфигурированные объекты, это соответствовало ситуации, продемонстрированной в правой части рисунка 12-4. Если нам нужно сконфигурировать экземпляр так, как это

показано в левой части рисунка, мы должны явным образом определить, какие объекты необходимо использовать.

Это легко сделать с помощью именованных объектов, поскольку это более или менее идиоматичный способ конфигурирования Spring.NET, который позволяет использовать отдельный XML элемент `list` для обращения к конкретному сценарию. В следующем листинге продемонстрирован соответствующий пример.

Листинг 12-3: Внедрение именованных объектов в последовательность

```
1. <object id="Entrée" type="Rillettes" />
2. <object id="Entrée1" type="LobsterBisque" />
3. <object id="MainCourse" type="CordonBleu" />
4. <object id="Dessert" type="MousseAuChocolat" />
5. <object id="Meal" type="Meal">
6.     <constructor-arg>
7.         <list element-type="ICourse">
8.             <ref object="Entrée" />
9.             <ref object="MainCourse" />
10.            <ref object="Dessert" />
11.        </list>
12.    </constructor-arg>
13. </object>
```

Строка 7: Указывает на список

Строка 8-10: Именованные объекты

Элемент `list` можно использовать для указания на то, что следующие элементы являются элементами списка. Когда Spring.NET интегрирует список, он создаст массив, тип которого задан атрибутом `element-type`. Элемент `list` может содержать множество различных дочерних элементов. Элемент `ref` используется для обращения к другим именованным объектам.

При разрешении объекта `Meal` вы получите экземпляр `Meal` с `Rillettes`, `CordonBleu` и `MousseAuChocolat` в качестве содержащихся в нем блюд, при этом `LobsterBisque` не используется.

Еще раз вы видите, что Spring.NET по существу работает с массивами. Несмотря на отсутствие поддержки других типов последовательностей вы можете обойти это ограничение посредством заворачивания последовательностей в класс наподобие `ArrayEnumerable<T>`.

Потребители, которые полагаются на последовательности зависимостей, могут выступать в роли самого интуитивного использования составных экземпляров одной и той же абстракции. Но до того как мы перестанем заниматься этим вопросом, нам необходимо рассмотреть еще один случай, когда в дело вступают составные экземпляры.

Интеграция Decorator'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования `Decorator` полезен при реализации сквозных сущностей. По определению `Decorator`'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам `Decorator` и вложенный в

него тип. Если бы мы помещали Decorator'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных объектов, которые реализуют одну и ту же абстракцию. В отличие от предыдущих разделов эти объекты не являются концептуально равносильными, а зависят друг от друга. Я продемонстрирую вам два разных способа конфигурирования Spring.NET, используемых для работы с этим паттерном.

Создание обертки с помощью именованных объектов

На протяжении всей этой главы вы видели множество примеров того, как обращаться к именованным объектам как к аргументам конструктора. Кроме того, вы можете использовать этот идиоматичный подход для конфигурирования Decorator'ов.

Класс Breeding – это Decorator для IIngredient. Для получения экземпляра, который необходимо в него вложить, он использует Constructor Injection:

```
public Breeding(IIngredient ingredient)
```

Для того чтобы получить Cotoletta, вам хотелось бы вложить VealCutlet (еще один IIngredient) в класс Breeding. Поскольку вы уже знаете, как соединять именованные объекты с аргументами конструктора, вам будет привычно выполнять действия, аналогичные следующим:

```
<object id="Breeding" type="Breeding">  
  <constructor-arg ref="Cutlet" />  
</object>  
<object id="Cutlet" type="VealCutlet" />
```

Строка 2: Ссылка на именованный объект

На данный момент этот подход должен быть вам знаком. Для интеграции объекта Breeding с объектом Cutlet вы используете ссылку на именованный объект. Поскольку Spring.NET явным образом не работает с преобразованиями абстракций в конкретные типы, каждый из этих двух элементов является таким же самым объектом, как и остальные элементы object. То, что они оба реализуют интерфейс IIngredient, никак не влияет на способ их конфигурирования.

При разрешении имени Breeding вы получаете экземпляр Breeding, в который вложен VealCutlet.

Это общепринятый способ создания обертки для компонента, но в тех случаях, когда вас не интересует вложенный экземпляр, вы можете воспользоваться более неявным способом.

Создание обертки с помощью вложенных объектов

Если вам никогда не потребуется разрешать вложенные компоненты напрямую, вы можете воспользоваться более неявным способом создания их обертки. Представьте себе, что вам никогда не нужно будет разрешать VealCutlet непосредственно как IIngredient. Когда вам нужен IIngredient, вам всегда нужен Cotoletta.

В таких случаях нет нужды явным образом конфигурировать `VealCutlet` в виде независимого объекта. Вместо этого вы можете воспользоваться преимуществами синтаксиса вложенных объектов Spring.NET:

```
<object id="Breeding" type="Breeding">
  <constructor-arg>
    <object type="VealCutlet" />
  </constructor-arg>
</object>
```

Spring.NET позволяет вам задавать объекты в виде вложенных элементов. Вместо того чтобы ссылаться на именованный объект, элемент `constructor-arg` может содержать конфигурации всего объекта. Поскольку предполагается, что кроме как из конфигурации ссылаться на объект `VealCutlet` откуда-то еще вам не нужно будет, вы можете предоставить неименованный элемент `object` с корректным атрибутом `type`. Будучи вложенным в элемент `constructor-arg`, тип `VealCutlet` будет разрешаться в виде первого аргумента конструктора класса `Breeding`.

Существует несколько доступных вариантов конфигурирования Decorator'ов. В отличие от Castle Windsor Spring.NET явно не понимает Decorator'ы, что может показаться слегка удивительным, поскольку, как и Windsor, он предполагает максимальную поддержку паттерна Decorator: механизм перехвата.

Создание перехватчиков

В разделе 9.3.3 "Пример: перехват с помощью Windsor" вы видели пример того, как добавить в WCF-приложение обработчик ошибок и `Circuit Breaker` с помощью возможности динамического перехвата, предлагаемой Castle Windsor. Чтобы продемонстрировать возможности перехвата в Spring.NET и сравнить их с Castle Windsor и Unity, я разберу точно такой же пример, но реализованный с помощью Spring.NET.

Как показано на рисунке 12-5, добавление аспекта в Spring.NET – довольно простой процесс.

Рисунок 12-5: Простой процесс добавления аспекта в Spring.NET.



Основная часть работы заключается в разработке самого перехватчика, но после создания перехватчика необходимо добавить его в контейнер. Это, как и все другое, делается в XML конфигурации.

Однако мы не можем конфигурировать перехватчики, пока не реализуем их. Поэтому первый шаг – написать некоторый код для обработчика ошибок и перехватчиков `Circuit Breaker`. После создания обработчика ошибок и перехватчика `Circuit Breaker` мы можем с помощью них сконфигурировать контейнер.

Реализация перехватчика обработчика исключений

Реализация перехватчика в Spring.NET требует от нас реализации интерфейса `IMethodInterceptor`. В следующем листинге демонстрируется, как реализовать

стратегию обработки исключений из главы 9 "Механизм перехвата". Эта конкретная реализация, приведенная для Spring.NET, соответствует листингу 9-8, приведенному для Castle Windsor и листингу 14-13, приведенному для Unity.

Листинг 12-4: Реализация обработчика исключений `IMethodInterceptor`

```
1. public class ErrorHandlerInterceptor : IMethodInterceptor
2. {
3.     public object Invoke(IMethodInvocation invocation)
4.     {
5.         try
6.         {
7.             return invocation.Proceed();
8.         }
9.         catch (CommunicationException e)
10.        {
11.            this.AlertUser(e.Message);
12.        }
13.        catch (InvalidOperationException e)
14.        {
15.            this.AlertUser(e.Message);
16.        }
17.        return null;
18.    }
19.    private void AlertUser(string message)
20.    {
21.        var sb = new StringBuilder();
22.        sb.AppendLine("An error occurred.");
23.        sb.AppendLine("Your work is likely lost.");
24.        sb.AppendLine("Please try again later.");
25.        sb.AppendLine();
26.        sb.AppendLine(message);
27.        MessageBox.Show(sb.ToString(), "Error",
28.            MessageBoxButton.OK, MessageBoxImage.Error);
29.    }
30. }
```

Строка 3: Реализация логики перехвата

Строка 5-8: Попытка вернуть результат

Строка 9-16: Обработка исключений

Класс `ErrorHandlerInterceptor` реализует интерфейс `IMethodInterceptor`, который определяет только единственный метод `Invoke`. Именно здесь вы должны определить логику перехвата. Единственный аргумент метода – экземпляр интерфейса `IMethodInvocation`. Благодаря его методу `Proceed` вы пытаетесь вызвать вложенный метод и вернуть результат. Однако, поскольку цель перехватчика заключается в обработке известных исключений, вызов метода `Proceed` вставляется в блок `try`.

Если метод `Proceed` (или лучше вложенный метод, который вызывает `Proceed`) выдает одно из известных исключений, перехватчик ловит его и предупреждает пользователя об ошибке. В данном примере совокупность известных исключений жестко закодирована в самом перехватчике, но более общепринятая реализация могла бы вместо такого поведения подавить исключения или выдать их повторно в соответствии с внедренной спецификацией.

Поскольку метод `Invoke` должен возвращать объект, он возвращает `null`, когда выдается исключение и завершает свое выполнение. Это корректное значение для тех случаев, когда вложенный метод возвращает значение типа `void`, но когда метод возвращает реальные значения, это может стать проблемой, поскольку все это легко может привести к `NullReferenceExceptions`. Тем не менее, мы можем создать другой перехватчик, который устанавливает соответствующие значения по умолчанию для различных типов возвращаемого результата. Это было бы более корректно, нежели попытка предугадать корректное значение по умолчанию в рамках `ExceptionHandler`, являющегося универсальным перехватчиком, который можно использовать для перехвата любого интерфейса. Кроме того, это соответствовало бы принципу единичной ответственности.

`ExceptionHandler` заботится об обработке конкретных исключений, полученных для вложенного компонента. Данный компонент сам по себе может являться другим перехватчиком в виде `Circuit Breaker`.

Реализация `CircuitBreaker` перехватчика

`Circuit Breaker` перехватчик слегка более сложен, поскольку для него нужна зависимость `ICircuitBreaker`, но, как показывает следующий листинг, мы обращаемся к нему путем применения стандартного паттерна `Constructor Injection`. Когда дело доходит до компоновки класса, `Spring.NET` поступает с ним так же, как и с любым другим объектом, поэтому, пока он может разрешать зависимость, все идет хорошо.

Листинг 12-5: Реализация `Circuit Breaker` перехватчика `IMethodInterceptor`

```
1. public class CircuitBreakerInterceptor :
2.     IMethodInterceptor
3. {
4.     private readonly ICircuitBreaker breaker;
5.     public CircuitBreakerInterceptor(
6.         ICircuitBreaker breaker)
7.     {
8.         if (breaker == null)
9.         {
10.            throw new ArgumentNullException("breaker");
11.        }
12.        this.breaker = breaker;
13.    }
14.    public object Invoke(IMethodInvocation invocation)
15.    {
16.        this.breaker.Guard();
17.        try
18.        {
19.            var result = invocation.Proceed();
20.            this.breaker.Succeed();
21.            return result;
22.        }
23.        catch (Exception e)
24.        {
25.            this.breaker.Trip(e);
26.            throw;
27.        }
28.    }
29. }
```

Строка 1-2: Реализация `IMethodInterceptor`

Строка 4-13: Constructor Injection

Строка 19: Получения результата вложенного метода

Строка 20: Запись успешного результата

Строка 25: Отключение прерывателя

`CircuitBreakerInterceptor` необходимо делегировать его реальную реализацию экземпляру `ICircuitBreaker`. Поскольку Spring.NET, как и любой другой объект, может автоматически интегрировать перехватчик для внедрения `ICircuitBreaker` вы можете воспользоваться стандартным паттерном Constructor Injection.

В методе `Invoke` вам необходимо реализовать идиоматическое выражение `Guard-Succeed/Trip`, которое вы уже видели в листингах 9-4 и 9-9. Как и в листинге 12-4, вы вызываете вложенный метод путем вызова метода `Proceed`, но вместо того, чтобы сразу же вернуть значение, вам необходимо присвоить его локальной переменной `result`, чтобы вы могли идентифицировать успешный результат для `Circuit Breaker`. Не забывайте, что это может привести к закрытию другого открытого прерывателя.

Любое исключение, которое может выдаваться вложенным методом, выдается заново неизменным с помощью метода `Proceed`, поэтому вы можете поймать его и отключить прерыватель, как вы обычно и делаете.

После того как и `ErrorHandlerInterceptor`, и `CircuitBreakerInterceptor` реализованы, приходит время конфигурирования контейнера таким образом, чтобы он создавал обертку для объекта `IProductManagementAgent`.

Конфигурирование механизма перехвата

Все, что нам фактически необходимо сделать, – перехватить объект `IProductManagementAgent` вместе с `Circuit Breaker` и обработчиком ошибок таким образом, чтобы в тех случаях, когда при взаимодействии с веб-сервисом возникает исключение, открывался `Circuit Breaker` и обрабатывалось исключение, что давало бы приложению возможность восстановиться, как только веб-сервис или сеть восстановят свою работу.

Конфигурировать механизм перехвата в Spring.NET довольно легко. Все, что вам нужно сделать, – выполнить конфигурацию самих перехватчиков:

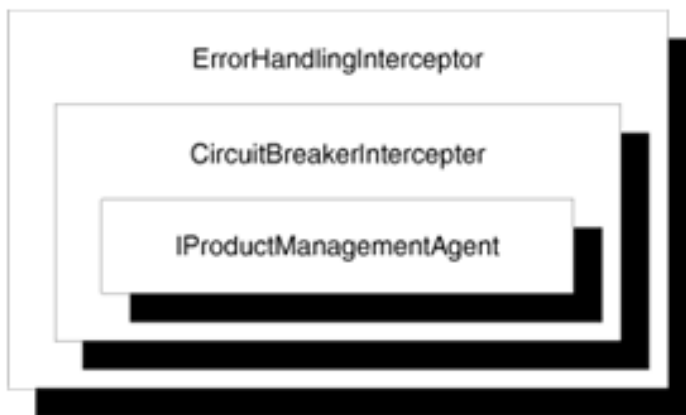
```
<objects xmlns="http://www.springframework.net"
    default-autowire="constructor">
    <object id="ErrorHandlerInterceptor"
        type="ErrorHandlerInterceptor" />
    <object id="CircuitBreakerInterceptor"
        type="CircuitBreakerInterceptor" />
</objects>
```

Не обладая особой фантазией, я присвоил объектам `id`, совпадающие с их `type`. Обратите внимание на то, что Constructor Injection, основанный на автоматической интеграции, включен по умолчанию. В то время как `ErrorHandlerInterceptor` обладает конструктором по умолчанию, `CircuitBreakerInterceptor` для запрашивания

ICircuitBreaker использует Constructor Injection. Механизм автоматической интеграции работает как для CircuitBreakerInterceptor, так и для ErrorHandlerInterceptor, а также для большинства других объектов конфигурации, поэтому включение его по умолчанию является самым простым способом.

После того как вы разместили перехватчики, осталось только сконфигурировать объект IProductManagementAgent с необходимыми перехватчиками. На рисунке 12-6 продемонстрирована нужная вам конфигурация.

Рисунок 12-6: IProductManagementAgent должен быть вложен в Circuit Breaker перехватчик таким образом, чтобы при выдаче агентом исключения цепь на некоторое время открывалась. Поскольку Circuit Breaker лишь регистрирует исключения, а не обрабатывает их, за это несет ответственность перехватчик обработчика ошибок, который должен, по крайней мере, уметь обрабатывать исключения, возникающие в обоих агентах, а также в Circuit Breaker.



Как показывает следующий листинг, делается это при помощи синтаксиса конфигурации XML, а также специального пространства имен и классов, которые Spring.NET предоставляет для этих целей. Основная концепция конфигурирования механизма перехвата заключается в отделении того, что делать, от того, где это делать. Мы должны предоставить ответ на оба эти вопроса, но отвечаем на них по отдельности, а затем связываем ответы вместе.

Листинг 12-6: Конфигурирование перехватчиков

```
1. <object type="WcfProductManagementAgent" />
2. <object id="AgentPointCut"
3.     type="RegularExpressionMethodPointcut">
4.     <property name="patterns">
5.         <list>
6.             <value>.*WcfProductManagementAgent.*</value>
7.         </list>
8.     </property>
9. </object>
10. <aop:config>
11.     <aop:advisor pointcut-ref="AgentPointCut"
12.         advice-ref="ErrorHandlerInterceptor"
13.         order="1" />
14.     <aop:advisor pointcut-ref="AgentPointCut"
15.         advice-ref="CircuitBreakerInterceptor"
16.         order="2" />
17. </aop:config>
```

Строка 1: Объект для перехвата

Строка 2-9: Указание того, в каком месте выполнять перехват

Строка 10-17: Привязка перехватчиков к спецификации

В предыдущем коде вы регистрировали перехватчики, но помимо этого вам необходимо зарегистрировать классы, которые нужно будет перехватить. В этом примере один такой класс, но при желании вы можете перехватить множество различных классов с помощью одного и того же набора перехватчиков.

Чтобы указать, какие классы или методы нужно перехватить, вы должны задать так называемое *Pointcut*, придуманное название для правила, определяющего, что нужно перехватить. Если вы вспомните первоначальное введение в механизм перехвата, приведенное в главе 9, то поймете, что *Pointcut* соответствует

`IModelInterceptorsSelector`, используемому в Castle Windsor и реализованному в листинге 9-10. Как и Castle Windsor, Spring.NET позволяет писать императивный код, определяющий *Pointcut*, но помимо этого он предоставляет некоторые декларативные *Pointcut*'ы. Одним из таких статических *Pointcut*'ов является

`RegularExpressionMethodPointcut`, который можно использовать для задания соответствующего правила с регулярным выражением. При каждом вызове метода он будет пытаться сопоставить полное имя метода с регулярным выражением. В этом конкретном случае вы планируете сопоставить только члены класса

`WcfProductManagementAgent`.

Наконец, вам нужно связать *Pointcut* с перехватчиками, которые вы уже регистрировали ранее. Делается это с помощью последовательности элементов `advisor`, которые объявляют перехватчики и порядок их компоновки. Заметьте, что поскольку вы сначала указываете `ErrorHandlingInterceptor`, он становится крайним перехватчиком, перехватывающим `CircuitBreakerInterceptor`.

Последнее, что вам нужно сделать для того, чтобы сконфигурировать приложение, в котором используется сильное управление внешним взаимодействием, – убедиться, что все зависимости могут удовлетворять требуемым условиям. Поскольку для `CircuitBreakerInterceptor` нужен `ICircuitBreaker`, вы также должны сконфигурировать и этот объект:

```
<object type="CircuitBreaker" autowire="no">
  <constructor-arg value="00:01:00" />
</object>
```

Для конструктора `CircuitBreaker` нужна задержка в виде экземпляра `TimeSpan`, и вы будете задавать это элементарное значение как вложенное. Для этого вы должны отключить используемые по умолчанию настройки автоматического интегрирования с тем, чтобы явно установить задержку, равной одной минуте.

Для большей эффективности важно наличие только одного экземпляра `Circuit Breaker` (по крайней мере, для одного внешнего ресурса), но, поскольку по умолчанию используется область применения `Singleton`, вам не нужно явным образом выражать данное требование.

Этот пример продемонстрировал, как реализовать механизм динамического перехвата в Spring.NET. По-моему, я получил сложность, сравнимую с поддержкой контейнерами Castle Windsor и Unity механизма перехвата. Будучи не таким уже незначительным, полученное преимущество становится довольно значимым.

Механизм перехвата – динамическая реализация паттерна Decorator, а сам паттерн Decorator является сложным применением составных объектов одного и того же типа. Spring.NET дает нам возможность работать с составными компонентами несколькими различными способами. Мы можем конфигурировать их в виде альтернатив друг другу, в виде пиров, которые разрешаются в виде последовательностей, или в виде иерархических Decorator'ов, или даже в виде перехватчиков. Когда дело дойдет до массивов, Spring.NET поймет, что делать, но при этом мы часто можем преобразовывать в массивы и другие типы последовательностей, используя такой адаптер, как класс `ArrayEnumerable<T>`. Кроме того, все это позволяет нам явным образом задавать, как komponуются сервисы в случае, если нам необходим более явный контроль.

Помимо этого такие ситуации могут возникать, когда нам приходится иметь дело с API, отклоняющимися от Constructor Injection. До настоящего момента вы наблюдали за тем, как конфигурировать объекты, включая то, как определять области применения, и как работать с составными компонентами. Но до этого момента мы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют Constructor Injection. Поскольку это не всегда происходит именно так, в следующем разделе мы сделаем краткий обзор того, как работать с классами, экземпляры которых должны создаваться особым образом.

12.4. Конфигурирование сложных API

До настоящего момента мы рассматривали то, как можно конфигурировать компоненты, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры, например, Spring.NET, могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей.

Все становится менее понятным, когда API не столь хорошо функционируют. В этом разделе вы увидите, как работать с простейшими аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Давайте начнем с рассмотрения классов, которые принимают в качестве параметров простейшие типы, например, строки и целые числа.

Конфигурирование простейших зависимостей

Пока мы внедряем абстракции в потребителей, все в порядке. Но этот процесс усложняется, если конструктор зависит от простейшего типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве компонента контейнера не имеет особого смысла, а в Spring.NET невозможно зарегистрировать значение простейшего типа в качестве объекта.

Рассмотрим в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются *code smell*'ами и их нужно преобразовывать в полиморфные классы (имеющие разное состояние). Тем не менее, для этого примера они вполне нам подходят.

Необходимо явным образом передать значение параметра конструктора `spiciness` в виде составляющей конфигурации объекта `ChiliConCarne`:

```
1. <object id="Course" type="ChiliConCarne">
2.   <constructor-arg value="Hot" />
3. </object>
```


Строка 2: Передача значения

Spring.NET снабжен конвертерами нескольких типов, которые конвертируют текстовые элементы в экземпляры необходимых типов. Один из встроенных конвертеров преобразует текст в элемент перечисляемого типа, что позволяет нам сделать текст "Hot" значением элемента `constructor-arg`. Spring.NET смотрит на тип параметра конструктора класса `ChiliConCarne`, определяет, что это параметр перечисляемого типа и использует конвертер соответствующего типа для преобразования текста "Hot" в значение `Spiciness.Hot`.

Эта возможность Spring.NET предназначена для тех ситуаций, в которых нам необходимо передавать простейшие значения в виде аргументов. В примере аргумент `ChiliConCarne` был передан в конструктор, но иногда встречаются ситуации, когда класс не имеет открытого конструктора.

Конфигурирование статических фабрик

Экземпляры некоторых классов нельзя создать с помощью открытого конструктора. В этом случае для создания экземпляров типа мы должны использовать некоторого рода фабрику. Это всегда проблематично для DI-контейнеров, поскольку по умолчанию им нужны открытые конструкторы.

Рассмотрим приведенный ниже пример конструктора открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже если класс `JunkFood` является открытым, конструктор расположен внутри него. Очевидно, экземпляры `JunkFood` должны создаваться с помощью статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С точки зрения Spring.NET, это проблемное API, поскольку в нем отсутствуют точно выраженные и заданные соглашения касательно статических фабрик. Тут требуется помощь — и мы можем предоставить ее при конфигурировании объекта:

```
<object id="Meal"
        type="JunkFoodFactory"
        factory-method="Create">
    <constructor-arg value="chicken meal" />
</object>
```

Как и всегда, конфигурация объекта выражается в элементе `object`, но вместо задания типа самого объекта, вы задаете тип фабрики. Помимо этого вы должны определить, что именем `factory-method` будет `Create`. Заметьте, что хотя метод `Create` по существу и не является конструктором, вы все равно используете элемент `constructor-arg` для того, чтобы задать значение аргумента `name` метода `Create`.

Хотя атрибут `type` задается как тип фабрики, а не тип результата, Spring.NET достаточно умен, чтобы понять, что выходной тип метода `Create` — это `JunkFood`. Это означает, что вы можете не использовать атрибут `id` и создать неименованный объект, как вы это делали в разделе 12.3.1 "Выбор из составных кандидатов", и вы все равно сможете разрешить объект `JunkFood` с помощью метода `GetObjectsOfType`.

Последним рассматриваемым нами отклонением от паттерна `Constructor Injection` является `Property Injection`.

`Property Injection` — это менее определенная форма механизма внедрения зависимостей, поскольку компилятор не принуждает нас задавать значение свойства, доступного для записи. Все-таки, Spring.NET понимает `Property Injection` и работает с ним интуитивным образом. Если мы разрешим автоматическое интегрирование, то `Property Injection` будет работать, но, помимо этого, он будет работать и посредством прямого интегрирования.

Рассмотрим класс `CaesarSalad`:

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

По всеобщему заблуждению в состав салата "Цезарь" входит курица, но это не правда. По существу "Цезарь" является салатом, но, поскольку с курицей он вкуснее, то ее часто предлагают использовать в нем в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Если вы конфигурируете только класс `CaesarSalad`, явно не обращаясь к свойству `Extra`, то этому свойству не будет присвоено значение. Вы все равно можете разрешать экземпляр, но свойство `Extra` будет иметь значение по умолчанию, которое ему присвоил конструктор (если только это имеет место).

Явное интегрирование свойств

Существует несколько способов, с помощью которых можно сконфигурировать `CaesarSalad` таким образом, чтобы свойство `Extra` заполнялось соответствующим образом. Один из таких способов — явным образом интегрировать свойство с именованным объектом:

```
<object id="Course" type="CaesarSalad">
  <property name="Extra" ref="Chicken" />
</object>
<object id="Chicken" type="Chicken" />
```

Элемент `property` указывает на то, что именем свойства является "Extra", и что в качестве его значения необходимо использовать объект `Chicken`. Вместо того чтобы использовать ссылку на именованный объект, вы можете использовать вложенный объект:

```
<object id="Course" type="CaesarSalad">
  <property name="Extra">
    <object type="Chicken" />
  </property>
</object>
```

Вы всегда можете явным образом интегрировать свойства с элементом `property`, но поскольку свойство идентифицируется по имени, это довольно хрупкий подход. Если вы позднее переименуете свойство, большинство инструментов рефакторинга не смогут идентифицировать и изменять значение атрибута `name` в различных XML-файлах. Это может привести к ошибкам во время выполнения.

Наилучшее решение – автоматически интегрировать объекты.

Автоматическое интегрирование свойств

Как вы помните из раздела 12.1.2 "Конфигурирование контейнера", вы должны явно разрешить автоматическое интегрирование в Spring.NET, но как только вы это сделаете, заработает Property Injection. Если зависимость не удовлетворяет условиям, то свойство игнорируется:

```
<objects xmlns="http://www.springframework.net">
  <object id="Course" type="CaesarSalad"
    autowire="autodetect" />
</objects>
```

В этом примере объект `Course` – единственный объект, сконфигурированный для контейнера. Несмотря на то, что он сконфигурирован с возможностью автоматической интеграции, свойству `Extra` никогда не будет присвоено значение, поскольку отсутствует доступный объект `IIngredient`. Выдается исключение, а свойство просто игнорируется.

Все меняется, как только объект `IIngredient` становится доступным:

```
<object id="Course" type="CaesarSalad"
  autowire="autodetect" />
<object type="Chicken" />
```

Теперь при разрешении объекта `Course` вы будете получать экземпляр `CaesarSalad` со свойством `Extra`, имеющим значение `Chicken`.

Использование автоматического интегрирования – более ясный способ, поскольку мы можем переименовывать свойства, не опасаясь, что конфигурация Spring.NET будет разрушена во время выполнения.

Вы видели, как использовать Spring.NET для работы с более сложными API разработки. В общем случае вы всегда можете явным образом конфигурировать интегрирование с помощью XML-конфигурации, но вы также видели и то, что паттерн Property Injection можно сконфигурировать таким образом, что он будет поддерживать возможность автоматической интеграции.

12.5. Резюме

Среди рассматриваемых в этой главе DI-контейнеров Spring.NET уникален тем, что это единственный контейнер, реализованный как порт из Java. И это более очевидный факт, чем то, что Spring.NET в большой степени полагается на XML-конфигурацию.

Большинство из старейших DI-контейнеров в начале своего пути были сильно сфокусированы на XML. Но если другие DI-контейнеры продвинулись вперед, Spring.NET не сделал ни одного шага в эту сторону.

Несмотря на то, что в последующих версиях Spring.NET возможно в большей степени будут поддерживаться технологии использования кода в качестве конфигурации и даже конфигурация на основании соглашений, применение Spring.NET на сегодняшний день несет за собой большое количество XML-кода.

Возможно, самым слабым местом Spring.NET является ограниченная поддержка механизма управления жизненным циклом. С одной стороны, Spring.NET, как и StructureMap, явным образом не поддерживает высвобождение диаграммы объектов (это можно рассматривать как отсутствие каких-либо возможностей). С другой стороны, Spring.NET не поддерживает пользовательские области применения объектов.

Хотя возможности механизма управления жизненными циклами не столь примечательны, Spring.NET – один из нескольких DI-контейнеров, предлагающих встроенный механизм перехвата.

Spring.NET все еще базируется на .NET 1.1, что, возможно, для некоторых является и преимуществом. Мы получаем полноценный пакет документации и возможность покупки коммерческого сопровождения. Поэтому, несмотря на то, что в целом Spring.NET может показаться старомодным, он все равно является одним из профессиональных пакетов.

Мы можем рассматривать Spring.NET как устаревший, но проверенный фреймворк. В противоположность этому в следующей главе рассматривается один из самых современных DI-контейнеров: Autofac.

13. Autofac

Меню:

- Знакомство с Autofac
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В предыдущих главах вы наблюдали за тем, как применять к конкретным DI-контейнерам принципы и паттерны, описанные в частях 1-3. В этой главе мы то же самое будем делать для DI-контейнера Autofac.

Autofac – это пример так называемых "DI-контейнеров второго поколения".

Примечание

Примерами DI-контейнеров второго поколения являются DI-контейнеры Ninject и Unity.

В основе DI-контейнеров второго поколения лежит .NET 3.5, поэтому их структуры чаще всего базируются напрямую на возможностях языка и платформы, которые были доступны в то время. Они не только отлично понимают дженерики, но и используют множество лямбда-выражений в качестве центральных API элементов. Несмотря на то, что большинство развитых DI-контейнеров также поддерживают эти более современные конструкции языка, в основе их центральных движков обычно лежат более ранние версии .NET. Поскольку контейнеры второго поколения не обладают таким наследием, обычно в их основе заложены самые базовые составляющие этих возможностей.

Autofac – это достаточно исчерпывающий DI-контейнер, имеющий тщательно спроектированный и логичный API. Он начал развиваться с конца 2007 года и, кажется, имеет достаточно обширную клиентскую базу.

В этой главе мы рассмотрим то, как можно использовать Autofac для применения принципов и паттернов, описанных ранее в частях 1-3. На рисунке 13-1 продемонстрирована структура этой главы.

Рисунок 13-1: Эта глава состоит из четырех разделов. Первый раздел знакомит нас с API Autofac и должен рассматриваться в качестве предпосылки следующих трех разделов. Каждый из четырех разделов может быть прочитан независимо от остальных разделов, несмотря на то, что в четвертом разделе используются методы, которые вводятся в третьем разделе. Тема четвертого раздела значительно отличается от темы третьего, поэтому читать их можно независимо друг от друга. Но, все-таки, вам может понадобиться более детальная информация о некоторых составляющих API.



Эта глава должна дать вам возможность приступить к работе, а также справиться с большинством общепринятых проблем, которые могут возникнуть при ежедневном использовании Autofac. Это не окончательная трактовка Autofac, поскольку всецелое его рассмотрение заняло бы несколько глав или, может быть, даже целую книгу. Но если вам хочется получить больше информации об Autofac, то лучше всего начать изучения с главной страницы сайта Autofac – <http://autofac.org>.

Вы можете прочитать эту главу, не читая при этом остальные главы части 4, специально для того, чтобы познакомиться с Autofac, или можете прочитать ее совместно с остальными главами части 4, чтобы сравнить DI-контейнеры. Цель данной главы – показать, как Autofac связан с паттернами и принципами, описанными в частях 1-3, и как он их реализует.

[13.1. Знакомство с Autofac](#)

[13.2. Управление жизненным циклом](#)

[13.3. Работа с составными компонентами](#)

[13.4. Регистрация сложных API](#)

[13.5. Резюме](#)

13.1. Знакомство с Autofac

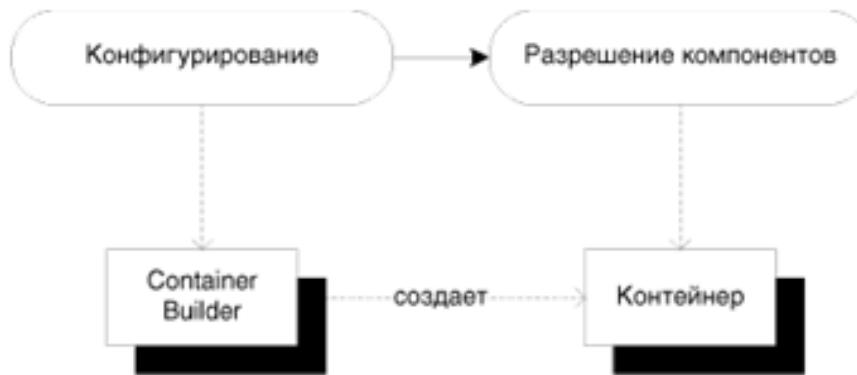
Из этого раздела вы узнаете, где можно взять Autofac, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим универсальные варианты конфигурирования, а также то, как пакетировать настройки конфигурации в повторно используемые компоненты. В таблице 13-1 содержится основополагающая информация, которая, скорее всего, понадобится вам для того, чтобы приступить к работе с Autofac

Таблица 13-1: Краткая информация об Autofac

Вопрос	Ответ
Откуда мне его получить?	Перейдите на страницу http://autofac.org и нажмите на соответствующую ссылку в списке рекомендуемых загрузок Из Visual Studio 2010 можно получить его посредством NuGet. Имя пакета – <i>Autofac</i> .
Что находится в загруженном файле?	Можно загрузить zip-файл, содержащий предварительно скомпилированные бинарные файлы. Кроме того, можно загрузить исходный код и скомпилировать его самостоятельно, хотя при этом будет трудно определить, какие изменения к какому релизу относятся. Последняя составляющая номера сборки (например, для используемой в этой главе сборки это 724) соответствует ревизии исходного кода, но для того, чтобы ее определить, понадобится система управления версиями Mercurial.
Какие платформы поддерживаются?	Поддерживаются версии .NET 3.5 SP1, .NET 4, Silverlight 3, Silverlight 4. Кроме того, доступны и другие версии, поддерживающие .NET 2.0, 3.0 и Silverlight 2 (для этого во вкладке Download (Загрузить) выберите пункт All Releases (Все релизы)).
Сколько он стоит?	Нисколько. Это программное обеспечение с открытым исходным кодом.
Откуда мне получить помощь?	Коммерческую поддержку можно получить от компаний, связанных с разработчиками Autofac. Подробную информацию можно получить на сайте http://code.google.com/p/autofac/wiki/CommercialSupport . Невзирая на коммерческую поддержку, Autofac все еще остается программным обеспечением с открытым исходным кодом, которое обладает процветающей экосистемой, поэтому, скорее всего (но не гарантированно), помощь можно получить на официальном форуме http://groups.google.com/group/autofac
На какой версии Autofac основана эта глава?	2.4.5.724.

Процесс использования Autofac слегка отличается от процесса использования других DI-контейнеров. Как показывает рисунок 13-2, это более явный процесс, состоящий из двух шагов: сначала мы конфигурируем ContainerBuilder, а затем с помощью него создаем контейнер, который можно использовать для разрешения компонентов.

Рисунок 13-2: При работе с Autofac сначала создается и конфигурируется экземпляр `ContainerBuilder`. Затем с помощью него создается контейнер, который впоследствии можно использовать для разрешения компонентов. Обратите внимание на то, что последовательность действий больше похожа на работу с контейнерами Castle Windsor или StructureMap: конфигурируем, а затем разрешаем. Тем не менее, концепция разделения понятий при работе с Autofac гораздо понятнее. `ContainerBuilder` не умеет разрешать компоненты, и поэтому мы не сможем сконфигурировать контейнер.



После прочтения этого раздела вы приобретете полноценное понимание всей сущности паттерна применения Autofac и сможете начать использовать его в сценариях, в которых все компоненты руководствуются должными DI-паттернами, например, Constructor Injection. Давайте начнем с простейшего сценария и посмотрим, как можно разрешать объекты с помощью контейнера Autofac.

Разрешение объектов

Основная услуга, предоставляемая любым DI-контейнером – разрешение компонентов. В этом разделе мы рассмотрим API, которое позволяет разрешать компоненты при помощи Autofac.

Вспоминая обсуждение процесса разрешения компонентов при помощи Castle Windsor и StructureMap, вы, возможно, припомните, что Windsor требует, чтобы перед тем, как стало возможным разрешение компонентов, все соответствующие компоненты должны быть зарегистрированы. В отличие от Castle Windsor StructureMap делает все возможное, чтобы сделать это за нас, когда мы запрашиваем конкретные типы с открытыми конструкторами. Autofac может вести себя и как Castle Windsor, и как StructureMap, хотя по умолчанию его поведение совпадает с поведением Castle Windsor. Перед разрешением компонентов мы должны их зарегистрировать, поэтому одним из самых простых способов применения Autofac является приведенный ниже пример:

```
var builder = new ContainerBuilder();
builder.RegisterType<SauceBéarnaise>();
var container = builder.Build();
SauceBéarnaise sauce = container.Resolve<SauceBéarnaise>();
```

Как уже было замечено ранее на рисунке 13-2, для конфигурирования компонентов необходим экземпляр `ContainerBuilder`. В примере выше регистрируется конкретный класс `SauceBéarnaise` с помощью `builder`, чтобы при запросе создания контейнера выходящий экземпляр `container` компоновался классом `SauceBéarnaise`. Все это дает возможность разрешать класс `SauceBéarnaise` из контейнера.

Если компонент `SauceBéarnaise` не был зарегистрирован, то попытка разрешить его приведет к исключению `ComponentNotRegisteredException`.

Если сравнивать этот самый простейший из всех сценариев с аналогичными фрагментами кода `Castle Windsor` и `StructureMap`, то `Autofac` покажется слегка более подробным. Однако вся эта подробность возникает, главным образом, благодаря дополнительному шагу создания контейнера из `ContainerBuilder`, поэтому в более крупных и сложных конфигурациях `Autofac` будет сравним с другими DI-контейнерами.

По умолчанию `Autofac` требует явно регистрировать все соответствующие компоненты. Такое поведение присуще и `Castle Windsor`. Если вам необходимо поведение, более похожее на `StructureMap`, то можно поступить следующим образом:

```
var builder = new ContainerBuilder();
builder.RegisterSource(
    new AnyConcreteTypeNotAlreadyRegisteredSource() );
var container = builder.Build();
SauceBéarnaise sauce = container.Resolve<SauceBéarnaise>();
```

Единственное отличие от предыдущего примера – отсутствие явной регистрации класса `SauceBéarnaise`. Вместо этого вы регистрируете `IRegistrationSource`, называемый `AnyConcreteTypeNotAlreadyRegisteredSource`. Это название довольно трудно произносить, но оно более или менее отражает функцию

`AnyConcreteTypeNotAlreadyRegisteredSource`: он выступает в роли источника регистраций любого конкретного типа, который ранее не был зарегистрирован. При добавлении `AnyConcreteTypeNotAlreadyRegisteredSource` не нужно явно добавлять тип `SauceBéarnaise`, поскольку `SauceBéarnaise` является конкретным классом с открытым конструктором, а источник регистраций может автоматически обеспечивать его регистрацию.

Источники регистраций

Дополнительная особенность `Autofac` – способность предоставлять большее количество дополнительных источников регистраций, нежели при прямом использовании API, которое раскрывается с помощью `ContainerBuilder`. Это механизм расширяемости, который `Autofac` использует для реализации различных возможностей, но, поскольку он основан на открытом интерфейсе под названием `IRegistrationSource`, можно использовать его и как механизм расширяемости.

Единственной открытой реализацией `IRegistrationSource`, которая входит в состав `Autofac`, является `AnyConcreteTypeNotAlreadyRegisteredSource`, которую вы уже видели ранее, но в `Autofac` есть и другие внутренние реализации интерфейса.

Идея, лежащая в основе `IRegistrationSource`, заключается в том, что реализации могут выступать в роли резервных механизмов или более эвристических источников регистраций компонентов, чем те, которые может обеспечить обычное API. Помимо вышеупомянутых источников конкретных типов использовать `IRegistrationSource` можно также и для того, чтобы превратить `Autofac` в автоматический mock-контейнер. Все это выходит за рамки этой книги, и для эффективного использования `Autofac` нет необходимости более подробно рассматривать `IRegistrationSource`.

Фактически, регистрация `AnyConcreteTypeNotAlreadyRegisteredSource` приводит к тому, что контейнер начинает вести себя как `StructureMap`, а не как `Castle Windsor`. Теперь контейнер может не только разрешать конкретные типы, обладающие конструкторами по умолчанию, но и автоматически интегрировать тип с другими конкретными зависимостями, не используя при этом явную регистрацию. Кроме того, сразу после того, как было введено слабое связывание, `Autofac` необходимо сконфигурировать на преобразование абстракций в конкретные типы.

Преобразование абстракций в конкретные типы

Несмотря на то, что время от времени способность `Autofac` автоматически интегрировать конкретные типы может становиться довольно полезной, обычно слабое связывание подразумевает преобразование абстракций в конкретные типы. Создание экземпляров на основании таких преобразований – ключевая возможность, которую предлагает любой `DI`-контейнер, но для начала нужно еще определить такие преобразования.

В приведенном ниже примере выполняется преобразование интерфейса `IIngredient` в конкретный класс `SauceBéarnaise`, что позволяет успешно разрешать `IIngredient`:

```
var builder = new ContainerBuilder();
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
var container = builder.Build();
IIngredient ingredient = container.Resolve<IIngredient>();
```

Для регистрации типов и определения преобразований используется экземпляр `ContainerBuilder`. Метод `RegisterType` дает возможность зарегистрировать конкретный тип. Как вы уже видели в первом примере этой главы, можно остановиться и на этом, если вашей целью является только регистрация класса `SauceBéarnaise`. Кроме того, при помощи метода `As` можно продолжить пример и определить, каким образом необходимо регистрировать конкретный тип.

Примечание

При использовании `Autofac` мы начинаем работать с конкретным типом, а затем преобразовываем его в абстракцию. Такая последовательность полностью противоположна работе с большинством других `DI`-контейнеров: в них абстракция преобразуется к конкретному типу.

Предупреждение

По сравнению с `Castle Windsor` и `StructureMap` в `Autofac`, в сущности, отсутствуют ограничители `generic`-типа между типами, определенными методами `RegisterType` и `As`. А это означает, что можно преобразовывать несовместимые типы. Код будет компилироваться, но во время выполнения, когда `ContainerBuilder` будет создавать контейнер, возникнет исключение.

В большинстве случаев нам нужно только `generic API`. Несмотря на то, что `generic API`, в отличие от других `DI`-контейнеров, не обеспечивает должную типовую безопасность, это все равно хороший способ конфигурирования контейнера. Однако в некоторых ситуациях необходим и более слабо типизированный способ разрешения сервисов. Это тоже возможно.

Разрешение слабо типизированных сервисов

В некоторых случаях нельзя использовать generic API, поскольку на этапе проектирования может быть неизвестно, какой тип нужен. Все, что у нас есть – экземпляр `Type`, но, все же, нам может потребоваться и экземпляр этого типа. Пример такой ситуации приведен в разделе 7.2 "Построение ASP.NET MVC приложений", где мы обсуждали ASP.NET MVC класс `DefaultControllerFactory`. Соответствующий метод приведен ниже:

```
protected internal virtual IController GetControllerInstance(  
    RequestContext requestContext, Type controllerType);
```

Поскольку в наличии у нас только экземпляр `Type`, мы не можем использовать дженерики, и должны прибегнуть к слабо типизированному API. В Autofac есть слабо типизированная перегрузка метода `Resolve`, которая позволяет реализовывать метод `GetControllerInstance` следующим образом:

```
return (IController) this.container.Resolve(controllerType);
```

Слабо типизированная перегрузка метода `Resolve` позволяет передавать параметр `controllerType` прямо в Autofac, но при этом необходимо приводить возвращаемое значение к `IController`.

Независимо от используемой перегрузки метода `Resolve` Autofac гарантирует, что эта перегрузка будет возвращать экземпляр необходимого типа или выдавать исключение в случае отсутствия подходящих зависимостей. После того, как все необходимые зависимости должным образом сконфигурированы, Autofac может автоматически интегрировать необходимый тип.

В предыдущем примере `this.container` – это экземпляр `Autofac.IContainer`. Для обеспечения возможности разрешать необходимый тип все слабо связанные зависимости необходимо сначала сконфигурировать. Autofac можно конфигурировать несколькими способами. В следующем разделе приведена информация о самых универсальных из этих способов.

Конфигурирование ContainerBuilder

Как уже говорилось в разделе 3.2 "Конфигурирование DI-контейнеров", существует несколько концептуально разных способов конфигурирования DI-контейнера. На рисунке 13-3 представлен обзор возможных вариантов.

Рисунок 13-3: Концептуально разные варианты конфигурирования. Использование кода в качестве конфигурации подразумевает строгую типизированность и имеет тенденцию к явному определению. XML, с другой стороны, предполагает позднее связывание, но тоже склонно к явному определению. Автоматическая регистрация, напротив, полагается на соглашения, которые могут быть и строго типизированными, и более слабо определенными.



Будучи DI-контейнером второго поколения, Autofac первоначально не имел в своей основе XML конфигурации, но позднее его API было заменено на программное API конфигурации подобно некоторым наиболее развитым контейнерам. Конечно, его создавали с заложенной в него возможностью эффективно использовать множество различных источников конфигурации, и XML – это один из возможных вариантов.

Центральное API конфигурации сконцентрировано на коде и поддерживает как технологию конфигурации в коде, так и автоматическую регистрацию на основании соглашений, а XML остается одним из возможных вариантов.

Autofac поддерживает все три подхода и даже позволяет сочетать их в рамках одного и того же контейнера. Поэтому в этом отношении он обеспечивает нас всем, что только могло бы нам понадобиться. В этом разделе описаны способы применения всех трех типов источников конфигурации.

Конфигурация в коде

В разделе 13.1.1 "Разрешение объектов" уже приводился краткий обзор строго типизированного API конфигурации контейнера Autofac. В этом разделе мы рассмотрим его более подробно.

В Autofac в любом типе конфигурации применяется API, раскрываемое классом `ContainerBuilder`, несмотря на то, что большинство используемых нами методов являются методами расширений. Один из общепринятого используемых методов – это метод `RegisterType`, о котором уже упоминалось ранее:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
```

Как и при работе с Castle Windsor, регистрация класса `SauceBéarnaise` в виде `IIngredient` скрывает конкретный класс так, что разрешать класс `SauceBéarnaise` с помощью этой регистрации больше невозможно. Тем не менее, с этим легко можно справиться с помощью перегрузки метода `As`, которая позволяет определить, что конкретный тип преобразуется к нескольким зарегистрированным типам:

```
builder.RegisterType<SauceBéarnaise>()  
    .As<SauceBéarnaise, IIngredient>();
```

Вместо того чтобы регистрировать класс только в виде `IIngredient`, вы можете зарегистрировать его и как класс, и как интерфейс, который он реализует. Это позволяет контейнеру разрешать запросы и `SauceBéarnaise`, и `IIngredient`.

Помимо этого еще один вариант – это связать в цепочку выводы метода `Call`:

```
builder.RegisterType<SauceBéarnaise>()  
    .As<SauceBéarnaise>().As<IIngredient>();
```

Получаем результат, аналогичный только что рассмотренному примеру.

Существует три generic-перегрузки метода `As`, которые позволяют задать один, два или три типа соответственно. Если необходимо задать большее количество типов, то для этого существует не generic-перегрузка. С помощью нее можно задать любое количество типов.

Подсказка

Если необходимо с помощью метода `As` задать более трех типов, то следует рассматривать его как smell-эскиз регистрируемого класса. Если он будет реализовывать столь много интерфейсов, то, скорее всего, он нарушит принцип единичной ответственности.

В подлинных приложениях нам всегда приходится преобразовывать более одной абстракции, поэтому приходится конфигурировать несколько преобразований. Делается это при помощи нескольких выводов метода `RegisterType`:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();  
builder.RegisterType<Course>().As<ICourse>();
```

Здесь `IIngredient` преобразуется в `SauceBéarnaise`, а `ICourse` – в `Course`. Наложение типов не происходит, поэтому вполне очевидно, что будет происходить. Тем не менее, одну и ту же абстракцию можно зарегистрировать несколько раз:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();  
builder.RegisterType<Steak>().As<IIngredient>();
```

В этом примере `IIngredient` регистрируется дважды. При разрешении `IIngredient` получаем экземпляр `Steak`. Выигрывает последняя регистрация, но предыдущие

регистрации тоже не забыты. Autofac правильно обрабатывает составные конфигурации одной и той же абстракции, но к этому мы еще вернемся в разделе 13.3.

При конфигурировании Autofac можно использовать и дополнительные опции, но полноценно сконфигурировать приложение можно и с помощью продемонстрированных в этом разделе методов. Однако чтобы оградить себя от слишком явного сопровождения конфигурации контейнера, вместо этих методов можно использовать другой подход основанный на соглашениях – автоматическую регистрацию.

Автоматическая регистрация

Чаще всего большинство регистраций будут аналогичны друг другу. Такие регистрации очень утомительно сопровождать, а явная регистрация каждого компонента, возможно, не является самым продуктивным подходом.

Рассмотрим библиотеку, в которой содержится множество реализаций интерфейса `IIngredient`. Можно выполнять индивидуальную конфигурацию каждого класса, но это приведет к многочисленным схожим вызовам метода `RegisterType`. Что еще хуже, всякий раз при добавлении новой реализации `IIngredient` необходимо явно регистрировать ее вместе с `ContainerBuilder`, чтобы она была доступна для использования. Было бы более продуктивно установить, что все реализации `IIngredient`, найденные в этой сборке, необходимо регистрировать.

Это можно выполнить с помощью метода расширения `RegisterAssemblyTypes`. Метод `RegisterAssemblyTypes` позволяет указывать сборку и конфигурировать все выбранные классы этой сборки посредством одного единственного оператора. Для получения экземпляра `Assembly` можно использовать представительский класс. В этом примере таким классом является `Steak`:

```
builder.RegisterAssemblyTypes(typeof(Steak).Assembly)
    .As<IIngredient>();
```

Метод `RegisterAssemblyTypes` возвращает тот же интерфейс, что и метод `RegisterType`, поэтому можно использовать большинство таких же опций конфигурации, что и при использовании метода `RegisterType`. Это поистине значительная возможность, поскольку она означает, что для использования автоматической регистрации не нужно разбираться ни с каким новым API. В предыдущем примере для регистрации всех типов сборки в виде сервисов `IIngredient` использовался метод `As`.

Примечание

Несмотря на то, что типы возвращаемых значений совпадают, возвращаемый методом `RegisterAssemblyTypes` интерфейс – это сложный generic-интерфейс. А поскольку большинство используемых нами API реализуются в виде методов расширений, не все методы можно использовать в любой ситуации. Возможность их использования зависит от типа аргументов возвращаемого generic-интерфейса.

В предыдущем примере, безусловно, конфигурируются все реализации интерфейса `IIngredient`, но существует возможность использовать фильтры, которые позволяют выбирать только необходимое подмножество реализаций. Ниже приведен пример основанного на соглашениях отбора, при котором добавляются только те классы, чье имя начинается со слова "Sauce":

```
builder.RegisterAssemblyTypes(typeof(Steak).Assembly)
    .Where(t => t.Name.StartsWith("Sauce"))
    .As<IIngredient>();
```

При регистрации всех типов сборки для определения критерия отбора можно использовать предикат. Единственное отличие от предыдущего примера – это импликация метода `Where`, в котором отбираются только те типы, имена которых начинаются со слова "Sauce". Обратите внимание на то, что это тот же самый синтаксис, который использовался для фильтрации типов и в Castle Windsor, и в StructureMap.

Существует множество других методов, позволяющих задавать различные критерии отбора. Метод `Where` позволяет отфильтровывать только те типы, которые совпадают с предикатом, но есть еще метод `Except`, который работает совсем по-другому.

Помимо отбора из сборки соответствующих типов еще одной составляющей автоматической регистрации является определение соответствующего преобразования. В предыдущих примерах использовался метод `As` с конкретным интерфейсом с тем, чтобы зарегистрировать все выбранные типы относительно этого интерфейса.

Тем не менее, иногда бывает необходимо использовать различные соглашения. Предположим, что вместо интерфейсов мы используем абстрактные базовые классы и собираемся зарегистрировать все типы сборки, имена которых заканчиваются на "Policy". Для этих целей существует несколько перегрузок метода `As`, включая ту, которая в качестве входного параметра принимает `Func<Type, Type>`:

```
builder.RegisterAssemblyTypes(typeof(DiscountPolicy).Assembly)
    .Where(t => t.Name.EndsWith("Policy"))
    .As(t => t.BaseType);
```

Блок кода, заданный для метода `As`, будет использоваться для всякого типа, имя которого заканчивается на "Policy". Такой подход гарантирует, что все классы, имеющие суффикс "Policy" будут зарегистрированы относительно их базового класса таким образом, что при запросе базового класса контейнер будет разрешать его в тип, преобразованный с помощью этого соглашения.

В рамках Autofac достаточно легко выполнять регистрацию на основании соглашений, при этом используется API, которое практически отражает API, раскрываемое единичным методом `RegisterType`.

Подсказка

`RegisterAssemblyTypes` считается множественной формой метода `RegisterType`.

Метод `RegisterAssemblyTypes` принимает в качестве параметра массив `params`, состоящий из экземпляров `Assembly`, поэтому в одном соглашении можно использовать сколь угодно много сборок. Поиск сборок в папке и применение всех найденных сборок для реализации дополнительной функциональности, при которой возможно добавлять расширения без повторной компиляции основного приложения – это не надуманная идея. Это один из способов реализации позднего связывания. Еще один способ реализации позднего связывания – использование XML конфигурации.

XML конфигурация

XML конфигурацию хорошо использовать в тех ситуациях, когда должна существовать возможность изменять конфигурацию без повторной компиляции приложения.

Подсказка

Используйте XML конфигурацию только для тех типов, изменить которые вам нужно, не выполняя при этом повторной компиляции приложения. В остальных случаях применяйте автоматическую регистрацию или технологию конфигурирования в коде.

Самый естественный способ использования XML конфигурации – внедрить ее в стандартный конфигурационный файл .NET приложения. Это возможно, но при этом, чтобы изменить конфигурацию Autofac независимо от стандартного *.config* файла, придется использовать еще и автономный XML-файл. Независимо от выбранного способа используется практически одинаковое API.

Примечание

Поддержка контейнером Autofac XML конфигурации реализована в отдельной сборке, поэтому, чтобы воспользоваться этой возможностью необходимо добавить ссылку на сборку `Autofac.Configuration`.

После добавления ссылки на `Autofac.Configuration` можно отправить `Container.Builder` запрос на считывание регистраций компонентов из стандартного *.config* файла следующим образом:

```
builder.RegisterModule(new ConfigurationSettingsReader());
```

Детальное обсуждение модулей Autofac приведено в разделе 13.1.3, а пока все, что вам необходимо знать – это то, что `ConfigurationSettingsReader` – это класс, который отвечает за объединение XML конфигурации с остальными регистрациями, применяемыми к `Container.Builder`. При использовании конструктора по умолчанию происходит автоматическое считывание из раздела конфигурации стандартного конфигурационного файла приложения, но с помощью еще одной перегрузки можно указать и другой XML-файл, из которого будет считываться конфигурация.

Примечание

К сожалению, API, которое позволило бы нам считывать XML из других источников, к примеру, из потоков или узлов, не существует.

Чтобы разместить конфигурацию Autofac в конфигурационном файле, для начала необходимо добавить раздел конфигурации с помощью стандартного .NET API, предназначенного для определения пользовательских разделов конфигурации:

```
<configSections>
  <section name="autofac"
    type="Autofac.Configuration.SectionHandler,
      ➡Autofac.Configuration"/>
</configSections>
```


Этот код позволяет добавить в конфигурационный файл раздел конфигурации `autofac`. Ниже приведен простой пример преобразования интерфейса `IIngredient` в класс `Steak`:

```
<autofac defaultAssembly="Ploeh.Samples.MenuModel">
  <components>
    <component type="Ploeh.Samples.MenuModel.Steak"
      service="Ploeh.Samples.MenuModel.IIngredient" />
  </components>
</autofac>
```

В элемент `components` можно добавить сколько угодно элементов `component`. В каждом элементе с помощью атрибута `type` необходимо задать конкретный тип. Это единственный обязательный атрибут, а для преобразования класса `Steak` в `IIngredient` можно использовать также необязательный атрибут `service`. Тип указывается с помощью полностью квалифицированного имени типа, но, если тип задан в сборке по умолчанию, то имя сборки можно опустить. Атрибут `defaultAssembly` является необязательным, но при этом довольно значительной возможностью, которая в случае наличия в одной и той же сборке множества типов позволяет избавиться от большого количества определений типов.

XML конфигурацию хорошо использовать, когда необходимо изменить конфигурацию одного или нескольких компонентов без повторной компиляции приложения. Но поскольку XML конфигурация может быть довольно хрупкой, необходимо прибегать к ней только в указанных случаях и использовать для основной части конфигурации контейнера либо автоматическую регистрацию, либо конфигурацию в коде.

Подсказка

Не забыли, что выигрывает последняя конфигурация? Такое поведение можно использовать для переписывания жестко закодированной с помощью XML конфигурации. Для этого необходимо не забывать считывать XML конфигурацию после завершения конфигурирования любого другого компонента.

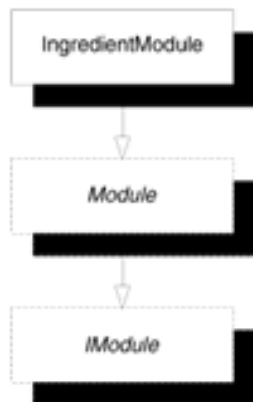
В этом разделе, главным образом, рассматривались различные API конфигурации контейнера `Autofac`. Хотя можно написать один большой блок неструктурированного кода конфигурации, лучше всего разделить конфигурацию на отдельные составляющие. `Autofac` поддерживает эту возможность с помощью "модулей".

Пакетирование конфигурации

Иногда существует необходимость упаковать логику конфигурации в повторно используемые группы. И даже в тех случаях, когда само по себе повторное использование не является высшим приоритетом, при конфигурировании больших и сложных приложений может появиться необходимость создать некоторого рода структуру.

При работе с `Autofac` конфигурацию можно упаковывать в модули. Модуль – это класс, реализующий интерфейс `IModule`, но в большинстве случаев проще всего выполнить наследование от абстрактного класса `Module`. На рисунке 13-4 продемонстрирована иерархия типов.

Рисунок 13-4: Повторно используемые конфигурации можно упаковывать в реализации интерфейса `IModule`. Самый простой способ реализации `IModule` – наследование от абстрактного класса `Module`, что и делает `IngredientModule`.



Все, что делается в дальнейшем, можно выполнить и внутри модуля. В следующем листинге представлен модуль, который регистрирует все реализации `IIngredient`.

Листинг 13-1: Реализация модуля в Autofac

```

public class IngredientModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        var a = typeof(Steak).Assembly;
        builder.RegisterAssemblyTypes(a).As<IIngredient>();
    }
}
  
```

`IngredientModule` наследуется от абстрактного класса `Module` и переопределяет метод `Load` этого класса. Метод `Load` – это метод, определенный классом `Module` для облегчения реализации интерфейса `IModule`. Посредством метода `Load` вы получаете экземпляр `ContainerBuilder`, который можно использовать для регистрации компонентов таким же самым способом, что и без применения `Module`. Знание способов использования API `ContainerBuilder` упрощает реализацию `Module`.

Чтобы применить `Module`, можно воспользоваться вызовом одной из перегрузок `RegisterModule`. Когда `Module` имеет конструктор по умолчанию, можно использовать сокращенную generic-версию:

```
builder.RegisterModule<IngredientModule>();
```

Кроме того, существует перегрузка, которая позволяет создать экземпляр, используемый в ситуациях, в которых `Module` требуется создавать вручную:

```
builder.RegisterModule(new IngredientModule());
```

Конфигурировать модули можно и в XML:

```

<modules>
  <module
    type="Ploeh.Samples.Menu.Autofac.IngredientModule,
    Ploeh.Samples.Menu.Autofac" />
</modules>
  
```

Эти три примера функционально эквивалентны.

Подсказка

Модули Autofac позволяют пакетировать и структурировать код конфигурации контейнера. Используйте модули вместо однострочной конфигурации, и ваша Composition Root станет более читабельной.

С помощью модулей конфигурацию контейнера Autofac можно выполнять любым из следующих способов: осуществлять конфигурацию в коде, использовать автоматическую регистрацию или XML. Кроме того, можно сочетать между собой указанные подходы. После завершения конфигурации контейнера можно отправлять запросы на разрешение сервисов.

Раздел 13.1 "Знакомство с Autofac" познакомил нас с DI-контейнером Autofac и продемонстрировал фундаментальные принципы как сконфигурировать ContainerBuilder и впоследствии использовать созданный контейнер для разрешения сервисов. Разрешение сервисов с легкостью выполняется посредством единичного вызова метода Resolve, поэтому вся сложность заключается в конфигурировании контейнера. Конфигурировать контейнер можно несколькими способами, включая императивный код и XML. До настоящего момента мы рассматривали только самое основное API, а более продвинутые вопросы в этом разделе еще не охватывались. Одна из самых важных проблем – управление жизненным циклом компонентов.

13.2. Управление жизненным циклом

В главе 8 обсуждался процесс управления жизненным циклом, в том числе наиболее универсальные стили существования, к примеру, Singleton и Transient. Autofac поддерживает несколько стилей существования и позволяет конфигурировать жизненные циклы всех сервисов. Продемонстрированные в таблице 13-2 стили существования являются частью API.

Примечание

В Autofac стили существования называются *областями применения экземпляров*.

Таблица 13-2: Стили существования Autofac

Название	Комментарии
Per Dependency	Стандартный Transient. Эта область применения экземпляра используется по умолчанию. Отслеживание экземпляров выполняет контейнер.
Single Instance	Стандартный Singleton.
Per Lifetime Scope	Связывает жизненные циклы компонентов с областью применения контейнера (см. раздел 13.2.1).
Contextual	Более расширенная версия Per Lifetime Scope

Реализации в Autofac стилей существования Transient и Singleton эквивалентны стилям существования, описанным в главе 8, поэтому в этой главе я не буду тратить время на их рассмотрение.

Подсказка

Используемый по умолчанию стиль Transient – самый безопасный, но не всегда является самым эффективным. Для потоко-безопасных сервисов Singleton – самый эффективный стиль, но вы не должны забывать о явной регистрации этих сервисов.

В этом разделе вы познакомитесь со способами определения стилей существования для компонентов – как с помощью кода, так и при помощи XML. Кроме того, мы рассмотрим сущность *областей применения* и то, как их можно использовать для реализации Web Request Context и других аналогичных стилей существования. После прочтения этого раздела вы уже сможете использовать стили существования Autofac в своем собственном приложении.

Начнем с рассмотрения способов конфигурирования областей применения для компонентов.

Конфигурирование областей применения экземпляров

В этом разделе мы рассмотрим процесс управления областями применения экземпляров с помощью Autofac. Конфигурирование области применения выполняется в рамках регистрации компонентов, а определить ее можно либо с помощью кода, либо при помощи XML. Поочередно рассмотрим каждую из этих возможностей.

Конфигурирование области применения с помощью кода

Область применения определяется как часть регистраций, которые мы выполняем для экземпляра `ContainerBuilder`. Это столь же просто, как и приведенный ниже код:

```
builder.RegisterType<SauceBéarnaise>().SingleInstance();
```

В этом примере конкретный класс `SauceBéarnaise` конфигурируется в виде `Singleton` таким образом, что каждый раз при запросе `SauceBéarnaise` возвращается один и тот же экземпляр. Если нам необходимо преобразовать абстракцию в конкретный класс, который имеет конкретный жизненный цикл, то мы можем воспользоваться обычным методом `As` и разместить вызов метода `SingleInstance` в любом удобном для нас месте. Приведенные ниже регистрации эквивалентны:

```
builder
    .RegisterType<SauceBéarnaise>()
    .As<IIngredient>()
    .SingleInstance();
builder
    .RegisterType<SauceBéarnaise>()
    .SingleInstance()
    .As<IIngredient>();
```

Единственное различие между двумя примерами заключается в том, что мы меняем местами вызовы методов `As` и `SingleInstance`. Лично я предпочитаю использовать код продемонстрированный сверху, поскольку вызовы методов `RegisterType` и `As` формируют преобразование между конкретным классом и абстракцией. Размещение их рядом друг с другом делает регистрацию более читабельной, и после этого мы можем определять область применения как изменение этого преобразования.

Несмотря на то, что стиль `Transient` используется по умолчанию, мы все равно можем определить его явным образом. Приведенные ниже примеры эквивалентны:

```
builder
    .RegisterType<SauceBéarnaise>();
builder
    .RegisterType<SauceBéarnaise>()
    .InstancePerDependency();
```

Конфигурирование областей применения для регистраций, основанных на соглашениях, выполняется тем же самым методом, который используется и для обычных регистраций:

```
builder.RegisterAssemblyTypes(typeof(Steak).Assembly)
    .As<IIngredient>()
    .SingleInstance();
```

Чтобы задать область применения для всех регистраций соглашения, можно воспользоваться методом `SingleInstance` и другими подходящими методами. В предыдущем примере мы определяли все `IIngredient` как `Singleton`'ы.

Аналогично регистрации компонентов с помощью кода и XML, мы также можем сконфигурировать область применения этими же способами.

Конфигурирование области применения с помощью XML

При возникновении необходимости определения компонентов с помощью XML вам также захочется иметь возможность конфигурировать в этом же самом месте области применения этих компонентов. Осуществить это можно в виде составляющей части XML-схемы, которую вы уже видели в разделе 13.1.2 "Конфигурирование ContainerBuilder".

Для объявления стиля существования можно использовать необязательный атрибут `instance-scope`:

```
<component type="Ploeh.Samples.MenuModel.Steak"
           service="Ploeh.Samples.MenuModel.IIngredient"
           instance-scope="single-instance" />
```

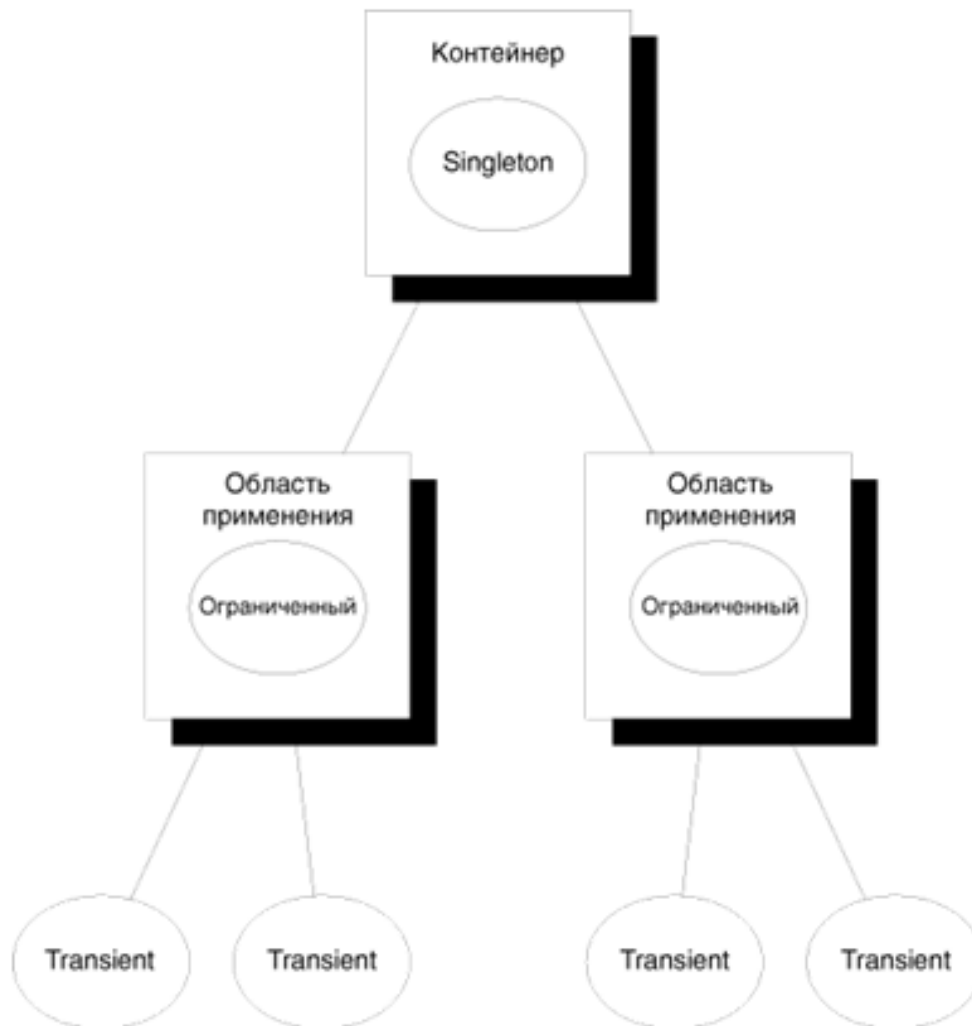
Отличие этого примера от примера, приведенного в разделе 13.1.2 "Конфигурирование ContainerBuilder", заключается в добавленном атрибуте `instance-scope`, который конфигурирует экземпляр в виде `Singleton`. Ранее, когда вы опускали этот атрибут, автоматически использовался стиль `Transient`.

Как с помощью кода, так и с помощью XML конфигурировать области применения экземпляров для компонентов довольно легко. Во всех случаях выполняется это в слегка декларативной форме. Несмотря на то, что конфигурация выполняется довольно просто, вы не должны забывать о том, что некоторые стили существования содержат объекты-долгожители, которые пользуются ресурсами на протяжении всего своего жизненного цикла.

Высвобождение компонентов

Как уже говорилось в разделе 8.2.2 "Управление устранимыми зависимостями", важно высвободить объекты после завершения работы с ними. Autofac не имеет явного метода `Release`, и вместо него использует так называемые *области применения жизненного цикла*. Область применения жизненного цикла можно рассматривать как неиспользуемую копию контейнера. Как демонстрирует рисунок 13-5, область применения определяет границу, в рамках которой могут повторно использоваться компоненты.

Рисунок 13-5: Области применения контейнера Autofac выступают в роли контейнеров, которые могут совместно использовать компоненты в течение ограниченного времени или для ограниченного круга целей. Компонент с ограниченной областью применения – это Singleton-компонент, расположенный в пределах этой области применения. Независимо от того, сколько раз мы запрашиваем этот компонент, мы получаем один и тот же экземпляр. Другая область применения содержит свой собственный экземпляр, и родительский контейнер управляет поистине совместно используемыми Singleton-компонентами. Transient-компоненты никогда не используются совместно, но их срок действия заканчивается, когда устраняется область применения.



Область применения жизненного цикла определяет производный контейнер, который можно использовать для конкретного срока действия или для конкретной цели. Наиболее очевидным примером является веб-запрос. Мы порождаем область применения от контейнера таким образом, что область применения наследует все Singleton'ы, отслеживаемые родительским контейнером, а область применения, помимо этого, играет роль контейнера "локальных Singleton'ов". Когда из области применения запрашивается компонент, относящийся к этой области применения, мы всегда получаем один и тот же экземпляр. Отличие от настоящих Singleton'ов – при запросе второй области применения мы получаем другой экземпляр.

Однако Transient-компоненты функционируют так как им и положено, независимо от того, разрешаем ли мы их из корневого контейнера или из области применения.

Подсказка

Области применения можно использовать для реализации контекстных стилей существования, например, стиля существования `Web Request Context`: создайте новую область применения в начале каждого контекста и используйте его для разрешения компонентов. А после завершения запроса уберите область применения. Тем не менее, что касается области применения веб-запроса, `Autofac` имеет встроенную возможность интеграции как с `Web Forms`, так и с `ASP.NET MVC`, поэтому нам не нужно выполнять эту интеграцию самостоятельно.

Одна из важных характеристик областей применения – они позволяют нам должным образом высвобождать компоненты по истечении срока действия этих областей применения. Новая область применения создается с помощью метода `BeginLifetimeScope` и высвобождает все соответствующие компоненты посредством вызова метода `Dispose`:

```
using (var scope = container.BeginLifetimeScope())
{
    var meal = scope.Resolve<IMeal>();
}
```

Строка 3: Уничтожение meal

Область применения создается из `container` посредством вызова метода `BeginLifetimeScope`. Возвращаемое значение реализует интерфейс `IDisposable`, поэтому можно поместить это значение в оператор `using`. Поскольку возвращаемое значение и контейнер реализуют один и тот же интерфейс, `scope` можно использовать для разрешения компонентов таким же образом, как и в случае работы с самим контейнером.

После окончания работы с областью применения ее можно устранить. При использовании оператора `using` область применения автоматически устраняется при выходе из оператора. Но это также можно сделать и явным образом путем вызова метода `Dispose`. При устранении `scope` мы также высвобождаем все компоненты, созданные с помощью области применения. В этом случае это означает, что вы высвобождаете диаграмму объектов `meal`.

Примечание

Не забывайте, что высвобождение устранимого компонента и устранение компонента – это не одно и то же. Это сигнал контейнеру о том, что компонент может завершить свой срок действия. Если это `Transient`-компонент или он ограничен областью применения, то он будет устранен. А если это `Singleton` компонент, то он будет продолжать действовать.

Ранее в этом разделе уже рассматривались способы конфигурирования компонентов в виде `Singleton`-ов или `Transient`-компонентов. Конфигурирование компонента с целью получения границ этого экземпляра, ограниченных областью применения, выполняется аналогичным образом:

```
builder.RegisterType<SauceBéarnaise>()
    .As<IIIngredient>()
    .InstancePerLifetimeScope();
```


Подробнее методам `Singleton` и `InstancePerDependency` метод `InstancePerLifetimeScope` можно использовать для указания того, что жизненный цикл компонента должен соответствовать области применения, которая создала экземпляр этого компонента.

Подсказка

Autofac отслеживает большинство `Transient`-компонентов, даже устранимых, поэтому важно не забывать разрешать все компоненты области применения и уstrаниать эту область после окончания ее использования.

Пока жизненный цикл компонента не закончен, `Singleton`'ы, в соответствии с их сущностью не высвобождаются. Однако если нам больше не нужен контейнер, мы можем высвободить даже эти компоненты. Выполняется это посредством устраниения самого контейнера:

```
container.Dispose();
```

На практике это не столь важно, поскольку жизненный цикл контейнера находится в близких взаимоотношениях с жизненным циклом поддерживаемого им приложения. Обычно мы сохраняем контейнер активным до тех пор, пока выполняется приложение, поэтому мы уstrаем его только, когда приложение закрывается, при этом операционная система восстанавливает память.

Области применения позволяют нам обращаться ко множеству сценариев, в которых мы по обыкновению используем `Web Request Context` или другой контекстный стиль существования. Это идиоматический способ реализации пользовательского жизненного цикла с помощью Autofac.

На этом наш обзор процесса управления жизненным циклом в рамках Autofac завершается. Компоненты можно конфигурировать с помощью сочетания областей применения экземпляров, и это справедливо даже при регистрации составных реализаций одной и той же абстракции. Мы еще не касались способов работы с составными компонентами, поэтому давайте направим все наше внимание в этом направлении.

13.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями
- Используются Decorator'ы

Мы рассмотрим каждую из этих ситуаций и увидим, как Autofac поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь регистрировать и разрешать компоненты даже тогда, когда в дело вступают составные реализации одной и той же абстракции.

Рассмотрим сначала способы предоставления более тщательного контроля, нежели тот, который предоставляет механизм автоматической интеграции.

Выбор среди составных кандидатов

Автоматическая интеграция – удобный и мощный инструмент, но предоставляет нам меньшие возможности контроля. Пока все абстракции преобразуются в конкретные типы отдельно друг от друга, никаких трудностей не возникает, но как только мы вводим большее количество реализаций для одного и того же интерфейса, возникает неопределенность.

Давайте для начала повторим, как Autofac работает с составными регистрациями одной и той же абстракции.

Конфигурирование составных реализаций одного и того же сервиса

Как вы уже видели в разделе 13.1.2 "Конфигурирование ContainerBuilder", вы можете конфигурировать составные реализации одного и того же интерфейса:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();  
builder.RegisterType<Steak>().As<IIngredient>();
```

В этом примере классы `Steak` и `SauceBéarnaise` регистрируются в виде сервиса `IIngredient`. Выигрывает последняя регистрация, поэтому, если мы будем разрешать `IIngredient` с помощью `container.Resolve<IIngredient>()`, то получим экземпляр `Steak`.

Подсказка

Выигрывает последняя регистрация данного сервиса. Она определяет экземпляры этого типа, используемые по умолчанию.

Можно попросить контейнер разрешить все компоненты `IIngredient`. Autofac обладает специально предназначенным для этих целей методом, но вместо того, чтобы его использовать Autofac полагается на *типы взаимосвязей*. *Тип взаимосвязи* — это тип, обозначающий взаимосвязь, которую может интерпретировать контейнер. К примеру, для обозначения того, что нам нужны все сервисы, мы можем использовать `IEnumerable<T>`:

```
var ingredients = container.Resolve<IEnumerable<IIngredient>>();
```

Обратите внимание на то, что мы используем обычный метод `Resolve`, но запрашиваем `IEnumerable<IIngredient>`. Autofac интерпретирует эту конструкцию как соглашение и отдает нам все компоненты `IIngredient`, которыми он обладает.

Подсказка

В противоположность `IEnumerable<T>` мы можем также запросить массив. Результаты в обоих случаях будут одинаковы: мы получим все компоненты запрашиваемого типа.

В случае наличия регистраций, которые невозможно разрешить при запрашивании всех сервисов определенного типа, Autofac выдает исключение, поясняющее, что существуют зависимости, неудовлетворяющие условиям. Такое поведение соответствует разрешению единичного компонента, но отличается от поведения Castle Windsor или MEF.

При регистрации компонентов каждой регистрации можно присвоить свое собственное имя, которое в дальнейшем можно использовать для осуществления выбора между различными компонентами:

```
builder.RegisterType<Steak>()  
    .Named<IIngredient>("meat");  
builder.RegisterType<SauceBéarnaise>()  
    .Named<IIngredient>("sauce");
```

Как обычно мы начинаем с метода `RegisterType`, но вместо того, чтобы после него использовать метод `As`, мы используем метод `Named`, чтобы задать тип сервиса, а также его название. Это позволяет нам разрешать именованные сервисы путем передачи этого же имени в метод `ResolveNamed`:

```
var meat = container.ResolveNamed<IIngredient>("meat");  
var sauce = container.ResolveNamed<IIngredient>("sauce");
```

Примечание

Именованный компонент не считается компонентом по умолчанию. Если мы регистрируем только именованные компоненты, то не сможем разрешать экземпляр сервиса, используемый по умолчанию. Тем не менее, ничто не мешает нам зарегистрировать компонент по умолчанию (неименованный) с помощью метода `As`, а сделать это можно в одном и том же операторе с помощью цепочки методов.

Присваивание имен компонентам с помощью строк – довольно универсальная возможность DI-контейнеров, но Autofac также позволяет идентифицировать компоненты с помощью произвольных ключей:

```
var meatKey = new object();  
builder.RegisterType<Steak>().Keyed<IIngredient>(meatKey);
```

В роли ключа может выступать любой объект, который впоследствии можно использовать для разрешения компонента:

```
var meat = container.ResolveKeyed<IIngredient>(meatKey);
```

Подразумеваемая необходимость, всегда разрешать сервисы в одном Composition Root, мы скорее всего, не должны ждать появления такой неопределенности на этом уровне.

Подсказка

Если вы обнаружите, что вызываете метод `Resolve` с конкретным именем или ключом, подумайте над тем, сможете ли вы сменить свой подход на менее неопределенный.

Тем не менее, именованные экземпляры или экземпляры, идентифицируемые с помощью ключа, можно использовать для осуществления выбора среди нескольких экземпляров при конфигурировании зависимостей данного сервиса.

Регистрация именованных зависимостей

Иногда бывает необходимо переопределить обычное поведение для того, чтобы обеспечить более разветвленный контроль над тем, куда какая зависимость отправляется. Кроме того, возможны ситуации, при которых приходится сталкиваться с неопределенным API. В качестве примера рассмотрим следующий конструктор:

```
public ThreeCourseMeal(ICourse entrée,  
    ICourse mainCourse, ICourse dessert)
```

В этом примере присутствуют три одинаковым образом типизированных зависимости, каждая из которых является отдельной сущностью, не похожей на другие. В большинстве случаев необходимо преобразовывать каждую из этих зависимостей в отдельный тип. В следующем листинге продемонстрированы способы регистрации преобразований `ICourse`.

Листинг 13-2: Регистрация именованных `course`

```
builder.RegisterType<Rillettes>()  
    .Named<ICourse>("entrée");  
builder.RegisterType<CordonBleu>()  
    .Named<ICourse>("mainCourse");  
builder.RegisterType<MousseAuChocolat>()  
    .Named<ICourse>("dessert");
```

В этом примере вы регистрируете три именованных компонента, преобразуя `Rillettes` в экземпляр под названием `"entrée"`, `CordonBleu` – в экземпляр с именем `"mainCourse"`, а `MousseAuChocolat` – в экземпляр под названием `"dessert"`.

При такой конфигурации вы теперь можете регистрировать класс `ThreeCourseMeal` с помощью именованных регистраций. Оказывается, это довольно сложно. В следующем листинге я сначала продемонстрирую вам, на что это похоже, а затем мы подробно разберем этот пример, чтобы понять, что происходит.

Листинг 13-3: Переопределение автоматической интеграции

```
1. builder.RegisterType<ThreeCourseMeal>()
2.   .As<IMeal>()
3.   .WithParameter(
4.     (p, c) => p.Name == "entrée",
5.     (p, c) =>
6.       c.ResolveNamed<ICourse>("entrée"))
7.   .WithParameter(
8.     (p, c) => p.Name == "mainCourse",
9.     (p, c) =>
10.      c.ResolveNamed<ICourse>("mainCourse"))
11.  .WithParameter(
12.    (p, c) => p.Name == "dessert",
13.    (p, c) =>
14.      c.ResolveNamed<ICourse>("dessert"));
```

Строка 3: Определение параметра

Строка 4: Фильтры

Строка 5-6: Определение значений

Метод `WithParameter` позволяет предоставлять значения параметров для конструктора `ThreeCourseMeal`. Одна из этих перегрузок принимает в качестве входных параметров два аргумента. Первый аргумент – предикат, который определяет, является ли этот параметр результатом этого конкретного вызова метода. Для первого параметра вы задаете условие, что он касается только параметра под названием `entree`. Если это выражение имеет значение `true`, то выполняется второй блок кода, определяющий значение для параметра `entree`. Параметр `c` – это экземпляр `IComponentContext`, который можно использовать для разрешения именованного компонента `entree`.

Подсказка

Аргументы метода `WithParameter` – это разновидность паттерна `Tester-Doer`.

Рассмотрим подробнее то, что происходит. Метод `WithParameter` действительно является оберткой класса `ResolvedParameter`, который обладает следующим конструктором:

```
public ResolvedParameter(
    Func<ParameterInfo, IComponentContext, bool> predicate,
    Func<ParameterInfo, IComponentContext, object> valueAccessor);
```

Параметр `predicate` – это тест, который определяет, будет ли вызываться делегат `valueAccessor`: если параметр `predicate` возвращает `true`, то для определения значения параметра вызывается `valueAccessor`. Оба делегата принимают в качестве входных данных одну и ту же информацию: информацию о параметре в виде объекта `ParameterInfo` и `IComponentContext`, который можно использовать для разрешения других компонентов. Когда `Autofac` использует экземпляры `ResolvedParameter`, то при вызове делегатов он возвращает оба эти значения. Иногда нет другого варианта, кроме как

старательно использовать метод `WithParameter` для каждого параметра конструктора. Но в остальных случаях можно воспользоваться преимуществами соглашений.

Разрешение именованных компонентов с помощью соглашения

Если вы внимательно проанализировали листинг 13-3, то возможно обратили внимание на повторяющийся паттерн. Каждый вызов метода `WithParameter` относится только к одному параметру конструктора, но каждый `valueAccessor` выполняет то же самое: он использует `IComponentContext` для разрешения компонента `ICourse`, имеющего то же название, что и параметр.

Нет никакого требования, согласно которому мы обязаны были бы называть компонент в честь параметра конструктора. Но всякий раз, когда мы так поступаем, мы можем воспользоваться преимуществами этого соглашения и переписать листинг 13-3 более простым образом. В следующем листинге продемонстрировано, как это сделать.

Листинг 13-4: Переопределение автоматической интеграции с помощью соглашения

```
builder.RegisterType<ThreeCourseMeal>()
    .As<IMeal>()
    .WithParameter(
        (p, c) => true,
        (p, c) => c.ResolveNamed(p.Name, p.ParameterType));
```

Возможно, это покажется удивительным, но обратиться ко всем трем параметрам конструктора класса `ThreeCourseMeal` можно с помощью одного и того же вызова метода `WithParameter`. Осуществить это можно, сформулировав соглашение о том, что этот экземпляр будет управлять всяким параметром, какой `Autofac` смог бы ему передать. Поскольку вы используете этот метод только для конфигурирования класса `ThreeCourseMeal`, соглашение применяется только в рамках этой ограниченной области.

Поскольку предикат всегда возвращает значение `true`, второй блок кода будет вызываться для всех трех параметров конструктора. Во всех трех случаях этот блок кода будет просить `IComponentContext` разрешить компонент, имеющий те же самые имя и тип, что и параметр. Функционально это аналогично тому, что вы делали в листинге 13-3.

Предупреждение

Идентифицировать параметры по их именам удобно, но это не безопасно для рефакторинга. Если вы переименуете параметр, то можете разрушить конфигурацию (это зависит от используемого вами инструмента рефакторинга).

Переопределение автоматической интеграции посредством явного преобразования параметров в именованные компоненты — это повсеместно используемое решение. Сделать это можно даже в тех случаях, когда именованные компоненты конфигурируются в одном модуле, а потребитель находится совсем в другом модуле. Это возможно, поскольку единственной идентификацией, связывающей именованный компонент с параметром, является имя.

Такой подход можно использовать всегда, но, если нам придется управлять большим количеством имен, то это будет довольно хрупким решением. Когда первоначальной причиной, побуждающей нас использовать именованные компоненты, является борьба с неопределенностью, наилучшим решением этой проблемы станет создание своего

собственного API, которое позволит избавиться от этой неопределенности. Все это приводит к улучшению всего дизайна приложения.

В следующем разделе мы рассмотрим возможность использования менее неопределенного и более гибкого подхода, при котором обед может состоять из нескольких блюд. Перед этим нам необходимо изучить то, как Autofac работает со списками и последовательностями.

Интеграция последовательностей

В разделе 10.3.2 "Разработка пользовательского стиля существования" мы обсуждали, как выполнить рефакторинг явного класса `ThreeCourseMeal` к более универсальному классу `Meal`, который обладает приведенным ниже конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

В этом разделе мы рассмотрим то, как можно сконфигурировать Autofac, чтобы он интегрировал экземпляры `Meal` с соответствующими зависимостями `ICourse`. После рассмотрения этого вопроса вы должны будете приобрести хорошее понимание тех возможностей, которые доступны в случае необходимости конфигурирования экземпляров, имеющих последовательности зависимостей.

Автоматическая интеграция последовательностей

Autofac довольно хорошо разбирается в последовательностях, поэтому, если нам необходимо использовать все зарегистрированные компоненты этого сервиса, то нам нужен как раз механизм автоматической интеграции. Например, имея сконфигурированные в листинге 13-2 экземпляры `ICourse`, можно сконфигурировать сервис `IMeal` следующим образом:

```
builder.RegisterType<Meal>().As<IMeal>();
```

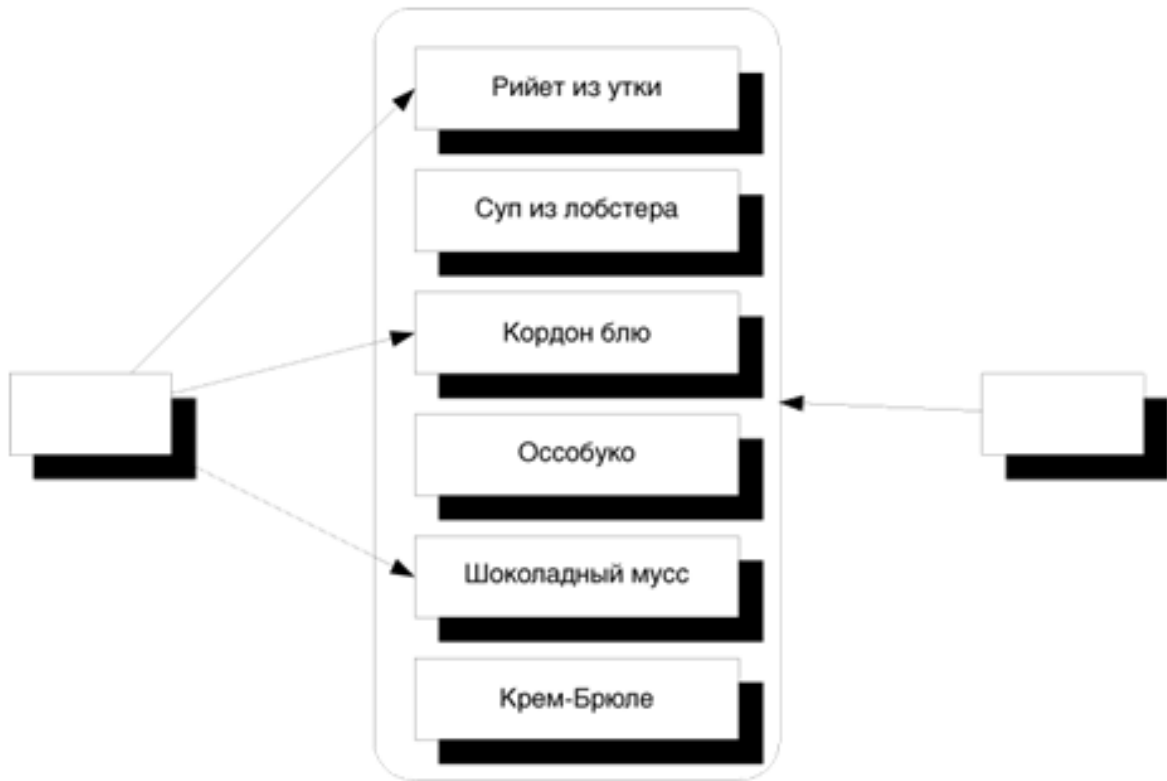
Обратите внимание на то, что это совершенно стандартное преобразование конкретного типа в абстракцию. Autofac будет автоматически понимать конструктор `Meal` и определять, что правильным направлением действия является разрешение всех компонентов `ICourse`. При разрешении `IMeal` вы получите экземпляр `Meal`, компонентами которого являются `ICourse` из листинга 13-2: `Rillettes`, `CordonBleu` и `MousseAuChocolat`.

Autofac автоматически управляет последовательностями и, пока мы не укажем обратное, выполняет то, чего мы от него и ждем: разрешает последовательность зависимостей всех зарегистрированных компонентов этого типа. Только если нам нужно отобрать лишь некоторые компоненты из большого набора, нам приходится выполнять больше действий. Рассмотрим то, как это можно сделать.

Отобор нескольких компонентов из большого набора

Используемая Autofac по умолчанию стратегия внедрения всех компонентов зачастую является правильной линией поведения. Но, как показывает рисунок 13-6, возможны случаи, когда нам нужно отобрать только несколько зарегистрированных компонентов из большого набора всех зарегистрированных компонентов.

Рисунок 13-6: В ситуации, продемонстрированной слева, мы хотим явным образом отобразить определенные зависимости из большого списка всех зарегистрированных компонентов. Это отличается от ситуации, приведенной справа, когда мы отбираем все без разбора.



Когда мы ранее позволяли Autofac автоматически интегрировать все сконфигурированные экземпляры, это соответствовало ситуации, изображенной в правой части рисунка 13-6. Если нам нужно зарегистрировать компонент так, как это показано в левой части рисунка, то мы должны явно определить, какие компоненты необходимо использовать.

Для получения такого результата можно еще раз применить метод `WithParameter` так, как вы уже использовали его в листингах 13-3 и 13-4. Все это время вы работали с конструктором `Meal`, который в качестве входной информации принимал только один-единственный параметр. В следующем листинге продемонстрировано, как можно реализовать составляющую метода `WithParameter`, которая предоставляет значение параметра, так, чтобы явным образом отбирать именованные компоненты из `IComponentContext`.

Листинг 13-5: Внедрение именованных компонентов в последовательность

```
builder.RegisterType<Meal>()
    .As<IMeal>()
    .WithParameter(
        (p, c) => true,
        (p, c) => new[]
        {
            c.ResolveNamed<ICourse>("entrée"),
            c.ResolveNamed<ICourse>("mainCourse"),
            c.ResolveNamed<ICourse>("dessert")
        }
    );
```


Как вы уже видели в разделе 13.3.1 "Выбор среди составных кандидатов", метод `WithParameter` в качестве входных параметров принимает два делегата. Первый – это предикат, который используется для того, чтобы определить, должен ли вызываться второй делегат. В этом примере мне захотелось полениться, и я вернул значение `true`. Вы знаете, что у конструктора класса `Meal` есть только один параметр, поэтому метод `WithParameter` будет работать. Тем не менее, если вы впоследствии измените класс `Meal` таким образом, что у его конструктора будет два параметра, метод `WithParameter` уже не будет работать корректно. Поэтому безопаснее всего будет установить явную проверку имени параметра.

Второй делегат предоставляет значение для параметра. Для разрешения трех именованных компонентов в массив вы используете `IComponentContext`. В результате получаем массив `ICourse`, который сравним с `IEnumerable<ICourse>`.

Autofac понимает последовательности. До тех пор пока не появляется необходимость отбирать только некоторые компоненты из всех сервисов указанного типа, Autofac автоматически все делает правильно. Автоматическая интеграция применяется не только для единичных экземпляров, но также и для последовательностей. А контейнер преобразует последовательность во все сконфигурированные экземпляры соответствующего типа.

Потребители, которые полагаются на последовательности зависимостей, могут быть самыми интуитивно понятными пользователю составными экземплярами одной и той же абстракции. Но перед тем как мы полностью отойдем от данной темы, нам необходимо рассмотреть последний (и, возможно, слегка неожиданный) случай, когда в дело вступают составные экземпляры.

Интеграция Decorator'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования `Decorator` полезен при реализации сквозных сущностей. По определению `Decorator`'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам `Decorator` и вложенный тип. Если бы мы помещали `Decorator`'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных регистраций одного и того же сервиса. В отличие от предыдущих разделов эти регистрации не являются концептуально равносильными, а зависят друг от друга. В этом разделе я рассмотрю два разных способа конфигурирования Autofac для работы с данным паттерном.

Создание обертки с помощью метода `WithParameter`

Метод `WithParameter` предлагает универсальный способ определения того, как создаются и внедряются компоненты. В разделах 13.3.1 "Выбор среди составных кандидатов" и 13.3.2 "Интеграция последовательностей" вы уже видели, как можно использовать метод `WithParameter` для отбора конкретных компонентов для параметров конструктора. Кроме того, метод `WithParameter` – это отличный способ предоставления параметров для `Decorator`'ов.

Рассмотрим способы использования метода `WithParameter` для конфигурирования класса `Breading`, который является оберткой `IIngredient`. Для получения экземпляра, оберткой которого должен стать `WithParameter`, он использует паттерн `Constructor Injection`:

```
public Breading(IIngredient ingredient)
```

Чтобы получить `Cotoletta`, вам следует обернуть `VealCutlet` (еще один `IIngredient`) в класс `Breading`. То есть вы собираетесь внедрить `VealCutlet` в `Breading`. В следующем листинге продемонстрировано, как для этих целей можно использовать метод `WithParameter`.

Листинг 13-6: Создание обертки с помощью метода `WithParameter`

```
builder.RegisterType<VealCutlet>().Named<IIngredient>("cutlet");
builder.RegisterType<Breading>()
    .As<IIngredient>()
    .WithParameter(
        (p, c) => p.ParameterType == typeof(IIngredient),
        (p, c) => c.ResolveNamed<IIngredient>("cutlet"));
```

`Breading` — это `Decorator`, но вам же нужно что-то обертывать, поэтому вы регистрируете `VealCutlet` в виде именованного компонента. В этом примере вы регистрируете `VealCutlet` перед `Breading`, но можно сделать это и по-другому. Порядок регистраций не имеет значения.

При регистрации `Breading` для определения параметра `ingredient` конструктора класса `Breading` вы используете метод `WithParameter`. Вы реализуете предикат, проверяя, что тип параметра — `IIngredient`, и предоставляете значение для параметра путем разрешения именованного компонента `cutlet` из заданного `IComponentContext`.

В этом примере вы использовали именованную регистрацию `IIngredient` для регистрации компонента `VealCutlet`. Это делает компонент `Breading` компонентом `IIngredient` по умолчанию. Еще один вариант — регистрировать `VealCutlet` и как `IIngredient`, и как `VealCutlet`. В следующем примере продемонстрирован этот подход в сочетании со строго типизированным делегатом.

Создание обертки с помощью делегатов

Вместо того чтобы обращаться к параметрам конструктора по типу или имени, мы можем написать строго типизированный блок кода, в котором используется конструктор:

```
builder.RegisterType<VealCutlet>()
    .As<IIngredient, VealCutlet>();
builder.Register(c => new Breading(c.Resolve<VealCutlet>()))
    .As<IIngredient>();
```

В качестве альтернативы регистрации `VealCutlet` в виде именованного компонента можно также регистрировать его и как `IIngredient`, и как `VealCutlet`. При таком подходе важно делать это до регистрации `Breading`, поскольку в противном случае `VealCutlet` станет компонентом `IIngredient` по умолчанию.

Вместо метода `RegisterType`, который вы в основном использовали до настоящего момента, можно также зарегистрировать сервис и с помощью метода под названием `Register`. Существует две перегрузки этого метода, и каждая из них принимает в качестве

входного параметра делегат, который создает рассматриваемый сервис. Чтобы зарегистрировать сервис `IIngredient`, вы реализуете блок кода, который создает новый экземпляр `Breading` путем прямого вызова конструктора. Чтобы передать значение в параметр конструктора `ingredient`, вы разрешаете тип `VealCutlet` из переданного `IComponentContext`. Это возможно, поскольку вы зарегистрировали `VealCutlet` как конкретный тип, а также как `IIngredient`.

Примечание

Вы также могли бы разрешить `VealCutlet` по имени, если бы зарегистрировали его в виде именованного компонента, как делали это в предыдущем примере.

Если вы попросите контейнер разрешить `IIngredient`, он передаст `IComponentContext` в качестве входного параметра в блок кода, который вы определили в методе `Register`. При выполнении блока кода из контекста разрешается экземпляр `VealCutlet` и передается в конструктор `Breading`, который возвращает экземпляр `Breading`.

Преимущество такого подхода заключается в том, что в блоке кода вы пишете код, в котором используется конструктор `Breading`. Это обычная строка кода, поэтому она проверяется компилятором. Это придает вам уверенности в том, что если метод `Register` компилируется, значит, обертка для `VealCutlet` будет создаваться корректно.

Несмотря на то, что строгая типизированность более безопасна, ее еще и сложно сопровождать. Если вы впоследствии решите добавить в конструктор `Breading` еще один параметр, то блок кода больше не будет компилироваться, и вам придется вручную решить эту проблему. Это не было бы столь необходимо, если бы вы использовали метод `WithParameter`, поскольку `Autofac` смог бы разобрать новый параметр с помощью механизма автоматической интеграции.

Как вы видели в этом разделе, существуют различные способы конфигурирования `Decorator`'ов. Строго типизированный подход более безопасен, но может потребовать больше затрат на сопровождение. Более слабо типизированное API является более гибким решением и позволяет `Autofac` справляться с изменениями вашего API, но ценой меньшей типовой безопасности.

Примечание

В этом разделе мы не обсуждали механизм перехвата во время выполнения. Несмотря на то, что `Autofac` имеет `Seam`'ы, которые позволяют использовать механизм перехвата, он также обладает встроенной поддержкой динамически создаваемых прокси. Эти `Seam`'ы можно применять, чтобы использовать для создания таких классов другие библиотеки (например, `Castle Dynamic Proxy`). Но поскольку они не являются частью `Autofac`, эта тема выходит за рамки этой главы.

`Autofac` позволяет нам работать с составными экземплярами разными способами. Мы можем регистрировать компоненты в виде альтернатив друг другу, в виде пиров, которые разрешаются в виде последовательностей, или в виде иерархических `Decorator`'ов. В большинстве случаев `Autofac` поймет, что надр делать, но мы всегда можем задать, как компонуются сервисы, если нам нужен более явный контроль.

Кроме того, это может происходить в ситуации, когда нам нужно иметь дело с API, отклоняющимися от Constructor Injection. До настоящего момента вы наблюдали за тем, как регистрировать компоненты, включая то, как определять области применения, и как работать с составными компонентами. Но до этого момента мы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют Constructor Injection. Поскольку это не всегда происходит именно так, в следующем разделе мы сделаем краткий обзор того, как работать с классами, экземпляры которых должны создаваться особым образом.

13.4. Регистрация сложных API

До настоящего момента мы рассматривали то, как можно конфигурировать компоненты, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры, например, Autofac, могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей.

Все становится менее понятным, когда API не столь хорошо функционируют. В данном разделе вы увидите, как работать с простейшими аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Начнем с рассмотрения классов, конструкторы которых принимают в качестве параметров простейшие типы, например, строки и целые числа.

Конфигурирование простейших зависимостей

Пока мы внедряем абстракции в потребителей, все в порядке. Но этот процесс усложняется, если конструктор зависит от простейшего типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве компонента контейнера не имеет особого смысла. Но в рамках Autofac это, по крайней мере, осуществимо.

Рассмотрим в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются `code smell`'ами и их нужно преобразовывать в полиморфные классы (имеющие разное состояние). Тем не менее, для данного примера они вполне нам подходят.

Если вы хотите, чтобы все потребители `Spiciness` использовали одно и то же значение, можно зарегистрировать `Spiciness` и `ChiliConCarne` независимо друг от друга:

```
builder.Register<Spiciness>(c => Spiciness.Medium);
builder.RegisterType<ChiliConCarne>().As<ICourse>();
```

Когда вы впоследствии будете разрешать `ChiliConCarne`, его `Spiciness` будет иметь значение `Medium`, как и все остальные компоненты, зависящие от `Spiciness`.

Если вы будете достаточным образом контролировать взаимосвязь `Spiciness` и `ChiliConCarne`, вы сможете использовать метод `WithParameter` таким же образом, как и в листингах 13-4, 13-5, 13-6:

```
builder.RegisterType<ChiliConCarne>()
    .As<ICourse>()
    .WithParameter("spiciness", Spiciness.Hot);
```

Поскольку вы собираетесь передать в параметр `spiciness` конкретное значение, вы можете воспользоваться другой перегрузкой метода `WithParameter`, которая в качестве входных данных принимает имя и значение параметра. Эта перегрузка делегирует полномочия другому `WithParameter` путем создания экземпляра `NamedParameter` из имени и значения параметра. `NamedParameter` также наследуется от `Parameter`, как и `ResolvedParameter`.

Оба описанных здесь варианта стимулируют автоматическую интеграцию на предоставление конкретного значения для компонента. Как уже обсуждалось в разделе 13.3 "Работа с составными компонентами", такой подход имеет как свои преимущества, так и недостатки. Если вам нужна строго типизированная конфигурация, которая вызывает конструктор или статическую фабрику, вы также сможете это сделать.

Регистрация объектов с помощью блоков кода

Еще один вариант создания компонента с примитивным значением — использовать метод `Register`, позволяющий передавать делегат, который создает компонент:

```
builder.Register<ICourse>(c =>
    new ChiliConCarne(Spiciness.Hot));
```

Вы уже видели метод `Register`, когда мы обсуждали `Decorator`'ы в разделе 13.3.3 "Интеграция `Decorator`'ов". Всякий раз при разрешении компонента `ICourse` будет вызываться конструктор `ChiliConCarne` с параметром `Spiciness.Hot`.

Примечание

Метод `Register` имеет безопасный тип, но не позволяет использовать автоматическую интеграцию.

Когда дело касается класса `ChiliConCarne`, вам предоставляется выбор между автоматической интеграцией и использованием блока кода. Но другие классы более ограничены: их экземпляры нельзя создать с помощью открытого конструктора. Чтобы создать экземпляры типа, вместо открытого конструктора вам приходится использовать некоторого рода фабрику. Для DI-контейнеров это всегда проблематично, поскольку по умолчанию им нужны открытые конструкторы.

Рассмотрим приведенный ниже пример конструктора открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже если класс `JunkFood` является открытым, конструктор расположен внутри него. Очевидно, экземпляры `JunkFood` должны создаваться с помощью статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С точки зрения `Autofac`, это проблемное API, поскольку в нем отсутствуют точно выраженные и заданные соглашения касательно статических фабрик. Тут требуется помощь — и мы можем предоставить ее посредством блока кода, который `Autofac` может исполнять для того, чтобы создать экземпляр:

```
builder.Register(c =>
    JunkFoodFactory.Create("chicken meal"));
```

В этот раз вы используете метод `Register` для создания компонента, вызывая статическую фабрику в рамках блока кода. Всякий раз при разрешении `IMeal` будет вызываться `JunkFoodFactory.Create` и возвращаться результат.

Является ли написание блока кода для создания экземпляра лучшим вариантом, нежели прямой вызов кода? При использовании блока кода внутри вызова метода `Register` мы приобретаем следующие преимущества:

- `IMeal` преобразуется в `JunkFood`.
- Область применения экземпляра остается доступной для конфигурирования. Несмотря на то, что для создания экземпляра вызывается блок кода, он может и не вызываться всякий раз при запросе экземпляра. По умолчанию он вызывается, но если мы изменим область применения экземпляра на `Singleton`, то блок кода будет вызываться только один раз, а результат будет кэшироваться и впоследствии повторно использоваться.

Последним рассматриваемым нами отклонением от `Constructor Injection` является `Property Injection`.

Интегрирование с помощью `Property Injection`

`Property Injection` — это менее определенная форма механизма внедрения зависимостей, поскольку компилятор не принуждает нас задавать значение свойства, доступного для записи. Это касается и `Autofac`, который будет оставлять доступные для записи свойства незаполненными до тех пор, пока мы явно не попросим его заполнить их.

Рассмотрим класс `CaesarSalad`:

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

По всеобщему заблуждению в состав салата "Цезарь" входит курица, но это не правда. По существу "Цезарь" является салатом, но, поскольку с курицей он вкуснее, то ее часто предлагают использовать в нем в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Если вы конфигурируете только класс `CaesarSalad`, явно не обращаясь к свойству `Extra`, то этому свойству не будет присвоено значение. Вы все равно можете разрешать экземпляр, но свойство `Extra` будет иметь значение по умолчанию, которое ему присвоил конструктор (если только это имеет место).

Существует несколько способов, с помощью которых можно сконфигурировать `CaesarSalad` таким образом, чтобы свойство `Extra` заполнялось соответствующим образом. Самым простым способом является использование метода `PropertiesAutowired`:

```
builder.RegisterType<CaesarSalad>()
    .As<ICourse>()
    .PropertiesAutowired();
builder.RegisterType<Chicken>().As<IIIngredient>();
```

Поскольку метод `PropertiesAutowired` является частью простого API регистрации, вы можете вызвать его для того, чтобы сообщить `Autofac`, что ему необходимо автоматически интегрировать доступные для записи свойства класса `CaesarSalad`. `Autofac` будет автоматически интегрировать только те свойства, о способе заполнения которых он осведомлен, поэтому вы также регистрируете `Chicken` в виде `IIIngredient`. Если бы вы это не сделали, то свойство `Extra` было бы проигнорировано.

Когда вы на основании этой регистрации разрешаете `ICourse`, вы получаете экземпляр `CaesarSalad`, свойству `Extra` которого присвоен экземпляр `Chicken`.

Если вам нужен более подробный контроль, нежели тот, который предоставляется посредством метода `PropertiesAutowired`, то вы можете воспользоваться методом `WithProperty`, который похож на метод `WithParameter`, используемый вами ранее:

```
builder.RegisterType<Chicken>().As<IIIngredient>();
builder.RegisterType<CaesarSalad>()
    .As<ICourse>()
    .WithProperty(new ResolvedParameter(
        (p, c) => p.Member.Name == "set_Extra",
        (p, c) => c.Resolve<IIIngredient>()));
```

Метод `WithProperty` отражает уже полюбоившийся вам метод `WithParameter`: он принимает в качестве входных данных только один аргумент `Parameter` и также обладает перегрузкой, которая в качестве параметров принимает имя и значение свойства.

Чтобы должным образом разрешить метод `Extra`, можно воспользоваться доверенным классом `ResolvedParameter`. Что касается свойств, переданный нами предикат имеет небольшую отличительную черту, поскольку `Autofac` вызывает блок кода с аргументом `ParameterInfo`, а не с `PropertyInfo`. Параметр `p` олицетворяет параметр `value`, который всегда потенциально доступен при реализации свойства, поэтому нам необходимо перейти к `Member`, который определяет этот параметр. `Member` — это экземпляр `MethodInfo`,

поэтому нам нужно ознакомиться с тем, как реализуются C# свойства на уровне интерфейса: в действительности свойства Extra – это метод под названием `set_Extra`.

Когда предикат передан, легко реализовать получение значения путем разрешения `IIngredient` из переданного `IComponentContext`.

Использование метода `WithProperty` дает нам более разветвленный уровень контроля над Property Injection, при этом мы сохраняем слабое связывание. Если нам нужен другой строго типизированный подход, то это тоже возможно.

Autofac позволяет нам использовать блоки кода, которые будут вызываться в течение жизненного цикла компонента при возникновении определенного события. Мы можем перехватывать эти события, чтобы заполнять свойства по мере того, как создается компонент.

Одно из таких событий – событие `OnActivating`, которое Autofac вызывает всякий раз при создании нового экземпляра компонента. Это событие можно использовать для заполнения свойства Extra, пока Autofac не вернет экземпляр `CaesarSalad`:

```
builder.RegisterType<Chicken>().As<IIngredient>();
builder.RegisterType<CaesarSalad>()
    .As<ICourse>()
    .OnActivating(e =>
        e.Instance.Extra = e.Context.Resolve<IIngredient>());
```

Метод `OnActivating` дает вам возможность выполнить какое-нибудь действие над компонентом, пока Autofac не вернет его тому объекту, который его запрашивает. В качестве единственного параметра он принимает

`Action<IActivatingEventArgs<CaesarSalad>>`, который можно использовать для реализации выбранной вами логики постобработки. Параметр `e` олицетворяет аргументы события, а также обладает свойством `Instance` типа `CaesarSalad` и свойством `Context`, которое можно использовать для разрешения других компонентов. Это сочетание вы используете для того, чтобы разрешить `IIngredient` и вернуть результат в свойство Extra. При разрешении `ICourse` вы получите экземпляр `CaesarSalad`, свойство Extra которого имеет значение `Chicken`.

Поскольку свойство `Instance` привязано к аргументу интерфейса `IActivatingEventArgs<T>`, имеющему generic-тип, этот подход является строго типизированным и влечет за собой свои преимущества и недостатки.

В этом разделе вы увидели, как можно использовать Autofac для работы с более трудными API разработки. Для интеграции конструкторов и свойств с сервисами можно использовать различные отклонения от абстрактного класса `Parameter`, чтобы при этом соблюсти сходство с автоматической интеграцией, или можно воспользоваться методом `Register` и блоком кода, чтобы соблюсти большую типовую безопасность.

В целом Property Injection отлично поддерживается и механизмом автоматической интеграции и строго типизированными присваиваниями.

13.5. Резюме

В этой главе вы опробовали в действии DI-контейнер Autofac. Несмотря на то, что Autofac – это контейнер второго поколения, он все равно достаточно комплексный и подходит для большинства каверзных ситуаций, с которыми мы сталкиваемся при использовании DI-контейнеров. Архитектура этого контейнера напрямую построена на особенностях .NET 3.5 и C# 3.0, но, хотя внутри Autofac использует делегаты и блоки кода, общее API просто в использовании.

Важным вопросом, касающимся Autofac, является недвусмысленность. Он не пытается предугадать, что мы имеем в виду, а предлагает легкое в использовании API, которое предоставляет нам возможность явным образом разрешать различные возможности.

Одним из примеров такой недвусмысленности является то, что в отличие от других DI-контейнеров Autofac навязывает более строгую концепцию разделения между процессом конфигурирования и потребления контейнера. Компоненты мы конфигурируем с помощью экземпляра `ContainerBuilder`, но `ContainerBuilder` не умеет разрешать компоненты. После окончания процесса конфигурирования `ContainerBuilder`, мы используем его для создания `IContainer`, который можно использовать для разрешения компонентов.

Мы можем конфигурировать `ContainerBuilder` любым возможным способом: с помощью императивного кода, посредством XML или путем задания соглашений. А конфигурации мы можем пакетировать в модули.

То, как Autofac использует механизм управления жизненным циклом, несколько отличается от того, как его используют другие DI-контейнеры. Такие стандартные стили существования, как `Transient` и `Singleton` являются встроенными. Но другие контекстные стили существования, например, `Web Request Context`, необходимы для областей применения, в которых мы явным образом взаимодействуем с контейнером с целью определить контекст, в рамках которого разрешаются и высвобождаются компоненты. Это в полной мере касается стиля `Web Request Context`. С другой стороны, эта модель не позволяет нам реализовывать пользовательский стиль существования `Pooled` или кэшированный стиль существования. Таким образом, простого способа реализации пользовательских стилей существования с помощью Autofac нет. По мнению Николааса Блумхардта (создателя Autofac), такие ситуации никогда не встречались на дискуссионных форумах, поэтому, вероятнее всего, это не столь большая проблема. Чаще всего предлагаемого Autofac механизма управления жизненным циклом бывает более чем достаточно.

Autofac – современный DI-контейнер, который предлагает достаточно исчерпывающий набор возможностей. В следующей главе мы рассмотрим еще один DI-контейнер второго поколения – `Unity`.

14. Unity

Меню:

- Знакомство с Unity
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В предыдущей главе мы рассматривали контейнер Autofac – один из недавно появившихся DI-контейнеров. Еще один современный DI-контейнер – это Unity. Его мы и рассмотрим в этой главе.

Autofac можно назвать DI-контейнером *второго поколения*, поскольку он зародился и развивался прямоком на .NET 3.5, и не был нагружен никакими чертами более ранних версий .NET. Несмотря на то, что Unity появился примерно в то же самое время, он унаследовал более консервативный подход. Unity 1.0 был выпущен в мае 2008 года, но создавался он под .NET 2.0, поскольку его разработчики осознавали, что большинству организаций потребуется некоторое время на то, чтобы перейти на версию .NET 3.5.

Unity – это *модуль приложений*, разработанный группой *Patterns&Practices* компании Microsoft. Но не обращайте внимание на это название: *модуль приложений* – это просто повторно используемая библиотека с соответствующей документацией и примерами.

Unity и Enterprise Library

Некоторые путают Unity и Enterprise Library (еще одно решение группы Patterns&Practices) или, по крайней мере, то, как они взаимосвязаны друг с другом. Давайте внесем некоторую ясность.

Unity – это самостоятельная библиотека. Ему не нужна Enterprise Library.

С другой стороны, в состав Enterprise Library входит Unity, хотя и выступает он в этом случае всего лишь в роли контейнера по умолчанию для Enterprise Library. При необходимости Unity в Enterprise Library может быть заменен другим DI-контейнером.

Согласно некоторому совершенно не научному интернет-исследованию, Unity является самым широко используемым DI-контейнером. Это довольно удивительно, но, скорее всего, связано с тем фактом, что разработала его компания Microsoft. Существующие конкуренты, к примеру, Castle Windsor или StructureMap, не были отмечены в этом исследовании, поэтому, вероятнее всего, Unity познакомил с понятием DI-контейнеров совершенно новый сегмент пользователей, которые ранее ничего о нем не знали.

В этой главе мы в такой же последовательности, как делали это и для других DI-контейнеров, рассмотрим контейнер Unity. Вы увидите, как можно использовать Unity для применения принципов и паттернов, приведенных ранее в частях 1-3. На рисунке 14-1 представлена структура главы.

Рисунок 14-1: Эта глава состоит из четырех разделов. Первый раздел знакомит нас с Unity и демонстрирует способы конфигурирования и разрешения компонентов. Следующие три раздела связаны с паттернами применения, которые требуют

дополнительного внимания. Их можно прочитать по порядку или же пропустить некоторые из этих разделов и прочитать только те, которые вас интересуют.



Первый раздел знакомит нас с API контейнера Unity и должен рассматриваться в качестве предпосылки следующих трех разделов. Каждый из четырех разделов может быть прочитан независимо от остальных разделов, несмотря на то, что в четвертом разделе используются некоторые классы, которые вводятся в третьем разделе. Эти классы имеют относительно ясные имена, поэтому четвертый раздел можно прочитать, не читая перед этим третий. Но, с другой стороны, у вас случайно может появиться необходимость вернуться к этому разделу.

Подобно Castle Windsor контейнер Unity поддерживает несколько дополнительных возможностей, например, пользовательские жизненные циклы и механизм перехвата. В этой главе приводятся примеры и того, и другого явления, а также множество других более универсальных возможностей DI-контейнера. Все эти возможности суммированы, именно поэтому эта глава является одной из самых длинных глав этой книги.

Эта глава должна дать вам возможность приступить к работе, а также справиться с большинством общепринятых проблем, которые могут возникнуть при ежедневном использовании Unity. Это не окончательная трактовка Unity, а более подробную информацию о Unity можно получить онлайн. Возможно, из доступных на сегодняшний момент DI-контейнеров Unity лучше всего задокументирован.

Вы можете прочитать эту главу, не читая при этом остальные главы части 4, специально для того, чтобы познакомиться с Unity, или можете прочитать ее совместно с остальными главами части 4, чтобы сравнить DI-контейнеры. Цель данной главы – показать, как Unity связан с паттернами и принципами, описанными в остальной части книги, и как он их реализует.

[14.1. Знакомство с Unity](#)

[14.2. Управление жизненным циклом](#)

[14.3. Работа с составными компонентами](#)

[14.4. Конфигурирование сложных API](#)

[14.5. Резюме](#)

14.1. Знакомство с Unity

Из этого раздела вы узнаете, где можно взять Unity, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим универсальные варианты конфигурирования, а также то, как пакетировать настройки конфигурации в повторно используемые компоненты. В таблице 14-1 содержится основополагающая информация, которая, скорее всего, понадобится вам для того, чтобы приступить к работе с Unity.

Таблица 14-1: Краткая информация об Unity

Вопрос	Ответ
Откуда мне его получить?	Лучше всего начать с сайта http://unity.codeplex.com/ . На главной странице приведены ссылки на самые последние релизы, которые обычно перенаправляют пользователей на веб-узел Microsoft Download Center. Из Visual Studio 2010 можно получить его посредством NuGet. Имя пакета – <i>Unity</i> .
Что находится в загруженном файле?	В отличие от других контейнеров Unity загружается в виде <i>.msi</i> файла. После его установки создаются ярлыки в меню Пуск, а бинарные файлы и исходный код помещаются в папку Program Files. Но поскольку у вас теперь есть бинарные файлы, вы можете развернуть их с помощью Хсору, если предпочитаете эту форму.
Какие платформы поддерживаются?	.NET 3.5 SP1 и .NET 4.0. Silverlight 3 и 4.
Сколько он стоит?	Нисколько. По сути Unity – это программное обеспечение с открытым исходным кодом, несмотря на то, что группа Patterns&Practices не принимает внесенные в исходный код изменения.
Откуда мне получить помощь?	Unity – это предложение группы Patterns&Practices, а не продукт компании Microsoft. В связи с этим он не сопровождается компанией Microsoft, но зато существует довольно оживленный дискуссионный форум – http://unity.codeplex.com/discussions .
На какой версии Unity основана эта глава?	2.0.

Как и при работе с Castle Windsor и StructureMap, при использовании Unity соблюдается простая цикличность, проиллюстрированная на рисунке 14-2.

Рисунок 14-2: Сначала конфигурируется контейнер, а затем из него разрешаются компоненты. В большинстве случаев создается экземпляр класса `UnityContainer`, который полностью конфигурируется перед началом разрешения компонентов. Компоненты разрешаются из того же экземпляра, который перед этим конфигурировался.



После прочтения этого раздела вы приобретете полноценное понимание всей сущности паттерна применения Unity и сможете начать использовать его в сценариях, в которых все компоненты руководствуются должными DI-паттернами, например, Constructor Injection. Давайте начнем с простейшего сценария и посмотрим, как можно разрешать объекты с помощью контейнера Unity.

Разрешение объектов

Основная услуга, предоставляемая любым DI-контейнером – разрешение компонентов. В этом разделе мы рассмотрим API, которое позволяет разрешать компоненты с помощью Unity.

Из предыдущих глав вы узнали, что при работе с некоторыми контейнерами (например, Castle Windsor) перед тем, как начать разрешать компоненты, необходимо обязательно явным образом сконфигурировать *все* компоненты с помощью этого контейнера. В то же время остальные контейнеры (например, StructureMap) полностью понимают, как автоматически интегрировать запрашиваемые компоненты, поскольку они являются конкретными типами с открытыми конструкторами. В сущности, можно сказать, что Unity *распознал* последнюю из указанных выше категорий, будучи первым DI-контейнером, поддерживающим эту возможность, поэтому самым простым возможным способом использования контейнера Unity является следующий:

```
var container = new UnityContainer();  
SauceBéarnaise sauce = container.Resolve<SauceBéarnaise>();
```

Благодаря экземпляру `UnityContainer` generic-метод `Resolve` можно использовать для получения экземпляра конкретного класса `SauceBéarnaise`. Поскольку класс `SauceBéarnaise` имеет конструктор по умолчанию, Unity автоматически понимает, как создавать экземпляр этого класса. Никакого явного конфигурирования контейнера не требуется.

Unity поддерживает механизм автоматической интеграции. Поэтому даже при отсутствии конструктора по умолчанию он сможет создавать экземпляры без предварительной конфигурации, пока все входящие в конструктор параметры являются конкретными

классами и все листья дерева параметров имеют тип, обладающий конструкторами по умолчанию.

Рассмотрим в качестве примера конструктор `Mayonnaise`:

```
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

Несмотря на то, что рецепт приготовления майонеза слегка упрощен, и `EggYolk`, и `OliveOil` являются конкретными классами, обладающими конструкторами по умолчанию. Сам `Mayonnaise` не обладает конструктором по умолчанию, но Unity все равно может создавать его без предварительной конфигурации:

```
var container = new UnityContainer();  
var mayo = container.Resolve<Mayonnaise>();
```

Это возможно, поскольку Unity способен разгадать, как создавать все необходимые параметры конструктора. Но как только мы ввели слабое связывание, мы должны сконфигурировать Unity, преобразовав абстракции в конкретные типы.

Преобразование абстракций в конкретные типы

Несмотря на то, что способность Unity автоматически интегрировать конкретные классы может, наверняка, время от времени быть полезной, для слабого связывания обычно бывает необходимо преобразовывать абстракции к конкретным типам. Создание экземпляров на основании таких преобразований – ключевая услуга, предоставляемая любым DI-контейнером, но вам все равно необходимо задать соответствующее преобразование.

В приведенном ниже примере вы преобразуете интерфейс `IIngredient` в конкретный класс `SauceBéarnaise`, который позволяет успешно разрешать `IIngredient`:

```
var container = new UnityContainer();  
container.RegisterType<IIngredient, SauceBéarnaise>();  
IIngredient ingredient = container.Resolve<IIngredient>();
```

Generic-метод `RegisterType` – один из тех нескольких методов расширения, которые вызывают слабо типизированный метод `RegisterType`, заданный для `IUnityContainer`. В предыдущем примере вы использовали перегрузку, в которой определяли абстракцию, а также конкретный тип как два типовых generic-аргумента. В этом примере вы преобразуете `IIngredient` в `SauceBéarnaise` таким образом, чтобы при дальнейшем разрешении `IIngredient` вы получали экземпляр `SauceBéarnaise`.

Generic-метод расширения `RegisterType` помогает предотвратить появление ошибок конфигурации, поскольку конечный тип обладает generic-ограничителем, который задает условие, согласно которому этот тип наследуется от исходного типа. Код приведенный в предыдущем примере, компилируется, поскольку `SauceBéarnaise` реализует `IIngredient`.

В большинстве случаев все, что вам нужно – это строго типизированное API, и, поскольку в таком API выполняется необходимая проверка во время компиляции, вам лучше использовать его при любой возможности. Однако встречаются ситуации, когда разрешать сервисы необходимо с помощью более слабо типизированных способов. Такое тоже возможно.

Разрешение слабо типизированных сервисов

Иногда у вас нет возможности использовать generic API, поскольку на этапе проектирования вы еще не знаете, какой тип вам понадобится. У вас есть только экземпляр `Type`, но все же вам хотелось бы получить экземпляр этого типа. Пример такой ситуации описан в разделе 7.2 "Построение ASP.NET MVC приложений", в котором обсуждался ASP.NET класс `DefaultControllerFactory`. Соответствующий метод приведен в следующем примере:

```
protected internal virtual IController GetControllerInstance(  
    RequestContext requestContext, Type controllerType);
```

Поскольку у вас имеется только экземпляр `Type`, вы не можете использовать generic'i, а должны прибегнуть к слабо типизированному API. К счастью, Unity имеет слабо типизированную перегрузку метода `Resolve`, которая позволяет реализовывать метод `GetControllerInstance` следующим образом:

```
return (IController) this.container.Resolve(controllerType);
```

Слабо типизированная перегрузка метода `Resolve` позволяет передавать аргумент `controllerType` напрямую в Unity, но при этом вам необходимо явным образом приводить возвращаемое значение к `IController`.

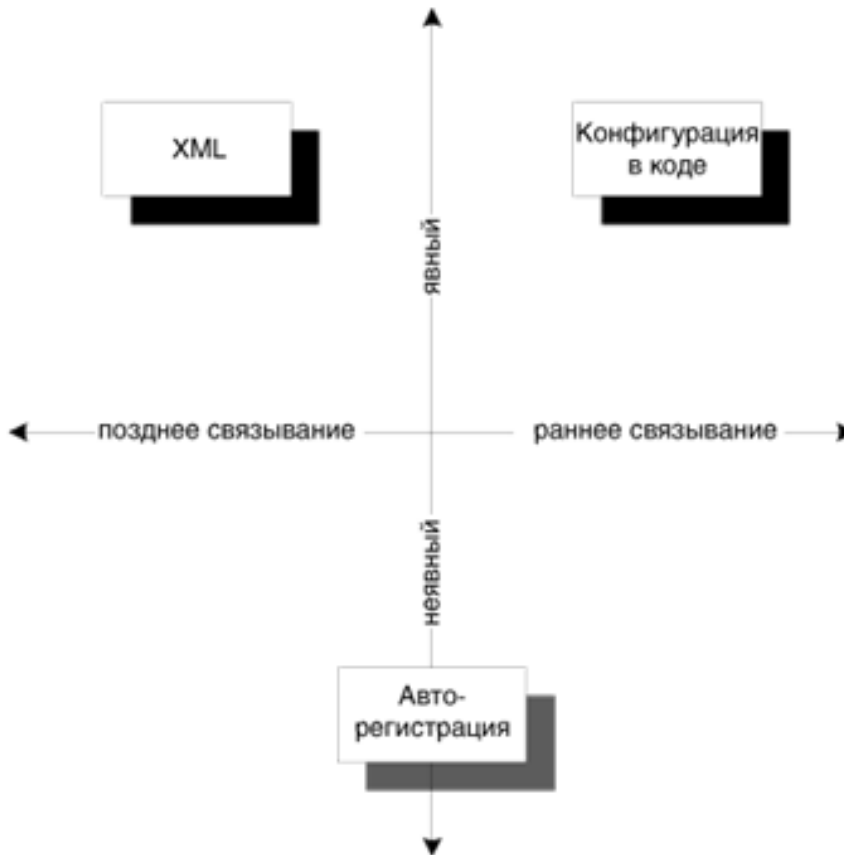
Независимо от того, какую из перегрузок метода `Resolve` вы используете, Unity гарантирует, что этот метод будет возвращать экземпляр запрашиваемого типа или выдавать исключение при наличии зависимостей, неудовлетворяющих условиям. После того, как все необходимые зависимости должным образом сконфигурированы, Unity может выполнить автоматическую интеграцию запрашиваемого типа.

В предыдущем примере `this.container` — это экземпляр `IUnityContainer`. Чтобы можно было разрешать запрашиваемый тип, сначала необходимо сконфигурировать все слабо связанные зависимости. Существует множество способов конфигурирования контейнера Unity, а в следующем разделе приведен обзор наиболее универсальных из этих способов.

Конфигурирование контейнера

Как уже говорилось в разделе 3.2 "Конфигурирование DI-контейнеров", существует несколько концептуально разных способов конфигурирования DI-контейнера. На рисунке 14-3 представлен обзор возможных вариантов.

Рисунок 14-3: Концептуально разные варианты конфигурирования. Технология конфигурирования в коде подразумевает строгую типизированность и имеет тенденцию к явному определению. XML, с другой стороны, предполагает позднее связывание, но тоже склонен к явному определению. Автоматическая регистрация, напротив, полагается на соглашения, которые могут быть и строго типизированными, и более слабо определенными, но в Unity не встроена поддержка механизма автоматической регистрации.



Несмотря на то, что Unity полностью поддерживает XML-конфигурацию, в его основе лежит императивная конфигурация. XML-конфигурация реализуется в произвольной библиотеке, для использования которой необходимо явным образом на нее ссылаться.

Хотя Unity поддерживает и конфигурацию в коде, и XML-конфигурацию, в него не встроена поддержка основанного на соглашениях механизма автоматической регистрации. Однако, как мы уже ранее упоминали, мы всегда можем реализовать соглашение посредством написания пользовательского императивного кода. В этом разделе мы будем рассматривать способы применения различных вариантов конфигурации.

Конфигурация в коде

В разделе 14.1.1 "Разрешение объектов" уже приводился краткий обзор строго типизированного API конфигурации контейнера Unity. В этом разделе мы рассмотрим его более подробно.

За исключением некоторых моментов вся конфигурация контейнера Unity выполняется в слабо типизированном методе `RegisterType`, определенном для `IUnityContainer`:

```
IUnityContainer.RegisterType(Type from, Type to, string name,
    LifetimeManager lifetimeManager,
    params InjectionMember[] injectionMembers);
```

Помимо этого метода в состав Unity входят еще несколько методов расширений. Некоторые из них являются строго типизированными generic-методами, а некоторые – слабо типизированными методами. В этой главе мы сосредоточим все свое внимание на строго типизированном API. Одним из наиболее часто используемых методов является приведенная ниже перегрузка:

```
container.RegisterType<IIngredient, SauceBéarnaise>();
```

В отличие от Castle Windsor или Autofac преобразование `IIngredient` в `SauceBéarnaise`, приведенное в предыдущем примере, не избавляет нас от разрешения самого `SauceBéarnaise`. И `sauce`, и `ingredient` будут соответствующим образом разрешаться в следующем коде:

```
container.RegisterType<IIngredient, SauceBéarnaise>();
var sauce = container.Resolve<SauceBéarnaise>();
var ingredient = container.Resolve<IIngredient>();
```

Возможно, из разделов 10.1.2 "Конфигурирование контейнера" и 13.1.2 "Конфигурирование ContainerBuilder" вы помните, что преобразование `IIngredient` в `SauceBéarnaise` с помощью Castle Windsor или Autofac будет приводить к исчезновению конкретного класса (`SauceBéarnaise`) до тех пор, пока вы не предпримите дополнительные шаги. При работе с Unity никаких дополнительных шагов предпринимать не нужно, поскольку Unity позволяет разрешать и `IIngredient`, и `SauceBéarnaise`. В большинстве случаев возвращаемые объекты – это экземпляры `SauceBéarnaise`.

В реальных приложениях нам всегда приходится преобразовывать более одной абстракции, поэтому нам необходимо сконфигурировать несколько преобразований. Сделать это можно с помощью нескольких вставок метода `RegisterType`:

```
container.RegisterType<IIngredient, SauceBéarnaise>();
container.RegisterType<ICourse, Course>();
```

Здесь выполняется преобразование `IIngredient` в `SauceBéarnaise` и преобразование `ICourse` в `Course`. Никаких наложений типов не происходит, поэтому все, что происходит дальше должно быть довольно очевидным. Кроме того, вы можете попытаться зарегистрировать одну и ту же абстракцию несколько раз, но, если вы сделаете это так, как приведено в примере ниже, то произойдет что-то неожиданное:

```
container.RegisterType<IIngredient, Steak>();
container.RegisterType<IIngredient, SauceBéarnaise>();
```

Регистрация типа без имени определяет значение по умолчанию для этого типа, но при этом переопределяется *предыдущее значение по умолчанию*. Конечным результатом этого примера является то, что, если вы будете разрешать `IIngredient`, то получите экземпляр `SauceBéarnaise`, но регистрация `Steak` при этом будет уже невозможна. Для типа возможно только одно значение по умолчанию, но зарегистрировать вы можете сколько угодно именованных компонентов. Чтобы использовать `Steak` в качестве `IIngredient` по умолчанию, вы можете зарегистрировать `SauceBéarnaise`, присвоив ему имя:

```
container.RegisterType<IIngredient, Steak>();  
container.RegisterType<IIngredient, SauceBéarnaise>("sauce");
```

Steak остается IIngredient по умолчанию, но, кроме того, вы можете разрешить IIngredient в SauceBéarnaise, запросив IIngredient с именем sauce. Чтобы точно указать, как будут интегрироваться зависимости, можно воспользоваться именованными компонентами. К этой теме мы вернемся в разделе 14.3.1.

Возможны и другие способы конфигурирования Unity, но с помощью приведенных в этом разделе методов можно полноценно сконфигурировать приложение. Чтобы оградить себя от слишком явного сопровождения конфигурации контейнера, было бы просто замечательно вместо этих методов использовать другой подход основанный на соглашениях – автоматическую регистрацию.

Автоматическая регистрация

Чаще всего большинство регистраций аналогичны друг другу. Такие регистрации очень утомительно сопровождать, а явная регистрация каждого компонента, возможно, не является самым продуктивным подходом.

Рассмотрим библиотеку, которая содержит реализации интерфейса IIngredient. Можно выполнять индивидуальную регистрацию каждого класса, но это приведет к многочисленным схожим вызовам метода RegisterType. Что еще хуже, всякий раз при добавлении новой реализации IIngredient необходимо явно регистрировать ее вместе с контейнером, чтобы она была доступна для использования. Было бы более продуктивно установить, что все реализации IIngredient, найденные в этой сборке, необходимо зарегистрировать.

К несчастью, в Unity не встроена поддержка автоматической регистрации, но, поскольку Unity имеет комплексное императивное API, мы можем написать пользовательский код, чтобы достичь того же эффекта. В следующем листинге мы рассмотрим простой пример, который демонстрирует, как этого можно достичь. Это никоим образом не является окончательной трактовкой этой темы, а скорее зарисовкой возможных вариантов.

Чтобы просмотреть сборку и зарегистрировать все реализации интерфейса IIngredient, можно воспользоваться комбинацией рефлексивного API.NET и слабо типизированного метода RegisterType, как это продемонстрировано в следующем листинге.

Листинг 14-1: Регистрация всех IIngredient сборки

```
foreach (var t in typeof(Steak).Assembly.GetExportedTypes())  
{  
    if (typeof(IIngredient).IsAssignableFrom(t))  
    {  
        container.RegisterType(typeof(IIngredient), t, t.FullName);  
    }  
}
```

Из этой сборки вы можете вытащить перечень всех открытых типов и выбрать только те из них, которые напрямую или косвенно реализуют интерфейс IIngredient. После применения фильтра для регистрации каждого типа IIngredient относительно интерфейса можно воспользоваться слабо типизированным методом Register. Чтобы предотвратить перетирание регистраций, необходимо не забывать присваивать каждой из

них уникальное имя. В этом примере вы используете полное имя каждого конкретного класса, но подойдет и что-нибудь другое, гарантирующее уникальность имени.

Несмотря на то, что Unity не обладает API, которое способно было бы справиться с основанным на соглашениях механизмом автоматической регистрации, для достижения того же эффекта, можно было бы написать свой собственный код. Конечно, предпочтительнее было бы иметь встроенную поддержку этого механизма, но, по крайней мере, API контейнера Unity не мешает нам определять соглашения вручную.

Подсказка

Проект с открытым исходным кодом *Unity Auto Registration* – одна из попыток определить повторно используемое API, которое позволило бы использовать механизм автоматической регистрации в Unity.

Будь мы более великодушны, мы могли бы сказать, что отсутствие API автоматической регистрации означает для нас отсутствие всяких ограничений. Если мы можем что-то закодировать, значит, мы можем это получить. Если бы мы захотели просмотреть папку на факт наличия в ней сборок или выполнить поиск типов в каждой сборке, то мы даже смогли бы реализовать дополнительную функциональность, при которой можно было бы добавлять в приложение расширения без повторной компиляции основного приложения. Это был бы один из способов реализации позднего связывания. Еще один способ – применение XML-конфигурации.

XML-конфигурация

XML-конфигурация является подходящим решением в тех ситуациях, когда нам необходимо уметь изменять конфигурацию без повторной компиляции приложения.

Подсказка

Рекомендую использовать XML-конфигурацию только для тех типов, изменять которые можно без повторной компиляции приложения. В остальных случаях используйте технологию конфигурирования в коде.

Unity подразумевает, что мы размещаем XML-конфигурацию в конфигурационном файле приложения. Для загрузки и интерпретации XML-конфигурации в Unity используется стандартное .NET API конфигурации.

Подсказка

Поддержка контейнером Unity возможности выполнения XML-конфигурации полностью сравнима с аналогичной возможностью других DI-контейнеров. В нем даже присутствует XSD-файл, который применяется для того, чтобы разрешить использовать в Visual Studio контекстную подсказку IntelliSense.

Примечание

Поскольку возможность поддержки контейнером Unity XML-конфигурации реализована в виде отдельной сборки, для использования этой возможности необходимо добавить ссылку на сборку `Microsoft.Practices.Unity.Configuration`.

После добавления ссылки на сборку `Microsoft.Practices.Unity.Configuration` мы должны добавить директиву `using` для пространства имен `Microsoft.Practices.Unity.Configuration`, чтобы нам стал доступен метод расширения `LoadConfiguration`. Это позволяет нам загружать XML-конфигурацию посредством всего лишь одного вызова метода:

```
container.LoadConfiguration();
```

Метод `LoadConfiguration` загружает XML-конфигурацию из стандартного конфигурационного файла приложения в контейнер.

Примечание

К сожалению, мы не можем считывать XML из других источников, например, из потоков или XML-узлов, поскольку соответствующее API в Unity отсутствует.

Подсказка

Несмотря на то, что мы не можем считывать XML из произвольных источников, мы можем считывать XML из любого конфигурационного файла с помощью API `ConfigurationManager`.

Чтобы разрешить выполнять конфигурацию контейнера Unity в конфигурационном файле, мы должны сначала добавить раздел конфигурации, используя стандартное .NET API конфигурации, предназначенное для определения пользовательских разделов конфигурации:

```
<configSections>
  <section name="unity"
    type="Microsoft.Practices.Unity.Configuration.
      ↳UnityConfigurationSection,
      ↳Microsoft.Practices.Unity.Configuration"/>
</configSections>
```

Это API позволяет нам добавить в конфигурационный файл раздел конфигурации `unity`. Ниже приведен простой пример преобразования интерфейса `IIngredient` в класс `Steak`:

```
<unity>
  <namespace name="Ploeh.Samples.MenuModel" />
  <assembly name="Ploeh.Samples.MenuModel" />
  <container>
    <register type="IIngredient" mapTo="Steak" />
  </container>
</unity>
```

XML-схема контейнера Unity позволяет определять осмысленные значения по умолчанию, которые, возможно, помогут сократить большие объемы кода, появляющиеся при работе с квалифицированными именами типов сборки в XML. Несмотря на то, что добавлять элементы `namespace` не обязательно, мы можем добавлять их в неограниченном количестве. Элементы `namespace` эквивалентны директивам `using` языка C#. В приведенном выше примере кода мы добавляем только одно пространство имен `Ploeh.Samples.MenuModel`, но могли бы добавлять и большее количество таких элементов или же вовсе их опустить. Если мы опускаем элемент `namespace`, то все равно можем

явным образом передавать полностью квалифицированное имя типа в виде составляющей части регистрации.

Работа с элементом `assembly` аналогична работе с элементом `namespace`. Мы можем добавлять неограниченное количество элементов `assembly` или же совсем их опускать. В приведенном выше примере кода мы добавляем сборку `Ploeh.Samples.MenuModel`, в которой определяются интерфейс `IIngredient` и класс `Steak`.

Это позволяет нам четко устанавливать соответствие между `IIngredient` и `Steak` с помощью элемента `register`. Поскольку мы добавили в контекст пространства имен и сборки, мы можем ссылаться на `IIngredient` и `Steak` по их сокращенным именам. Пока имена в рамках контекста четко определены, Unity раскрывает их за нас подобно тому, как это делает компилятор C#.

XML-конфигурация является подходящим решением в тех ситуациях, когда нам необходимо изменить конфигурацию одного или нескольких компонентов без повторной компиляции приложения. Но поскольку такой подход является довольно хрупким, мы должны использовать его только для таких случаев, а для остальной части конфигурации контейнера использовать технологию конфигурирования в коде.

Подсказка

Вы не забыли, что выигрывает последняя конфигурация? Этот факт вы можете использовать для того, чтобы переписать жестко закодированную конфигурацию с помощью XML-конфигурации. Для этого вы должны не забыть загрузить XML-конфигурацию после конфигурации всех остальных компонентов. И наоборот, если у вас имеется конфигурация, которую нельзя переписывать с помощью XML, примените эту конфигурацию после загрузки XML-конфигурации.

В этом разделе мы, главным образом, рассматривали различные API регистрации контейнера Unity. Несмотря на то, что вы, наверняка, можете записать один большой блок неструктурированного кода конфигурации, лучше всего разбить конфигурацию на модули. Хотя контейнер Unity явным образом не поддерживает такую возможность, оказывается, мы все равно можем достичь такого же результата.

Пакетирование конфигурации

Иногда существует необходимость упаковать логику конфигурации в повторно используемые группы. И даже в тех случаях, когда само по себе повторное использование не является высшим приоритетом, при конфигурировании больших и сложных приложений может появиться необходимость создать некоторого рода структуру.

В контейнере Castle Windsor есть инсталлеры, в StructureMap – регистры, а в Autofac – модули. Но в Unity нет ничего похожего. В нем отсутствует интерфейс, который был бы предназначен, главным образом, для пакетирования конфигурации в повторно используемые компоненты. Но зато в нем есть более или менее достаточная замена этим механизмам: *расширения контейнера* (Container Extensions).

Целью расширения контейнера Unity является не только пакетирование конфигурации в повторно используемые пакеты. Как следует из его названия, расширение контейнера также можно использовать для того, чтобы расширять функциональность контейнера

Unity. Например, такая возможность контейнера Unity, как механизм перехвата, реализуется в виде расширения контейнера (это конкретное расширение мы рассмотрим в разделе 14.3.4).

Несмотря на то, что расширения контейнера можно использовать для множества различных целей, их также можно использовать и для разбиения конфигурации на модули. Все, что нам нужно для реализации расширения контейнера, – выполнить наследование от абстрактного класса `UnityContainerExtension` и реализовать метод `Initialize` этого класса. В следующем листинге продемонстрировано, как можно с легкостью преобразовать код из листинга 14-1 в расширение контейнера.

Листинг 14-2: Реализация расширения контейнера

```
public class IngredientExtension : UnityContainerExtension
{
    protected override void Initialize()
    {
        var a = typeof(Steak).Assembly;
        foreach (var t in a.GetExportedTypes())
        {
            if (typeof(IIngredient).IsAssignableFrom(t))
            {
                this.Container.RegisterType(
                    typeof(IIngredient), t, t.FullName);
            }
        }
    }
}
```

Класс `IngredientExtension` наследуется от абстрактного класса

`UnityContainerExtension` для того, чтобы пакетировать основанную на соглашениях конфигурацию из листинга 14-1 в повторно используемый класс. При наследовании от класса `UnityContainerExtension` необходимо реализовать абстрактный метод `Initialize`, в котором вы можете выполнить все необходимые вам действия.

Единственное функциональное отличие от листинга 14-1 заключается в том, что теперь вместо локальной переменной вы вызываете метод `RegisterType` унаследованного свойства `Container`.

Чтобы воспользоваться расширением контейнера, можно вызвать метод `AddExtension` или соответствующий метод расширения. Если расширение обладает конструктором по умолчанию, то вы можете использовать характерное условное обозначение метода расширения:

```
container.AddNewExtension<IngredientExtension>();
```

Метод `AddNewExtension` вызывает метод `AddExtension`, который вы также можете использовать в тех ситуациях, когда вам необходимо создать модуль вручную:

```
container.AddExtension(new IngredientExtension());
```

Эти примеры функционально эквивалентны.

Подсказка

Расширения контейнера Unity позволяют пакетировать и структурировать конфигурационный код вашего контейнера. Даже если расширения разрабатываются совсем не для этих целей, у вас может появиться желание использовать их вместо однострочной конфигурации. Такой подход сделает вашу Composition Root более читабельной.

Благодаря функциональности расширений контейнера для конфигурирования контейнера Unity можно использовать как технологию конфигурирования в коде, так и XML-конфигурацию, и даже реализованный пользователем механизм автоматической регистрации (хотя этот подход занимает больше времени). После того как контейнер сконфигурирован, можно приступить к разрешению сервисов с помощью этого контейнера, что описано в разделе 14.1.1 "Разрешение объектов".

Этот раздел познакомил нас с DI-контейнером Unity и продемонстрировал фундаментальные принципы как сконфигурировать и впоследствии использовать созданный контейнер для разрешения сервисов. Разрешение сервисов с легкостью выполняется посредством единичного вызова метода `Resolve`, поэтому вся сложность заключается в конфигурировании контейнера. Конфигурировать контейнер можно несколькими способами, включая императивный код и XML. До настоящего момента мы рассматривали только самое основное API, а более продвинутые вопросы в этом разделе еще не охватывались. Одна из самых важных проблем — это управление жизненным циклом компонентов.

14.2. Управление жизненным циклом

В главе 8 обсуждался процесс управления жизненным циклом, в том числе наиболее универсальные стили существования, к примеру, Singleton и Transient. Unity поддерживает множество различных стилей существования и позволяет конфигурировать жизненные циклы всех сервисов. Продемонстрированные в таблице 14-2 стили существования являются частью API контейнера Unity.

В контейнере Unity реализации стилей существования Transient, Per Graph и Singleton эквивалентны основным стилям существования, описанным в главе 8. Поэтому я не буду тратить время на рассмотрение этих стилей существования.

Предупреждение

Несмотря на то, что стиль существования Per Resolve совпадает с описанием, приведенным в разделе 8.3.3 "Per Graph", он имеет некоторые известные дефекты, которые делают менее предпочтительным для использования.

Таблица 14-2: Стили существования Unity

Название	Комментарии
Transient	Этот стиль существования используется по умолчанию. Экземпляры контейнером не отслеживаются.
Container Controlled	В Unity это название используется для обозначения стиля Singleton.
Per Resolve	В Unity это название используется для обозначения стиля Per Graph. Экземпляры контейнером не отслеживаются.
Hierarchical	Связывает жизненные циклы компонентов с дочерним контейнером (см. раздел 14.2.1).
Per Thread	Для одного потока создается один экземпляр. Экземпляры контейнером не отслеживаются.
Externally Controlled	Разновидность стиля существования Singleton, при котором сам контейнер содержит только хрупкую ссылку на экземпляр, позволяющую уничтожить его сборщиком мусора в случае неиспользования.

Подсказка

Используемый по умолчанию стиль существования Transient является самым безопасным вариантом, но не всегда самым эффективным. Наиболее подходящий стиль существования для потоко-безопасных сервисов – стиль Singleton, но при этом нужно не забывать явным образом регистрировать эти сервисы.

В этом разделе вы познакомитесь со способами определения стилей существования для компонентов – как с помощью кода, так и посредством XML. Кроме того, чтобы продемонстрировать, что мы не ограничены встроенными стилями существования контейнера Unity, мы рассмотрим процесс реализации пользовательского стиля существования. После прочтения этого раздела вы уже сможете использовать стили существования контейнера Unity в своем собственном приложении.

Перед тем как приступить к разработке пользовательского стиля существования, необходимо рассмотреть способы конфигурирования и использования стилей существования.

Конфигурирование стиля существования

В этом разделе мы рассмотрим способы управления стилями существования компонентов, которые применяются в контейнере Unity. Жизненный цикл компонента конфигурируется в рамках регистрации компонентов и может задаваться как в коде, так и в XML. Мы поочередно рассмотрим каждый из этих способов.

Конфигурирование стиля существования с помощью кода

Стиль существования конфигурируется с помощью перегрузки метода `RegisterType`, которая используется для регистрации компонентов в целом. По своей простоте она равносильна следующему коду:

```
container.RegisterType<SauceBéarnaise>(
    new ContainerControlledLifetimeManager());
```

В этом примере конкретный класс `SauceBéarnaise` конфигурируется в виде `Singleton` таким образом, что всякий раз при запрашивании `SauceBéarnaise` возвращается один и тот же экземпляр. Если нам необходимо преобразовать абстракцию в конкретный класс, имеющий конкретный стиль существования, можно воспользоваться еще одной похожей перегрузкой метода `RegisterType`:

```
container.RegisterType<IIIngredient, SauceBéarnaise>(
    new ContainerControlledLifetimeManager());
```

В этом примере `IIIngredient` преобразуется в `SauceBéarnaise` и конфигурируется в виде `Singleton`. В двух предыдущих примерах вы использовали перегрузки метода `RegisterType`, которые в качестве аргумента принимали экземпляр `LifetimeManager`. Вместо `ContainerControlledLifetimeManager` вы можете использовать любой другой класс, унаследованный от абстрактного класса `LifetimeManager`. В контейнере Unity для каждого стиля существования, описанного в таблице 14-2, есть свой `LifetimeManager`. Но, как вы впоследствии увидите в разделе 14.2.2, можно создать и свой собственный `LifetimeManager`.

Несмотря на то, что стиль `Transient` используется по умолчанию, мы можем задать это явным образом. Приведенные ниже примеры эквивалентны:

```
container.RegisterType<SauceBéarnaise>();
container.RegisterType<SauceBéarnaise>(
    new TransientLifetimeManager());
```

Благодаря использованию технологии конфигурирования в коде мы можем регистрировать компоненты, имеющие различные стили существования, любым способом. Несмотря на то, что это намного более гибкий способ конфигурирования компонентов, иногда для получения позднего связывания нам необходимо прибегать к XML. В этом случае мы также можем объявлять стили существования.

Конфигурирование стиля существования с помощью XML

Когда нам нужно определять компоненты в XML, нам также может понадобиться возможность конфигурировать в этом же месте их стили существования. Это можно легко сделать в рамках XML-схемы, которая уже рассматривалась в разделе 14.1.2 "Конфигурирование контейнера". Для объявления стиля существования можно использовать и необязательный элемент `lifetime`:

```
<register type="IIngredient" mapTo="Steak">
  <lifetime type="ContainerControlledLifetimeManager" />
</register>
```

Отличие от примера из раздела 14.1.2 "Конфигурирование контейнера" заключается в том, что теперь вы добавили необязательный элемент `lifetime` для того, чтобы определить, какой из `LifetimeManager` должен использоваться для регистрации. Чтобы сконфигурировать компонент в виде `Singleton`, вы устанавливаете атрибут типа, равным псевдониму `ContainerControlledLifetimeManager`, но вместо него могли бы использовать и квалифицированное имя типа сборки или пользовательский псевдоним, если бы вам нужно было присвоить пользовательский `LifetimeManager`.

Конфигурировать жизненные циклы легко как в коде, так и в XML. В обоих случаях конфигурирование жизненных циклов выполняется в довольно декларативной манере. Несмотря на то, что конфигурация выполняется довольно просто, вы не должны забывать о том, что некоторые стили существования содержат объекты-долгожители, которые пользуются ресурсами на протяжении всего своего жизненного цикла.

Высвобождение компонентов

Как уже говорилось в разделе 8.2.2 "Управление устраняемыми зависимостями", важно высвободить объекты после завершения работы с ними, чтобы каждый устраняемый экземпляр можно было бы устранить по истечении его жизненного цикла. Это возможно, но в рамках контейнера Unity сделать это довольно-таки трудно.

Предупреждение

Контейнер Unity не уничтожает устраняемые зависимости до тех пор, пока кто-нибудь явно не прикажет ему это сделать.

`IUnityContainer` определяет метод `Teardown`, который с первого взгляда кажется похожим на эквивалентный метод `Release` контейнера Castle Windsor. Мы можем попробовать использовать его таким же образом:

```
container.Teardown(ingredient);
```

Однако независимо от того, какой из встроенных стилей существования мы выбрали, ни один из компонентов не уничтожается. Это приводит к нарушению принципа "наименьшего удивления" (*Principle of Least Surprise*).

Предупреждение

Метод `Teardown` не уничтожает устраняемые зависимости.

Несмотря на то, что метод `Teardown` не выполняет (по умолчанию) то, что нам хотелось бы, нам, тем не менее, доступны некоторые другие варианты. Один из таких вариантов – реализовать пользовательский стиль существования (что вы и сделаете в следующем разделе). Еще один вариант – использовать комбинацию дочерних контейнеров и стиля существования `Hierarchical`.

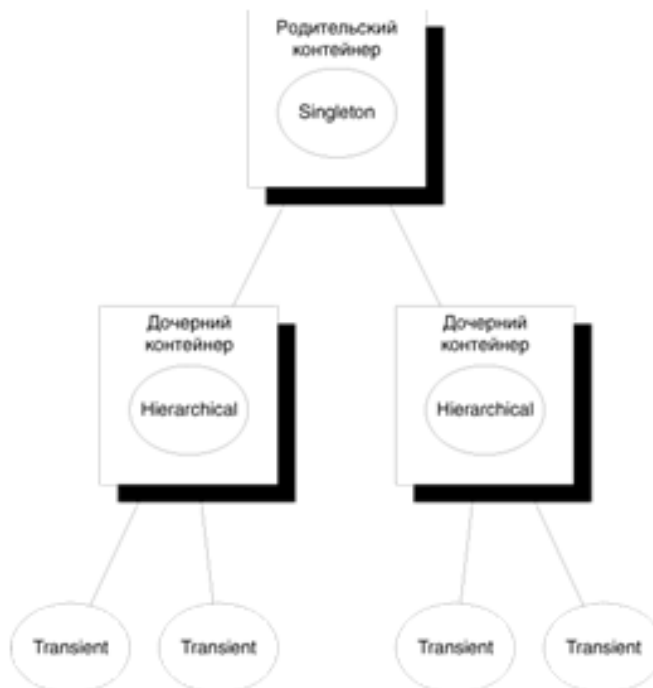
Сущность стиля существования `Hierarchical` заключается в том, что в рамках дочернего контейнера он выступает в роли `Singleton`, но при этом каждый дочерний контейнер обладает своим собственным локальным `Singleton`.

Примечание

Комбинация дочерних контейнеров и стиля существования `Hierarchical` аналогична областям применения контейнера `Autofac`, описанным в разделе 13.2.1 "Конфигурирование областей применения экземпляров".

Дочерний контейнер – это копия родительского контейнера. При создании дочернего контейнера из родительского дочерний контейнер наследует из него всю конфигурацию, но впоследствии мы можем изменять дочерний контейнер, не влияя при этом на родительский. Такая возможность может быть полезной, если нам необходимо переопределить только небольшую часть конфигурации родительского контейнера. Дочерний контейнер чаще всего имеет более ограниченную область применения. Как показывает рисунок 14-4, он также определяет границу, в пределах которой могут повторно использоваться компоненты

Рисунок 14-4: Дочерние контейнеры могут совместно использовать компоненты в течение ограниченного периода или для ограниченного круга целей. Компонент `Hierarchical` по существу играет роль `Singleton` в рамках этого контейнера. Независимо от того, сколько раз мы запрашиваем у дочернего контейнера этот компонент, мы получаем один и тот же экземпляр. Другой дочерний контейнер будет получать свой собственный экземпляр, а родительский контейнер управляет совместно используемыми `Singleton`'ами. `Transient`-компоненты нельзя использовать совместно.



При создании нового дочернего контейнера он наследует все `Singleton`'ы, которыми управляет родительский контейнер, но при этом выступает в роли контейнера "локальных `Singleton`'ов". Когда из дочернего контейнера запрашивается компонент `Hierarchical`, мы всегда получаем один и тот же экземпляр. Отличие от истинных `Singleton`'ов заключается в том, что, если мы запросим компонент `Hierarchical` у второго дочернего контейнера, то получим совсем другой экземпляр.

Однако `Transient`-компоненты функционируют так, как и должны, независимо от того, разрешаем ли мы их из родительского или из дочернего контейнера.

Подсказка

Дочерние контейнеры и стиль существования `Hierarchical` можно использовать в качестве еще одного варианта опускания стиля существования `Web Request Context`: создайте новый дочерний контейнер в начале каждого веб-запроса и используйте его для разрешения компонентов. После завершения запроса уничтожьте дочерний контейнер.

Одной из важных особенностей дочерних контейнеров является то, что они позволяют нам соответствующим образом высвобождать компоненты по истечении их жизненного цикла. С помощью метода `CreateChildContainer` мы создаем новый дочерний контейнер и высвобождаем все соответствующие компоненты посредством вызова метода `Dispose`:

```
1. using (var child = container.CreateChildContainer())
2. {
3.     var meal = child.Resolve<IMeal>();
4. }
```

Строка 3: Уничтожение обеда

Новый дочерний контейнер создается из `container` посредством вызова метода `CreateChildContainer`. Возвращаемое значение реализует интерфейс `IDisposable`, поэтому вы можете поместить его в директиву `using`. Получаем новый экземпляр `IUnityContainer`. В связи с этим вы можете использовать `child` для того, чтобы разрешать компоненты точно таким же способом, как и при использовании родительского контейнера.

После окончания работы с дочерним контейнером вы можете уничтожить его. При использовании директивы `using` дочерний контейнер автоматически уничтожается при выходе из этой директивы. Но, безусловно, вы можете сделать это и, явно уничтожив дочерний контейнер посредством вызова метода `Dispose`. При уничтожении `child` вы также высвобождаете все компоненты, созданные дочерним контейнером. В случае приведенного выше примера это означает, что вы высвобождаете диаграмму объекта `meal`.

Примечание

Не забывайте, что высвобождение устранимого компонента и его уничтожение — это не одно и то же. Это сигнал контейнеру о том, что срок эксплуатации этого компонента завершился. Если это `Hierarchical`-компонент, то он уничтожится автоматически, а если `Singleton`, то он не будет уничтожен автоматически.

Предупреждение

Устраиваемые объекты со стилями существования `Transient` или `Per Graph` не уничтожаются при уничтожении дочернего контейнера. Это может привести к утечкам памяти.

Ранее в этом разделе вы уже видели, как сконфигурировать компоненты в виде `Singleton` или `Transient`. Конфигурирование компонента в виде `Hierarchical` выполняется аналогичным образом:

```
container.RegisterType<IIngredient, SauceBéarnaise>(
    new HierarchicalLifetimeManager());
```

При регистрации компонента с определенным стилем существования всегда используется перегрузка метода `RegisterType`, принимающая в качестве аргумента `LifetimeManager`. Чтобы использовать стиль существования `Hierarchical`, вы передаете в метод экземпляр `HierarchicalLifetimeManager`.

По своей сущности `Singleton`'ы не высвобождаются по истечении жизненного цикла самого контейнера. Однако мы можем высвободить даже такие компоненты, если контейнер нам больше не нужен. Делается это посредством уничтожения самого контейнера:

```
container.Dispose();
```

На практике это не столь важно, поскольку жизненный цикл контейнера находится в близких взаимоотношениях с жизненным циклом поддерживаемого им приложения. Обычно мы сохраняем контейнер активным до тех пор, пока приложение запущено. Поэтому устраним мы его только тогда, когда приложение закрывается, при этом операционная система восстанавливает память.

Стили существования, встроенные в контейнер `Unity`, могут показаться довольно исчерпывающим набором стилей существования, удовлетворяющим практически всем повседневным нуждам. Но когда дело доходит до высвобождения компонентов, могут возникать проблемы в виде дефектов, а также проблемы несовместимости. С другой стороны, в контейнере `Unity` присутствует достаточное количество `Seam`'ов, чтобы можно было справиться с этими проблемами посредством разработки пользовательских стилей существования.

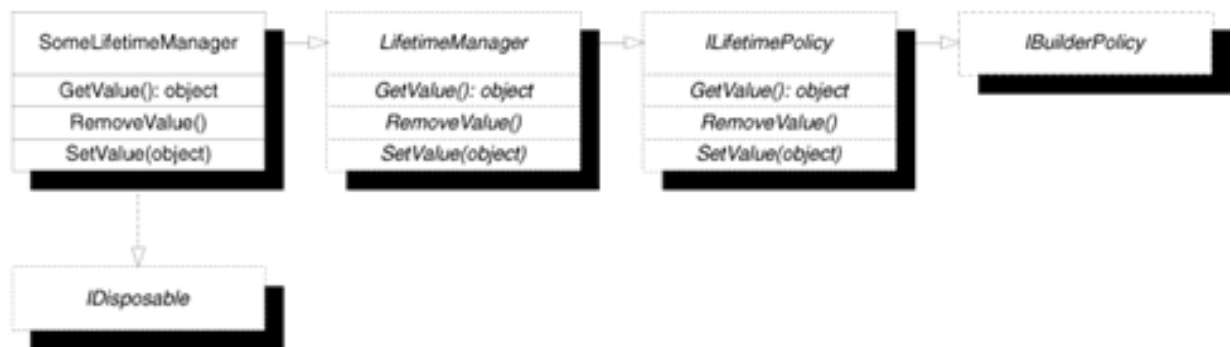
Разработка пользовательского стиля существования

В большинстве случаев мы должны уметь выходить из разных ситуаций с помощью исчерпывающего набора стилей существования, уже предоставляемых контейнером `Unity`, но если у нас имеются особые нужды или нам необходимо справиться с проблемами прекращения срока эксплуатации, можно реализовать пользовательский стиль существования. В данном разделе вы увидите, как это сделать. Мы рассмотрим `Seam`'ы, которые делают возможным создание пользовательского стиля существования, и часть времени потратим на рассмотрение примера, перемежая теорию с практикой.

Понимание API LifetimeManager

В разделе 14.2.1 "Конфигурирование стиля существования" мы уже мельком рассматривали API стилей существования контейнера Unity. Несколько перегрузок метода `RegisterType` принимают в качестве параметра экземпляр абстрактного класса `LifetimeManager`, который моделирует процесс взаимодействия стилей существования с остальной частью контейнера Unity. На рисунке 14-5 продемонстрирована небольшая иерархия типов, связанная с классом `LifetimeManager`.

Рисунок 14-5: `SomeLifetimeManager` реализует пользовательский стиль существования посредством наследования от абстрактного класса `LifetimeManager`, который, в свою очередь, реализует интерфейс `ILifetimePolicy`, унаследованный от интерфейса `IBuilderPolicy`. Пользовательский стиль существования может реализовать `IDisposable`, чтобы внедрить функциональность постобработки, в результате которой уничтожается контейнер.



При реализации пользовательского стиля существования важнейшим типом является абстрактный класс `LifetimeManager`. Даже если `LifetimeManager` реализует `ILifetimePolicy`, это нас никоим образом не касается, поскольку перегрузки метода `RegisterType` принимают в качестве параметра только экземпляры `LifetimeManager`, а не экземпляры `ILifetimePolicy` или `IBuilderPolicy`.

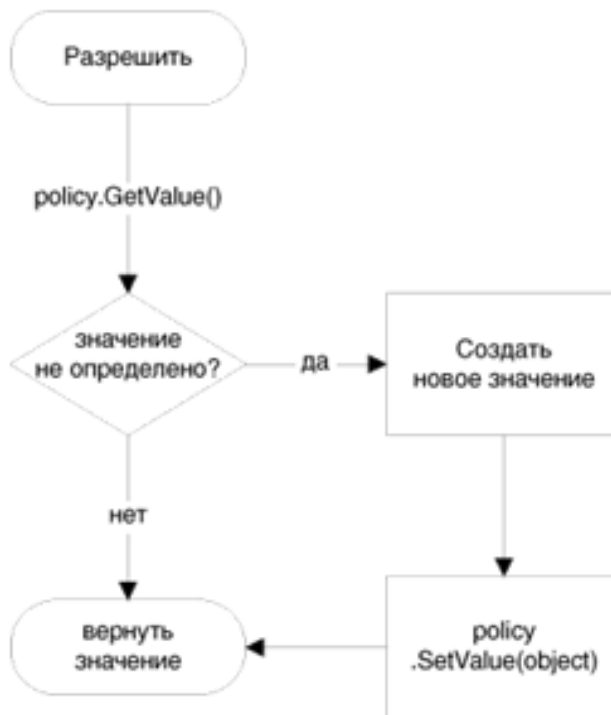
Мы можем реализовать `IDisposable`, чтобы внедрить функциональность постобработки, но по умолчанию эта функциональность работает не так, как нам бы хотелось. Метод `Dispose` вызывается не всегда. Позднее мы еще вернемся к рассмотрению этого вопроса.

Предупреждение

То, что мы реализуем `IDisposable`, еще не гарантирует, что будет вызван метод `Dispose`.

При разрешении компонента контейнер Unity взаимодействует с `LifetimeManager`, что проиллюстрировано на рисунке 14-6.

Рисунок 14-6: Контейнер Unity взаимодействует с интерфейсом `ILifetimePolicy`, вызывая сначала метод `GetValue`. Если `policy` возвращает какое-то значение, то это значение незамедлительно используется. Если значение не возвращается, то Unity создает новое значение и устанавливает его в `policy` перед тем, как вернуть это значение.



Примечание

Механизм, проиллюстрированный на рисунке 14-6, аналогичен взаимодействию StructureMap с `IObjectCache`, которое продемонстрировано на рисунке 11-5.

Сначала Unity пытается получить запрашиваемый экземпляр из метода `GetValue`. Если этот метод возвращает `null`, то Unity создает запрашиваемый экземпляр и добавляет его в `policy` с помощью метода `SetValue` перед тем, как вернуть это значение. Таким образом, один экземпляр `ILifetimePolicy` управляет одним компонентом.

Предупреждение

Метод `RemoveValue` никогда не вызывается контейнером Unity.

Несмотря на то, что методы `GetValue` и `SetValue` принимают участие в процессе разрешения запроса контейнером Unity, метод `RemoveValue` никогда не вызывается контейнером. Объяснение того, почему метод `Teardown` не работает так, как нам бы хотелось, займет слишком времени. Мы могли бы оставить реализацию пустой, но, оказывается, мы можем изменить назначение метода. Перед детальным рассмотрением этого вопроса изучение примера, охватывающего самые основы, могло бы прояснить некоторые моменты.

Разработка стиля существования Caching

В приведенном ниже примере мы будем разрабатывать стиль существования Caching, который уже создавали для контейнеров Castle Windsor и StructureMap в разделах 10.2.3

"Разработка пользовательского стиля существования" и 11.2.2 "Разработка пользовательского стиля существования". Если кратко, то этот стиль существования кэширует и повторно в течение некоторого времени использует экземпляры перед тем, как их высвободить.

Несмотря на то, что вы можете добавить некоторую дополнительную линию поведения, реализовав `IDisposable` и применив некоторые уловки, вам нужно будет реализовать, как минимум, три абстрактных метода, определенных `LifetimeManager`. Это продемонстрировано в следующем листинге.

Листинг 14-3: Реализация пользовательского `LifetimeManager`

```
1. public partial class CacheLifetimeManager :
2.     LifetimeManager, IDisposable
3. {
4.     private object value;
5.     private readonly ILease lease;
6.     public CacheLifetimeManager(ILease lease)
7.     {
8.         if (lease == null)
9.         {
10.            throw new ArgumentNullException("lease");
11.        }
12.        this.lease = lease;
13.    }
14.    public override object GetValue()
15.    {
16.        this.RemoveValue();
17.        return this.value;
18.    }
19.    public override void RemoveValue()
20.    {
21.        if (this.lease.IsExpired)
22.        {
23.            this.Dispose();
24.        }
25.    }
26.    public override void SetValue(object newValue)
27.    {
28.        this.value = newValue;
29.        this.lease.Renew();
30.    }
31. }
```

Строка 14-18: Получение значения

Строка 19-25: Удаление значения

Строка 26-30: Установка значения

Чтобы реализовать стиль существования `Caching` необходимо унаследовать класс `CacheLifetimeManager` от абстрактного класса `LifetimeManager`. Кроме того, класс `CacheLifetimeManager` реализует `IDisposable`, но мы немного повременим с изучением реализации, поэтому в листинге 14-3 метод `Dispose` пропущен.

`CacheLifetimeManager` для получения экземпляра `ILease` использует паттерн `Constructor Injection`. Интерфейс `ILease` – это локальный вспомогательный интерфейс, который вводится для реализации необходимой функциональности. Впервые этот интерфейс был

введен в разделе 10.2.3 "Разработка пользовательского стиля существования" и никак не влияет на контейнер Unity или любой другой DI-контейнер.

Примечание

Пример реализации `ILease` можно увидеть в разделе 10.2.3 "Разработка пользовательского стиля существования".

Метод `GetValue` сначала вызывает метод `RemoveValue`, чтобы обезопасить себя от недействительного срока аренды, а затем возвращает значение поля `value`. Поле `value` может иметь `null`-значение, но, как демонстрирует рисунок 14-6, это ожидаемый сценарий. С другой стороны, в поле может содержаться значение, если сначала был вызван метод `SetValue` и при этом срок аренды не просрочен.

Несмотря на то, что метод `RemoveValue` никогда не вызывается самим Unity, это все равно отличное место для реализации кода, позволяющего высвободить компонент. Поскольку целью `CacheLifetimeManager` является кэширование значения на некоторое время, вы устраняете компонент только по окончании срока аренды. В противном случае вы храните его несколько дольше. Метод `Dispose` не включен в листинг 14-3, но мы скоро к нему вернемся.

Метод `SetValue` сохраняет значение в поле `value` и продляет срок аренды. Согласно схеме, приведенной на рисунке 14-6, метод `SetValue` вызывается только тогда, когда Unity создает новое значение для рассматриваемого компонента, причем в этом случае уместно продлить срок аренды.

Примечание

Сравните конструктор из листинга 14-3 с более сложным кодом, приведенным в листинге 10.2. Это сравнение отчетливо демонстрирует превосходство паттерна `Constructor Injection` над `Method Injection`.

Все это реализует ключевую функциональность, необходимую для `LifetimeManager`. Хотя нам все равно нужно обсудить реализацию `IDisposable` и то, что под ним подразумевается, нам следует для начала вкратце рассмотреть то, как `CacheLifetimeManager` сопоставляется с экземпляром `UnityContainer`.

Регистрация компонентов с пользовательским стилем существования

Применять `CacheLifetimeManager` в рамках компонента довольно легко и делается это наподобие определения всех остальных стилей существования:

```
var lease = new SlidingLease(TimeSpan.FromMinutes(1));
var cache = new CacheLifetimeManager(lease);
container.RegisterType<IIngredient, SauceBéarnaise>(cache);
```

В этом примере кода контейнер конфигурируется таким образом, что он с односторонней задержкой использует `CacheLifetimeManager` для экземпляра `IIngredient`. В рамках одностороннего промежутка вы можете запрашивать какое угодно количество диаграмм объектов, и всякий раз, когда диаграмма содержит экземпляр `IIngredient`, вы будете получать один и тот же `SauceBéarnaise`. По истечении односторонней задержки при последующих запросах будет возвращаться новый экземпляр `SauceBéarnaise`.

Разрешение компонентов с пользовательским стилем существования выполняется обычным образом. Сюрпризы начинаются только, когда мы пытаемся высвободить разрешенные диаграммы объектов.

Высвобождение компонентов с пользовательским стилем существования

Как я уже упоминал, метод `RemoveValue` никогда не вызывается контейнером Unity. Тем не менее, если нам необходимо добавить функциональность постобработки для пользовательского `LifetimeManager`, мы можем заставить `LifetimeManager` реализовать `IDisposable`. Это приведет к тому, что пользовательский `LifetimeManager` будет уничтожен при уничтожении контейнера, являющегося владельцем этого `LifetimeManager`.

Однако могут возникать и другие сюрпризы, но не будем заострять на них внимание. В листинге 14-3 вы уже видели, что `CacheLifetimeManager` реализует `IDisposable`, но в следующем листинге вы впервые увидите саму реализацию.

Листинг 14-4: Уничтожение `LifetimeManager`

```
1. public void Dispose()
2. {
3.     GC.SuppressFinalize(this);
4.     this.Dispose(true);
5. }
6. protected virtual void Dispose(bool disposing)
7. {
8.     if (disposing)
9.     {
10.         var d = this.value as IDisposable;
11.         if (d != null)
12.         {
13.             d.Dispose();
14.         }
15.         this.value = null;
16.     }
17. }
```

Строка 10-15: Уничтожение устранимого объекта

Класс `CacheLifetimeManager` реализует `IDisposable`, руководствуясь при этом стандартным паттерном `Dispose`. Если полученное значение реализует `IDisposable`, вы его уничтожаете. Но в любом случае вы устанавливаете в поле `value` значение `null`, чтобы сборщик мусора мог уничтожить этот компонент.

Согласно теории, если вы регистрируете и разрешаете устранимый компонент с помощью `CacheLifetimeManager`, компонент должен быть уничтожен вместе с контейнером следующим образом:

```
var lease = new SlidingLease(TimeSpan.FromMinutes(1));
var cache = new CacheLifetimeManager(lease);
container.RegisterType<IIngredient, Parsley>(cache);
var ingredient = container.Resolve<IIngredient>();
container.Dispose();
```

Согласно документации, которая прилагается к контейнеру Unity, при устранении контейнера также выполняется высвобождение `ingredient`. Как известно, петрушку

нельзя разогревать, поэтому класс `Parsley`, очевидно, является устранимым. При уничтожении контейнера уничтожается и экземпляр `Parsley`. Пока все идет так, как надо.

Тем не менее, при создании и уничтожении дочернего контейнера вы будете рассчитывать на то, что устранимый `LifetimeManager` будет работать так же, как и `HierarchicalLifetimeManager`:

```
1. IIngredient ingredient;  
2. using (var child = container.CreateChildContainer())  
3. {  
4.     ingredient = child.Resolve<IIngredient>()  
5. }
```

Строка 4: `Ingredient` не уничтожается

При наличии конфигурации компонента `Parsley`, аналогичной той, что продемонстрирована в предыдущем примере, вы рассчитываете на то, что `ingredient` будет устранен при уничтожении дочернего контейнера. Увы, этого не происходит. `CacheLifetimeManager.Dispose` никогда не вызывается.

Предупреждение

Даже когда `LifetimeManager` реализует `IDisposable`, метод `Dispose` вызывается только при уничтожении контейнера-владельца.

Как такое может происходить, если в аналогичном коде, который вы уже видели в разделе 14.2.1 "Конфигурирование стиля существования", использовался `HierarchicalLifetimeManager`? Оказывается, контейнер Unity обладает соответствующим `BuilderStrategy`, который содержит особую логику `HierarchicalLifetimeManager`, позволяющую использовать эту функциональность. Хорошая новость – мы можем сделать то же самое.

Реализация пользовательской `LifetimeStrategy`

Причиной того, что `HierarchicalLifetimeManager` работает должным образом, является тот факт, что Unity обладает `BuilderStrategy`, который создает копию `HierarchicalLifetimeManager`, содержащейся в родительском контейнере, и связывает ее с дочерним контейнером. Это позволяет дочернему контейнеру уничтожать `LifetimeManager`, когда он сам устраняется. То же самое мы можем сделать, реализовав пользовательский `BuilderStrategy`, продемонстрированный в следующем листинге.

Листинг 14-5: Реализация пользовательского `LifetimeStrategy`

```
1. public class CacheLifetimeStrategy : BuilderStrategy  
2. {  
3.     public override void PreBuildUp(  
4.         IBuilderContext context)  
5.     {  
6.         if (context == null)  
7.         {  
8.             throw new ArgumentNullException("context");  
9.         }  
10.        IPolicyList policySource;  
11.        var lifetimePolicy = context  
12.            .PersistentPolicies
```

```

13.         .Get<ILifetimePolicy>(context.BuildKey,
14.             out policySource);
15.         if (object.ReferenceEquals(policySource,
16.             context.PersistentPolicies))
17.         {
18.             return;
19.         }
20.         var cacheLifetime =
21.             lifetimePolicy as CacheLifetimeManager;
22.         if (cacheLifetime == null)
23.         {
24.             return;
25.         }
26.         var childLifetime = cacheLifetime.Clone();
27.         context
28.             .PersistentPolicies
29.             .Set<ILifetimePolicy>(childLifetime,
30.                 context.BuildKey);
31.         context.Lifetime.Add(childLifetime);
32.     }
33. }

```

Строка 3-4: Переопределение `PreBuildUp`

Строка 10-14: Получение стратегии жизненного цикла

Строка 15-19: Проверка принадлежности

Строка 20-25: Проверка типа

Строка 26: Создание копии

Строка 27-31: Смена стратегии жизненного цикла

`CacheLifetimeStrategy` наследуется от абстрактного класса `BuilderStrategy` и реализует метод `PreBuildUp`, который вызывается всякий раз, когда контейнер Unity создает новый экземпляр. Это дает вам возможность изменить контекст до создания объекта.

Первое, что вам необходимо сделать, – получить текущий `ILifetimePolicy` для компонента. Контекст может предоставить эту информацию, а также информацию об источнике стратегии. Экземпляр `policySource` косвенным образом сообщает вам о том, где определен стиль существования. Если источником является родительский контейнер, но в настоящий момент вы выполняете настройку в рамках дочернего контейнера, то предполагается, что источник стиля существования будет отличаться от текущего контекста, если стиль существования первоначально был определен в родительском контейнере. Это то, что нам надо, поэтому вы преждевременно выходите из метода.

Эта конкретная реализация затрагивает только `CacheLifetimeManager`, поэтому если в качестве стиля существования используется какой-то другой стиль, вы также выходите из метода. С другой стороны, если в качестве стиля существования используется стиль `CacheLifetimeManager`, то вы создадите его копию, которая будет использоваться в дочернем контейнере.

Если вы точно уверены в том, что на данный момент выполняете настройку в рамках дочернего контейнера, то добавляете копию `CacheLifetimeManager` обратно в контекст, фактически, заменяя унаследованный из родительского контейнера стиль существования на стиль, необходимый для этого конкретного дочернего контейнера.

Это достаточно сложно, и потом, вы еще даже не все до конца выполнили. Несмотря на то, что вы реализовали пользовательский `BuilderStrategy`, вы еще не сообщили об этом `Unity`. К счастью, как демонстрирует следующий листинг, это намного проще, чем реализация `CacheLifetimeStrategy`.

Листинг 14-6: Расширение `Unity` с помощью `CacheLifetimeStrategy`

```
public class CacheLifetimeStrategyExtension : UnityContainerExtension
{
    protected override void Initialize()
    {
        this.Context.Strategies
            .AddNew<CacheLifetimeStrategy>(
                UnityBuildStage.Lifetime);
    }
}
```

Чтобы добавить `CacheLifetimeStrategy` в контейнер `Unity`, вы создаете новое расширение контейнера. Помните, как вы использовали расширения контейнера для пакетирования конфигурации в разделе 14.1.3 "Пакетирование конфигурации"? Здесь же представлен еще один пример, возможно, более идиоматического применения расширения контейнера.

В методе `Initialize` вы добавляете контекст `CacheLifetimeStrategy`, наряду с информацией о том, что этот конкретный `BuilderStrategy` предназначен для управления жизненным циклом.

Наконец, выполнив все это, вы можете расширить контейнер `Unity` таким образом, что `CacheLifetimeManager` теперь функционирует точно так же, как и `HierarchicalLifetimeManager`:

```
container.AddNewExtension<CacheLifetimeStrategyExtension>();
```

После добавления этого расширения контейнера сценарий, который ранее не работал, в конце концов, становится работоспособным: для высвобождения объектов со стилем `CacheLifetimeManager` вы можете воспользоваться дочерними контейнерами. Теперь, когда вы изучили `BuilderStrategy`, мы можем закончить цикл обучения и реализовать поддержку метода `Teardown`.

Реализация поддержки метода `Teardown`

Когда мы в разделе 14.2.1 "Конфигурирование стиля существования" приступали к обсуждению процесса высвобождения компонентов, мы сразу же опустили метод `Teardown`, поскольку он не высвобождал компоненты должным образом. С другой стороны, это не должно было привести вас к мысли о том, что метод `Teardown` бесполезен. Напротив, он вызывает различные методы зарегистрированных `BuilderStrategy`. Это означает, что мы можем реализовать пользовательский `BuilderStrategy`, который должным образом высвобождает компоненты в рамках `Teardown`.

Было бы хорошо, если бы `TearDown` поддерживал `CacheLifetimeManager`. К счастью, несмотря на то, что поддержка `CacheLifetimeManager` подразумевает создание еще одного `BuilderStrategy` (или расширение уже созданного нами `BuilderStrategy`), листинг 14-7 демонстрирует, что это намного проще, чем реализация `CacheLifetimeStrategy`, приведенного в листинге 14-5.

Листинг 14-7: Реализация стратегии высвобождения

```
1. public class CacheReleasingLifetimeStrategy : BuilderStrategy
2. {
3.     public override void PostTearDown(
4.         IBuilderContext context)
5.     {
6.         if (context == null)
7.         {
8.             throw new ArgumentNullException("context");
9.         }
10.        var lifetimes = context
11.            .Lifetime.OfType<CacheLifetimeManager>();
12.        foreach (var lifetimePolicy in lifetimes)
13.        {
14.            lifetimePolicy.RemoveValue();
15.        }
16.    }
17. }
```

Строка 3-4: Реализация `PostTearDown`

Строка 6-9: Граничный оператор

Строка 10-15: Высвобождение значения

Вместо того, чтобы переопределять метод `PreBuildUp`, как вы поступали в листинге 14-5, вы переопределяете метод `PostTearDown`, который вызывается из метода `TearDown` после того, как списывается большинство остальных ресурсов рассматриваемого компонента.

Контекст имеет целый список объектов, определяющих стили существования, но, предполагается, что в большинстве случаев контекст содержит только один единственный экземпляр `CacheLifetimeManager`. Все-таки, для чистоты эксперимента, вы делаете вид будто в контексте может присутствовать любое количество таких объектов, и вызываете метод `RemoveValue` для каждого из них. Как вы можете помнить из реализации `CacheLifetimeManager`, приведенной в листинге 14-3, метод `RemoveValue` удаляет и устраняет соответствующее значение только по истечении срока аренды.

И вот вы уже близки к завершению цикла. Можете добавить `CacheReleasingLifetimeStrategy` к `CacheLifetimeStrategyExtension`, приведенному в листинге 14-6:

```
this.Context.Strategies
    .AddNew<CacheLifetimeStrategy>(
        UnityBuildStage.Lifetime);
this.Context.Strategies
    .AddNew<CacheReleasingLifetimeStrategy>(
        UnityBuildStage.Lifetime);
```

В конце концов, это позволяет вам высвободить кэшированные компоненты с помощью метода `Teardown`:

```
container.AddNewExtension<CacheLifetimeStrategyExtension>();  
var lease = new SlidingLease(TimeSpan.FromTicks(1));  
var cache = new CacheLifetimeManager(lease);  
container.RegisterType<IIngredient, Parsley>(cache);  
var ingredient = container.Resolve<IIngredient>();  
container.Teardown(ingredient);
```

После создания такой инфраструктуры переменная `ingredient`, которая на самом деле является экземпляром устранимого класса `Parsley`, должным образом высвобождается при выводе метода `Teardown`. По истечении срока аренды экземпляр уничтожается, тем не менее, если срок аренды не истек, то ничего не происходит.

После того как мы добавили все эти `LifetimeManager`, `BuilderStrategy` и расширения контейнера `Unity` наконец-то начинает функционировать так, как нам и нужно, то есть, та его часть, которая относится к стилю существования `cache`. Вспомните, что стили `Transient` и `Per Graph` все равно не ведут себя так, как нам бы того хотелось.

Подсказка

Несмотря на то, что ни `TransientLifetimeManager`, ни `PerResolveLifetimeManager` не реализуют `IDisposable` и в их методах `RemoveValue` не исполняется никакой логики, они, по крайней мере, не изолированы. Если мы хотим, чтобы они должным образом высвобождали компоненты, то можем выполнить наследование от этих классов, а затем не забыть реализовать соответствующие `BuilderStrategy`.

В заключении, мы должны признать, что хотя `Unity` и обладает множеством `Seam'ов`, необходимых для желаемой нами реализации пользовательских стилей существования, сделать это очень сложно. С другой стороны, это хотя бы возможно, чего мы не можем сказать о некоторых других DI-контейнерах.

Можно было бы упаковать такой пользовательский стиль существования в повторно используемую библиотеку затем, чтобы нам, по крайней мере, не пришлось повторно реализовывать этот стиль существования для каждого нового приложения, которое мы планируем создать. Это тоже нужно, поскольку модель управления жизненным циклом, используемая контейнером `Unity`, не позволяет реализовать все наши потребности, несмотря на очевидную исчерпывающую поддержку им различных стилей существования.

На этом наш обзор механизма управления жизненным циклом контейнера `Unity` подошел к концу. По сравнению с остальными разделами, посвященными конкретным DI-контейнерам, этот раздел довольно объемный. Частично это зависит от множества различных возможностей, доступных нам при реализации пользовательских стилей существования. С другой стороны, такой объем объясняется некоторыми уникальными ловушками, связанными с механизмом управления жизненным циклом, которые мне хотелось бы рассмотреть. Конфигурировать компоненты можно и посредством сочетания различных стилей существования. Это справедливо и при регистрации составных реализаций одной и той же абстракции. Мы уже рассматривали процесс работы с составными компонентами, но в следующем разделе этот вопрос обсуждается более углубленно. `Unity` позволяет нам более подробно изучить этот процесс, поскольку он

поддерживает механизм перехвата. Следующий раздел можно рассматривать как расширение обсуждения Decorator'ов.

14.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями
- Используются Decorator'ы

В этом разделе мы рассмотрим каждую из этих ситуаций и увидим, как Unity поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь регистрировать и разрешать компоненты даже тогда, когда в дело вступают составные реализации одной и той же абстракции.

Рассмотрим сначала способы предоставления более тщательного контроля, нежели тот, который предоставляет механизм автоматической интеграции.

Выбор среди составных кандидатов

Автоматическая интеграция – удобный и мощный инструмент, но предоставляет нам меньшие возможности контроля. Пока все абстракции преобразуются в конкретные типы отдельно друг от друга, никаких трудностей не возникает, но как только мы вводим большее количество реализаций для одного и того же интерфейса, возникает неопределенность.

Способ работы контейнера Unity с составными регистрациями одной и той же абстракции слегка отличается от того, который используется другими DI-контейнерами. С рассмотрения этого вопроса мы и начнем изучение раздела, поскольку с помощью него устанавливаются некоторые фундаментальные правила, которые понадобятся нам в остальной части раздела.

Регистрация составных реализаций одного и того же компонента

Как вы уже видели в разделе 14.1.2 "Конфигурирование контейнера", вы можете регистрировать составные компоненты одного и того же сервиса:

```
container.RegisterType<IIngredient, Steak>();  
container.RegisterType<IIngredient, SauceBéarnaise>("sauce");
```

В этом коде может присутствовать только одна неименованная регистрация. Она называется *регистрацией по умолчанию*. Если вы впоследствии вызовете для `IIngredient`

метод `RegisterType` без имени, то регистрация `Steak` будет заменена новым компонентом.

Примечание

Регистрация по умолчанию для типа может быть только одна, а именованных регистраций может быть сколько угодно.

При вызове метода `Resolve` без имени мы получаем объект на основании регистрации по умолчанию. Благодаря предыдущей конфигурации метод `Resolve` возвращает экземпляр `Steak`:

```
var ingredient = container.Resolve<IIngredient>();
```

Об именованной регистрации `sauce` мы не забыли. Разрешать составные `IIngredient` можно следующим образом:

```
IEnumerable<IIngredient> ingredients =  
    container.ResolveAll<IIngredient>();
```

В соответствии с конфигурацией, приведенной в предыдущем примере, вы получаете последовательность, содержащую не экземпляр `Steak`, а экземпляр `SauceBéarnaise`.

Предупреждение

Метод `ResolveAll` возвращает все именованные регистрации, но не регистрацию по умолчанию.

Если существуют сконфигурированные экземпляры плагина, которые не могут быть разрешены при вызове метода `ResolveAll`, Unity выдает исключение, поясняющее, что существуют зависимости, не удовлетворяющие условиям. Такое поведение совместимо с поведением метода `Resolve`, но отлично от того, как поступают `Castle Windsor` или `MEF`.

В следующем листинге демонстрируется, как можно использовать именованные регистрации для предоставления подсказок, которые позднее можно применять при осуществлении выбора между различными сконфигурированными компонентами.

Листинг 14-8: Присваивание имен регистрациям

```
container.RegisterType<IIngredient, Steak>("meat");  
container.RegisterType<IIngredient, SauceBéarnaise>("sauce");
```

Каждой регистрации можно присвоить свое уникальное имя, которое в дальнейшем можно применять для различения похожих компонентов.

Примечание

Существует возможность регистрировать для типа только именованные компоненты. Если мы так сделаем, то у типа не будет регистрации по умолчанию.

Благодаря именованным регистрациям, приведенным в листинге 14-8, вы можете разрешить и `Steak`, и `SauceBéarnaise` следующим образом:

```
var meat = container.Resolve<IIIngredient>("meat");
var sauce = container.Resolve<IIIngredient>("sauce");
```

Обратите внимание на то, что вы передаете тот же самый идентификатор, который использовали для именования компонента во время регистрации.

Принимая во внимание тот факт, что мы всегда должны разрешать сервисы в единственном Composition Root, мы, скорее всего, не должны ждать появления такой неопределенности на этом уровне.

Подсказка

Если вы обнаружите, что вызываете метод `Resolve` с конкретным именем или идентификатором, подумайте над тем, сможете ли вы сменить свой подход на менее неопределенный.

Именованные регистрации можно использовать для осуществления выбора среди нескольких экземпляров при конфигурировании зависимостей данного сервиса.

Конфигурирование именованных зависимостей

Иногда бывает необходимо переопределить обычное поведение для того, чтобы обеспечить более разветвленный контроль над тем, куда какая зависимость отправляется. Кроме того, возможны ситуации, при которых приходится сталкиваться с неопределенным API. В качестве примера рассмотрим следующий конструктор:

```
public ThreeCourseMeal(ICourse entrée,
    ICourse mainCourse, ICourse dessert)
```

В этом примере присутствуют три одинаковым образом типизированных зависимости, каждая из которых является отдельной, не похожей на другие сущностью. В большинстве случаев необходимо преобразовывать каждую из этих зависимостей в отдельный тип. В следующем листинге продемонстрированы способы конфигурации преобразований `ICourse`.

Листинг 14-9: Регистрация именованных `course`

```
container.RegisterType<ICourse, Rillettes>("entrée");
container.RegisterType<ICourse, CordonBleu>("mainCourse");
container.RegisterType<ICourse, MousseAuChocolat>("dessert");
```

Как вы уже делали это в листинге 14-8, вы регистрируете три именованных компонента, преобразуя `Rillettes` в экземпляр под названием `"entrée"`, `CordonBleu` – в экземпляр с именем `"mainCourse"`, а `MousseAuChocolat` – в экземпляр под названием `"dessert"`.

Принимая во внимание эти регистрации, теперь вы можете зарегистрировать класс `ThreeCourseMeal` так как это продемонстрировано в следующем листинге.

Листинг 14-10: Переопределение механизма автоматической интеграции

```
container.RegisterType<IMeal, ThreeCourseMeal>(
    new InjectionConstructor(
        new ResolvedParameter<ICourse>("entrée"),
        new ResolvedParameter<ICourse>("mainCourse"),
        new ResolvedParameter<ICourse>("dessert")));
```

До этого момента мы еще не обсуждали подробно тот факт, что все перегрузки метода `RegisterType` принимают в качестве параметра массив объектов класса `InjectionMember`. `InjectionMember` – это стратегия, которую Unity использует в качестве инструкции при компоновке типов друг с другом. Например, `InjectionConstructor` позволяет определять параметры, используемые для паттерна `Constructor Injection`.

Один из способов переопределения механизма автоматической интеграции – задание его с помощью массива экземпляров `ResolvedParameter`. Каждый `ResolvedParameter` определяет тип, который необходимо разрешить, а также необязательное имя – это имя именованной регистрации, а не имя аргумента конструктора. `ResolvedParameter` *entrée* обозначает регистрацию *entrée*. Параметры конструктора заполняются позиционным способом, т.е. первый `ResolvedParameter` соответствует первому аргументу конструктора и т.д.

Примечание

В отличие от большинства остальных DI-контейнеров, позиционная стратегия позволяет контейнеру Unity оставаться стабильным при столкновении с такими формами рефакторинга, как изменение имени аргументов конструктора. С другой стороны, если мы переставим местами аргументы конструктора, то все рухнет. Другие DI-контейнеры могут с этим справиться.

Переопределение автоматической интеграции посредством явного преобразования параметров в именованные компоненты – общепринятый подход. Делать это можно даже, если именованные компоненты конфигурируются в одном расширении контейнера, а потребитель – в совершенно другом расширении, поскольку единственная идентификация, связывающая именованный компонент с параметром – это имя. Это возможно, но если нам приходится управлять множеством имен, такой подход является довольно хрупким. Когда основной причиной, побуждающей нас использовать именованные компоненты, является неопределенность, наилучшее решение – сконструировать свое собственное API, помогающее избавиться от этой неопределенности. Все это также приводит к лучшей конструкции в целом.

В следующем разделе вы увидите, как можно использовать менее неопределенный и более гибкий подход при котором разрешается использовать в обеде несколько блюд. Теперь вы должны изучить то, как Unity работает со списками и последовательностями.

Интеграция последовательностей

В разделе 10.3.2 "Разработка пользовательского стиля существования" мы обсуждали, как выполнить рефакторинг явного класса `ThreeCourseMeal` к более универсальному классу `Meal`, который обладает приведенным ниже конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

В этом разделе мы рассмотрим то, как можно сконфигурировать Unity, чтобы он интегрировал экземпляры `Meal` с соответствующими зависимостями `ICourse`. После рассмотрения этого вопроса вы должны будете приобрести хорошее понимание тех возможностей, которые доступны в случае необходимости конфигурирования экземпляров, имеющих последовательности зависимостей.

Автоматическая интеграция последовательностей

Unity довольно хорошо разбирается в массивах, но не в других видах последовательностей, например, `IEnumerable<T>` или `IList<T>`. Для эффективной работы с последовательностями мы должны определить или преобразовать их в массивы, чтобы Unity мог работать с ними соответствующим образом.

Если вы попытаетесь зарегистрировать `Meal`, не сообщая при этом контейнеру о том, как ему следует работать с зависимостью `IEnumerable<ICourse>`, то при попытке разрешить `IMeal` будет выдаваться исключение:

```
container.RegisterType<IMeal, Meal>();  
var meal = container.Resolve<IMeal>();
```

При разрешении `IMeal` выдается исключение, потому что Unity не знает, как нужно разрешать `IEnumerable<ICourse>`. Так произойдет, даже если вы перед этим зарегистрируете несколько компонентов `ICourse`, как вы и поступали в листинге 14-9.

Чтобы преобразовать все именованные регистрации `ICourse` в `IEnumerable<ICourse>`, можно воспользоваться преимуществами врожденного понимания контейнером Unity массивов. Самый простой способ сделать это – преобразовать два приведенных ниже типа:

```
container.RegisterType<IEnumerable<ICourse>, ICourse[]>();
```

Это может показаться слегка странным, но зато такой подход достаточно хорошо работает. Всякий раз, когда Unity сталкивается с зависимостью `IEnumerable<ICourse>`, он преобразует ее в запрос массива экземпляров `ICourse`, что даст нам такой же самый результат, как если бы мы вызвали `container.ResolveAll<ICourse>()`.

Примечание

Unity разрешает массивы подобно тому, как он возвращает результат метода `ResolveAll`. Возвращаются все запрашиваемые именованные компоненты, но не компонент по умолчанию.

После выполнения такого преобразования в результате разрешения `IMeal` возвращается корректный результат: экземпляр `Meal` с экземплярами `ICourse` из листинга 14-9: `Rillettes`, `CordonBleu` и `MousseAuChocolat`.

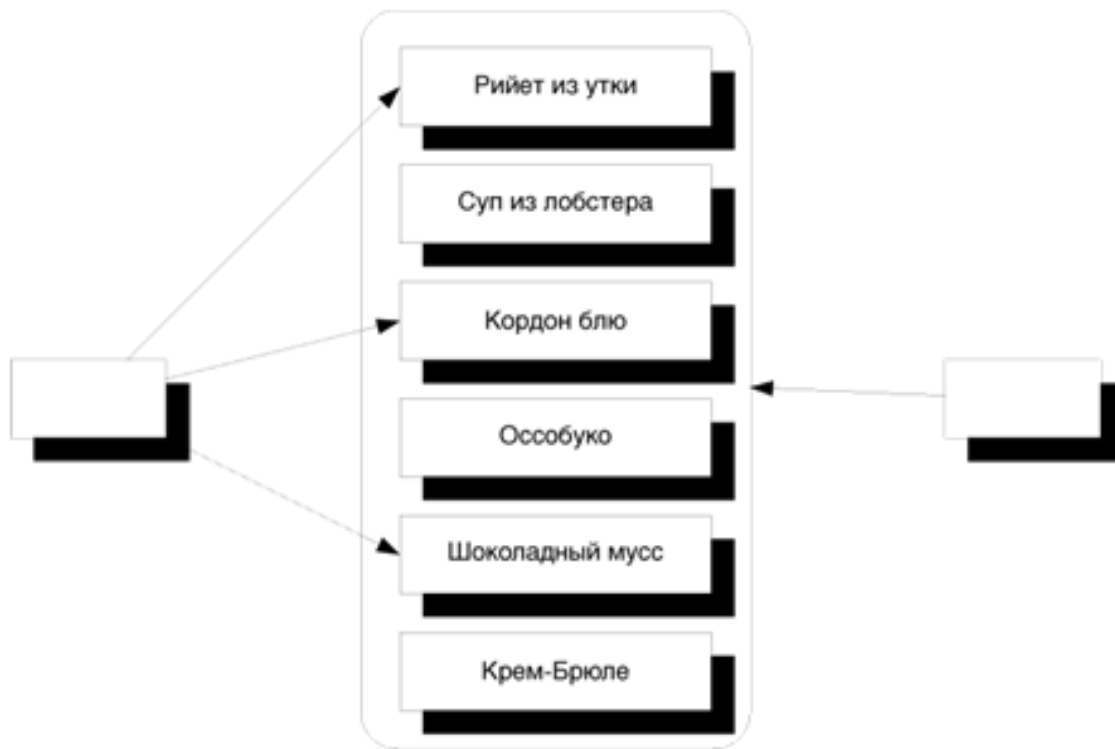
Unity управляет массивами сообразно тому, как он реализует `ResolveAll`. Но вот как работать с остальными видами последовательностей, он не знает, поэтому мы должны преобразовать их в массивы, чтобы заставить Unity их понимать. В результате мы получим все зарегистрированные компоненты данного типа, и этого чаще всего бывает достаточно. Только когда нам нужно из большого набора компонентов явно отобрать только несколько из них, нам необходимо выполнить больше действий. Это возможно благодаря более явной конфигурации.

Отбор нескольких компонентов из большого набора

Когда мы используем способность Unity разрешать массивы, все именованные компоненты внедряются в потребителей. Чаще всего это корректная линия поведения, но,

как показывает рисунок 14-7, возможны случаи, когда нам необходимо отобразить только несколько компонентов из большого набора всех зарегистрированных компонентов.

Рисунок 14-7: В ситуации, продемонстрированной слева, мы хотим явным образом отобразить определенные зависимости из большого списка всех зарегистрированных компонентов. Это отличается от ситуации, приведенной справа, когда мы отображаем все без разбора.



Когда мы ранее позволяли Unity автоматически интегрировать все сконфигурированные экземпляры, это соответствовало бы ситуации, изображенной в правой части рисунка 14-7. Если нам нужно сконфигурировать экземпляр так, как изображено в левой части рисунка, то мы должны явно определить, какие экземпляры необходимо использовать.

В листинге 14-10 вы использовали класс `InjectionConstructor` для того, чтобы определить вместо стратегии автоматической интеграции, используемой по умолчанию, другую стратегию. Когда дело касается класса `Meal`, вы можете сделать то же самое с единственной лишь разницей, что теперь вместо трех отдельных аргументов `ICourse` внедряется зависимость `IEnumerable<ICourse>`. Следующий листинг демонстрирует, как сконфигурировать явный массив с помощью `InjectionConstructor`.

Листинг 14-11: Внедрение именованных компонентов в последовательность

```
container.RegisterType<IMeal, Meal>(
    new InjectionConstructor(
        new ResolvedArrayParameter<ICourse>(
            new ResolvedParameter<ICourse>("entrée"),
            new ResolvedParameter<ICourse>("mainCourse"),
            new ResolvedParameter<ICourse>("dessert"))) );
```

Чтобы переопределить механизм автоматической интеграции и явным образом задать стратегию внедрения зависимостей в конструктор `Meal`, вы еще раз обращаетесь к классу `InjectionConstructor`. Поскольку для конструктора `Meal` нужен `IEnumerable<ICourse>`,

вы можете воспользоваться экземпляром `ResolvedArrayParameter<ICourse>` для определения массива, который будет вычисляться при разрешении контейнером класса `Meal`. Класс `ResolvedArrayParameter<ICourse>` определяет стратегию, при которой анализ массива экземпляров `ICourse` откладывается до тех пор, пока не разрешится сам `Meal`.

Чтобы определить значения, которые будут использоваться при разрешении `Meal`, можно использовать три именованных экземпляра `ResolvedParameter<ICourse>`, как вы уже делали в листинге 14-10. Единственное отличие – теперь они используются в качестве аргументов для конструктора `ResolvedArrayParameter<ICourse>` вместо того, чтобы использоваться напрямую в `InjectionConstructor`. При разрешении `IMeal` `ResolvedArrayParameter<ICourse>` разрешает три именованных регистрации *entrée*, *mainCourse* и *dessert*, а также создает массив `ICourse` из этих трех компонентов. Поскольку `ICourse[]` реализует `IEnumerable<ICourse>`, может подойти конструктор `Meal`.

В очередной раз вы видите, что контейнер Unity отлично понимает массивы. Несмотря на то, что он не поддерживает остальные виды последовательностей, это ограничение можно обойти путем преобразования зависимостей в массивы.

Потребители, которые полагаются на последовательности зависимостей, могут быть самым интуитивно понятным применением составных экземпляров одной и той же абстракции. Но перед тем как мы полностью отойдем от данной темы, нам необходимо рассмотреть последний (и, возможно, слегка неожиданный) случай, когда в дело вступают составные экземпляры.

Интеграция Decorator'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования Decorator полезен при реализации сквозных сущностей. По определению Decorator'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам Decorator и вложенный в него тип. Если бы мы помещали Decorator'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных регистраций одного и того же сервиса. В отличие от предыдущих разделов эти регистрации не являются концептуально равносильными, а зависят друг от друга. В следующем разделе я продемонстрирую вам два разных способа конфигурирования Unity для работы с данным паттерном.

Создание обертки для именованного компонента

Класс `Breading` – это обертка `IIngredient`. Этот класс использует паттерн `Constructor Injection` для получения того экземпляра, который ему необходимо обернуть:

```
public Breading(IIngredient ingredient)
```

Чтобы получить `Cotoletta`, вам необходимо будет вложить `VealCutlet` (еще один `IIngredient`) в класс `Breading`. Поскольку вы уже знаете, как соединить именованные компоненты с аргументами конструктора, было бы вполне естественным сделать что-то аналогичное в следующем листинге.

Листинг 14-12: Создание обертки с помощью именованного компонента

```
container.RegisterType<IIngredient, VealCutlet>("cutlet");
container.RegisterType<IIngredient, Breeding>(
    new InjectionConstructor(
        new ResolvedParameter<IIngredient>("cutlet")));
```

Компонент `Breeding` по умолчанию должен иметь тип `IIngredient`, поэтому вам нужно присвоить `VealCutlet` имя `IIngredient`, поскольку по умолчанию `IIngredient` должен быть только один. При регистрации компонента `Breeding` вы еще раз используете `InjectionConstructor`, чтобы указать, как контейнер `Unity` должен интегрировать аргумент конструктора `ingredient` класса `Breeding`. `ResolvedParameter<IIngredient>` позволяет вам указать, что должен разрешаться именно первый параметр конструктора (и только он) и интегрироваться с именованным компонентом `cutlet`.

При разрешении `IIngredient` вы получаете экземпляр `Breeding`, в который вложен `VealCutlet`.

Это общепринятый способ создания обертки для компонента, но в случае, когда нам не нужны вложенные компоненты, мы можем использовать более неявный метод.

Создание обертки для конкретного компонента

Если нам никогда не нужно будет напрямую разрешать вложенный компонент, мы можем использовать более неявный способ создания для него обертки. Представьте себе, что вам никогда не придется разрешать `VealCutlet` напрямую в виде `IIngredient`. В тех случаях, когда вам нужен будет `IIngredient`, вам всегда хотелось бы получать `Cotoletta`.

В таком случае вообще нет никакой необходимости конфигурировать `VealCutlet`. Вместо этого вы можете воспользоваться преимуществами того факта, что `Unity` автоматически разрешает конкретные типы, даже если они не зарегистрированы:

```
container.RegisterType<IIngredient, Breeding>(
    new InjectionConstructor(
        new ResolvedParameter<VealCutlet>()));
```

Вы уже знаете, что вам нужно внедрить в экземпляр `Breeding` именно `VealCutlet`, поэтому нет причин для неявного определения `ResolvedParameter<IIngredient>`, когда вы можете напрямую передать `ResolvedParameter<VealCutlet>`. Когда вы отправите запрос контейнеру на разрешение `IIngredient`, `ResolvedParameter<VealCutlet>` будет автоматически разрешен в экземпляр `VealCutlet`, поскольку это конкретный класс. В связи с тем, что `VealCutlet` реализует `IIngredient`, он нам подходит.

Несмотря на то, что вы не регистрировали компонент `VealCutlet`, вы все еще можете это сделать, если вам необходимо сконфигурировать другие аспекты, например, его стиль существования:

```
container.RegisterType<VealCutlet>(
    new ContainerControlledLifetimeManager());
container.RegisterType<IIngredient, Breeding>(
    new InjectionConstructor(
        new ResolvedParameter<VealCutlet>()));
```

В этом примере вы конфигурируете конкретный `VealCutlet` в виде `Singleton`, но, поскольку вы не планируете разрешать его в виде `IIngredient`, вы не преобразуете его в интерфейс. Все это превращает его в `VealCutlet` по умолчанию, а затем `ResolvedParameter<VealCutlet>` сможет должным образом его разрешить.

Как вы уже видели в этом разделе, при конфигурировании Decorator'ов существует несколько возможных вариантов. Во всех этих вариантах используется класс `InjectionConstructor`. В отличие от Castle Windsor контейнер Unity не поддерживает Decorator'ы явным образом, что может быть немного удивительным, поскольку подобно Windsor контейнер Unity максимально поддерживает паттерн Decorator: в виде механизма перехвата.

Создание перехватчиков

В разделе 9.3.3 "Пример: перехват с помощью Windsor" вы видели пример того, как добавить в WCF-приложение обработчик ошибок и `Circuit Breaker` с помощью возможности динамического перехвата, предлагаемой Castle Windsor. В этом разделе мы сделаем то же самое с помощью Unity.

Как показано на рисунке 14-8, добавление аспекта в Unity – процесс, включающий в себя несколько шагов.

Рисунок 14-8: Шаги, которые включает в себя процесс добавления аспекта в Unity



Основная часть работы заключается в разработке самого перехватчика, но после его создания мы должны добавить этот перехватчик в контейнер. Механизм перехвата – это расширение контейнера Unity, поэтому мы должны добавить это расширение и в контейнер, чтобы все это работало.

В этом разделе мы будем сначала создавать перехватчики для обработчика ошибок и `Circuit Breaker`, а в конце сконфигурируем контейнер с помощью этих перехватчиков.

Реализация перехватчика обработчика исключений

Реализация перехватчика для контейнера Unity требует от нас реализации интерфейса `IInterceptionBehavior`. Листинг 14-13 демонстрирует, как реализовать стратегию обработки исключений, приведенную в главе 9. Эта конкретная реализация стратегии для контейнера Unity соответствует листингу 9-8, который приводился при описании Castle Windsor, и листингу 12-4, который приводился при описании Spring.NET.

Листинг 14-13: Реализация интерфейса `IInterceptionBehavior`, предназначенного для обработки исключений.

```
1. public class ErrorHandlerInterceptionBehavior :
2.     IInterceptionBehavior
3. {
4.     public IEnumerable<Type> GetRequiredInterfaces ()
5.     {
6.         return Type.EmptyTypes;
7.     }
8.     public bool WillExecute
9.     {
10.         get { return true; }
11.     }
12.     public IMethodReturn Invoke (
13.         IMethodInvocation input,
14.         GetNextInterceptionBehaviorDelegate getNext)
15.     {
16.         var result = getNext()(input, getNext);
17.         if (result.Exception is CommunicationException
18.             || result.Exception is
19.                 InvalidOperationException)
20.         {
21.             this.AlertUser(result.Exception.Message);
22.             return input.CreateMethodReturn(null);
23.         }
24.         return result;
25.     }
26.     private void AlertUser(string message)
27.     {
28.         var sb = new StringBuilder();
29.         sb.AppendLine("An error occurred.");
30.         sb.AppendLine("Your work is likely lost.");
31.         sb.AppendLine("Please try again later.");
32.         sb.AppendLine();
33.         sb.AppendLine(message);
34.         MessageBox.Show (sb.ToString(), "Error",
35.             MessageBoxButton.OK,
36.             MessageBoxImage.Error);
37.     }
38. }
```

Строка 4-7: Передача интерфейсов

Строка 8-11: Подключение перехватчика

Строка 12-14: Реализация логики перехвата

Строка 16: Получение результата из вложенного объекта

Строка 17-22: Обработка исключений

Класс `ErrorHandlerInterceptionBehavior` реализует `IInterceptionBehavior`, который является интерфейсом с тремя членами. Двое из этих членов, в основном, относятся к инфраструктуре Unity, и реализовать их довольно легко. Метод `GetRequiredInterfaces` позволяет нам указать, к каким интерфейсам обращается этот перехватчик, но возвращая пустой массив, вы можете отложить принятие решения до того момента, когда вы будете конфигурировать, какие компоненты собираетесь перехватывать. Свойство `WillExecute` должно возвращать `true`, если вы хотите, чтобы перехватчик работал. Это дает нам

возможность сконфигурировать, должен ли выполняться конкретный перехватчик. Но в этом примере вы хотите всегда выполнять `ErrorHandlingInterceptionBehavior`, если он сконфигурирован для компонента.

Основная реализация `IInterceptionBehavior` выполняется в методе `Invoke`, который вызывается (так!) контейнером Unity при вызове перехватываемого компонента. Параметр `input` содержит некоторую информацию о текущем вызове метода, тогда как параметр `getNext` содержит делегат, который можно использовать для вызова вложенного компонента. Это приблизительно соответствует методу `Proceed`, который используется в Castle Windsor и продемонстрирован в листинге 9-8.

Вызов метода `getNext` дает вам метод `Invoke`, представляющий собой метод, который вложен в перехватчика. Это может быть другой перехватчик или сам вложенный компонент. Вызывая этот метод `Invoke` с первоначальными параметрами, вы получаете результат вложенного метода.

Если метод возвращает одно из тех исключений, которые необходимо обработать, то вы предупреждаете об этом пользователя путем вывода на экран диалогового окна. В случае получения обрабатываемого исключения вам хотелось бы пресечь его сразу же после того, как вы предупредите пользователя. Если бы вложенный метод возвращал `result`, то это позволило бы исключению распространиться, поэтому вместо этого вы создаете новое возвращаемое значение и возвращаете его. В противном случае при отсутствии исключений вы вернули бы `result` из вложенного метода.

`ErrorHandlingInterceptionBehavior` обрабатывает определенные исключения вложенного компонента. В роли этого компонента может выступать перехватчик в форме `Circuit Breaker`.

Реализация `CircuitBreaker` перехватчика

`Circuit Breaker` перехватчик – слегка более сложный перехватчик, поскольку для него необходима зависимость `ICircuitBreaker`. Но как демонстрирует следующий листинг, мы решаем эту проблему путем применения стандартного паттерна `Constructor Injection`. Когда дело доходит до компоновки класса, контейнер Unity выполняет ее так же, как и любой другой компонент: пока он может разрешать зависимость, все идет нормально.

Листинг 14-14: Реализация `IInterceptionBehavior` в виде `Circuit Breaker`

```
1. public class CircuitBreakerInteceptionBehavior :
2.     IInterceptionBehavior
3.     {
4.         private readonly ICircuitBreaker breaker;
5.         public CircuitBreakerInteceptionBehavior(
6.             ICircuitBreaker breaker)
7.         {
8.             if (breaker == null)
9.             {
10.                 throw new ArgumentNullException("breaker");
11.             }
12.             this.breaker = breaker;
13.         }
14.         public IMethodReturn Invoke(IMethodInvocation input,
15.             GetNextInterceptionBehaviorDelegate getNext)
16.         {
17.             try
```

```

18.      {
19.          this.breaker.Guard();
20.      }
21.      catch (InvalidOperationException e)
22.      {
23.          return
24.              input.CreateExceptionMethodReturn(e);
25.      }
26.      var result = getNext()(input, getNext);
27.      if (result.Exception != null)
28.      {
29.          this.breaker.Trip(result.Exception);
30.      }
31.      else
32.      {
33.          this.breaker.Succeed();
34.      }
35.      return result;
36.  }
37.  public IEnumerable<Type> GetRequiredInterfaces()
38.  {
39.      return Type.EmptyTypes;
40.  }
41.  public bool WillExecute
42.  {
43.      get { return true; }
44.  }
45.  }

```

Строка 17-25: Реализация граничного оператора

Строка 26: Возвращение `result` из вложенного метода

Строка 27-30: Обработка исключения

Строка 31-34: Обозначение успешного выполнения

Строка 37-44: Необходимые данные для `IInterceptionBehavior`

`CircuitBreakerInteceptionBehavior` должен делегировать свою реализацию экземпляру `ICircuitBreaker`. Поскольку `Unity`, как и любой другой компонент, будет автоматически интегрировать перехватчик, для внедрения `ICircuitBreaker` можно воспользоваться стандартным паттерном `Constructor Injection`.

В методе `Invoke` вам необходимо реализовать идиоматическое выражение *Guard-Succeed/Trip*, которое вы уже видели в листингах 9-4 и 9-9. Для начала вам необходимо вызвать метод `Guard`, и вернуть исключение, если этот метод возвращает исключение. `Unity` предполагает, что вы будете передавать исключения не посредством вывода их, а путем инкапсуляции их в экземпляры `IMethodReturn`. Поэтому вы должны явным образом перехватывать `InvalidOperationException` и из перехваченного исключения создавать возвращаемое значение.

Если метод `Guard` уже выполнен, то вы можете приступить к выполнению вложенного метода. Делается это точно так же, как вы делали это в листинге 14-13. После получения `result` из вложенного метода можно проанализировать это значение, чтобы определить, не было ли возвращено исключение. Если это так, то с помощью метода `Trip` вы

устанавливаете прерыватель, но, обратите внимание на то, что при этом вы не изменяете `result`. Не забудьте о том, что после вызова метода `Trip` все равно необходимо выдавать исключение, поэтому `result` вы можете оставить неизменным — он уже инкапсулирует исключение.

При отсутствии исключений вы можете аналогичным образом указать на это прерывателю, вызвав метод `Succeed`. Не забудьте о том, что это может привести к закрытию другого открытого прерывателя.

После реализации `ErrorHandlingInterceptionBehavior` и `CircuitBreakerInteceptionBehavior` настает время для того, чтобы сконфигурировать контейнер таким образом, чтобы `ErrorHandlingInterceptionBehavior` и `CircuitBreakerInteceptionBehavior` стали обертками для `IProductManagementAgent`.

Конфигурирование механизма перехвата

Все, что нам необходимо сделать, — перехватить компонент `IProductManagementAgent` с помощью `Circuit Breaker` и механизма обработки ошибок таким образом, чтобы при возникновении исключений во время взаимодействия с веб-сервисом открывался `Circuit Breaker` и обрабатывалось исключение. Это дает приложению возможность восстанавливаться при каждом последующем подключении веб-сервиса или сети.

Сначала нам необходимо добавить в контейнер `Unity` возможность поддержки механизма перехвата. Несмотря на то, что механизм перехвата является частью модуля приложений контейнера `Unity`, реализуется он в отдельной сборке и, поэтому его необходимо добавлять явным образом.

Примечание

Чтобы иметь возможность использовать механизм перехвата в рамках контейнера `Unity`, необходимо добавить ссылку на сборку

```
Microsoft.Practices.Unity.InterceptionExtension.
```

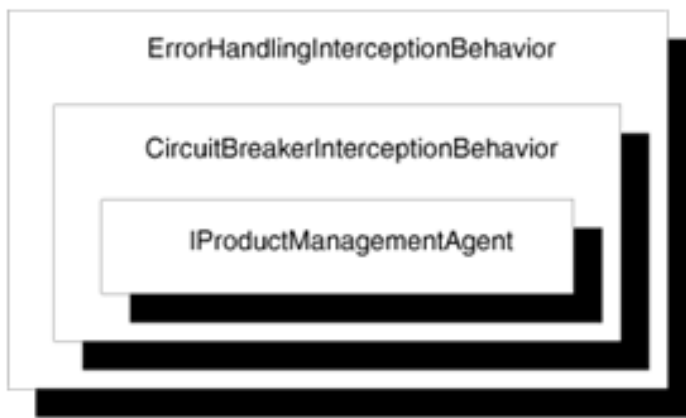
После добавления ссылки на сборку

`Microsoft.Practices.Unity.InterceptionExtension` необходимо добавить в контейнер расширение `Interception`. Это всего лишь еще одно расширение контейнера, поэтому добавить его можно с помощью метода `AddNewExtension`:

```
container.AddNewExtension<Interception>();
```

Несмотря на то, что приведенный код в принципе позволяет добавить в контейнер возможность использования механизма перехвата, нам все равно нужно добавить в компонент `IProductManagementAgent` необходимые перехватчики. На рисунке 14-9 продемонстрирована та конфигурация, к которой мы стремимся.

Рисунок 14-9: `IProductManagementAgent` должен быть вложен в `Circuit Breaker` перехватчик таким образом, чтобы при выдаче агентом исключения на некоторое время открывался цикл. `Circuit Breaker` не управляет исключениями, а всего лишь регистрирует их. Поэтому за управление исключениями отвечает перехватчик обработки ошибок, который должен располагаться как можно дальше для того, чтобы иметь возможность управлять исключениями, получаемыми как от агента, так и от `Circuit Breaker`.



При конфигурировании механизма перехвата для компонента используется тот факт, что все перегрузки метода `RegisterType` принимают в качестве параметра массив объектов `InjectionMember`. До настоящего момента вы использовали только класс `InjectionConstructor`, но для конфигурирования механизма перехвата можно использовать совокупность и других классов:

```
container.RegisterType<IProductManagementAgent,
    ➔ WcfProductManagementAgent>(
    new Interceptor<InterfaceInterceptor>(),
    new InterceptionBehavior<ErrorHandlingInterceptionBehavior>(),
    new InterceptionBehavior<CircuitBreakerInteceptionBehavior>());
```

Разобьем этот код на составляющие части. Сначала вы добавляете `Interceptor<InterfaceInterceptor>`. Это `InjectionMember`, который сообщает Unity о том, что далее следуют один или несколько `InterceptionBehavior`, которые выполняют перехват на уровне инетрфейса (в противоположность, например, перехвату виртуальных членов).

Следующие два `InjectionMember` добавляют реализованные вами перехватчики `ErrorHandlingInterceptionBehavior` и `CircuitBreakerInteceptionBehavior`. Обратите внимание, что, поскольку вы сначала указываете `ErrorHandlingInterceptionBehavior`, он становится самым дальним перехватчиком и, в свою очередь, перехватывает `CircuitBreakerInteceptionBehavior`.

В конце этого примера кода вам необходимо убедиться, что выполнены требования всех зависимостей. Поскольку для `CircuitBreakerInteceptionBehavior` необходим `ICircuitBreaker`, вы должны зарегистрировать этот компонент:

```
container.RegisterType<ICircuitBreaker, CircuitBreaker>(
    new ContainerControlledLifetimeManager(),
    new InjectionConstructor(TimeSpan.FromMinutes(1)));
```

Для обеспечения эффективности важно, чтобы в коде присутствовал только один экземпляр `Circuit Breaker` (хотя бы по одному на каждый внешний ресурс), поэтому мы регистрируем его как `Singleton`. Кроме того, для конструктора `CircuitBreaker` мы задаем одноминутную задержку, чтобы гарантировать, что приложению разрешается восстанавливать прерванное подключение один раз в минуту.

Этот раздел продемонстрировал, как можно применять механизм динамического перехвата в рамках контейнера Unity. Лично я считаю, что сложность использования

механизма перехвата в рамках Unity сравнима с поддержкой этого механизма контейнерами Castle Windsor и Spring.NET. Несмотря на то, что это применение не является полностью тривиальным, пользу оно приносит колоссальную.

Механизм перехвата – это динамическая реализация паттерна Decorator, а паттерн Decorator сам по себе является совместным применением составных компонентов одного и того же типа. Контейнер Unity позволяет нам использовать различные подходы при работе с составными компонентами. Мы можем регистрировать компоненты в виде альтернатив друг другу, в виде пиров, которые разрешаются в виде последовательностей, в виде иерархических Decorator'ов или даже в виде перехватчиков. Что касается массивов, то тут Unity сам поймет, как ему поступать дальше, а остальные типы последовательностей мы можем преобразовывать в массивы. Это также позволяет нам явным образом определять способ компоновки сервисов в случае, когда нам нужен более явный контроль.

Кроме того, это может происходить в ситуации, когда нам нужно иметь дело с API, отклоняющимися от Constructor Injection. До настоящего момента вы наблюдали за тем, как конфигурировать экземпляры, включая то, как определять стили существования и как работать с составными компонентами. Но до этого момента мы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют Constructor Injection. Поскольку это не всегда происходит именно так, в следующем разделе мы сделаем краткий обзор того, как работать с классами, экземпляры которых должны создаваться особым образом.

14.4. Конфигурирование сложных API

До настоящего момента мы рассматривали то, как можно конфигурировать компоненты, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры, например, Unity, могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей.

Все становится менее понятным, когда API не столь хорошо функционируют. В этом разделе вы увидите, как работать с простейшими аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Начнем с рассмотрения классов, конструкторы которых принимают в качестве параметров простейшие типы, например, строки и целые числа.

Конфигурирование простейших зависимостей

Пока мы внедряем абстракции в потребителей, все в порядке. Но данный процесс усложняется, если конструктор зависит от простейшего типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве компонента контейнера не имеет особого смысла. Но в рамках Unity это, по крайней мере, осуществимо.

Рассмотрим в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются code smell'ами и их нужно преобразовывать в полиморфные классы (имеющие разное состояние). Тем не менее, для данного примера они вполне нам подходят.

Если вы хотите, чтобы все потребители `Spiciness` использовали одно и то же значение, можно зарегистрировать `Spiciness` и `ChiliConCarne` независимо друг от друга:

```
container.RegisterInstance(Spiciness.Medium);
container.RegisterType<ICourse, ChiliConCarne>();
```

Когда вы впоследствии будете разрешать `ChiliConCarne`, его `Spiciness` будет иметь значение `Medium`, как и все остальные компоненты, зависящие от `Spiciness`.

Если вы будете достаточным образом контролировать взаимосвязь `Spiciness` и `ChiliConCarne`, то сможете таким же образом поступить и с вездесущим `InjectionConstructor`, передав значение в конструктор:

```
container.RegisterType<ICourse, ChiliConCarne>(
    new InjectionConstructor(Spiciness.Hot));
```

Ранее в этой главе `InjectionConstructor` в основном использовался с `ResolvedParameter<T>`, но можно поступить и по-другому: передать значение, которое затем будет передано прямо в конструктор компонента. В приведенном примере вы передаете значение `Spiciness.Hot`, которое затем будет передано прямо в конструктор `ChiliConCarne`, возвращая значение `Hot`.

Оба описанных здесь варианта стимулируют автоматическую интеграцию на предоставление конкретного значения для компонента. Если вам нужна более строгая типизированная конфигурация, которая вызывает конструктор или статическую фабрику, вы также сможете это сделать.

Регистрация компонентов с помощью блоков кода

Еще один вариант создания компонента с примитивным значением – использовать еще один `InjectionMember`, позволяющий передавать делегат, который создает компонент:

```
container.RegisterType<ICourse, ChiliConCarne>(
    new InjectionFactory(
        c => new ChiliConCarne(Spiciness.Hot)));
```

`InjectionFactory` – это еще один класс, унаследованный от абстрактного класса `InjectionMember`. Он обладает двумя перегруженными конструкторами, но мы используем самый простой из них, который в качестве входных данных принимает `Func<IUnityContainer, object>`. Это дает нам возможность определить блок кода, с помощью которого будет создаваться компонент. В этом примере всякий раз при разрешении компонента `ICourse` будет вызываться конструктор `ChiliConCarne` с параметром `Spiciness.Hot`.

Примечание

Блок кода, приведенный в примере выше, полностью идентичен соответствующему блоку кода, приведенному для контейнера `Autofac` в разделе 13.4.2 "Регистрация объектов с помощью блоков кода".

Когда дело касается класса `ChiliConCarne`, вам предоставляется выбор между автоматической интеграцией и использованием блока кода. Но другие классы более ограничены: их экземпляры нельзя создать с помощью открытого конструктора. Чтобы создать экземпляры типа, вместо открытого конструктора вам приходится использовать некоторого рода фабрику. Для DI-контейнеров это всегда проблематично, поскольку по умолчанию им нужны открытые конструкторы.

Рассмотрим приведенный ниже пример конструктора открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже если класс `JunkFood` является открытым, конструктор расположен внутри него. Очевидно, экземпляры `JunkFood` должны создаваться с помощью статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С точки зрения Unity, это проблемное API, поскольку в нем отсутствуют точно выраженные и заданные соглашения касательно статических фабрик. Тут требуется помощь – и мы можем предоставить ее посредством блока кода, который Autofac может исполнять для того, чтобы создать экземпляр:

```
container.RegisterType<IMeal, JunkFood>(
    new InjectionFactory(
        c => JunkFoodFactory.Create("chicken meal")));
```

В этот раз вы используете класс `InjectionFactory` для создания компонента, вызывая статическую фабрику в рамках блока кода. Всякий раз при разрешении `IMeal` будет вызываться `JunkFoodFactory.Create` и возвращаться результат.

Является ли написание блока кода для создания экземпляра лучшим вариантом, нежели прямой вызов кода? При использовании блока кода внутри конструктора `InjectionFactory` мы приобретаем следующие преимущества:

- `IMeal` преобразуется в `JunkFood`.
- Стиль существования остается доступным для конфигурирования. Несмотря на то, что для создания экземпляра вызывается блок кода, он может и не вызываться всякий раз при запросе экземпляра. По умолчанию он вызывается, но если мы изменим стиль существования на `Singleton`, то блок кода будет вызываться только один раз, а результат будет кэшироваться и впоследствии повторно использоваться.

Последним рассматриваемым нами отклонением от `Constructor Injection` является `Property Injection`.

Интегрирование с помощью `Property Injection`

`Property Injection` – это менее определенная форма механизма внедрения зависимостей, поскольку компилятор не принуждает нас задавать значение свойства, доступного для записи. Это касается и Unity, который будет оставлять доступные для записи свойства незаполненными до тех пор, пока мы явно не попросим его заполнить их.

Рассмотрим класс `CaesarSalad`:

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

По всеобщему заблуждению в состав салата "Цезарь" входит курица. По своей сути "Цезарь" является салатом, но, поскольку с курицей он вкуснее, то ее часто предлагают использовать в нем в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Если вы конфигурируете только класс `CaesarSalad`, явно не обращаясь к свойству `Extra`, то этому свойству не будет присвоено значение. Вы все равно можете разрешать экземпляр, но свойство `Extra` будет иметь значение по умолчанию, которое ему присвоил конструктор (если только это имеет место).

Помимо этого, чтобы присвоить значение свойству, вы можете использовать `InjectionMember`:

```
container.RegisterType<IIngredient, Chicken>();
container.RegisterType<ICourse, CaesarSalad>(
    new InjectionProperty("Extra"));
```

Точно так же, как вы можете использовать класс `InjectionConstructor` для конфигурирования `Constructor Injection`, вы можете использовать класс `InjectionProperty` для конфигурирования `Property Injection`. `InjectionProperty` — это еще один класс, унаследованный от `InjectionMember`. Чтобы его использовать, вы должны указать имя того свойства, которое необходимо заполнить. В этом примере вы собираетесь заполнить свойство `Extra`. Это приведет к автоматической интеграции свойства, поэтому очень важно, чтобы Unity умел разрешать этот тип. Свойство `Extra` имеет тип `IIngredient`, поэтому Unity разрешит это свойство в `Chicken` благодаря тому, что ранее вы зарегистрировали `Chicken` как `IIngredient`.

Предупреждение

Если для свойства задается паттерн `Property Injection`, то Unity должен уметь разрешать тип этого свойства. Если он не сможет это сделать, то при попытке разрешения такого типа будет возникать исключение.

Когда вы на основании этой регистрации разрешаете `ICourse`, вы получаете экземпляр `CaesarSalad`, свойству `Extra` которого присвоен экземпляр `Chicken`.

В предыдущем примере использовался `IIngredient` по умолчанию, но для того чтобы присвоить значение свойству, мы можем использовать и другую перегрузку класса `InjectionProperty`. Мы можем присвоить ему непосредственное значение или можем воспользоваться проверенным классом `ResolvedParameter<T>`, чтобы сослаться на именованный компонент.

```
container.RegisterType<IIngredient, Chicken>("chicken");
container.RegisterType<IIngredient, Steak>("steak");
container.RegisterType<ICourse, CaesarSalad>(
    new InjectionProperty("Extra",
        new ResolvedParameter<IIngredient>("chicken")));
```

Экземпляр `ResolvedParameter` ссылается на ранее зарегистрированный компонент `chicken`, гарантируя, что при разрешении `ICourse` вы получите `CaesarSalad`, свойству `Extra` которого присвоен экземпляр `Chicken`.

В этом разделе вы увидели, как можно использовать Unity для работы с более трудными API разработки. Для указания конкретных экземпляров или блоков кода, которые будут использоваться для создания экземпляров, можно применять различные классы, унаследованные от `InjectionMember`, а также конфигурировать `PropertyInjection`. Я считаю, что это API становится довольно простым для изучения, как только вы осознаете его смысл.

14.5. Резюме

Unity – это DI-контейнер, разрабатываемый группой Patterns&Practices компании Microsoft. Несмотря на то, что он не является как таковым решением компании Microsoft, многие организации, занимающиеся разработкой приложений, все еще считают его некоторого рода полуофициальным продуктом компании Microsoft. Теми компаниями, которые строго соблюдают принцип использования только продуктов компании Microsoft, этот факт может восприниматься как преимущество, поскольку в Unity часто входят модули приложений группы Patterns&Practices. Вторым фактором, влияющим на их выбор, является наличие официальной документации.

API, предлагаемое контейнером Unity, является постоянным, но, в отличие от большинства других контейнеров, не гибким. Иногда это может приводить к усложнению проверки, но как только вы поймете суть контейнера Unity, вы сможете оценить его постоянность: за исключением необязательных имен и `LifetimeManager`'ов, вся дальнейшая конфигурация выполняется с помощью классов, унаследованных от абстрактного класса `InjectionConstructor`.

Помимо императивного API контейнер Unity также обладает комплексной XML-схемой, которая позволяет определять конфигурацию контейнера как в XML, так и в коде. С другой стороны, он не поддерживает основанный на соглашениях механизм автоматической регистрации.

Unity – один из тех относительно немногих DI-контейнер'ов, которые поддерживают механизм перехвата. Эта поддержка обеспечивается за счет расширения основного контейнера, но при этом является частью модуля приложений.

Одной из относительно слабых сторон контейнера Unity является механизм управления жизненным циклом. Несмотря на то, что встроенные стили существования кажутся вполне достаточным набором стилей, управление ресурсами становится довольно трудным процессом, поскольку после прекращения использования компоненты не высвобождаются должным образом. К счастью, не все так уж плохо, поскольку мы можем реализовать пользовательские стили существования, которые будут решать эти проблемы. Получается, что управлять жизненными циклами в рамках контейнера Unity достаточно сложно, но вполне возможно.

В общем Unity является довольно добротным DI-контейнер'ом. Он имеет свои слабые стороны, но, с другой стороны, предоставляет довольно полный набор возможностей. Единственная важная упущенная возможность – автоматическая регистрация, но поскольку API контейнера Unity является открытым, при желании его можно усовершенствовать.

Эта глава познакомила нас с полуофициальным DI-контейнер'ом компании Microsoft. В следующей главе мы рассмотрим контейнер, слегка отличающийся от контейнера Unity. Некоторые вообще не считают его DI-контейнер'ом. С другой стороны, он входит в состав стандартной библиотеки классов .NET 4 и имеет множество схожих с DI-контейнер'ами черт. Поэтому стоит все-таки рассмотреть *Managed Extensibility Framework* (MEF – управляемая платформа расширений).

15. MEF

Меню:

- Знакомство с MEF
- Управление жизненным циклом
- Работа с составными компонентами
- Конфигурирование сложных API

В пяти предыдущих главах вы наблюдали за тем, как можно использовать различные DI-контейнеры в качестве средств реализации паттернов и принципов, описанных в остальной части книги. В этой главе мы будем заниматься несколько другим, поскольку *Managed Extensibility Framework* (MEF – управляемая платформа расширений) в действительности не является DI-контейнером.

Судя по названию, MEF – это фреймворк, предназначенный для решения проблем расширяемости приложений. Целью MEF является подключение к стандартному приложению дополнительных сценариев. Вероятно, Visual Studio 2010 является самым первым и наиболее известным приложением, в котором MEF используется для поддержки плагинов. Но с другой стороны, любое другое приложение, созданное на платформе .NET 4 или Silverlight 4, также может использовать MEF для раскрытия возможностей расширяемости.

Если MEF не является DI-контейнером, то почему тогда на ее рассмотрение мы выделили целую главу? Самая важная причина заключается в том, что MEF *настолько похож* на DI-контейнер, что вы потратите уйму времени на то, чтобы разобраться, в чем же его отличие от реального DI-контейнера. Поскольку он является частью .NET 4 или Silverlight 4, возможно, использование его в качестве DI-контейнера было бы довольно заманчивым, если бы вы не понимали тонкой границы между ними. Цель этой главы – выявить эти различия для того, чтобы вы могли принять верное решение.

Примечание

Не забывайте, что если вам не интересен MEF и вы уже решили использовать другой DI-контейнер, вы всегда можете пропустить эту главу.

Является ли MEF DI-контейнером?

Вокруг того, является ли MEF DI-контейнером или нет, возникает слишком много путаницы. Если вкратце, то нет, MEF не является DI-контейнером, но у него настолько много общего с "настоящими" DI-контейнерами, что в будущем он может стать полноценным DI-контейнером.

MEF не создавался как DI-контейнер. Его целью является предоставление универсального фреймворка для подключения дополнительной функциональности в стандартные приложения. С точки зрения стандартного приложения, расширение – это неизвестный компонент. Несмотря на то, что расширение, по всей видимости, необходимо для раскрытия определенного интерфейса, это и есть все, что знает о нем приложение. В зависимости от среды разработки в приложении могут присутствовать одно или несколько расширений, или же в приложение может не входить ни одного расширения. Эти MEF

отличается от DI-контейнера, в котором мы обычно на этапе компиляции бываем осведомлены обо всех (или о большинстве) его компонентах.

Когда мы используем DI-контейнер в качестве инструмента для компоновки приложения, мы точно знаем, из каких компонентов состоит приложение, и используем эти знания для того, чтобы сконфигурировать контейнер в Composition Root приложения.

С другой стороны, когда дело касается плагинов, мы знаем только о том, что плагины должны реализовывать некоторого рода абстракцию, но не можем скомпилировать приложение, состоящее из конфигурации конкретных плагинов, поскольку на этапе разработки они еще неизвестны. В этом случае нам необходим механизм, позволяющий обнаруживать компоненты приложения.

Традиционный механизм обнаружения расширений заключается в просмотре определенного каталога сборок с целью обнаружения всех классов, реализующих необходимую абстракцию. Однако это не решает той проблемы, которая возникает, когда у самого расширения есть зависимости. С другой стороны, MEF решает именно эту проблему посредством расширенной модели обнаружения, которая для определения потребителей и их сервисов использует атрибуты.

DI-контейнер поддерживает раздельную компоновку сервисов. Такая возможность обеспечивает высокую степень гибкости, но является довольно затратной: будучи разработчиками уже на этапе конфигурирования контейнера мы должны быть осведомлены о том, какие компоненты собираемся компоновать.

MEF поддерживает *механизм обнаружения компонентов*. Такой подход успешно решает проблему, возникающую в ситуациях, когда на этапе проектирования мы почти ничего не знаем о расширениях. Компромисс заключается в том, что механизм обнаружения расширений сильно связан с компонентами, поэтому мы теряем некоторую гибкость.

При рассмотрении внутренней инфраструктуры MEF, оказывается, что процессы обнаружения расширений и компоновки не связаны друг с другом. Это означает, что компания Microsoft может усовершенствовать MEF в направлении настоящего DI-контейнера. С другой стороны, некоторые DI-контейнеры обладают столь огромным багажом основанных на соглашениях возможностей, что в будущем могут посягнуть на MEF.

Даже на сегодняшний день MEF имеет столь много общего с DI-контейнерами, что некоторые считают его DI-контейнером, а некоторые отвергают этот факт.

Подсказка

Если вы уже пытались использовать MEF в качестве DI-контейнера и были разочарованы и сбиты с толку, то эта глава объяснит вам, почему так произошло.

Несмотря на то, что, когда MEF зарождался и разрабатывался, возможность использования его в качестве DI-контейнера не имела первостепенной важности, оказывается, его все-таки можно использовать как DI-контейнер. Со временем это стало возможным, хотя и довольно затруднительным. Существует несколько сценариев, при которых имеет смысл использовать MEF в качестве DI-контейнера: в частности, в приложениях, в которых он уже используется для реализации возможностей расширяемости приложения.

Примечание

Общая предпосылка этой главы (непреднамеренно) заключается в том, что использование MEF не принесет нам успешного результата. Это не означает, что MEF делает что-то не так. Это означает, что мы пытаемся заставить MEF сделать то, для чего он не предназначен. Мы пытаемся сделать невозможное.

В этой главе мы рассмотрим, как можно использовать MEF для применения принципов и паттернов, описанных в частях 1-3. На рисунке 15-1 продемонстрирована структура этой главы

Рисунок 15-1: Эта глава состоит из четырех разделов. Первый раздел знакомит нас с MEF и демонстрирует способы конфигурирования и разрешения компонентов. Следующие три раздела связаны с паттернами применения, которые требуют дополнительного внимания. Их можно прочитать по порядку или же пропустить некоторые из этих разделов и прочитать только те, которые вас интересуют.



Структура главы является отражением структуры всех остальных глав части 4, поскольку я считаю, что это облегчает сравнение различных DI-контейнеров. В случае с MEF это приводит к четырем неравноценным по размеру и важности разделам. Много времени мы потратим на знакомство с MEF для того, чтобы получить полноценное понимание этого контейнера и его API. На рассмотрение механизма управления жизненным циклом выделим только несколько страниц, поскольку в этой области MEF не обладает обширными возможностями. Далее большая часть оставшейся главы отведена на изучение возможностей работы с составными компонентами, а последний раздел снова занимает совсем немного страниц. Первый раздел является предпосылкой остальных разделов, оставшиеся три раздела можно читать независимо друг от друга.

Эта глава должна дать вам возможность приступить к работе, а также справиться с большинством общепринятых проблем, которые могут возникнуть при использовании MEF. Однако это не окончательная трактовка MEF, поскольку мы рассматриваем его в терминах DI-контейнеров, а не с точки зрения его реального предназначения.

Вы можете прочитать эту главу, не читая при этом остальные главы части 4, специально для того, чтобы познакомиться с MEF, или можете прочитать ее совместно с остальными главами части 4, чтобы сравнить MEF с "настоящими" DI-контейнерами. Цель данной главы – показать, как MEF связан с паттернами и принципами, описанными в остальной части книги, и как он их реализует.

[15.1. Знакомство с MEF](#)

[15.2. Управление жизненным циклом](#)

[15.3. Работа с составными компонентами](#)

[15.4. Конфигурирование сложных API](#)

[15.5. Резюме](#)

15.1. Знакомство с MEF

Из этого раздела вы узнаете, где можно взять MEF, что вы при этом получите и как начать его использовать. Кроме того, мы рассмотрим варианты конфигурирования и пакетирования компонентов. В таблице 15-1 содержится основополагающая информация, которая, скорее всего, понадобится вам для того, чтобы приступить к работе с Unity.

Таблица 15-1: Краткая информация об MEF

Вопрос	Ответ
Откуда мне его получить?	MEF является частью .NET 4 и Silverlight 4.
Что находится в загруженном файле?	Вы получаете MEF при установке .NET 4 или Silverlight 4. MEF является частью стандартной библиотеки классов и упакован в сборку <code>System.ComponentModel.Composition</code> . Если вы посетите сайт http://mef.codeplex.com/ , то сможете также загрузить исходный код для дальнейшего внимательного изучения.
Какие платформы поддерживаются?	.NET 4 и Silverlight 4. На сайте http://mef.codeplex.com/ вы также можете найти неподдерживаемые версии для .NET 3.5 SP1 и Silverlight 3.
Сколько он стоит?	Нисколько. MEF входит в состав .NET 4 и Silverlight 4.
Откуда мне получить помощь?	Поскольку MEF является частью .NET 4 и Silverlight 4, вы можете обратиться за поддержкой в компанию Microsoft.
На какой версии MEF основана эта глава?	.NET 4

В отличие от остальных DI-контейнеров MEF имеет другой цикл применения. Мы никогда не конфигурируем контейнер, зато снабжаем сами компоненты атрибутами. На рисунке 15-2 продемонстрирована взаимосвязь компонентов с движком компоновки.

Рисунок 15-2: При работе с MEF мы снабжаем части (к примеру, классы и члены) атрибутами в рамках отдельных рабочих фаз. При компоновке приложения мы сначала отбираем соответствующие части в каталог, а затем используем этот каталог для определения контейнера, из которого можно разрешать компоненты.



Терминология MEF

MEF использует терминологию, которая слегка отличается от той, которой мы пользовались при обсуждении других DI-контейнеров.

Обычно взаимодействующие классы мы называли *компонентами*, но в MEF для обозначения таких классов используется термин *часть*. *Часть* – это класс или член класса, который передает или использует зависимость.

Когда часть использует зависимость, мы говорим, что она *импортирует* ее. Напротив, когда часть предоставляет сервис, она его *экспортирует*. В классическом случае в профессиональной сфере слова `import` и `export` могут использоваться и как существительные: импорт и экспорт.

Возможность экспорта и импорта определяется путем снабжения частей соответствующими атрибутами.

При компоновке приложения мы сопоставляем экспортируемые и импортируемые компоненты в соответствии с *контрактами*. Часто в качестве контрактов мы используем типы (например, интерфейсы), но MEF более гибкий, нежели контракты. Контракт, в действительности, – это всего лишь строка.

При работе с другими DI-контейнерами мы используем расщепленное API конфигурации для определения доступных компонентов, для определения того, как конкретные типы преобразуются в абстракции, как создаются компоненты, и должны ли экземпляры использоваться совместно.

Напротив, при работе с MEF мы привязываем эту информацию к каждой части путем применения атрибутов к типам и членам. Это довольно легко понять, но такой подход сильно связывает конфигурацию компонента с самим компонентом.

Примечание

Помните о том, что MEF, по первой версии, использует атрибуты в качестве метода обнаружения по умолчанию (и только по умолчанию), но по сути, не привязан к атрибутам как к средству обнаружения.

Чтобы скомпоновать приложение, мы отбираем соответствующие части и пакетируем их в *каталог*, а затем создаем контейнер, который может разрешать компоненты из этого каталога.

После прочтения этого раздела вы приобретете полноценное понимание всей сущности паттерна применения MEF и сможете начать использовать его в сценариях, в которых все части определяют простые экспортируемые и импортируемые компоненты. Давайте начнем с простейшего сценария и посмотрим, как можно разрешать объекты с помощью контейнера MEF.

Разрешение объектов

Основная услуга, предоставляемая любым DI-контейнером – компоновка и разрешение компонентов. В этом разделе мы рассмотрим API, которое позволяет разрешать компоненты с помощью MEF. Что касается работы с любым другим контейнером,

разрешать объекты в них так же просто, как и вызывать простой метод. Но при работе с MEF мы не можем что-либо разрешать до тех пор, пока не станут доступными соответствующие экспортируемые компоненты.

Вспоминая обсуждение процесса разрешения компонентов при помощи Castle Windsor и StructureMap, вы, возможно, припомните, что Windsor требует, чтобы перед тем, как стало возможным разрешение компонентов, все соответствующие компоненты были зарегистрированы. MEF предъявляет аналогичное требование, несмотря на то, что мы никоим образом не можем *зарегистрировать* компонент. Вместо этого часть должна экспортировать необходимый сервис.

Чтобы разрешить сервис `SauceBéarnaise`, мы должны его экспортировать. Самый простой способ осуществить это – путем аннотирования самого класса следующим образом:

```
[Export]
public class SauceBéarnaise : IIngredient { }
```

Обратите внимание на атрибут `[Export]`, которым снабжен класс `SauceBéarnaise`. Это MEF атрибут, который объявляет, что класс `SauceBéarnaise` экспортирует самого себя. Это означает, что, если вы поместите этот класс в каталог, то после этого сможете разрешать класс `SauceBéarnaise`, но никакой другой, поскольку атрибутом экспорта отмечен только этот класс:

```
var catalog = new TypeCatalog(typeof(SauceBéarnaise));
var container = new CompositionContainer(catalog);
SauceBéarnaise sauce =
    container.GetExportedValue<SauceBéarnaise>();
```

На рисунке 15-2 вы уже видели некоторый намек на понятие каталога. Более детально мы рассмотрим его в разделе 15.1.3. А сейчас будет достаточно заметить, что аннотированный класс `SauceBéarnaise` помещается в каталог, который вы используете для определения контейнера. После получения контейнера вы можете использовать его для разрешения сервиса `SauceBéarnaise`.

Примечание

Метод `GetExportedValue` полностью соответствует методам `Resolve` контейнеров Windsor, Autofac и Unity.

Помимо метода `GetExportedValue` MEF поддерживает и еще один стиль, при котором мы сначала получаем экспортируемые компоненты, а затем извлекаем значение этих компонентов. Самая простая форма этого стиля выглядит следующим образом:

```
Lazy<SauceBéarnaise> export =
    container.GetExport<SauceBéarnaise>();
SauceBéarnaise sauce = export.Value;
```

Метод `GetExport` – отличный пример экспорта, который является первостепенным понятием MEF. Метод `GetExport` инкапсулирует экспортируемые компоненты, не создавая при этом экземпляр части. Создание части можно отложить до тех пор, пока мы не запросим свойство `Value` этой части, но все это зависит от стиля существования части.

И метод `GetExportedValue`, и метод `GetExport` обладают многочисленными аналогами, что позволяет нам разрешать последовательности частей. Выглядят они следующим образом:

```
IEnumerable<IIngredient> ingredients =  
    container.GetExportedValues<IIngredient>();  
IEnumerable<Lazy<IIngredient>> exports =  
    container.GetExports<IIngredient>();
```

До настоящего момента класс `SauceBéarnaise` экспортировал только свой собственный тип. Даже если он реализует `IIngredient`, он все равно не будет экспортировать этот интерфейс до тех пор, пока вы явно не укажете ему на это. При преобразовании абстракций к конкретным типам также используется атрибут `[Export]`.

Преобразование абстракций в конкретные типы

Атрибут `[Export]` экспортирует ту часть, к которой он относится. Иногда экспортируемая часть уже является абстракцией, но при аннотировании класса по умолчанию экспортируется конкретный класс, даже если он реализует один или несколько интерфейсов.

В связи со слабым связыванием нам необходимо преобразовывать абстракции в конкретные типы. Создание экземпляров на основании таких преобразований – основная услуга, предоставляемая любым DI-контейнером, но нам все равно нужно определить преобразование. При работе с MEF мы делаем это путем явного указания в атрибуте `[Export]` того, что он должен экспортировать.

В приведенном ниже примере вы позволяете конкретному классу `SauceBéarnaise` экспортировать интерфейс `IIngredient`:

```
[Export(typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }
```

По сравнению с предыдущим примером вы изменили атрибут `[Export]` таким образом, что теперь используется перегрузка, которая позволяет нам указывать, что экспортируемым типом является `IIngredient`. И снова вы пакетируете класс `SauceBéarnaise` в каталог, а затем из этого каталога создаете контейнер.

```
IIngredient ingredient = container.GetExportedValue<IIngredient>();
```

При разрешении `IIngredient` из контейнера, оказывается, что значение `ingredient`, как и ожидалось, является экземпляром класса `SauceBéarnaise`. Однако если вы попытаетесь разрешить `SauceBéarnaise` таким образом, как делали это в первом примере, то получите исключение, в связи с отсутствием частей, которые экспортировали бы контракт `SauceBéarnaise`.

Вы легко можете это сделать, несколько раз применив атрибут `[Export]`:

```
[Export]  
[Export(typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }
```

Атрибут `[Export]` можно применять столько раз, сколько потребуется, поэтому при таком варианте экспортируется и класс `SauceBéarnaise`, и интерфейс `IIngredient`.

До настоящего момента мы рассматривали строго типизированный generic-метод, используемый для разрешения сервисов. Однако существуют ситуации, когда для разрешения сервисов, нам необходим более слабо типизированный подход. Это возможно, но подразумевает выполнение несколько большего количества действий, чем планировалось.

Разрешение слабо типизированных сервисов

Иногда у вас нет возможности использовать generic API, поскольку на этапе проектирования вы еще не знаете, какой тип вам понадобится. У вас есть только экземпляр `Type`, но все же вам хотелось бы получить экземпляр этого типа. Пример такой ситуации описан в разделе 7.2 "Построение ASP.NET MVC приложений", в котором обсуждался ASP.NET класс `DefaultControllerFactory`. Соответствующий метод приведен в следующем примере:

```
protected internal virtual IController GetControllerInstance(  
    RequestContext requestContext, Type controllerType);
```

Поскольку у вас имеется только экземпляр `Type`, вы не можете использовать generic'и, а должны прибегнуть к слабо типизированному API. К несчастью, единственное нетипизированное API, раскрываемое с помощью `CompositionContainer`, является слегка громоздким. Нетипизированных версий методов `GetExportedValue` или `GetExportedValues` не существует, поэтому для реализации `GetControllerInstance` нам необходимо прибегнуть к не generic-версии `GetExports`:

```
var export = this.container.GetExports(  
    controllerType, null, null).Single();  
return (IController) export.Value;
```

Существует несколько перегрузок метода `GetExports`, но в этом примере мы используем ту, которая позволяет нам передавать результат напрямую в `controllerType`. Два других параметра можно использовать для создания ограничителей запроса, но если они нам не нужны, то мы можем передать для них значения типа `null`. Метод `GetExports` возвращает последовательность экспортируемых компонентов, но согласно нашему условию экспортировать можно только один компонент, удовлетворяющий параметрам запроса. Поэтому для получения единичного экземпляра из последовательности мы вызываем метод расширения `Single`.

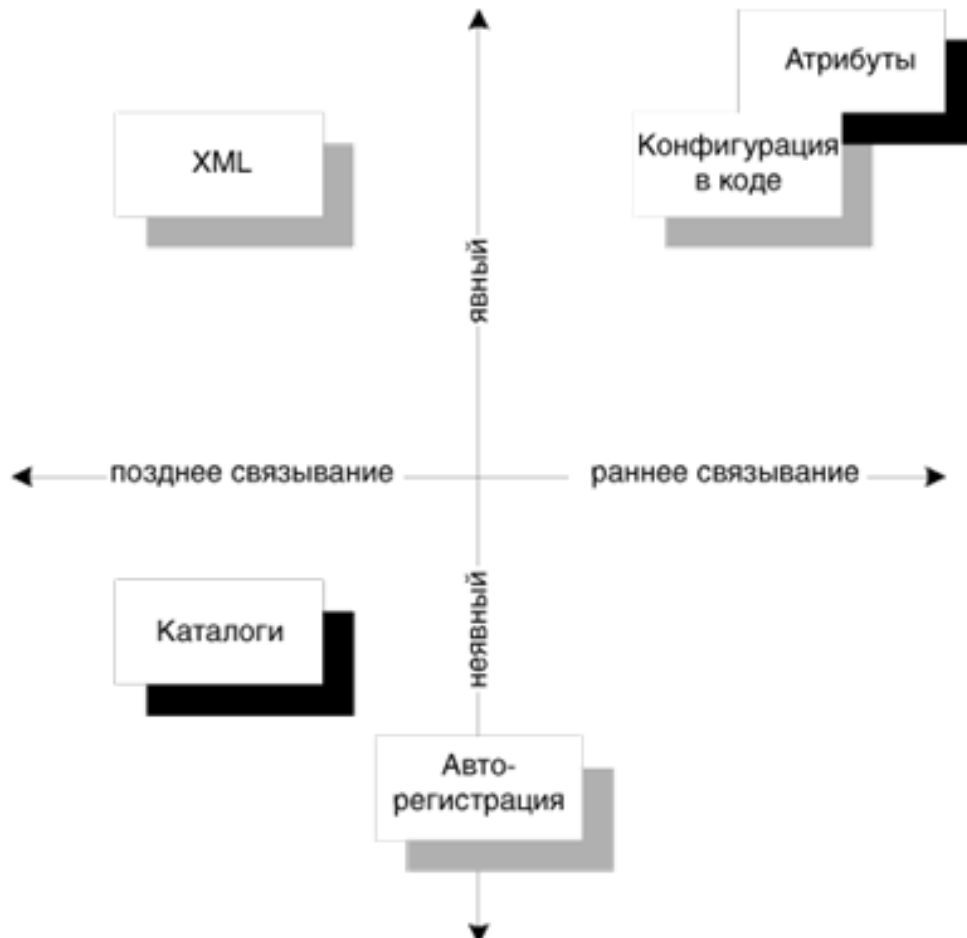
Поскольку метод `GetExports` является слабо типизированным, мы должны привести экспортируемое значение к `IController` перед тем, как его вернуть. В любом случае независимо от того, какой конкретный метод вы используете для разрешения частей, MEF компонует части посредством сопоставления импортируемых и экспортируемых компонентов. MEF может это делать только, когда мы заранее явно определили это.

Определение импортируемых и экспортируемых компонентов

В разделе 3.2 "Конфигурирование DI-контейнеров" мы обсуждали несколько различных способов конфигурирования DI-контейнера. На рисунке 15-3 представлен обзор

возможных вариантов, а также показано то, что MEF совершенно не подходит под эту модель.

Рисунок 15-3: MEF не подходит под нашу стандартную концептуальную модель вариантов конфигурирования, поскольку мы не можем конфигурировать контейнер. Все обычные варианты, которые мы рассматривали ранее, являются недоступными и обозначены серым цветом. Снабжение частей атрибутами статистически наряду с типом определяет конфигурацию, тогда как каталоги обеспечивают гибкость.



Мы не можем сконфигурировать контейнер с помощью используемых нами ранее способов – ни с помощью императивного кода, ни посредством XML конфигурации. MEF дает нам возможность определять только импортируемые и экспортируемые части, а именно посредством применения атрибутов к этим частям. Атрибуты являются частями тех типов, которыми они отмечены, поэтому мы должны рассматривать этот механизм как более явный и более рано связанный, нежели, например, технология конфигурирования в коде. С другой стороны, каталоги делают MEF довольно гибким, поскольку позволяют отбирать типы, которые мы хотим включить в компоновку.

Предупреждение

Применение атрибутов сильно привязывает конфигурацию к реализации. В разделе 9.3.1 "Использование атрибутов для объявления аспектов" мы обсуждали недостатки использования атрибутов для аспектов, но это обсуждение применимо и ко всему

остальному. Не забывайте, что в будущем в MEF может появиться некоторая альтернатива атрибутам

В этом разделе мы рассмотрим варианты импорта и экспорта частей, а затем в разделе 15.1.3 перейдем к изучению каталогов. Несмотря на то, что мы не можем использовать наши стандартные варианты конфигурирования контейнера, мы можем привести некоторые соответствия, чтобы рассмотреть другие сценарии. В этом разделе вы познакомитесь с различными способами экспорта и импорта частей.

Экспорт типов

В этом разделе мы рассмотрим сценарий, при котором мы управляем классами, которые планируем экспортировать. Когда мы полностью контролируем исходный код тех классов, которые собираемся экспортировать, мы можем экспортировать класс с помощью атрибута `[Export]`:

```
[Export]
[Export(typeof(IIngredient))]
public class SauceBéarnaise : IIngredient { }
```

Свойство `[Export]` можно применять столько раз, сколько потребуется, поэтому один и тот же класс может экспортировать различные контракты. Класс `SauceBéarnaise`, продемонстрированный в этом примере, экспортируется и как конкретный класс, и как интерфейс `IIngredient`.

Перегрузка `[Export]`, позволяющая указывать экспортируемый тип, не нуждается в проверке на этапе компиляции. Вы можете объявить невалидный атрибут экспорта, и при этом не произойдет никаких ошибок компиляции:

```
[Export(typeof(ICourse))]
public class SauceBéarnaise : IIngredient { }
```

Класс `SauceBéarnaise` не реализует интерфейс `ICourse`, хотя вы можете составить требование, что он должен его реализовывать. Однако при попытке разрешения `ICourse` выдается исключение, поскольку MEF не умеет приводить `SauceBéarnaise` к `ICourse`.

Предупреждение

Можно объявлять невалидные атрибуты экспорта.

Очевидно, вы можете позволить различным классам экспортировать разные контракты, и при этом не возникнет никаких конфликтов:

```
[Export(typeof(IIngredient))]
public class SauceBéarnaise : IIngredient { }
[Export(typeof(ICourse))]
public class Course : ICourse { }
```

Поскольку каждый отдельный класс экспортирует отличные от других контракты, конфликта не возникает, и вы можете отправить контейнеру запрос на разрешение и `ICourse`, и `IIngredient`, получив в итоге экземпляры `Course` и `SauceBéarnaise` соответственно.

Однако если мы экспортируем одну и ту же абстракцию несколько раз, то картина изменится:

```
[Export(typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }  
[Export(typeof(IIngredient))]  
public class Steak : IIngredient { }
```

В этом примере вы экспортируете `IIngredient` дважды. Если вы попытаетесь разрешить `IIngredient`, то контейнер выдаст исключение, поскольку в коде приведено несколько атрибутов экспорта. Под вывоном метода `GetExport` или `GetExportedValue` подразумевается запрос повсеместно встречающейся части. Вы все равно можете получить и `SauceBéarnaise`, и `Steak`, вызвав множественные методы `GetExports` или `GetExportedValues`.

Примечание

В MEF нет такого понятия как *компонент по умолчанию*. Все экспортируемые компоненты равны между собой.

Это и есть первый намек на такое важное понятие MEF, как *мощность* импортируемых и экспортируемых компонентов. Количество экспортируемых компонентов должно совпадать с количеством импортируемых компонентов. В таблице 15-2 показано, как MEF сопоставляет импортируемые и экспортируемые компоненты по мощности.

Таблица 15-2: Сопоставление количества атрибутов импорта и экспорта

	Export.Single	Export.Many
Import.Single	Равно	Не равно
Import.Many	Равно	Равно

В этом контексте термин "many" используется для обозначения последовательности частей, обычно массива `IEnumerable<T>`. Если мы будем явным образом импортировать множество частей одного и того же контракта, то MEF всегда будет находить соответствие, поскольку нулевое количество атрибутов экспорта есть особый случай множественных атрибутов экспорта.

С другой стороны, явно импортируя один экземпляр, мы получаем несоответствие мощностей в случае наличия нулевого или множественного количества атрибутов экспорта, потому что импорт одного экземпляра означает, что у нас должен быть только один экземпляр.

Примечание

Мощность – это один из тех видов размерностей, по которым атрибуты экспорта и импорта должны соответствовать друг другу.

Как вы увидите в разделе 15.2.1, механизм управления жизненным циклом также может играть важную роль в вопросе соответствия частей, но мощность учитывается всегда. Далее в этом разделе вы увидите, как определять единичные и множественные атрибуты импорта. Но перед тем как приступить к рассмотрению этого вопроса, нам следует

рассмотреть процесс экспорта частей в тех ситуациях, когда мы не контролируем классы, участвующие в этом процессе.

Экспорт адаптеров

Самый простой способ экспортировать часть – применить к классу атрибут `[Export]`, но это не всегда возможно сделать. Нам может понадобиться экспортировать классы, которые уже были скомпилированы, а доступа к исходному коду у нас нет. В таких случаях мы не можем применять атрибуты, хотя нам и хотелось бы включить этот класс в компоновку.

И все равно мы можем достичь этой цели, усилив способность MEF экспортировать свойства так же, как и классы. Для примера рассмотрим конструктор `Mayonnaise`:

```
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

Представьте себе, что класс `Mayonnaise` и составляющие его зависимости `EggYolk` и `OliveOil` не подвластны нам. Одним из возможных вариантов в таком случае было бы наследование от первоначального класса и применение атрибута `[Export]` к унаследованному классу:

```
[Export(typeof(OliveOil))]  
[Export(typeof(IIngredient))]  
public class MefOliveOil : OliveOil { }
```

Обратите внимание, что если вам необходимо экспортировать и первоначальный конкретный класс, и интерфейс `IIngredient`, то вы должны явным образом установить, что базовый класс (который и является конкретным классом) также должен быть экспортирован. Если бы вы использовали атрибут `[Export]` без указания типа, то вместо базового класса вы бы экспортировали класс `MefOliveOil`.

Однако если рассматриваемые классы заблокированы, то вы не сможете экспортировать их таким образом. В этом случае, как показано в следующем листинге, можно создать адаптер и экспортировать часть через свойство.

Листинг 15-1: Экспорт `OliveOil` через адаптер

```
1. public class OliveOilAdapter  
2. {  
3.     private readonly OliveOil oil;  
4.     public OliveOilAdapter()  
5.     {  
6.         this.oil = new OliveOil();  
7.     }  
8.     [Export]  
9.     public OliveOil OliveOil  
10.    {  
11.        get { return this.oil; }  
12.    }  
13. }
```

Строка 8: Экспорт свойства

Класс `OliveOilAdapter` – это совершенно новый класс, который обортывает первоначальный класс `OliveOil` и экспортирует его посредством аннотированного

свойства. Атрибут [Export] можно применять как к свойствам, так и к типам, но в остальном он работает аналогичным образом. Свойство OliveOil имеет тип OliveOil, являющийся, в свою очередь, контрактом, который вы собираетесь экспортировать, поэтому в этом случае вы можете использовать свойство [Export], не задавая явно тип.

Подсказка

Тип всегда можно экспортировать путем создания адаптера.

Когда у класса, который вам необходимо скомпоновать, есть зависимости, вам нужно экспортировать их посредством адаптера. Как показывает следующий листинг, этот процесс становится более сложным, но пока что вполне управляемым.

Листинг 15-2: Настройка класса, имеющего зависимости

```
1. public class MayonnaiseAdapter
2. {
3.     private readonly Mayonnaise mayo;
4.     [ImportingConstructor]
5.     public MayonnaiseAdapter(
6.         EggYolk yolk, OliveOil oil)
7.     {
8.         if (yolk == null)
9.         {
10.            throw new ArgumentNullException("yolk");
11.        }
12.        if (oil == null)
13.        {
14.            throw new ArgumentNullException("oil");
15.        }
16.        this.mayo = new Mayonnaise(yolk, oil);
17.    }
18.    [Export]
19.    public Mayonnaise Mayonnaise
20.    {
21.        get { return this.mayo; }
22.    }
23. }
```

Строка 5-6: Имитатор сигнатуры конструктора Mayonnaise

Строка 16: Создание Mayonnaise

Строка 18: Экспорт Mayonnaise

Чтобы экспортировать класс Mayonnaise посредством адаптера, вы должны учитывать тот факт, что у него есть зависимости, которые вам нужно будет импортировать. Чтобы иметь возможность создать экземпляр класса, вы должны симитировать сигнатуру конструктора Mayonnaise в конструкторе адаптера для того, чтобы вы смогли импортировать все необходимые части. После прохождения соответствующих граничных операторов вы создаете новый экземпляр Mayonnaise из параметров конструктора и сохраняете результат в приватном поле. Это и есть паттерн Constructor Injection в действии.

Чтобы экспортировать класс Mayonnaise, вы можете раскрыть поле mayo как свойство и отметить его атрибутом [Export].

Благодаря `EggYolkAdapter`, похожему на `OliveOilAdapter` из листинга 15-1, вы можете создать каталог, состоящий из трех адаптеров, и успешно разрешить экземпляр `Mayonnaise`, даже если вы никогда не изменяли первоначальные классы.

Возможно, вы обратили внимание на атрибут `[ImportingConstructor]`, который появился в листинге 15-2. Это часть другой стороны уравнения. До настоящего момента мы рассматривали процесс экспорта частей. Теперь давайте изучим, как можно импортировать части.

Импорт частей

В рамках MEF присутствует некоторого рода симметрия. Большинство из тех утверждений, которые мы можем применить к атрибутам экспорта, также применимы к атрибутам импорта. Однако когда дело доходит до паттерна `Constructor Injection`, нам необходимо прибегнуть к атрибуту `[ImportingConstructor]`, эквивалентов которого для экспортируемых компонентов не существует. Мы видели, как этот атрибут применялся к `MayonnaiseAdapter` в листинге 15-2, но он должен применяться всякий раз, когда нам нужно применять паттерн `Constructor Injection`.

В приведенном примере мы предположили, что класс `Mayonnaise` нами не контролируется. Благодаря невероятному стечению обстоятельств мы смогли невзначай перехватить исходный код и теперь можем изменять типы напрямую. В этом случае нам не придется создавать адаптеры, и мы можем применять атрибуты `[Export]` напрямую к классам `Mayonnaise`, `OliveOil` и `EggYolk`.

MEF не распознал паттерн `Constructor Injection`, поэтому, даже если у `Mayonnaise` имеется всего один конструктор, при попытке разрешить этот класс мы сначала получим исключение. Нам необходимо явным образом сообщить MEF о том, какой конструктор он должен использовать при отсутствии конструктора по умолчанию:

```
[ImportingConstructor]
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

`[ImportingConstructor]` — это сигнал MEF о том, что тот конструктор, к которому относится этот атрибут, должен использоваться для компоновки типа.

Подсказка

Для конструкторов по умолчанию `[ImportingConstructor]` не нужен. Используйте этот атрибут, если у класса нет конструктора по умолчанию, или если компоновка осуществляется с помощью другого конструктора, а не конструктора по умолчанию.

Кроме того, мы можем использовать атрибут `[Import]` для поддержки паттерна `Property Injection`, но к этому вопросу мы вернемся в разделе 15.4.3, который посвящен этому паттерну. Более того, существует атрибут `[ImportMany]`. Который используется для импорта последовательностей частей, но его мы рассмотрим в разделе 15.3.2.

Импорт и экспорт частей основывается на применении атрибутов, а поскольку атрибуты компилируются в типы, это делает MEF негибким. Свою гибкость MEF приобретает благодаря *каталогам*.

Работа с каталогами

Каталог инкапсулирует набор частей, которые контейнер может использовать для компоновки диаграммы объектов. В этом разделе мы приведем обзор различных видов каталогов, доступных в MEF.

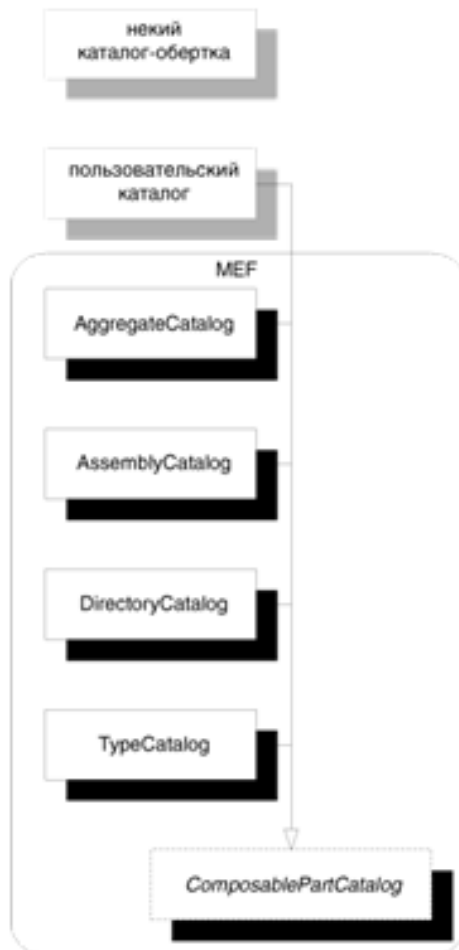
Использование каталогов в рамках контейнеров

В разделе 15.1.1 "Разрешение объектов" вы уже видели пример взаимодействия каталога и контейнера:

```
var catalog = new
    TypeCatalog(typeof(SauceBéarnaise));
var container = new
    CompositionContainer(catalog);
```

В этом примере используется `TypeCatalog` конкретного типа, но вы можете создать `CompositionContainer` с любым `ComposablePartCatalog`. `TypeCatalog` — это всего лишь один из множества дочерних классов. На рисунке 15-4 приведена схема иерархии типов.

Рисунок 15-4: В MEF входит четыре конкретных каталога, но помимо них мы можем определять пользовательские каталоги. Возможно, было бы достаточно просто реализовать каталог, который выступал бы в роли Decorator для других каталогов (например, каталог фильтрации), между тем, как настоящий пользовательский каталог был бы в него вложен.



Определение

Каталог – это любой класс, унаследованный от абстрактного класса `ComposablePartCatalog`.

Как и подразумевает его имя, `ComposablePartCatalog` – это каталог частей, которые `CompositionContainer` использует для сопоставления импортируемых и экспортируемых компонентов. Одна из перегрузок конструктора класса `CompositionContainer` позволяет передавать `ComposablePartCatalog`, и именно этот конструктор мы использовали до настоящего момента:

```
public CompositionContainer(ComposablePartCatalog catalog,
    params ExportProvider[] providers)
```

Помимо того, что этот конструктор принимает в качестве параметра экземпляр `ComposablePartCatalog`, он также принимает и массив `params` типа `ExportProviders`, который является еще одним механизмом расширяемости, не рассматриваемым в этой книге.

Поскольку `ComposablePartCatalog` – это абстрактный класс, а `CompositionContainer` принимает в качестве параметра любой унаследованный класс, теоретически мы можем создать пользовательские каталоги с самых азов. Это главный *Seam* контейнера MEF. Его даже можно использовать для определения других вариантов атрибутивной модели MEF по умолчанию, которая используется для определения импортируемых и экспортируемых компонентов. Несмотря на то, что такой подход возможен, он слишком трудозатратный, поэтому в этой главе мы не будем его рассматривать.

Подсказка

В проекте с открытым исходным кодом *MEF Contrib* содержится пример пользовательского `ComposablePartCatalog`, полностью замещающего атрибутивную модель конфигурирования более открытой моделью, которая больше похожа на другие DI-контейнеры.

Все каталоги, входящие в состав MEF в .NET 4, используют атрибуты `[Import]` и `[Export]` для определения импортируемых и экспортируемых данных, но по-разному определяют местоположение частей. Например, `TypeCatalog` определяет местоположение частей, считывая атрибуты типов, содержащихся в каталоге.

Использование каталогов типов

Класс `TypeCatalog` позволяет определять каталог, состоящий из перечня типов, на основании предположения о том, что эти типы определяют импортируемые и экспортируемые компоненты с помощью атрибутов. Существует два перегруженных конструктора, которые позволяют нам создавать произвольное количество экземпляров `Type`:

```
public TypeCatalog(params Type[] types)
public TypeCatalog(IEnumerable<Type> types)
```

В качестве примера для того, чтобы иметь возможность компоновать `Mayonnaise` из адаптеров, созданных вами в листингах 15-1 и 15-2, вы можете создать каталог следующим образом:

```
var catalog = new TypeCatalog(  
    typeof(MayonnaiseAdapter),  
    typeof(EggYolkAdapter),  
    typeof(OliveOilAdapter));
```

Это небольшой каталог, позволяющий разрешать `Mayonnaise`. Если вы удалите любой из трех типов адаптеров, то экспортируемые компоненты будут упущены. Помимо того, что этот каталог позволяет вам разрешать `Mayonnaise`, он также позволяет разрешать `EggYolk` и `OliveOil`, но не более того.

Очевидно, что вы могли бы передать в `TypeCatalog` большее количество типов, чтобы экспортировать больше данных, но при этом вы должны четко определить список типов. Это имеет смысл для небольших сценариев с ограниченной областью применения. Преимущество заключается в том, что вы можете отбирать только те типы, которые вы собираетесь использовать при компоновке. При наличии типов, экспортирующих конкурирующие типы, вы можете отобрать только те типы, которые вам нужны.

Подсказка

Вы можете создать несколько адаптеров, которые совершенно по-разному экспортируют одну и ту же часть, и передать в `TypeCatalog` только один из них. При написании шаблонного кода для этой главы я использовал этот прием для того, чтобы варьировать атрибуты `MayonnaiseAdapter`, не редактируя при этом код.

Недостаток применения `TypeCatalog` заключается в том, что вы должны явно передавать все типы. При добавлении в сборку нового типа вам также нужно добавить его в `TypeCatalog`, если вы хотите, чтобы он вошел в каталог. Это приводит к нарушению *принципа DRY* (Don't Repeat Yourself – Не повторяйся). Вы могли бы решить эту проблему путем написания кода, в котором для просмотра сборки с целью обнаружения всех открытых типов используется рефлексия. Но вам не придется этого делать, поскольку уже существует каталог, выполняющий то же самое.

Использование каталогов сборки

Класс `AssemblyCatalog` предназначен для просмотра сборки с целью поиска всех импортируемых и экспортируемых компонентов, определенных в этой сборке. Этот класс позволяет нам продолжать добавлять части в сборку без необходимости помнить о том, что нужно добавить эту часть еще и в каталог.

Использовать `AssemblyCatalog` так же просто, как и создавать экземпляр `Assembly` с помощью конструктора:

```
var assembly = typeof(Steak).Assembly;  
var catalog = new AssemblyCatalog(assembly);
```

В этом примере вы используете неизбирательный представительский тип (`Steak`) для определения сборки, но подойдет и любой другой метод, который создает соответствующий экземпляр `Assembly`.

Кроме того, существует перегрузка конструктора, которая вместо экземпляра `Assembly` принимает в качестве параметра имя файла. Это делает возможными более слабо связанные сценарии, поскольку мы можем заменить `.dll` файл, не компилируя повторно остальную часть приложения. Это приближает нас к смыслу подключения сценариев расширений в контейнере MEF. Благодаря `AssemblyCatalog` мы могли бы написать императивный цикл и создать для каждого найденного нами файла каталог по этому пути. Однако нам не приходится действовать подобным образом, поскольку в MEF уже есть специально предназначенный для этого каталог.

Использование каталогов директорий

Основная цель MEF – разрешить использовать сценарии расширений. Универсальная архитектура расширений – определить для расширений особую директорию. Главное приложение будет загружать и использовать любую сборку, размещенную в этой директории.

MEF способствует выполнению такого сценария посредством класса `DirectoryCatalog`. В конструктор мы передаем путь к директории, а он, в свою очередь, ищет в этой директории файлы с расширением `.dll` и загружает все, что обнаружено в этих сборках:

```
var catalog = new DirectoryCatalog(directory);
```

Еще одна перегрузка конструктора также позволяет нам задавать шаблон поиска посредством универсальных специальных символов.

Примечание

При использовании MEF в качестве фреймворка расширяемости, что является ключевым его предназначением, предполагается, что `DirectoryCatalog` является самым универсальным из используемых каталогов.

Несмотря на то, что в директории мы можем разместить произвольное количествоборок, а затем отбирать их с помощью `DirectoryCatalog`, мы можем захотеть сочетать каталоги из нескольких отличных друг от друга источников. Даже если для включения возможности расширяемости мы используем `DirectoryCatalog`, нам также могут понадобиться некоторые реализации по умолчанию или внутренние реализации соответствующих импортируемых и экспортируемых элементов. Эти реализации не должны размещаться в папке расширений, потому что в этом случае пользователи смогут удалить из приложения жизненно важную функциональность. Возможно, было бы лучше создать такие реализации по умолчанию с помощью `TypeCatalog`, но это означает, что нам придется бы соединить разные каталоги в один.

Использование каталогов агрегатов

Для соединения каталогов можно использовать класс `AggregateCatalog`, который является тем же самым `Composite` только с другим именем. Он объединяет в единое целое произвольное количество каталогов и в то же самое время сам является каталогом:

```
var catalog = new AggregateCatalog(catalog1, catalog2);  
var container = new CompositionContainer(catalog);
```

Входящие в MEF четыре каталога уже обеспечивают достаточную гибкость приложения, но для получения наибольшего контроля мы реализуем также пользовательские каталоги. Один из примеров, который легко реализовать и использовать, – это каталог фильтрации.

Реализация каталога фильтрации

Несмотря на то, что реализация пользовательского каталога с нуля возможно, является довольно замысловатой затеей, мы можем довольно легко реализовать Decorator, который изменит поведение другого каталога.

Наиболее очевидный пример – каталог фильтрации, который отфильтровывает вложенный каталог. В следующем листинге продемонстрирован пользовательский каталог, который является оберткой для другого каталога и может вступить в игру только с помощью тех составляющих, которые экспортируют контракт, содержащий строку "Sauce". Этот пользовательский каталог можно использовать, чтобы отобразить из каталога всех ингредиентов только соусы.

Листинг 15-3: Реализация пользовательского каталога

```
1. public class SauceCatalog : ComposablePartCatalog
2. {
3.     private readonly ComposablePartCatalog catalog;
4.     public SauceCatalog(ComposablePartCatalog cat)
5.     {
6.         if (cat == null)
7.         {
8.             throw new ArgumentNullException("cat");
9.         }
10.        this.catalog = cat;
11.    }
12.    public override
13.        IQueryable<ComposablePartDefinition> Parts
14.    {
15.        get
16.        {
17.            return this.catalog.Parts.Where(def =>
18.                def.ExportDefinitions.Any(x =>
19.                    x.ContractName
20.                        .Contains("Sauce")));
21.        }
22.    }
23. }
```

Строка 1: Наследование от ComposablePartCatalog

Строка 3-11: Constructor Injection

Строка 12-22: Реализация фильтра

Чтобы реализовать пользовательский каталог, вы выполняете наследование от абстрактного класса ComposablePartCatalog. Поскольку вы пожелали создать обертку для другого каталога, вы запрашиваете этот каталог через Constructor Injection.

Свойство Parts – единственный абстрактный член ComposablePartCatalog, поэтому это и есть тот единственный член класса, который *необходимо* реализовать. При желании вы можете реализовать и другие виртуальные члены, но для этого примера это не

потребуется. Фильтр реализуется с помощью выражения `Where`, которое отфильтровывает все `ComposablePartDefinitions`, не экспортирующие никакого контракта и содержащие слово "Sauce".

`SauceCatalog` – это конкретный класс, но вы можете обобщить реализацию, чтобы создать универсальный `FilteringCatalog`. В документации к MEF есть соответствующий пример.

Пользовательские каталоги

Возможно, листинг 15-3 вас удивил если нам для создания пользовательского каталога необходимо реализовать только одно единственное свойство, то, как этот процесс может быть сложным? Проблема заключается в том, что `ComposablePartDefinition` является абстрактным типом и не имеет ни одной открытой реализации. При реализации унаследованного `ComposablePartCatalog` необходимо также реализовать пользовательский `ComposablePartDefinition`. Теперь паттерн повторяется, поскольку `ComposablePartDefinition` определяет еще один абстрактный метод, возвращаемое значение которого относится к типу, не имеющему открытой реализации. Хотя создать пользовательский каталог возможно, но в этой книге этот процесс не рассматривается.

Каталоги – существенные строительные блоки MEF. Несмотря на то, что атрибуты статичны, каталоги придают приложению большую гибкость. В MEF встроены четыре каталога, которые содержат элементы, полученные из явных типов, из одной и той же сборки или из найденных в папке сборок. В случае необходимости сочетать части из разных каталогов можно воспользоваться `AggregateCatalog`.

MEF поддерживает только возможность конфигурирования частей с помощью атрибутов. Но если нам нужно скомпоновать части, не используя атрибуты, мы всегда можем создать адаптеры, которые импортируют и экспортируют эти части. Такие адаптеры можно использовать для того, чтобы преодолеть разрыв между статической атрибутивной моделью и конфигурацией контейнера, которую мы использовали для других DI-контейнеров. Мы можем разделить существующий каталог типов на подклассы с тем, чтобы расфасовать совокупность частей или адаптеров, которые впоследствии можно сочетать с `AggregateCatalog` для компоновки приложения.

До настоящего момента мы рассматривали только то, как можно определить импортируемые и экспортируемые компоненты таким образом, что MEF мог компоновать диаграмму объектов. Существуют и другие стороны механизма внедрения зависимостей, которые мы еще не рассматривали. Один из наиболее важных вопросов – как управлять жизненным циклом объектов.

15.2. Управление жизненным циклом

В главе 8 обсуждался процесс управления жизненным циклом, в том числе наиболее универсальные стили существования, к примеру, Singleton и Transient. Сделать обзор доступных стилей существования MEF довольно легко, поскольку MEF обладает только двумя стилями существования, продемонстрированными в таблице 15-3.

Таблица 15-3: Стили существования MEF

Название	Комментарии
Shared	Этот стиль считается стилем по умолчанию, несмотря на то, что он зависит от соответствия импортируемых и экспортируемых элементов. Так в MEF называется стиль существования Singleton.
NonShared	Так в MEF называется стиль существования Transient. Контейнер отслеживает экземпляры.

Примечание

В MEF стили существования называются *политиками создания*.

Реализации в MEF стилей существования Transient и Singleton эквивалентны стилям существования, описанным в главе 8, поэтому в этой главе я не буду тратить время на их рассмотрение.

Подсказка

В MEF используемым по умолчанию является стиль Singleton. Этим он отличается от других DI-контейнеров. Как уже обсуждалось в главе 8, Singleton – самый эффективный, но не всегда самый безопасный стиль. Поэтому в MEF эффективность приоритетнее безопасности.

В MEF существует только две политики создания, а реализовать пользовательские жизненные циклы в нем невозможно, поэтому по сравнению с остальными главами части 4 этот раздел довольно краток. Вы увидите, как объявлять стили существования для частей и как высвобождать компоненты. После прочтения этого раздела вы сможете создавать политики создания в своем собственном приложении.

Сообразно остальному API контейнера MEF политика создания определяется с помощью атрибутов.

Объявление политики создания

Объявить политику создания можно посредством добавления в класс атрибута `[PartCreationPolicy]`:

```
[Export(typeof(IIngredient))]  
[PartCreationPolicy(CreationPolicy.NonShared)]  
public class SauceBéarnaise : IIngredient { }
```

Для атрибута `[PartCreationPolicy]` необходимо, чтобы вы указали значение для `CreationPolicy`. В этом примере в качестве значения `CreationPolicy` вы указали

"NonShared" для того, чтобы объявить SauceBéarnaise как Transient-компонент. Но как демонстрирует таблица 15-4, у перечисления CreationPolicy есть еще несколько вариантов.

Таблица 15-4: Значение CreationPolicy

Значение	Описание
Any	Это значение используется по умолчанию. Часть может быть как Singleton-компонентом, так и Transient. Но пока явно не будет запрашиваться значение NonShared, часть будет вести себя как Shared.
Shared	Часть является Singleton-компонентом.
NonShared	Часть всегда выступает в роли Transient –компонента.

Примечание

Атрибут [PartCreationPolicy] можно применять только к классам. Он отличается от атрибутов [Import] и [Export], которые можно применять к классам, членам классов и параметрам.

Неудивительно, что мы можем использовать значения Shared и NonShared, но вот то, что мы можем указать значение Any, возможно, слегка вас удивило. Мы вкратце рассмотрим значение Any, но сначала завершим рассмотрение значений Shared и NonShared.

Экспорт с помощью политик создания

Как мы только что обсуждали, политику создания мы задаем с помощью атрибута [PartCreationPolicy]. Если мы не указываем значение этого атрибута, то используется значение Any. Однако если мы поставим этот атрибут рядом с атрибутами [Import] и [ImportingConstructor], которые вы уже видели в этой главе, то по умолчанию будет использоваться стиль Singleton.

В этом контексте два приведенных ниже примера эквивалентны

```
[Export(typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }  
[Export(typeof(IIngredient))]  
[PartCreationPolicy(CreationPolicy.Shared)]  
public class SauceBéarnaise : IIngredient { }
```

Единственное отличие между двумя верхними строчками кода и тремя нижними заключается в том, что в нижнем примере вы явно указываете на то, что часть является Singleton-компонентом. До тех пор, пока импортер не запрашивает конкретную политику создания, поведение приведенных примеров кода будет идентично.

Разница состоит в том, что в верхнем примере не указана явная политика создания, т.е. используется значение Any. Несмотря на то, что в большинстве случаев по умолчанию используется поведение, соответствующее стилю Singleton, все намного сложнее.

Импорт с помощью требований политики создания

`CreationPolicy.Any` явно указывает на то, что политика создания части не определена, и что жизненный цикл будет определен посредством сопоставления атрибутов экспорта и импорта.

Среди многочисленных вариантов импорта зависимостей в MEF есть возможность, позволяющая нам потребовать, чтобы часть имела конкретную политику создания. Это могло бы выглядеть следующим образом:

```
[ImportingConstructor]
public Mayonnaise(
    [Import(RequiredCreationPolicy = CreationPolicy.NonShared)]
    EggYolk eggYolk,
    OliveOil oil)
{ }
```

Этот конструктор `Mayonnaise` явно указывает на то, что принимаются только свежие желтки. С точки зрения кулинарии, возможно, это и не кажется таким уж сложным, но, когда дело касается юда, это требование задает строгое ограничение для импортируемой части. Это требование компилируется в класс `Mayonnaise` посредством атрибута

`[Import]`, которым отмечается аргумент конструктора `eggYolk`. Обратите внимание на то, что этим атрибутом отмечен только атрибут `eggYolk`, что позволяет вам для создания нескольких порций майонеза брать оливковое масло из той же самой бутылки.

Предупреждение

Задание требования политики создания в компилируемом атрибуте потребителя – это разновидность анги-паттерна `Control Freak`. MEF позволяет это сделать, но вам необходимо воздержаться от такого поведения, поскольку это ограничивает ваши возможности компоновки.

Свойство `RequiredCreationPolicy` потенциально может изменить контекст сопоставления импортируемых и экспортируемых компонентов. Если мы не используем это свойство, то принимается любое значение (`Shared` и `NonShared`), но при использовании свойства `RequiredCreationPolicy` несовместимые экспортируемые элементы будут отбрасываться.

Помните ли вы таблицу 15-2, в которой описывается, как сопоставляются импортируемые и экспортируемые компоненты по мощности? Сопоставление по политике создания – это еще одна сторона алгоритма сопоставления, используемого в MEF. В таблице 15-5 демонстрируется, как сопоставляются политики создания.

Таблица 15-5: Сопоставление импортируемых и экспортируемых компонентов по политике создания

	Export.Any	Export.Shared	Export.NonShared
Import.Any	Shared	Shared	NonShared
Import.Shared	Shared	Shared	Не сопоставимы
Import.NonShared	NonShared	Не сопоставимы	NonShared

Не забывайте, что сопоставление по политике создания – это всего лишь одна из сторон сопоставления импортируемых и экспортируемых компонентов, и перед тем, как будет выполнено сопоставление, должны быть проверены все ограничения. Не указывать для импортируемых элементов политику создания – наиболее предпочтительный вариант.

В этом разделе вы увидели, что политика создания определяется с помощью атрибутов, в общем случае подобных атрибутам импорта и экспорта. Указание политик создания – первая составляющая процесса управления жизненным циклом. Но после разрешения диаграмм объектов, которые могут содержать части, имеющие смешанные жизненные циклы, мы должны не забыть их высвободить.

Высвобождение объектов

Как мы уже обсуждали в разделе 8.2.2 "Управление устраняемыми зависимостями", важно высвободить объекты после того, как мы завершили работу с ними, чтобы можно было уничтожить любые устраняемые экземпляры при завершении их жизненного цикла. В MEF выполнить это довольно легко. Мы можем явным образом высвободить экспортируемые элементы или же устранить весь контейнер, если он нам больше не нужен.

Высвобождение экспортируемых компонентов

Высвобождать экспортируемые компоненты очень легко, но особенностью MEF является то, что, несмотря на возможность высвободить экспортируемые компоненты, мы не можем высвободить экспортированные значения. В чем отличие?

Ниже приведено экспортированное значение:

```
var ingredient = container.GetExportedValue<IIngredient>();
```

Как вы уже видели в разделе 15.1.1 "Разрешение объектов", метод `GetExportedValue` возвращает экземпляр запрашиваемого типа, таким образом, `ingredient` является экземпляром `IIngredient`. Вместо того чтобы запрашивать экспортированное значение, можно запросить экспортируемый компонент:

```
var x = container.GetExport<IIngredient>();
```

Метод `GetExport` возвращает `Lazy<IIngredient>`, а не экземпляр `IIngredient`. Однако вы можете получить экспортированное значение из свойства `Value` экспортируемого компонента:

```
var ingredient = x.Value;
```

Поскольку `x` – это экземпляр `Lazy<IIngredient>`, `ingredient` является экземпляром `IIngredient`. Если вы собираетесь высвободить разрешенные компоненты, вы должны использовать экспортируемые компоненты, поскольку для `CompositionContainer` возможен только один метод `Release`:

```
public void ReleaseExport<T>(Lazy<T> export)
```

Для метода `ReleaseExport` нужен экспортируемый компонент и неэкспортированное значение. Это означает, что вы не можете напрямую высвободить экспортированное

значение посредством передачи переменной `ingredient`, а для того, чтобы высвободить его, должны использовать экспортируемый компонент:

```
container.ReleaseExport(x);
```

Поскольку `ingredient` был создан из `x`, он высвобождается, когда высвобождается подобный экспортируемый компонент. Устраняемые зависимости должным образом уничтожаются, если заканчивается их жизненный цикл. Высвобождать части столь же просто, как и вызывать метод `ReleaseExport`. Но для того чтобы иметь возможность вызывать этот метод, вы должны использовать первоначальный экспортируемый компонент.

В сценариях, когда один и тот же контейнер разрешает множество экземпляров, важно иметь возможность высвобождать экспортируемые компоненты, не устраняя при этом сам контейнер. Такой сценарий обычно используется в веб-приложениях и веб-сервисах, в которых один и тот же контейнер управляет сразу несколькими запросами. С другой стороны, в клиентских приложениях необходимо разрешать только одну диаграмму зависимостей, при этом мы можем уничтожить контейнер после закрытия приложения.

Высвобождение контейнера

Такие клиентские приложения, как WPF, Windows Forms или консольные приложения, должны руководствоваться простым паттерном Register Release Resolve, создавая только одну диаграмму объектов для всего жизненного цикла приложения. Это означает, что нам нужно всего лишь высвободить диаграмму объектов после окончания жизненного цикла приложения.

Несмотря на то, что мы можем высвободить экспортируемый компонент с помощью метода `ReleaseExport`, проще устранить сам контейнер, и при этом нам не придется хранить ссылку на экспортируемый компонент. После закрытия приложения контейнер уже не нужен, поэтому мы можем соответствующим образом высвободить все части посредством уничтожения самого контейнера:

```
container.Dispose();
```

При устранении контейнера высвобождаются все части, при этом устраняются как Singleton-, так и Transient-компоненты.

На этом наше путешествие по процессу управления жизненным циклом в MEF подходит к концу. Части можно компоновать из компонентов со смешанными стилями существования, и это справедливо даже тогда, когда мы определяем несколько экспортируемых компонентов одной и той же абстракции. Мы еще не рассматривали процесс работы с составными частями, поэтому давайте обратим наше внимание на эту тему.

15.3. Работа с составными компонентами

DI-контейнеры процветают благодаря их индивидуальности, но их неопределенность порождает ряд трудностей. При использовании Constructor Injection единичный конструктор предпочтительнее перегружаемых конструкторов, поскольку в этом случае ясно, какой конструктор использовать в ситуации, когда у вас нет выбора. То же самое касается и преобразования абстракций к конкретным типам. Если мы пытаемся преобразовать конкретные составные типы к одной и той же абстракции, это приводит к неопределенности.

Несмотря на столь нежелательную особенность как неопределенность, нам часто приходится работать с составными реализациями единичного интерфейса. Это может происходить в следующих ситуациях:

- Для разных потребителей должны использоваться разные специфичные типы
- Зависимости являются последовательностями
- Используются Decorator'ы

В этом разделе мы рассмотрим каждую из этих ситуаций и увидим, как MEF поочередно справляется с каждой из них. После прочтения раздела вы должны будете уметь добавлять атрибуты и успешно высвобождать части даже в тех ситуациях, когда в игру вступают составные реализации одной и той же абстракции.

Примечание

В этом разделе мы не обсуждаем процесс перехвата во время выполнения, поскольку MEF не поддерживает этот механизм.

Иногда нам необходим более тщательный контроль, нежели тот, который предоставляет механизм автоматической интеграции. В следующем разделе описывается, как можно обеспечить более тщательный контроль при работе с MEF.

Выбор среди составных кандидатов

Автоматическая интеграция – удобный и мощный инструмент, но предоставляет нам меньшие возможности контроля. Пока все абстракции преобразуются в конкретные типы отдельно друг от друга, никаких трудностей не возникает, но как только мы вводим большее количество реализаций для одного и того же интерфейса, возникает неопределенность.

Для начала мы должны повторить то, как MEF справляется с составными экспортируемыми компонентами одной и той же абстракции.

Работа с составными реализациями одного и того же экспортируемого компонента

Как вы уже видели в разделе 15.1.2 "Определение экспортируемых и импортируемых элементов", можно создать составные части одного и того же экспорта:

```
[Export(typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }  
[Export(typeof(IIngredient))]  
public class Steak : IIngredient { }
```

В этом примере и `SauceBéarnaise`, и `Steak` определяются как экспортируемые компоненты типа `IIngredient`. Однако в отличие от большинства остальных DI-контейнеров в MEF не используется понятие компонента по умолчанию. Существует либо единственный экспортируемый компонент части, либо несколько компонентов. Это и есть та мощьность, которая проиллюстрирована в таблице 15-2. Если вы экспортируете и `SauceBéarnaise`, и `Steak` как `IIngredient`, то получаете составные экспортируемые компоненты, а разрешить их вы можете только посредством импорта составных экземпляров.

При наличии двух экспортируемых компонентов попытка разрешить один экземпляр `IIngredient` приводит к исключению:

```
var ingredient = container.GetExportedValue<IIngredient>();
```

В этом случае выдается исключение, поскольку существуют составные экспортируемые компоненты `IIngredient`, и MEF отказывается отбирать только один из них. Это имеет значение при рассмотрении ключевого сценария MEF: в сценариях расширяемости обычно мы будем получать экспортируемые компоненты из сборок, находящихся в соответствующей папке. Экспортируемые компоненты являются расширениями, поэтому во время проектирования мы еще не знаем, какие экспортируемые компоненты будут нам доступны, если они вообще будут нам доступны. В таком контексте нет смысла отбирать только один экспортируемый компонент, теряя при этом остальные. Мы должны либо сделать экспортируемые компоненты неопределенными каким-то другим способом, либо должны уметь справляться с множественностью. Далее мы снова ненадолго вернемся к вопросу импорта составных компонентов, но сначала рассмотрим варианты наделения экспортируемых компонентов наибольшей индивидуальностью.

Один из способов сделать экспортируемый компонент более индивидуальным – присвоить ему имя. Перегрузка конструктора атрибута `[Export]` позволяет присвоить экспортируемому компоненту имя:

```
[Export("sauce", typeof(IIngredient))]  
public class SauceBéarnaise : IIngredient { }  
[Export("meat", typeof(IIngredient))]  
public class Steak : IIngredient { }
```

В этом примере вместо двух частей, порождающих один и тот же экспортируемый компонент (`IIngredient`), определяются два разных экспортируемых компонента: один из которых экспортирует сочетание `IIngredient` и имени "sauce", а другой – сочетание `IIngredient` и имени "meat". Теперь нет ни одного неименованного контракта `IIngredient`.

Примечание

При экспорте только именованных типов отсутствуют экспортируемые компоненты неименованного типа.

При попытке разрешить неименованный `IIngredient` будет выдаваться исключение:

```
var ingredient = container.GetExportedValue<IIngredient>();
```

Несмотря на то, что этот код приводит к выдаче исключения, оба именованных экспортируемых компонента можно разрешить, выполнив запрос конкретно этих компонентов:

```
var meat = container.GetExportedValue<IIIngredient>("meat");  
var sauce = container.GetExportedValue<IIIngredient>("sauce");
```

Явное разрешение именованного экспортируемого компонента с помощью соответствующей перегрузки `GetExportedValue` – хороший способ продемонстрировать, как разрешаются части, но если мы руководствуемся паттерном `Register Release Resolve`, то нет необходимости запрашивать таким способом конкретный именованный компонент.

Подсказка

Если вы обнаружите, что вызываете метод `GetExportedValue` с конкретным именем или идентификатором, подумайте над тем, сможете ли вы сменить свой подход на менее неопределенный.

Когда части определяются таким образом, чтобы сопоставляться с данным потребителем, для осуществления выбора среди нескольких вариантов можно использовать именованные экспортируемые компоненты.

Импорт именованных экспортируемых компонентов

Иногда бывает необходимо переопределить обычное поведение для того, чтобы обеспечить более разветвленный контроль над тем, куда какая зависимость отправляется. Кроме того, возможны ситуации, при которых приходится сталкиваться с неопределенным API. В качестве примера рассмотрим следующий конструктор:

```
public ThreeCourseMeal(ICourse entrée,  
    ICourse mainCourse, ICourse dessert)
```

В этом примере присутствуют три одинаковым образом типизированных зависимости, каждая из которых является отдельной, не похожей на другие сущностью. В большинстве случаев вам необходимо, чтобы три, отличающихся друг от друга, экспортируемых компонента заполнялись соответствующими параметрами. В следующем листинге продемонстрировано, как можно отметить нужные классы, чтобы создать соответствующие экспортируемые компоненты.

Листинг 15-4: Определение именованных экспортируемых компонентов

```
[Export("entrée", typeof(ICourse))]  
public class Rillettes : ICourse { }  
[Export("mainCourse", typeof(ICourse))]  
public class CordonBleu : ICourse { }  
[Export("dessert", typeof(ICourse))]  
public class MousseAuChocolat : ICourse { }
```

Вы помечаете `Rillettes` экспортируемым компонентом под названием `"entrée"`, класс `CordonBleu` – экспортируемым компонентом под названием `"mainCourse"`, а `MousseAuChocolat` – экспортируемым компонентом под названием `"dessert"`.

Принимая во внимание эти экспортируемые компоненты, теперь вы можете следующим образом отметить конструктор класса `ThreeCourseMeal` соответствующими атрибутами `[Import]`:

```
[ImportingConstructor]
public ThreeCourseMeal(
    [Import("entrée", typeof(ICourse))] ICourse entrée,
    [Import("mainCourse", typeof(ICourse))] ICourse mainCourse,
    [Import("dessert", typeof(ICourse))] ICourse dessert)
```

Обратите внимание на то, что вы можете применить атрибут `[Import]` к аргументам конструктора. Обычно в тех случаях, когда конструктор уже отмечен атрибутом `[ImportingConstructor]`, вам не нужно явным образом применять атрибут `[Import]` к аргументам конструктора. Но в этом примере вам необходимо отметить каждый параметр атрибутом, чтобы они соответствовали разным именованным экспортируемым компонентам. Поскольку в листинге 15-4 присутствуют соответствующие экспортируемые компоненты, из этих частей вы теперь можете успешно разрешить класс `ThreeCourseMeal`.

Подсказка

Если вы не можете (или не хотите) явным образом изменять классы, можете вместо них создать экспортирующие адаптеры.

Метаданные

Сопоставление импортируемых и экспортируемых компонентов с помощью именованных контрактов – отличный способ справиться с неопределенностью. Однако использование таких жестко закодированных строк приводит к небезопасному рефакторингу. Можно попытаться исправить это, определив константы, которые мы будем использовать вместо жестко закодированных строк, но это не дает гарантии, что все разработчики будут помнить об использовании этих констант.

Еще один вариант – использовать такую возможность MEF, как метаданные. Эта возможность позволяет определять пользовательские атрибуты экспорта, инкапсулирующие дополнительные метаданные, которые мы собираемся добавить в экспортируемый компонент. Полноценная трактовка метаданных не будет рассматриваться в этой книге, но если вы хотите узнать о них больше, обратитесь к статье Глена Блока "Managed Extensibility Framework" в журнале "MSDN Magazine".

Чтобы избавиться от неопределенности, мы всегда можем сопоставить именованные импортируемые компоненты со схожими именованными экспортируемыми компонентами, но лучшим решением будет сконструировать вышеупомянутое API. Это приводит и к лучшему дизайну в целом.

В следующем разделе вы увидите, как можно использовать менее неопределенный и более гибкий подход, при котором разрешается использовать в обеде несколько блюд. Теперь вы должны изучить то, как MEF работает со списками и последовательностями.

Интеграция последовательностей

В разделе 10.3.2 "Разработка пользовательского стиля существования" мы обсуждали, как выполнить рефакторинг явного класса `ThreeCourseMeal` к более универсальному классу `Meal`, который обладает приведенным ниже конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

В этом разделе мы рассмотрим то, как можно сконфигурировать MEF, чтобы он интегрировал экземпляры `Meal` с соответствующими зависимостями `ICourse`. После рассмотрения этого вопроса вы должны будете приобрести хорошее понимание тех возможностей, которые доступны в случае необходимости конфигурирования частей, имеющих последовательности зависимостей.

Автоматическая интеграция последовательностей

Как мы уже говорили в разделах 15.1.2 "Определение экспортируемых и импортируемых элементов" и 15.3.1 "Выбор среди составных кандидатов", в MEF используется такое понятие, как мощност. Это также означает, что MEF разбирается в составных импортируемых и экспортируемых компонентах, но чтобы мы должны их явно определять. В разделе 15.1.2 вы видели, как нужно применять атрибут `[ImportingConstructor]`, чтобы обеспечить возможность использования паттерна `Constructor Injection`. Несмотря на то, что нужно применять к конструктору `Meal` атрибут `[ImportingConstructor]`, этого недостаточно. Такое поведение указывает MEF на то, что конструктор `Meal` необходимо использовать для композиции, но при этом предполагается использовать импортируемый компонент `IEnumerable<ICourse>`.

Экспортировать части `ICourse` можно так, как это продемонстрировано в листинге 15-4. Однако поскольку теперь вы не хотите явно различать эти части, ни одной из них не присваивается имя:

```
[Export(typeof(ICourse))]  
public class Rilletes : ICourse { }  
[Export(typeof(ICourse))]  
public class CordonBleu : ICourse { }  
[Export(typeof(ICourse))]  
public class MousseAuChocolat : ICourse { }
```

Обратите внимание, что единственное отличие от листинга 15-4 заключается в том, что ни один из экспортируемых компонентов не имеет имени. Сейчас у вас имеются составные экспортируемые компоненты типа `ICourse`, но это, само по себе, еще не устраняет несоответствия между составными экспортируемыми компонентами `ICourse` и единственным импортируемым компонентом `IEnumerable<ICourse>`. Последним шагом будет применение атрибута `[ImportMany]`:

```
[ImportingConstructor]  
public Meal([ImportMany]IEnumerable<ICourse> courses)
```

Атрибут `[ImportMany]` используется для того, чтобы явно преобразовать составные экспортируемые компоненты в единственный оператор импорта последовательностей. Экспортируемые компоненты могут браться из разных сборок, но при этом они будут компоноваться в одну последовательность. При разрешении `IMeal` вы получаете

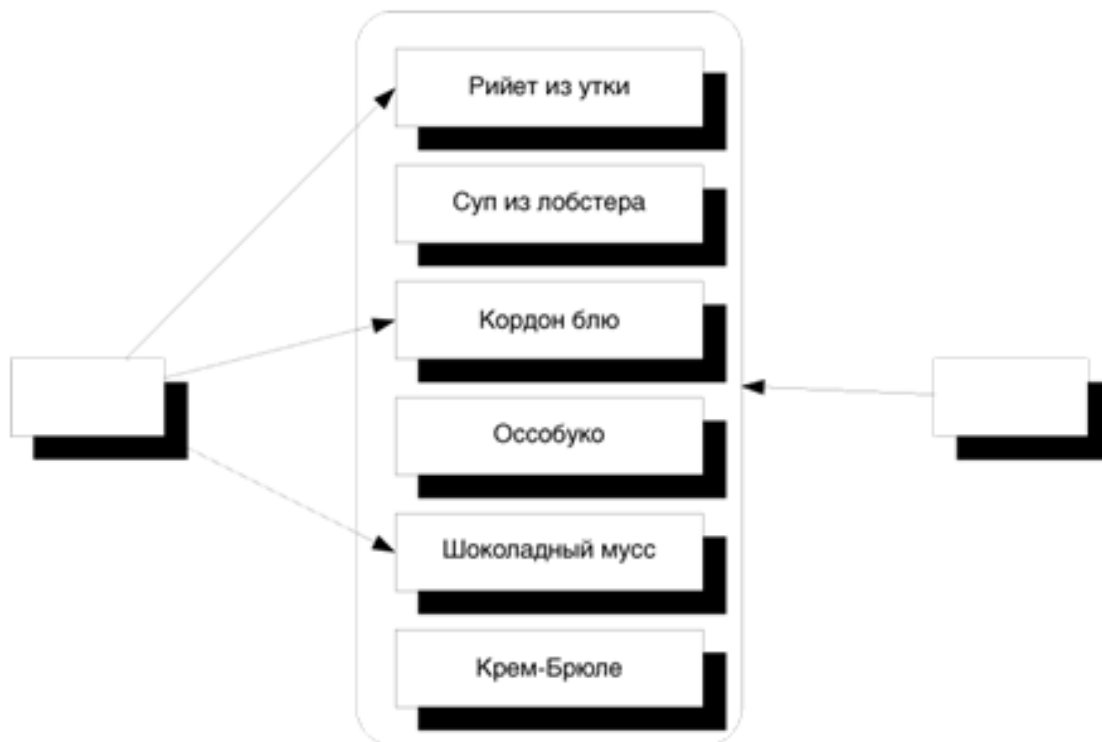
экземпляр `Meal`, имеющий три экспортируемых компонента `ICourse:Rillettes`, `CordonBleu` и `MousseAuChocolat`.

С помощью атрибута `[ImportMany]` часть может импортировать последовательность всех соответствующих экспортируемых компонентов. Только когда нам нужно из большого набора экземпляров явно отобрать только несколько из них, нам необходимо выполнить больше действий. Рассмотрим, как это сделать.

Отбор нескольких экспортируемых компонентов из большого набора

Когда мы имеем дело с множественностью экспортируемых компонентов, та стратегия, которая подразумевается под использованием атрибута `[ImportMany]`, является корректной линией поведения. Это позволяет сопоставить импортеру все экспортируемые компоненты необходимого контракта. Но, как показывает рисунок 15-5, возможны случаи, когда из большого набора всех экспортируемых компонентов нам необходимо отобрать только некоторые из них.

Рисунок 15-5: В ситуации, продемонстрированной слева, мы хотим явным образом отобрать определенные зависимости из большого списка всех экспортируемых компонентов. Это отличается от ситуации, приведенной справа, когда мы отбираем все без разбора.



Когда мы ранее позволяли MEF автоматически интегрировать все экспортируемые компоненты, это соответствовало бы ситуации, изображенной в правой части рисунка 15-5. Если нам нужно скомпоновать часть так, как изображено в левой части рисунка, то мы должны явно определить, какие экспортируемые компоненты необходимо использовать.

Единственный способ это сделать – еще раз прибегнуть к именованным экспортируемым компонентам. Однако выбранная нами стратегия немного отличается от используемой в листинге 15-4, поскольку теперь для того, чтобы отметить все те экспортируемые

компоненты, которые необходимо импортировать в класс `Meal`, мы собираемся использовать именованный атрибут экспорта. Как показано в следующем листинге, это не исключает экспорта других контрактов, а также контракта, лежащего в основе совокупности компонентов.

Листинг 15-5: Отбор экспортируемых компонентов из набора

```
1. [Export(typeof(ICourse))]  
2. [Export("meal", typeof(ICourse))]  
3. public class Rillettes : ICourse { }  
4. [Export(typeof(ICourse))]  
5. public class LobsterBisque { }  
6. [Export(typeof(ICourse))]  
7. [Export("meal", typeof(ICourse))]  
8. public class CordonBleu : ICourse { }  
9. [Export(typeof(ICourse))]  
10. [Export("meal", typeof(ICourse))]  
11. public class MousseAuChocolat : ICourse { }
```

Строка 2, 7, 10: Искомые экспортируемые компоненты

Строка 1, 6, 9: Обычные атрибуты экспорта

Строка 4-5: Не выполняется отбор компонентов

Все три класса `Rillettes`, `CordonBleu` и `MousseAuChocolat` экспортируют контракт с именем "meal". Этот именованный контракт можно использовать для импорта только тех частей, которые экспортируют этот конкретный контракт. Однако для других потребителей, которым могут понадобиться все экспортируемые компоненты `ICourse` независимо от их имени, можно также экспортировать эти три класса как неименованный контракт `ICourse`. К части можно добавлять сколько угодно атрибутов `[Export]`.

Класс `LobsterBisque` экспортирует не именованный контракт `meal`, а только неименованный контракт `ICourse`. Это означает, что те потребители, которые собираются импортировать все экспортируемые компоненты `ICourse`, могут сделать это с помощью атрибута `[ImportMany]` по умолчанию. Однако вы все равно можете установить, что часть импортирует только те части, которые явно экспортируют именованный контракт `meal`:

```
[ImportingConstructor]  
public Meal(  
    [ImportMany("meal", typeof(ICourse))] IEnumerable<ICourse> courses)
```

Вместо конструктора по умолчанию атрибута `[ImportMany]` можно использовать перегрузку конструктора, которая позволяет импортировать только именованный контракт. Атрибут помечает параметр `courses`, что означает, что в последовательность `courses` будут отбираться только те части, которые экспортируют именованный контракт `meal`. Благодаря экспортируемым компонентам из листинга 15-5 вы получите `Meal`, содержащий `Rillettes`, `CordonBleu` и `MousseAuChocolat`, но без `LobsterBisque`.

Именованные экспортируемые компоненты можно использовать в качестве маркеров таким образом, чтобы помеченные экспортируемые компоненты можно было выборочно отбирать в потребители. Поскольку можно использовать сколько угодно атрибутов `[Export]`, помечать экспортируемые компоненты можно для разных целей.

В обоих случаях, продемонстрированных на рисунке 15-5, атрибут `[ImportMany]` является ключевым элементом, позволяющим импортировать множественные экспортируемые компоненты в одного потребителя. Импорт последовательностей – хороший способ избавиться от неопределенности, а используемые в MEF понятие мощности и явные атрибуты делают этот процесс более понятным.

Потребители, которые полагаются на последовательности зависимостей, могут быть самым интуитивно понятным применением составных экспортируемых компонентов одной и той же абстракции. Но перед тем как мы полностью отойдем от данной темы, нам необходимо рассмотреть последний (и, возможно, слегка неожиданный) случай, когда в дело вступают составные экземпляры.

Интеграция Decorator'ов

В разделе 9.1.2 "Паттерны и принципы механизма перехвата" мы обсуждали то, насколько паттерн проектирования Decorator полезен при реализации сквозных сущностей. По определению Decorator'ы представляют собой составные типы одной и той же абстракции. У нас есть, по крайней мере, две реализации абстракции: сам Decorator и вложенный в него тип. Если бы мы помещали Decorator'ы в стек, то у нас было бы еще больше реализаций.

Это еще один пример составных экспортируемых компонентов одного и того же контракта. В отличие от предыдущих разделов эти экспортируемые компоненты не являются концептуально равносильными, а зависят друг от друга. В этом разделе вы увидите, как сконфигурировать части, чтобы они могли использовать этот паттерн. В рамках MEF компоновать Decorator'ы можно несколькими способами, но поскольку все они похожи друг на друга, мы рассмотрим только один из них.

Создание обертки с помощью конкретных контрактов

Рассмотрим проверенный класс `Breading`, который является Decorator'ом для `IIngredient`. Для получения экземпляра, оберткой для которого он должен стать, класс `Breading` использует паттерн Constructor Injection:

```
public Breading(IIngredient ingredient)
```

Чтобы получить `Cotoletta`, вам необходимо будет вложить `VealCutlet` (еще один `IIngredient`) в класс `Breading`. Один из способов это сделать – связать `VealCutlet` с классом `Breading`, используя конкретный класс `VealCutlet` в качестве контракта:

```
[Export(typeof(VealCutlet))]  
public class VealCutlet : IIngredient { }
```

Обратите внимание, что часть `VealCutlet` экспортирует не `IIngredient`, а только конкретный тип, даже если она реализует интерфейс. Теперь конструктор `Breading` может явно утверждать, что он импортирует конкретный контракт `VealCutlet`:

```
[ImportingConstructor]  
public Breading(  
    [Import(typeof(VealCutlet))]  
    IIngredient ingredient)
```


MEF сопоставляет экспортируемые и импортируемые компоненты, поэтому до тех пор, пока отсутствует неопределенное сопоставление, композиция выполняется успешно. `VealCutlet` реализует `IIngredient`, поэтому даже если в алгоритме сопоставления в качестве контракта используется конкретный тип, части все равно можно сравнивать. Однако обратите внимание, что компилятор этого не гарантирует.

Примечание

Этот подход концептуально схож с подходом, описанным в разделе 14.3.3, где мы использовали `Unity` для компоновки `Breading` и `VealCutlet` через класс `VealCutlet`.

Примечание

Поскольку атрибут компилируется в класс, будет проще изменить конструктор `Breading` таким образом, чтобы он принимал в качестве параметра `VealCutlet`, а не `IIngredient`. По-моему, это хорошая демонстрация недостатков использования атрибутов для управления компоновкой.

Несмотря на то, что класс `VealCutlet` реализует `IIngredient`, он его не экспортирует. Это существенная составляющая этого подхода. Если бы `VealCutlet` экспортировал `IIngredient`, то это приводило бы к неопределенности, поскольку `Breading` уже экспортирует интерфейс. Это приводило бы к несовпадению по мощности, потому что в этом случае имелось бы два экспортируемых компонента `IIngredient`, и, импортируя `IIngredient`, вы не смогли бы разрешить `Breading`.

Предупреждение

Нельзя интегрировать `Decorator`'ы, если вложенный экспортируемый компонент должен экспортировать и его абстракцию.

Части `Breading` и `VealCutlet` компонуются вместе, поскольку обладают сопоставляемыми контрактами. Точная форма контракта не так важна. В этом примере вы использовали конкретный тип вложенного класса, но могли бы использовать именованный контракт или, как в этом конкретном случае, любую явную строку. Важно, что соответствие между двумя частями является определенным.

MEF позволяет работать с составными экспортируемыми компонентами несколькими способами. Мы можем конфигурировать их в виде альтернатив друг другу, в виде пиров, которые разрешаются в виде последовательностей, или в виде иерархических `Decorator`'ов. В любом случае мы должны явно указывать MEF на то, как сопоставлять экспортируемые и импортируемые компоненты.

Кроме того, это может происходить в ситуации, когда нам нужно иметь дело с API, отклоняющимися от `Constructor Injection`. До настоящего момента вы наблюдали за тем, как компоновать части, включая то, как определять политики создания и как работать с составными экспортируемыми компонентами. Но до этого момента мы позволяли контейнеру подключать зависимости, явным образом предполагая, что все компоненты используют `Constructor Injection`. Поскольку это не всегда происходит именно так, в следующем разделе мы сделаем краткий обзор того, как работать с классами, экземпляры которых должны создаваться особым образом.

15.4. Конфигурирование сложных API

До настоящего момента мы рассматривали то, как можно компоновать части, использующие Constructor Injection. Одним из главных преимуществ Constructor Injection является то, что DI-контейнеры могут с легкостью понимать, как компоновать и создавать все классы диаграммы зависимостей. В MEF, с другой стороны, необходимо явно использовать атрибут `[ImportingConstructor]`, поэтому для MEF это не совсем справедливо.

В этом разделе вы увидите, как работать с простейшими аргументами конструктора, статическими фабриками и Property Injection. Все это требует особого внимания. Начнем с рассмотрения классов, конструкторы которых принимают в качестве параметров простейшие типы, например, строки и целые числа.

Конфигурирование простейших зависимостей

Пока мы внедряем абстракции в потребителей, все в порядке. Но данный процесс усложняется, если конструктор зависит от простейшего типа, например, строкового, числового или перечисляемого. Наиболее часто это случается в реализациях доступа к данным, которые принимают в качестве параметра конструктора строку соединения. Но в то же время это является более общей проблемой, касающейся всех строковых и числовых типов.

В сущности, регистрация строкового или числового типа в качестве контракта не имеет особого смысла. Что значит импортировать строку или число, если мы руководствуемся только типом? Нужна ли нам любая строка? В основном нам нужна конкретная строка, например, строка соединения. Такие же рассуждения можно применить и к простейшему значению, включая строки, числа и перечисления.

Рассмотрим в качестве примера приведенный ниже конструктор:

```
public ChiliConCarne (Spiciness spiciness)
```

В этом примере `Spiciness` имеет перечисляемый тип:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

Предупреждение

Согласно эмпирическому правилу перечисления являются code smell'ами и их нужно преобразовывать в полиморфные классы (имеющие разное состояние). Тем не менее, для данного примера они вполне нам подходят.

Чтобы соответствующим образом пометить `ChiliConCarne`, вы добавляете к конструктору атрибут `[ImportingConstructor]`. При экспорте `Spiciness` лучше всего сделать это с помощью адаптера:

```
public class SpicinessAdapter
{
    [Export]
    public Spiciness Spiciness
    {
        get { return Spiciness.Hot; }
    }
}
```

Этот адаптер экспортирует значение `Spiciness.Hot` таким образом, что, если вы компонуете `ChiliConCarne` из каталога, в котором содержатся эти части, то вы получите горячее *Chili con Carne*.

Подсказка

Вместо того чтобы экспортировать и импортировать сам тип `Spiciness`, можно использовать пользовательскую строку в качестве совместно используемого контракта. Для этого потребуется добавить дополнительный атрибут `[Import]` в аргумент конструктора `spiciness` с целью определения контракта.

С помощью адаптеров и контрактов можно должным образом сопоставить простейшие типы и импортируемые компоненты. Такой подход можно использовать в ситуациях, когда все типы и конструкторы являются открытыми. А что нам делать с типами, у которых нет открытых конструкторов?

Компоновка частей без открытых конструкторов

Экземпляры некоторых классов нельзя создать с помощью открытого конструктора. Вместо открытых конструкторов для создания экземпляров такого типа нам приходится использовать некоторого рода фабрику. Для DI-контейнеров это всегда проблематично, поскольку по умолчанию им нужны открытые конструкторы.

Рассмотрим приведенный ниже пример конструктора открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже если класс `JunkFood` является открытым, конструктор расположен внутри него. Очевидно, экземпляры `JunkFood` должны создаваться с помощью статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static IMeal Create(string name)
    {
        return new JunkFood(name);
    }
}
```

Предположим, что вы не можете изменить это API. Как вы тогда поступите в этой ситуации, чтобы суметь должным образом интегрировать и компоновать `JunkFood`? Ответ на этот вопрос точно такой же, как и для всех остальных случаев, когда вы не можете изменить первоначальный экспортированный тип: используйте адаптер подобно тому, как это продемонстрировано в следующем листинге.

Листинг 15-6: Экспорт типа, имеющего внутренний конструктор

```
public class JunkFoodAdapter
{
    private readonly IMeal junk;
    public JunkFoodAdapter()
    {
        this.junk = JunkFoodFactory.Create("chicken meal");
    }
    [Export]
    public IMeal JunkFood
    {
        get { return this.junk; }
    }
}
```

В `JunkFoodAdapter` инкапсулировано знание о том, что экземпляр `JunkFood` создается с помощью метода `JunkFoodFactory.Create`. Этот метод создает экземпляр в конструкторе и импортирует его через свойство. Поскольку тип свойства – `IMeal`, то он также является и экспортированным контрактом.

С помощью расположенного в каталоге класса `JunkFoodAdapter` вы можете успешно разрешать `IMeal` и возвращать экземпляра блюда из курицы `JunkFood`.

Последним рассматриваемым нами отклонением от `Constructor Injection` является `Property Injection`.

Интегрирование с помощью `Property Injection`

`Property Injection` – это менее определенная форма механизма внедрения зависимостей, поскольку компилятор не принуждает нас задавать значение свойства, доступного для записи. По иронии `MEF` был задуман скорее как использующий паттерн `Property Injection`, а не `Constructor Injection`. Это объясняет тот факт, что нам не нужно явно применять атрибуты ко всему, что мы собираемся скомпоновать: с точки зрения `MEF`, паттерн `Property Injection` (являющийся неопределенным) является используемым по умолчанию, а `Constructor Injection` – менее идиоматичным вариантом.

Несмотря на то, что я считаю эту точку зрения одновременно и устаревшей, и неверной, это все же позволяет без затруднений использовать в `MEF` паттерн `Property Injection`. Все, что нам приходится делать, – применять к свойству атрибут `[Import]`.

Рассмотрим класс `CaesarSalad`:

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

По всеобщему заблуждению в состав салата "Цезарь" входит курица. По своей сути "Цезарь" является салатом, но, поскольку с курицей он вкуснее, то ее часто предлагают использовать в нем в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует такую возможность посредством доступного для записи свойства под названием `Extra`.

Чтобы разрешить использовать для CaesarSalad паттерн Property Injection, вам необходимо просто применить атрибут [Import]:

```
[Import(AllowDefault = true)]
public IIngredient Extra { get; set; }
```

В этой книге я последовательно рассматриваю паттерн Property Injection, который применяется в тех случаях, когда есть возможность передавать внешние зависимости. Это имеет смысл, поскольку компилятор не принуждает вас присваивать свойству значение (в отличие от аргумента конструктора). Но MEF не придерживается этой точки зрения. По умолчанию импорт *должен* выполняться, пока вы явно с помощью свойства AllowDefault не укажете, что делать это необязательно. Чтобы продолжать соответствовать описанному выше паттерну Property Injection, свойству AllowDefault вы присваиваете значение true. Это означает, что MEF не будет выдавать исключение, если не сможет импортировать IIngredient.

Вы должны знать, что если свойство AllowDefault имеет значение true, то вместо того, чтобы игнорировать свойство, когда импорт не может быть выполнен, MEF будет явным образом присваивать свойству значение по умолчанию (в этом случае null). Чтобы применить эту возможность, вы должны быть готовы к работе с null-значениями, но это приведет к разрушению инвариантов класса. Вы должны использовать большие значения, чтобы избежать присваивания null-значений приватным полям.

Один из способов работы с null-значениями – молча поглотить такое значение:

```
1. [Import(AllowDefault = true)]
2. public IIngredient Extra
3. {
4.     get { return this.extra; }
5.     set
6.     {
7.         if (value == null)
8.         {
9.             return;
10.        }
11.        this.extra = value;
12.    }
13. }
```

Строка 7-10: Молчаливое игнорирование null-значений

Вы можете явным образом проверить, имеет ли свойство значение null, и выйти, если вызывающий оператор пытается внедрить null-значение. Такое поведение приводит к нарушению принципа "наименьшего удивления" (*Principle of Least Surprise*), поскольку вызывающих операторов может удивить тот факт, что присваивание значения не дало никакого результата, даже если при этом не выдавалось никакого исключения. И снова вы пришли к тому, что Property Injection – очень проблематичный паттерн, и лучше всего избегать его использования, пока оно не будет оправданно.

Известно, что Property Injection характерен для MEF, но, как часто бывает, все зло кроется в деталях. Даже при работе с MEF я предпочитаю использовать Constructor Injection, мой любимый паттерн.

В этом разделе вы увидели, как можно использовать MEF для работы с более трудными API разработки. Процесс применения Property Injection, а также всего остального, к чему мы можем обратиться с помощью экспортируемых адаптеров, довольно прост. Такой подход всегда является универсальным решением, если все перестает работать, а мы не можем модифицировать части.

15.5. Резюме

Среди всех остальных DI-контейнеров, охваченных в части 4, MEF является самым особенным. Во-первых, это единственная технология компоновки, официально поставляемая и поддерживаемая компанией Microsoft. Во-вторых, MEF, в действительности, является не DI-контейнером, а фреймворком расширяемости (как видно из его названия *Managed Extensibility Framework*), поэтому рассматривать его с той позиции, как если бы он был DI-контейнером, было бы не правильно.

Между MEF и другими DI-контейнерами столько схожих черт, что эта глава не только является обоснованной, но и необходимой. Вы должны четко понимать, почему MEF не является DI-контейнером, чтобы принять правильное решение о том, когда его можно использовать.

Эта глава продемонстрировала, что мы можем выдавить из MEF большую часть функциональности обычного DI-контейнера, но при этом используя достаточно неудобные способы. Наиболее проблемная сторона MEF – зависимость от атрибутов, потому что атрибуты сильно привязывают к типам такие вопросы, как жизненный цикл и отбор импортируемых компонентов из совокупности компонентов. В сценариях расширяемости это не является проблемой, но в процессе компоновки полноценного приложения это ограничение приводит к возникновению трудностей.

В ситуациях, когда мы не можем или не хотим помечать типы атрибутами MEF, мы можем создавать адаптеры, которые импортируют и экспортируют соответствующие части ради реальной реализации. Мы можем рассматривать такие адаптеры MEF в качестве конфигурационного API, но в отличие от большинства других строго типизированных свободных интерфейсов DI-контейнеров это API довольно громоздко. Однако адаптер MEF – универсально используемая хитрость, которую можно использовать, чтобы справиться с конкретными проблемами, возникающими в MEF. Можно использовать адаптеры не только для компоновки не атрибутивных типов, но также для экспорта частей из методов фабрики и для много другого.

Имеет ли смысл использовать в приложениях MEF в качестве DI-контейнера? Ответ на этот вопрос зависит от обстоятельств. Одним из сильных аргументов в пользу использования MEF является тот факт, что MEF является частью .NET 4 и Silverlight 4, поэтому если для приложения будут использоваться эти платформы, MEF в них уже будет доступен. Дело не только в удобстве применения. Для тех организаций, в которых согласно их внутренней политике должны использоваться только официальные технологии компании Microsoft, использование MEF может стать большим преимуществом.

Поскольку MEF – это официальный продукт компании Microsoft, нам предоставляется другой уровень поддержки, нежели тот, который мы получали при использовании других DI-контейнеров. Нам предоставляется такая же поддержка, как и при использовании .NET и Silverlight, при этом мы можем быть уверены, что в MEF не возникнет неполадок.

Однако эти преимущества могут не перевесить недостатки использования MEF в тех ролях, для которых она не была сконструирована. MEF был создан для сценариев расширяемости, поэтому имеет смысл использовать его в приложениях, в которых расширяемость является важной чертой. В таких приложениях, возможно, имеет смысл

расширить ответственность MEF до композиции всего приложения, поскольку эта возможность уже используется.

Для приложений, в которых расширяемость не играет значительной роли, возможно, имеет смысл выбрать специально предназначенный для композиции объектов DI-контейнер.

Неважно, какой DI-контейнер вы выбрали, или даже если вы предпочитаете Poor Man's DI, я надеюсь, что эта книга передала вам следующую мысль: механизм внедрения зависимостей не основывается на какой-то конкретной технологии, например, на конкретном DI-контейнере. Приложения могут и должны создаваться с помощью DI-дружественных паттернов и приемов, описанных в этой книге. Если мы в этом преуспели, то огромное значение имеет выбор DI-контейнера. DI-контейнер – это инструмент, который компонует наше приложение, но в идеале мы должны уметь заменять один контейнер на другой, не переписывая при этом другие части нашего приложения, кроме как Composition Root.