ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ В .NET

Марк Симан

MANNING

ПИТЕР®

Dependency Injection in .NET

MARK SEEMANN

MANNING
SHELTER ISLAND

ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ В .NET

Марк Симан

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.973.2-018
УДК 004.42
С37

Марк Симан

С37 Внедрение зависимостей в .NET. — СПб.: Питер, 2014. — 464 с.: ил.

ISBN 978-5-496-00657-6

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.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

ISBN 978-1935182504 англ.
ISBN 978-5-496-00657-6

Краткое содержание

Часть 1. Последовательность изучения внедрения зависимостей

Часть 2. Каталог паттернов внедрения зависимостей

Часть 3. Самостоятельное создание внедрения зависимостей

Часть 4. Контейнеры внедрения зависимостей

Оглавление

Часть 1. Последовательность изучения внедрения зависимостей

Часть 2. Каталог паттернов внедрения зависимостей

Часть 3. Самостоятельное создание внедрения зависимостей

Часть 4. Контейнеры внедрения зависимостей

Сесиль.
Я бы не сделал это без тебя

Предисловие

Мое первое знакомство с внедрением зависимостей состоялось почти 10 лет тому назад. Я трудился в одной ISV-компании (Independent Software Vendor, независимый поставщик программного обеспечения) в качестве архитектора фреймворка уровня предприятия для наших приложений. Тогда возникало ощущение, что все мои друзья, работавшие в индустрии программного обеспечения, разрабатывали похожие фреймворки. Фреймворк работал на разных уровнях в многоуровневых приложениях и поддерживал доступ к данным, бизнес-логику и пользовательский интерфейс. Бизнес-объекты, поставляемые как продукты, должны обеспечивать сохранность при работе с разными базами данных и быть представлены в различных пользовательских интерфейсах; проблема была в поиске способа сделать систему расширяемой и хорошо поддерживаемой. Мы нашли ответ, обратившись к внедрению зависимостей. С его помощью мы четко определили контракты между уровнями, что позволило нам упростить тестирование уровней, а также менять реализации зависимостей без изменения кода.

Марк говорит в этой книге о «внедрении зависимостей для бедных», и это было именно то, что делали мы. В те дни у нас не было контейнеров внедрений. У нас также не было руководств типа того, что вы найдете в этой книге. Как результат, мы совершили множество ошибок — ошибок, которые вам не захотелось бы повторять.

В следующие четыре года я лично работал с сотнями заказчиков, и мне известно о тысячах достигших успеха при помощи технологий, которые описаны в этом издании.

Начинается все это с шаблонов.

Контейнеры внедрений зависимостей — это всего лишь инструменты. Инструменты становятся полезными только тогда, когда вы создаете системы, которые построены на шаблонах, реализуемых этими инструментами. Они не являются средством решения любой проблемы. В идеале вам сначала нужно понять, что такое внедрение зависимостей, какие виды проблем оно решает и что представляют собой шаблоны, используемые им. Затем вы должны овладеть различными инструментами и вспомогательными средствами, чтобы применять эти шаблоны.

Эта книга поможет вам во всем этом. Начальные главы содержат обзор проблем, возникающих, если программное обеспечение разработано как сильно связанное. Затем обсуждаются способы использования различных технологий для решения этих проблем. Среди прочего классифицируются разные шаблоны и определяется, когда они могут применяться наилучшим образом в конкретных ситуациях. Вторая половина книги представляет всесторонний обзор наиболее распространенных контейнеров внедрения зависимостей и фреймворков в .NET и объясняет, как с их помощью могут применяться разные технологии.

Читая это издание, вы познакомитесь с многолетним опытом применения соответствующих технологий. Это настоящее удовольствие; часто те, кто начинает использовать внедрение зависимостей, ощущают себя затерявшимися в океане путаницы. Эта книга устраняет множество потенциальных недоразумений, начиная от простейших вопросов типа «Где я должен применить инверсию управления?» или «Должен ли я разрабатывать свой контейнер?». Марк отвечает на эти вопросы и на множество других.

Марк не только описывает технологии, но и углубляется в них, объясняя, когда вы можете — и, что еще более важно, не можете — использовать их. При описании проблемы он приводит реальные примеры, чтобы не потерять из виду всю картину.

Если вы новичок в области инверсии управления (IoC), я полагаю, что для вас эта книга будет хорошим источником знаний. Даже если у вас имеется опыт работы с инверсией управления, вы получите пользу от тщательной работы Марка по классификации различных шаблонов и созданию таксономии инверсии управления.

Независимо от вашей квалификации я желаю, чтобы эта книга оказалась полезной для вас.

Гленн Блок,
старший менеджер программ Microsoft

Введение

Существует своеобразный связанный с Microsoft феномен, именуемый Microsoft Echo Chamber — эхо-камера Microsoft. Microsoft — это огромная организация, и окружающая ее экосистема сертифицированных партнеров увеличивает этот размер на несколько порядков. Если вы успешно внедрились в данную экосистему, может быть непросто увидеть что-либо за ее границами. Всякий раз, когда вы ищете решение проблемы с помощью продукта или технологии Microsoft, вы, вероятно, найдете ответ, который потребует привлечения еще большего количества продуктов Microsoft. Не имеет значения, что именно вы кричите внутри этой эхо-камеры, ответ будет: Microsoft!

Когда Microsoft приняла меня на работу в 2003-м, я уже принимал непосредственное участие в этой Echo Chamber, работая у сертифицированных партнеров Microsoft в течение долгих лет, — и мне это нравилось! Вскоре меня отправили на внутреннюю техническую конференцию в Новый Орлеан с целью познакомиться с новейшими и величайшими технологиями Microsoft.

Сегодня я не могу вспомнить, в секциях каких именно продуктов Microsoft я тогда участвовал, но я хорошо запомнил последний день. В этот день, не найдя секцию, которая могла бы удовлетворить мой интерес к свежим технологиям, я был главным образом занят поиском подходящего рейса, чтобы лететь домой, в Данию. Моей основной целью в тот момент было найти такое место, где я мог бы устроиться так, чтобы иметь доступ к своей электронной почте, поэтому я выбрал секцию, более-менее подходившую моим целям, и включил свой ноутбук.

Секция не имела четко выраженной тематики, в ней выступало несколько докладчиков. Одним из них был бородатый парень по имени Мартин Фаулер, рассказывавший о разработке через тестирование (Test-Driven Development, TDD) и динамических моках. До этого я никогда не слышал о нем и слушал невнимательно, но что-то, по-видимому, оставило след в моей памяти.

Вскоре после возвращения в Данию передо мной была поставлена задача написания большой ETL-системы (Extract, Transform, Load, «выделить, преобразовать, загрузить») с нуля, и я решил попытаться использовать TDD (это оказалось очень хорошим решением). Вслед за этим естественным оказалось применение динамических моков, что выявило необходимость управления зависимостями. Эта проблема показалась мне очень трудной, но весьма захватывающей, и я не мог перестать думать о ней.

То, что началось как сторонний эффект моего интереса к TDD, стало объектом моего интереса. Я выполнил огромное количество исследований, прочитал безумное количество блог-постов на эту тему, сам несколько раз писал в блоги, экспериментировал с кодом и дискутировал на эту тему с любым, кто соглашался выслушать

меня. По возрастающей я начал выходить за пределы эхо-камеры Microsoft в поисках вдохновения и рекомендаций. И все это время люди ассоциировали меня с движением ALT.NET, хотя я никогда не был его активистом.

Я сделал все ошибки, которые только можно было сделать, но постепенно начал понимать и осваивать разработку с использованием внедрения зависимостей.

Когда издательство Manning обратилось ко мне с идеей написать книгу о внедрении зависимостей в .NET, моей первой реакцией было: «А это нужно?» Я чувствовал, что все концепции, необходимые для того, чтобы понять внедрение зависимостей, были уже описаны в огромном количестве блог-постов. Можно ли еще что-либо добавить? Честно говоря, я думал, что внедрение зависимостей в .NET — это тема, которая уже должна умереть.

После некоторых раздумий, однако, меня осенило, что хотя знания и имеются в наличии, но они очень разбросаны, плюс еще используется большое количество конфликтующих технологий. До этой книги не было источников, в которых была бы сделана попытка дать ясное описание внедрения зависимостей. После дальнейших размышлений я пришел к выводу, что Manning делает мне огромный вызов и в то же время предоставляет великолепную возможность собрать воедино и систематизировать все, что я знаю о внедрении зависимостей.

Результатом всего этого является данное издание. В нем используются технологии .NET и C#, чтобы представить и описать исчерпывающую терминологию и руководство по внедрению зависимостей, но я надеюсь, что значимость этой книги выйдет за пределы данных платформ. Я думаю, что введенный здесь язык шаблонов является универсальным. Являетесь вы разработчиком на платформе .NET или используете другую объектно-ориентированную платформу, я надеюсь, что это издание поможет вам стать более хорошим инженером-программистом.

Благодарности

Благодарность может выглядеть как клише, но это только потому, что она является основой человеческой природы. Пока я писал эту книгу, многие люди давали мне повод быть им благодарным, и я хочу сказать им всем спасибо.

Прежде всего работа над изданием в мое свободное время дало мне новое понимание того, насколько обременителен такой проект для брака и семейной жизни. Моя жена Сесилия оставалась со мной и активно поддерживала меня в течение всего этого времени. Что важнее всего, она поняла, насколько этот проект важен для меня. Мы вместе и сейчас, и я планирую провести больше времени с нею и с нашими детьми Линеа и Ярлом (которые были лишены меня, хотя я и находился рядом все это время).

Как мои родители, так и родители моей жены также оказали большую помощь в течение всего времени, когда я все свои усилия отдавал этой книге. Я бы не справился без них.

На профессиональном уровне я благодарен издательству Manning за предоставленную мне возможность. Карен Тегтмейер, можно сказать, «открыла» меня и помогла мне установить отношения с Manning. Майкл Стефенс был инициатором этого проекта и верил в меня, когда дела выглядели мрачно. Было время, когда стало казаться, что я никогда не закончу эту книгу без посторонней помощи, но Майкл предоставил мне шанс, и я чрезвычайно благодарен за то, что мне была предоставлена возможность завершить это издание самому. Синтия Кейн выступала в роли моего редактора и острым взором следила за качеством текста. Она помогла мне определить слабые места в рукописи и подвергла ее обширной, но конструктивной критике. Несмотря на возникавшие по ходу дела разочарования, я особенно благодарен ей за то, что она убедила меня переделать главы с первой по третью. Я намного больше удовлетворен окончательным результатом, и я говорю Синтии спасибо за это.

Хотя написание книги было неоплачиваемой дополнительной работой, я никогда не сомневался, что ее результатом будет повышение производительности моего труда. Когда я только начал работу над этим проектом, меня очень поддержал мой тогдашний менеджер Петер Хааструп. Я хочу выразить благодарность как ему, так и нашему CEO, Нильсу Фленстед-Йенсену, за предоставление необходимого для работы оборудования. К несчастью, та компания вышла из бизнеса, но и мой новый работодатель, Йорн Флоор Андерсен, был исключительно терпелив ко мне.

Карстен Стробэк и Брайан Расмуссен прочли огромное количество ранних черновиков и сделали много полезных замечаний. Карстен также выступала в качестве технического корректора во время выпуска книги.

Следующие люди читали рукопись на разных этапах ее готовности, и я благодарен им за комментарии и понимание: Кристиан Сигерс, Амос Баннистер, Рама Кришна Вавилала, Даг Фергюсон, Даррен Неймке, Чак Дюфре, Пол Гребен, Лестер Лобо, Йонас Банди, Брай Панда, Алан Руч, Тимоти Бинкли-Джонс, Эндрю Симер, Хавьер Лозано, Дэвид Баркол и Патрик Стигер.

Многие участники Manning Early Access Program (MEAP) тоже комментировали книгу и отвечали на сложные вопросы, касавшиеся слабых мест текста.

Я был очень счастлив, что существующее сообщество, занимающееся контейнерами внедрения для .NET, восприняло проект книги весьма положительно. Некоторым разработчикам конкретных контейнеров было предложено познакомиться с главами, касающимися «их» контейнеров. Например, Кшиштоф Козмич рассматривал главу о Castle Windsor, а Джереми Миллер отвечал на мои глупые вопросы в Twitter и на форуме StructureMap. Я благодарен им всем за участие, а также за полученное от них подтверждение, что мой способ представления их работы согласуется с их собственным. Я также благодарен Гленну Блоку за написанное им предисловие.

Могенс Хеллер Грабе учтиво разрешил использовать его изображение фена, встроенного прямо в стенную розетку, а Патрик Смачья предоставил мне копию NDepend и просмотрел посвященный ему подраздел.

Во многих смыслах Мартин Гилденпфеннинг посеял больше семян для этой книги, чем он может вообразить. Еще до того, как в 2003 году я услышал презентацию Мартина Фаулера, касающуюся разработки через тестирование, Мартин Гилденпфеннинг уже познакомил меня с принципами модульного тестирования, хотя мы никогда не думали о его использовании в то время. Много позже у меня было ложное убеждение, что локатор сервисов есть благо, но на нескольких простых примерах он убедил меня, что существуют лучшие альтернативы.

Мой бывший коллега Миккель Христенсен всегда был готов поработать вместе, пока я писал большие фрагменты книги. Мы много обсуждали вопросы проектирования API и шаблоны, при этом я часто выдавал ему свои сумасшедшие идеи, и всегда у нас происходило очень квалифицированное их обсуждение.

В заключение я хочу поблагодарить Томаса Яскулу за поддержку и одобрение в течение всей работы. Нам никогда не доставляли удовольствия наши встречи, но Томас неоднократно восхищался моей работой. Он может не отдавать себе отчета в этом, но временами это было единственным, что заставляло меня продолжать работу.

Об этой книге

Эта книга в основном и в первую очередь посвящена внедрению (или инъекциям) зависимостей (ВЗ, Dependency Injection). Кроме того, это издание о технологии .NET, но это не так существенно. Коды примеров в нем написаны на языке C#, но большинство обсуждаемого в книге материала может быть легко применено к другим языкам и платформам. Я, например, изучал большинство рассматриваемых принципов и паттернов, читая книги, в которых коды примеров были написаны на языках Java или C++.

Внедрение зависимостей — это набор связанных принципов и паттернов. Оно является в большей степени процессом обдумывания и проектирования кода, а не конкретной технологией. Основная цель использования ВЗ — создание удобного в сопровождении программного обеспечения, построенного на объектно-ориентированной парадигме.

Все применяемые в данном издании концепции относятся к объектно-ориентированному программированию (ООП). Проблема, с которой справляется ВЗ (создание удобного в сопровождении программного кода), универсальна, но решение, предлагаемое ВЗ, находится в области объектно-ориентированного программирования на языках со статической типизацией: C#, Java, Visual Basic .NET, C++ и им подобных. Вы не сможете применить ВЗ в процедурном программировании. Этот подход — внедрение зависимостей — также, по-видимому, плохо применим в функциональных или динамических языках.

Само по себе ВЗ невелико, но оно тесно связано с большим комплексом принципов и паттернов, используемых в проектировании и разработке объектно-ориентированного программного обеспечения. Несмотря на то что основным предметом этой книги является последовательное и всестороннее рассмотрение ВЗ, в ней также обсуждается множество других тем с точки зрения той специфической перспективы, которую дает ВЗ. Целью этого издания является не просто дать вам представление о тонкостях ВЗ, но и повысить вашу квалификацию в области разработки объектно-ориентированного программного обеспечения.

Для кого предназначена эта книга

Мне бы хотелось сказать, что эта книга предназначена для всех разработчиков платформы .NET. Но сообщество .NET-программистов сегодня очень велико и объединяет разработчиков веб-приложений, настольных приложений, приложений для смартфонов, RIA (Rich Internet Application — насыщенные интернет-приложения), интеграции, офисной автоматизации, систем управления контентом (CMS) и даже игр. И хотя .NET является объектно-ориентированным инструментом, не все из этих разработчиков создают объектно-ориентированный код.

Это издание об объектно-ориентированном программировании, поэтому читатели, как минимум, должны быть заинтересованы в применении ООП и понимать суть интерфейсов[1]. Наличие нескольких лет опыта в программировании, а также знание паттернов или SOLID (http://ru.wikipedia.org/wiki/SOLID_(объектно-ориентированное_программирование — *Примеч. ред.*), несомненно будет дополнительным преимуществом. Не уверен, что начинающие программисты получат много пользы от чтения этой книги, так как она в большей степени адресована опытным разработчикам и архитекторам программного обеспечения.

Все примеры в издании написаны на языке C#, поэтому читатели, программирующие на других языках платформы .NET, должны понимать C#. Читатели, работающие с объектно-ориентированными языками типа Java или C++, не поддерживаемыми платформой .NET, также могут счесть эту книгу полезной, поскольку информация, специфичная для платформы .NET, излагается здесь относительно поверхностно. Мне довелось прочесть множество книг по паттернам проектирования, примеры в которых были даны на языке Java, и я легко разобрался в большинстве из них, поэтому надеюсь, что и читатель не растеряется в подобной ситуации.

Наш маршрут

Содержимое книги разбито на четыре части. В идеале мне бы хотелось, чтобы вы прочли ее полностью, а затем использовали нужные части в качестве справочника, но я понимаю, что у вас могут быть и другие приоритеты. Из этих соображений большая часть глав написана таким образом, что вы можете выбрать нужную главу и начать чтение прямо с нее.

Однако первая часть книги все же обязательна для прочтения. Она содержит общую вводную информацию о внедрении зависимостей, и ее лучше читать по порядку. Вторая часть представляет собой каталог паттернов и других технологических компонентов. Третья часть содержит анализ ВЗ, выполненный с трех различных точек зрения. Четвертая часть книги содержит описание двух библиотек *контейнеров внедрения зависимостей*.

- Часть 1 — это общее введение в ВЗ. Если вы не в курсе, что именно представляет собой ВЗ, то эта глава является хорошей отправной точкой. Но даже если вы знаете что-либо об ВЗ, вы, возможно, захотите ознакомиться с содержимым части 1, поскольку она определяет терминологию, используемую в остальной части книги. В главе 1 дается общее представление о ВЗ, а также обсуждаются назначение и достоинства этой технологии. Глава 2 описывает большой и всесторонний практический пример. В главе 3 объясняется, как библиотеки контейнеров внедрения зависимостей используются вместе с другими компонентами. По сравнению с другими частями книги, часть 1 имеет гораздо более линейную структуру. Поэтому, чтобы максимально полно усвоить материал, каждую из глав этой части нужно прочесть от начала и до конца.
- Часть 2 является каталогом паттернов, антипаттернов и рефакторинга. В ней вы найдете инструкции по применению ВЗ, а также описание сложностей, с ко-

[1] В контексте ООП. — *Примеч. пер.*

торыми вы можете столкнуться на этом пути. Глава 4 содержит каталог паттернов, а глава 5, соответственно, — антипаттернов. В главе 6 описываются обобщенные решения наиболее распространенных проблем. Поскольку главы из этой части предназначены для использования в качестве каталога, то все они состоят из практически не связанных между собой разделов, которые можно читать независимо друг от друга и в произвольной последовательности.

- Часть 3 содержит анализ ВЗ по трем различным аспектам: композиция объектов (object composition), управление их жизненным циклом (lifecycle management) и перехват (interception).

  В главе 7 обсуждаются вопросы реализации ВЗ на основе уже существующих фреймворков приложений, таких как WCF, ASP.NET MVC, WPF и некоторых других. Во многих случаях материал главы 7 может быть использован в качестве каталога способов реализации ВЗ для определенного набора фреймворков. В главе 8 описывается, как осуществлять управление жизненным циклом зависимостей, избегая при этом утечки ресурсов. Поскольку структура этой главы менее строга, чем структура предыдущих глав, значительная часть ее может использоваться в качестве каталога широко известных жизненных стилей. Глава 9 описывает порядок сборки приложений с использованием сквозных аспектов приложения (cross-cutting concerns). В этой главе мы получим реальные результаты от проделанной ранее работы, так что я считаю ее кульминацией всей части.

- Часть 4 содержит описание библиотек контейнеров внедрения зависимостей. В двух главах рассмотрены контейнеры Castle Windsor и StructureMap.

Чтобы разговор о принципах и паттернах ВЗ не был привязан к API определенных контейнеров, большинство материала в этой книге, за исключением части 4, написано без ссылок на какой-нибудь конкретный контейнер. По этой причине контейнеры интенсивно обсуждаются только в части 4. Я твердо уверен, что благодаря общему обсуждению это издание будет полезным в течение более длительного периода времени.

Условные обозначения и возможности для скачивания

В этой книге содержится много примеров кода. Большинство из них написано на C#, но в разных местах встречается и код XML. Исходные коды в листингах и тексте книги приводятся моноширинным шрифтом, чтобы отделить его от обычного текста.

Весь исходный код в издании написан на языке C# в среде Visual Studio 2010. Приложения, имеющие архитектуру ASP.NET MVC, созданы в соответствии со стандартами ASP.NET MVC 3.

Лишь немногие технологии, рассмотренные в этой книге, базируются исключительно на возможностях современных языков программирования. Я начинал со слабо связанного кода на платформе .NET 1.1, и большинство примеров могло бы быть реализовано без каких-либо дополнительных ухищрений. Как бы то ни было, я хотел достичь разумного баланса между консервативным и современным

стилями программирования. Когда я пишу профессиональный код, я использую самые передовые возможности языка для достижения наилучшего результата. Но в этой книге нет ничего сложнее дженериков (generics) и LINQ. Меньше всего мне хотелось бы произвести неверное впечатление: дескать, внедрение зависимостей применимо только в ультрасовременных языках.

Написание кода в качестве примеров для любой книги связано с характерными проблемами. В отличие от экрана современного компьютера, книга позволяет размещать в строке только очень короткие фрагменты кода. Было очень заманчиво писать код в сжатом стиле, используя краткие, но малопонятные имена методов и переменных. Такой код труден для понимания, даже когда написан в реальных программах, и работа с ним ведется в интегрированной среде разработки и отладчике. Но в книге работать с ним крайне сложно. Я убежден, что следует использовать настолько удобочитаемые имена, насколько это возможно. Чтобы соответствовать всем этим требованиям, мне зачастую приходилось прибегать к нестандартным переносам. Весь код при этом остается компилируемым, но время от времени такое форматирование выглядит немного смешно.

В коде интенсивно используется ключевое слово `var`. В профессиональном коде я практически не применяю это слово, но нахожу его полезным в коде примеров, когда оно используется совместно с явными объявлениями, так как интегрированная среда разработки в данной ситуации не может помочь. В целях экономии места я также использую слово `var` всюду, где, по моему мнению, явные объявления не являются необходимыми.

Ключевое слово `class` часто применяется как синоним слову `type`. В .NET классы, структуры, интерфейсы, перечисления и т. д. являются типами, но поскольку слово `type` в обычной речи, не связанной с программированием, имеет множество разнообразных смыслов, то его использование сделало бы текст менее понятным.

Большинство кода в этом издании относится к всеобъемлющему примеру, используемому на протяжении всей книги: онлайновому магазину, дополненному вспомогательными внутренними приложениями. Возможно, вы надеялись увидеть более захватывающий пример, но я выбрал его по нескольким причинам.

- Это хорошо известная проблемная область для большинства читателей. И хотя она может показаться скучной, я думаю, что она обладает определенными достоинствами, поскольку до сих пор мало исследовалось применение внедрения зависимостей в данной области.
- Я не представляю какую-либо другую предметную область, настолько широкую, чтобы в ней можно было реализовать все разнообразные сценарии, которые я держу в голове.

Я написал массу кода для использования в качестве примеров, и большая часть этого кода даже не вошла в книгу. Фактически я написал почти весь мой код, следуя принципам разработки через тестирование (Test-Driven Development, TDD), но поскольку эта книга не о TDD, я практически не включал в нее модульные тесты.

Исходные коды всех примеров из этой книги доступны на сайте издательства Manning: http://manning.com/DependencyInjectionin.NET. Файл `ReadMe.txt` в корневом каталоге скачанных материалов содержит инструкции по компиляции и запуску кода.

Об авторе

Марк Симан — программист, программный архитектор и лектор, проживает в Копенгагене (Дания). Он занимается программированием с 1995 года, а разработкой через тестирование — с 2003 года. Шесть лет проработал в Microsoft в качестве консультанта, разработчика и архитектора программного обеспечения (ПО). В настоящее время Марк профессионально занимается разработкой программного обеспечения. Он увлекается чтением, рисованием, игрой на гитаре, любит хорошее вино, а также изысканную еду.

Иллюстрация на обложке

На обложке книги представлена «Женщина из Водняна», маленького городка на полуострове Истрия в Адриатическом море в Хорватии. Эта иллюстрация взята из альбома Николы Арсеновича «Хорватский национальный костюм середины девятнадцатого столетия», изданного Этнографическим музеем в Сплите (Хорватия) в 2003 году. Иллюстрации были получены из библиотеки Этнографического музея в Сплите, которая находится в Римском квартале средневекового центра города: руины дворца римского императора Диоклетиана, около 304 года нашей эры. Альбом содержит прекрасные цветные иллюстрации персонажей из разных регионов Хорватии, дополненные описаниями костюмов и сцен из повседневной жизни.

Воднян — это значимый в культурном и историческом смысле город, расположенный на холме с великолепным видом на Адриатическое море, известный своими многочисленными церквями и сокровищами обрядового искусства. Женщина на обложке одета в длинную черную нижнюю юбку и короткий черный жакет поверх белой нижней рубашки. Обрез жакета отделан голубой вышивкой. Довершает костюм голубой фартук. Черная шляпа с полями украшена цветочным узором, кроме того, на женщине шарф и большие серьги в виде обруча. Ее элегантный костюм свидетельствует, что она жительница города, а не села. Народные костюмы в окружающей сельской местности более красочны, изготовлены из шерсти и украшены богатой вышивкой.

Стили одежды и стиль жизни претерпели изменения за последние 200 лет, и региональные различия, такие заметные в прежние времена, ушли в прошлое. Сегодня тяжело отличить друг от друга даже жителей разных континентов, не то что городов, разделенных всего несколькими километрами. По-видимому, мы променяли разнообразие культур на большую индивидуальность в личной жизни — несомненно, из-за разнообразной и стремительной технологической жизни.

Издательство Manning отдает должное изобретательности и инициативности компьютерного бизнеса, издавая книги с иллюстрациями на обложках, которые отображают богатое разнообразие жизни в различных регионах два столетия тому назад, и таким образом воскрешая это разнообразие при помощи рисунков из старинных книг и коллекций, подобных упомянутому альбому.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ 1

Последовательность изучения внедрения зависимостей

Внедрение зависимостей (ВЗ) — одна из наиболее превратно понятых концепций в объектно-ориентированном программировании. Путаница в этом вопросе весьма велика и распространяется как на терминологию, так и назначение и механизмы внедрения зависимостей. Как ее нужно называть: внедрение (инъекция) зависимостей (dependency injection), инверсия управления (inversion of control) или, быть может, подключение посредника (third-party connect)? Предназначено ли внедрение зависимостей только для поддержки модульного тестирования (unit testing) или для нее существуют и более широкие области применения? Эквивалентны ли внедрение зависимостей и обнаружение сервисов (service location)? Необходимо ли использование контейнеров внедрения зависимостей?

На тему ВЗ написано множество постов в блогах, журнальных статей, подготовлено большое количество презентаций для различных конференций. Но, к сожалению, многие из них используют противоречивую терминологию или дают неверное понимание предмета. Это справедливо для всех направлений, и даже крупные и влиятельные игроки уровня Microsoft лишь подливают масла в огонь.

Такого не должно происходить. В этой книге я представляю и использую согласованную терминологию, которая, как я надеюсь, будет одобрена сообществом. По большей части я адаптировал и уточнил существующую терминологию, применяемую другими специалистами, но в некоторых случаях я добавил и кое-какие свои определения — в основном тогда, когда подходящих терминов не существовало. Это в значительной степени помогло мне расширить определение области применения внедрения зависимостей.

Одна из причин, приводящих к противоречиям и неверной трактовке, — размытость границ термина «внедрение зависимостей». Где заканчивается ВЗ и начинаются другие объектно-ориентированные концепции? Я думаю, что не представляется возможным провести разграничительную линию между ВЗ и другими аспектами написания качественного объектно-ориентированного кода. Обсуждение ВЗ невозможно без обсуждения других концепций, например таких, как SOLID или Clean Code (чистый код). Я знал, что не смогу достоверно написать об ВЗ, не затрагивая этих тем.

Первая часть книги поможет вам понять место ВЗ относительно других аспектов программной инженерии, так сказать, разметить карту.

Первая глава содержит краткий обзор внедрения зависимостей, включая его назначение, принципы и достоинства этой технологии. Кроме того, в ней излагается краткое содержание остальной части книги. Если вы хотите понять, что представляет собой ВЗ и почему вам стоит заняться его изучением, то вам следует начать именно с данной главы. Материал главы предполагает, что у вас нет предварительных знаний о ВЗ, но даже при наличии таких знаний вам все же будет полезно прочитать ее — может оказаться, что ВЗ не совсем соответствуют вашим представлениям о них.

Основной упор в главе 1 сделан на формирование общей картины, что не предполагает погружения в детали. А вот глава 2 полностью отведена под большой пример. Назначение этого примера — конкретизировать полученное общее представление о ВЗ. Она разделена на две части и написана почти в повествовательной форме. Для демонстрации отличий ВЗ от более «традиционного» стиля программирования эта глава сначала представляет типовую сильно связанную реализацию

приложения-примера, а затем демонстрирует это же приложение, но уже реализованное с использованием ВЗ.

Третья и последняя глава части 1 вводит концепцию контейнеров внедрения зависимости и объясняет, какую роль они играют в ВЗ. Обсуждение ВЗ в этой главе ведется на общем уровне, и хотя в текст главы включены примеры кода, иллюстрирующего функционирование типового контейнера ВЗ, она не содержит информации о конкретных деталях API. Основная задача главы 3 — показать, что контейнеры ВЗ являются хотя и полезным, но необязательно необходимым инструментом. Безусловно, ВЗ может быть реализовано без использования контейнеров, поэтому контейнеры в частях 2 и 3 в большей или меньшей степени игнорируются. Вместо этого в них обсуждается реализация ВЗ без использования контейнеров. Однако позднее, в части 4, мы вернемся к этой теме, чтобы проанализировать два типа контейнеров.

Часть 1 определяет содержание остальной книги. Она нацелена на читателей, не имеющих предварительных знаний в области ВЗ, но опытные специалисты по ВЗ также могут получить пользу даже от беглого просмотра этих глав. Так они смогут ознакомиться с терминологией, используемой в этой книге. После прочтения части 1 у вас должно сформироваться твердое понимание терминологии и общих концепций, даже если некоторые мелкие детали все еще будут неясны. Так и должно быть — книга становится все более конкретной по мере ее прочтения, и части 2, 3, 4 должны ответить на вопросы, возможно, появившиеся у вас после прочтения части 1.

1 Дегустационное меню внедрения зависимостей

Меню:

- заблуждения относительно внедрения зависимостей;
- назначение внедрения зависимостей;
- преимущества внедрения зависимостей;
- в каких случаях следует применять внедрения зависимостей.

Возможно, вам приходилось слышать, что приготовление беарнского соуса — трудная задача. Многие любители готовить даже не берутся за нее. Очень жаль, поскольку этот соус просто восхитителен (традиционно он подается со стейком, но является также потрясающим дополнением к белой спарже, яйцам пашот и другим блюдам). Кое-кто пытается выкрутиться, например, употребляя готовые купленные соусы либо не требующие длительного приготовления смеси, но все эти варианты нисколько не похожи на оригинал.

ОПРЕДЕЛЕНИЕ

Беарнский соус — это эмульгированный соус, который готовится из яичных желтков и масла, приправленных полынью, кервелем, луком-шалотом и уксусом. Не содержит воды.

Самое сложное в приготовлении беарнского соуса — то, что ваша готовка может пойти прахом: соус может свернуться или расслоиться на компоненты, и если такое произойдет, вы не сможете восстановить его. Приготовление этого соуса занимает около 45 минут, так что неудача может означать, что у вас не останется времени на вторую попытку.

С другой стороны, беарнский соус может приготовить любой профессиональный повар. Это входит в программу их обучения и, как они скажут вам, не составляет особого труда.

Я думаю, что внедрение зависимостей похоже на беарнский соус. Все выглядит сложным, и не многие возьмутся за него. Если вы попытаетесь реализовать ВЗ и потерпите неудачу, то, вполне вероятно, времени на вторую попытку у вас уже не останется.

ОПРЕДЕЛЕНИЕ

Внедрение зависимостей — это набор принципов проектирования программного обеспечения и шаблонов, облегчающих процесс создания слабосвязанного кода.

Несмотря на Страх, Неопределенность и Сомнение (СНС), окружающие ВЗ, освоить его не сложнее, чем приготовить беарнский соус. Вы можете совершать ошибки во время учебы, но после того, как набьете руку, вы уже никогда не ошибетесь, применяя эту технологию.

Посвященный вопросам разработки программного обеспечения сайт Stack Overflow провел опрос на тему «Как объяснить внедрение зависимостей пятилетнему ребенку». Лучшим был признан ответ, данный Джоном Манчем[1]. Он содержит неожиданно точную аналогию, предназначенную (воображаемому) пятилетнему инквизитору:

Когда ты идешь и сам берешь что-нибудь из холодильника, то могут возникнуть проблемы. Ты можешь оставить дверцу открытой, можешь взять что-то, что мама и папа предназначали не для тебя. Ты можешь даже взять что-то недозволенное или что-то испортившееся.

Все, что ты должен сделать, — это начать с просьбы. Скажи: «Я хочу выпить чего-нибудь во время еды», — и мы сделаем все, чтобы у тебя было то, что тебе нужно, когда ты сядешь за стол.

Вот что это означает, когда речь идет о разработке объектно-ориентированного программного обеспечения: «Взаимодействующие классы (“пятилетние дети”) должны обращаться к инфраструктуре (“родителям”) за получением необходимых услуг (сервисов)».

Как показано на рис. 1.1, данная глава совершенно линейна по своей структуре. Сначала я даю введение во внедрение зависимостей, включая описание его назначения и достоинств. Хотя я и привожу некоторые примеры, эта глава содержит меньше кода, чем любая другая глава в книге.

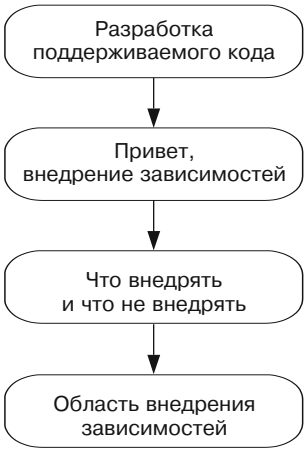


Рис. 1.1. Структура данной главы совершенно линейна: вы должны прочитать первый раздел, прежде чем возьметесь за второй, и т. д. Это представляется очевидным, но следующие главы в книге будут менее линейными по своей структуре

[1] *John Munsch* How to explain Dependency Injection to a 5-year old, 2009, http://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old.

Прежде чем я начну обзор ВЗ, я немного порассуждаю об основном его назначении: создании легкого в сопровождении и развитии кода. Это важно сделать, так как внедрение зависимостей может быть легко понято неправильно, ели вы не будете полностью подготовлены. Затем, после рассмотрения примера (Hello DI), я поговорю о достоинствах и области применения ВЗ, уделяя особенное внимание последовательности чтения книги. После изучения этой главы вы должны быть подготовлены к изучению более сложных концепций, представленных в следующих главах книги.

Многим разработчикам ВЗ может показаться похожей скорее на некий старинный способ программирования, и они свяжут с ней, как с беарнским соусом, множество СНС. Чтобы начать изучение ВЗ, вы прежде всего должны понять его назначение.

1.1. Написание удобного в сопровождении кода

Каково назначение ВЗ? Само внедрение не является нашей целью; скорее, оно должно служить для достижения цели. А целью большинства программных технологий является обеспечение максимальной эффективности разработки программного обеспечения, насколько это вообще возможно. Один из аспектов данной задачи — написание удобного и легкого в сопровождении кода.

Если только вы не разрабатываете прототип или приложение, которое никогда не будет модернизироваться после выпуска релиза номер один, вы скоро обнаружите, что пытаетесь использовать и расширять написанный ранее код. Чтобы работа с таким созданным ранее программным кодом была как можно более эффективной, код должен быть максимально удобным для сопровождения.

Один из многих способов улучшить сопровождаемость кода — сделать его слабо связанным. Еще с 1995 года, когда «Банда четырех»[1] написала книгу Design Patterns, стал известен принцип:

Программируйте в соответствии с интерфейсом, а не с реализацией.

Эта важная часть рекомендации является в разработке шаблонов не окончательным заключением, а скорее предпосылкой. Слабая связанность делает код расширяемым, а расширяемость делает его сопровождаемым.

Внедрение зависимостей — это не что иное, как технология, обеспечивающая слабую связанность. Однако по ее поводу существует множество заблуждений, и зачастую они приводят к неверному пониманию. И прежде чем вы начнете обучение, вам понадобится забыть все то, что вы (как вы считаете) уже знаете.

1.1.1. Отучиться от внедрения зависимостей

Подобно голливудскому клише, касающемуся боевых искусств, вы должны отучиться, прежде чем сможете начать обучение. Существует много заблуждений по поводу ВЗ, и если вы не освободитесь от них, то вы будете неправильно трактовать

[1] Подробнее о «Банде четырех» вы можете прочитать в «Википедии»: http://ru.wikipedia.org/wiki/Design_Patterns. — *Примеч. пер.*

то, что вы прочтете в этой книге. То есть вы должны предварительно очистить свой разум, чтобы понять, что такое ВЗ.

Существует как минимум четыре мифа о внедрении зависимостей:

- ВЗ применимо только для случаев позднего (динамического) связывания (Late binding);
- ВЗ применимо только для модульного тестирования;
- ВЗ — это «прокачанная» разновидность «Абстрактной фабрики»;
- ВЗ требует наличия специального контейнера.

Хотя ни один из этих мифов не соответствует действительности, они тем не менее весьма распространены. Мы должны их рассеять, прежде чем начнется изучение ВЗ.

Динамическое (позднее) связывание

В данном контексте термин «динамическое связывание» означает возможность заменять части приложения без перекомпиляции кода. Приложение, допускающее использование расширений сторонних разработчиков (такое как Visual Studio), — один из таких примеров.

Другой пример — это стандартное программное обеспечение, поддерживающее различные среды времени исполнения. Вы можете работать с приложением, которое может выполняться более чем на одном сервере баз данных, например приложение, поддерживающее как Oracle, так и SQL Server. Для реализации такой возможности основная часть приложения должна работать с базой данных через интерфейс. Код приложения должен при этом содержать разные реализации данного интерфейса, обеспечивающие доступ к Oracle, и SQL Server соответственно. Для управления тем, какой вариант реализации будет использован в данном конкретном варианте, используется соответствующая опция конфигурации.

Распространенным заблуждением является и то, что ВЗ уместно только в сценариях, подобных вышеописанным. Да, ВЗ допускает такие сценарии, но ошибочно будет думать, что данная зависимость будет симметричной (двусторонней). То, что ВЗ допускает динамическое связывание, вовсе не означает, что оно применимо только в таких сценариях. Рис. 1.2 показывает, что динамическое связывание — это только один из многочисленных аспектов внедерения зависимостей.

Рис. 1.2. Внедрение зависимостей допускает динамическое связывание, но предположение, что сценарии динамического связывания являются единственной областью применения ВЗ, означает применение узкой проекции к значительно более широкой перспективе

Итак, если вы думаете, что внедрение зависимостей применимо только к сценариям с динамическим связыванием, то вам следует как можно скорее забыть об этом. ВЗ далеко не ограничивается динамическим связыванием.

Модульное тестирование

Некоторые думают, что внедрение зависимостей может поддерживать только модульное тестирование. Это, однако, не так, хотя ВЗ, конечно же, является важной частью поддержки модульного тестирования.

Сказать по правде, мое знакомство с ВЗ началось с преодоления некоторых аспектов разработки через тестирование (Test-Driven Development, TDD). За прошедшее время я открыл для себя внедрение зависимостей и выяснил, что другие люди использовали его для реализации некоторых сценариев, которыми занимался и я.

Даже если вы не пишете модульные тесты (если вы еще не делаете этого, то вам следует начать), ВЗ тем не менее оказываются полезными благодаря другим своим достоинствам. Утверждение, что ВЗ годится только для поддержки модульного тестирования, не более справедливо, чем «оно полезно только для динамического связывания». Рисунок 1.3 показывает, что, хотя модульное тестирование (Unit testing) во многом отличается от позднего связывания, этот подход так же узок, как и показанный на рис. 1.2. В данной книге я стараюсь показать всю картину целиком.

Рис. 1.3. Хотя предположение, что модульное тестирование — это единственный способ использования внедрения зависимостей и может считаться иным вариантом решения той же проблемы, которая решается при помощи динамического связывания, модульное тестирование так же узко и не позволяет полностью увидеть очень широкую полную картину

Если вы думали, что ВЗ применимы только для модульного тестирования, забудьте это предположение. ВЗ способны на гораздо более значительные свершения.

«Абстрактная фабрика»

Пожалуй, самой серьезной ошибкой будет считать, что ВЗ включает некоторую разновидность многоцелевой «Абстрактной фабрики» (Abstract Factory), которую мы можем использовать для создания экземпляров тех зависимостей, в которых нуждаемся.

Во введении к этой главе я писал, что «взаимодействующие классы... должны обращаться к инфраструктуре... для получения необходимых услуг».

Как вы сначала поняли это предложение? Сочли ли вы инфраструктуру некой разновидностью сервиса, которую мы запрашиваем для получения зависимостей, которые нам нужны? Если это так, то вы не одиноки. Многие разработчики и архитекторы программного обеспечения воспринимают ВЗ как сервис, который может использоваться для нахождения других сервисов; этот паттерн называется «Локатор служб» (Service Locator), но он полностью противоположен ВЗ.

Если вы приравниваете ВЗ к «Локатору служб» — то есть считаете его многоцелевой фабрикой, то вам следует забыть это свое представление. Внедрение зависимостей является полной противоположностью «Локатору служб»; это способ структурировать код таким образом, что нам никогда не придется настоятельно запрашивать зависимости. Скорее, мы заставляем потребителей поставлять их.

Контейнеры внедрения зависимостей

С предыдущим заблуждением тесно связано еще одно — что ВЗ требует специального контейнера. Если вы не избавились от предыдущей ошибки и до сих пор считаете, что ВЗ включает в себя локатор служб, то вполне можете полагать, что контейнер ВЗ может взять на себя функции локатора служб. Такой вариант возможен, но это не единственный способ использования контейнера ВЗ.

Контейнер ВЗ — это опциональная библиотека, которая может облегчить нам сборку компонентов при компоновке приложения, но это не тот путь, который нам нужен. Когда мы реализуем приложения без контейнера ВЗ (я называю такой вариант «ВЗ для бедных»), над ними приходится поработать немного больше, но никаким отказом от принципов внедрения зависимостей это не является.

Если вы полагаете, что ВЗ *обязательно* требует наличия контейнера ВЗ, то знайте: это не так. ВЗ представляет собой набор принципов и шаблонов, при этом контейнер ВЗ является полезным, но *не необходимым* инструментом.

Вы можете подумать, что если я продемонстрировал четыре мифа относительно ВЗ, то я представлю и убедительные доводы против каждого из них. Это справедливо. В определенном смысле вся эта книга является одним большим аргументом против этих распространенных заблуждений.

Как подсказывает мой опыт, избавление от неправильных представлений является жизненно важным, поскольку люди склонны пытаться исправить прямо указанные мной ошибки, касающиеся внедрения зависимостей, и вернуться к знакомым методам работы. Когда такое происходит, требуется обычно много времени, прежде чем они окончательно удостоверятся в своей неправоте. Я бы хотел, чтобы вы учли этот опыт. Поэтому постарайтесь читать эту книгу так, как будто вы ничего не знаете о внедрениях зависимостей.

Давайте предположим, что вы не знаете ничего ни о внедрении зависимостей, ни о его назначении, и начнем с обзора — что же оно делает.

1.1.2. Назначение внедрения зависимостей

Внедрение зависимостей — это не цель, а инструмент достижения цели. Внедрение зависимостей создает условия для слабого связывания, а слабое связывание делает код более легким для сопровождения. Это абсолютная истина, и хотя я и отсылал вас к авторитетным источникам типа книги «Банды четырех», но мне все же кажется необходимым пояснить еще раз, почему это так.

Разработка программного обеспечения все еще остается относительно новой деятельностью, поэтому во многих аспектах мы до сих пор пытаемся разобраться, какие средства обеспечивают производство качественной продукции. Однако

опытные специалисты из других, более традиционных сфер деятельности (таких, например, как строительство) выяснили это достаточно давно.

Проживание в дешевом отеле

Если вы останавливаетесь в дешевом отеле, возможно, вы обнаружите нечто подобное тому, что показано на рис. 1.4. Отель, о котором я говорю, любезно предоставляет в ваше распоряжение фен для волос, но, по-видимому, они не уверены, что вы оставите этот фен следующим гостям: устройство подсоединено к скрытому источнику питания, и шнур питания просто выпущен через стенное отверстие. И хотя длины шнура хватает на то, чтобы предоставить вам определенную свободу перемещения, вы не можете взять этот фен с собой. По-видимому, руководство отеля считает, что стоимость замены украденных фенов достаточно высока, чтобы оправдать такое решение — далекое от совершенства.

Рис. 1.4. В недорогом отеле вы, возможно, обнаружите фен, вмонтированный прямо в стену. Это эквивалентно широко распространенной практике написания сильно связанного программного кода

Что происходит, когда фен выходит из строя? Отелю придется прибегнуть к услугам опытного специалиста, знающего, как разрешить возникшую проблему. Чтобы отремонтировать жестко прикрепленный к стене фен, придется отключить электричество во всей комнате, что сделает ее временно непригодной для жилья. Далее рабочий должен будет использовать специальные инструменты, чтобы без повреждений отсоединить фен и заменить его на новый. Если вам повезет, то техник вспомнит, что он должен снова включить электричество в комнате, после чего он вернется чтобы проверить, работает ли новый фен... Опять же, если вам повезет.

Напоминает ли это вам что-нибудь?

Это похоже на работу с *сильно связанным* кодом. В данном сценарии фен сильно связан со стеной, и вы не можете изменить одно из них, не воздействуя на другое.

Сравнение электропроводки и шаблонов проектирования

Как правило, мы не подключаем электрические устройства, врезая кабели прямо в стену. Вместо этого, как показано на рис. 1.5, мы используем вилки и розетки. Розетка определяет требования, которым должна соответствовать вилка. Если провести аналогию с программным кодом, то розетка играет роль *интерфейса*.

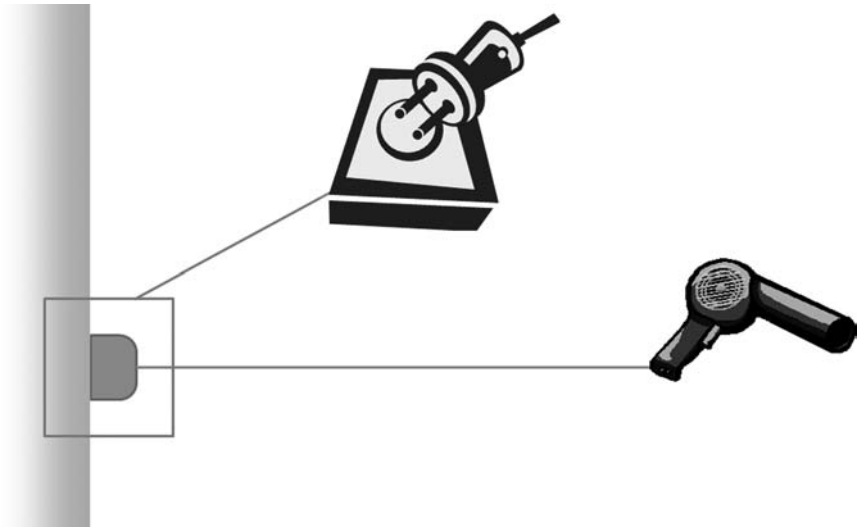


Рис. 1.5. За счет использования розеток и вилок фен становится слабо связанным со стенной розеткой

В противоположность жестко встроенному фену, вилки и розетки определяют модель *слабой связанности* для подключаемых электрических устройств. Пока вилки различных устройств соответствуют имеющимся в распоряжении розеткам, мы можем подключать наши устройства самыми разными способами. Особенно интересно то, что многие из существующих хорошо известных вариантов подключения можно сравнить с известными принципами и шаблонами разработки программного обеспечения.

Таким образом, мы больше не ограничены только фенами. Если вы — среднестатистический читатель, то я могу предположить, что электричество нужно вам для подключения компьютера, а не фена. Это не представляет никакой проблемы: мы просто выдергиваем вилку фена и включаем компьютер в ту же розетку, как показано на рис. 1.6.

Потрясает то, что концепция розеток опережает появление компьютеров на десятилетия, и тем не менее розетки великолепно обслуживают компьютеры. Первые конструкторы розеток вряд ли предусматривали появление персональных компьютеров, но поскольку конструкция розеток является универсальной, они могут обслуживать устройства, появление которых изначально не предполагалось. Способность заменять одну сторону пары без изменения другой похожа на основной принцип разработки программного обеспечения, называемый принципом подстановки Лисков.

Этот принцип означает, что мы должны быть способны заменить одну реализацию некоторого интерфейса на другую, не разрушая как клиента, так и реализацию.

Рис. 1.6. Используя розетки и вилки, мы можем заменить подключенный изначально фен на компьютер. Это соответствует принципу подстановки Лисков

Применительно к внедрения зависимостей принцип подстановки Лисков является одним из наиболее важных принципов разработки. Следование этому принципу позволяет нам решить проблемы, которые могут возникнуть в будущем, даже если мы не можем предвидеть их появление сегодня.

Как показано на рис. 1.7, мы можем отключить компьютер, если не хотим использовать его в данный момент. Даже если в розетку ничего не включено, она все равно остается работоспособной.

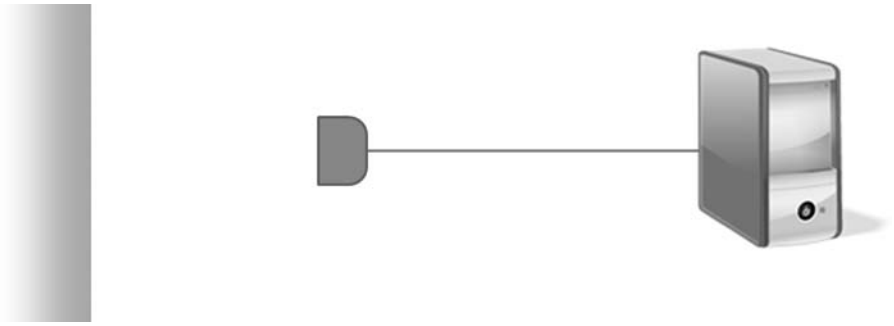


Рис. 1.7. Отключение компьютера не вызывает разрушений ни розетки, ни самого компьютера. Это соответствует паттерну «Нулевой объект» (Null Object)

Если мы выключим компьютер из розетки, не пострадают ни компьютер, ни розетка (фактически, если компьютер является ноутбуком, он сможет даже работать на батарее в течение некоторого периода времени). Однако если речь идет о программном обеспечении, клиент часто ожидает, что сервис по-прежнему является доступным.

Клиентское обращение к сервису, который был удален, приводит к возникновению исключения типа `NullReferenceException`. Чтобы исправить ошибку такого рода, мы можем создать и подключить реализацию того же интерфейса, которая «ничего не делает». Такая реализация представляет собой шаблон проектирования, называемый Null Object, и она достаточно точно соответствует ситуации отключения компьютера от сети. Поскольку мы используем слабое связывание, мы можем заменить реальную реализацию чем-то «ничего не делающим» и не приводящим к возникновению ошибок.

Помимо перечисленного, мы можем делать и другие вещи. Например, если мы живем в районе, где часто пропадает электричество, мы можем обеспечить продолжение работы компьютера путем подключения его к источнику бесперебойного питания (ИБП, Uninterrupted Power Supply — UPS), что продемонстрировано на рис. 1.8: мы включаем ИБП в стенную розетку, а компьютер подключаем к ИБП.

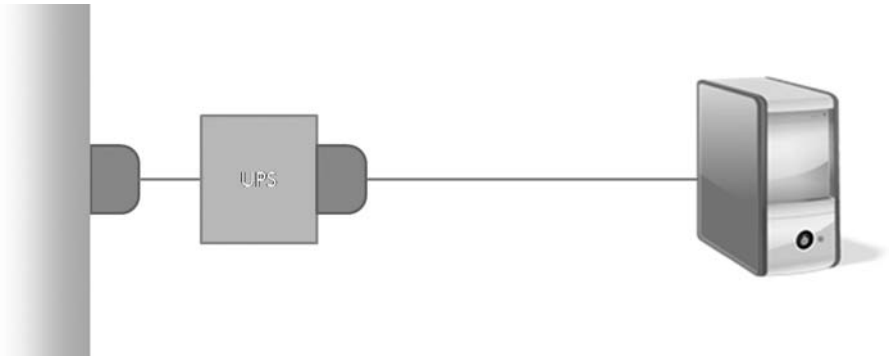


Рис. 1.8. Источник бесперебойного питания может быть подключен, чтобы обеспечить работу компьютера в случае пропадания электропитания. Это соответствует паттерну проектирования «Декоратор»

Компьютер и ИБП имеют разное назначение. Каждый из них выполняет единственную функцию, которая не пересекается с функцией другого устройства. Вполне вероятно, что ИБП и компьютер будут произведены двумя различными изготовителями, возможно, в разное время, и будут подключены также в различное время. Как показано на рис. 1.6, мы можем эксплуатировать компьютер и без ИБП, но еще мы можем, например, использовать и фен во время отключений электричества, подключая его к ИБП.

В разработке программного обеспечения такой способ подмены некоторой реализации другой реализацией того же интерфейса известен как шаблон проектирования «Декоратор» (Decorator). Он дает нам возможность постепенно добавлять новые возможности и *сквозные аспекты приложения* без полного переписывания или существенной модификации больших объемов уже существующего кода.

Другой способ добавления новой функциональности в уже разработанный код — компоновка существующей реализации интерфейса с новой реализацией. Когда несколько реализаций объединяются в одну, используется паттерн проектирования «Составной» (Composite). На рис. 1.9 показано, как это соответствует подключению различных устройств в удлинитель.

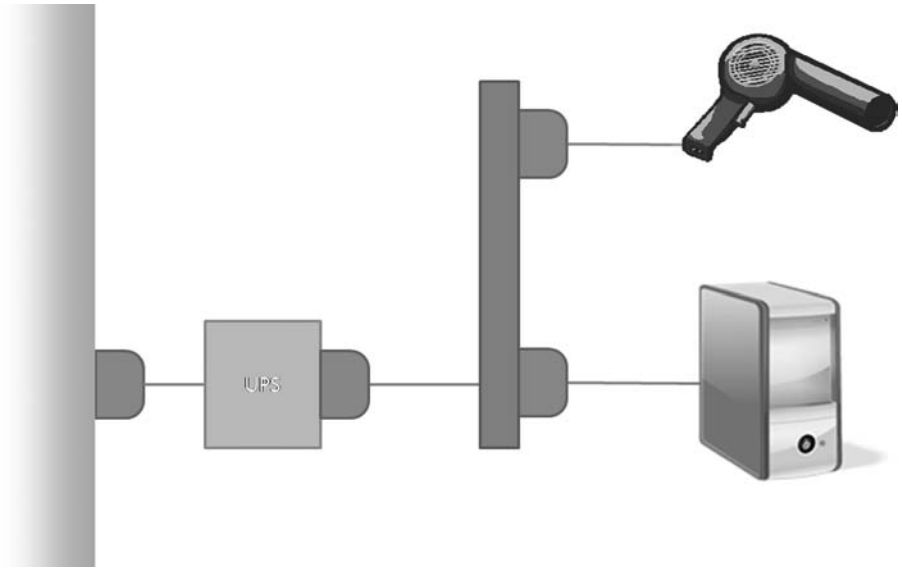


Рис. 1.9. Удлинитель позволяет подключать несколько устройств в одну розетку. Это соответствует паттерну проектирования «Составной»

Удлинитель имеет одну вилку, которую мы можем включить в одну розетку, а сам удлинитель представляет собой набор розеток, к которым может быть присоединено несколько устройств. Это позволяет нам подключать и отключать фен во время работы компьютера. По аналогии паттерн «Составной» упрощает задачу добавления или удаления функциональности за счет изменения набора объединенных реализаций интерфейса.

Наконец, заключительный пример. Иногда мы сталкиваемся с ситуацией, когда вилка не подходит к конкретной розетке. Если вам доводилось путешествовать по другим странам, то вы, вероятно, слышали, что розетки в разных уголках мира отличаются друг от друга. Если во время путешествия вы подключаете для зарядки какое-то устройство, например фотоаппарат, как показано на рис. 1.10, вам понадобится адаптер, чтобы сделать это. Соответственно, существует паттерн проектирования с таким названием.

Паттерн проектирования «Адаптер» работает подобно своему физическому тезке. Он может быть использован для совмещения двух связанных, но не стыкующихся (с точки зрения реализации) интерфейсов друг с другом. Это особенно полезно в ситуации, когда у вас есть некоторый API (интерфейс программирования приложений), разработанный неким сторонним разработчиком, и вы хотите использовать его как экземпляр интерфейса, с которым работает ваше приложение.

В модели розетки и вилки восхищает то, что и по прошествии десятилетий эта модель остается легкой для понимания и многофункциональной. До тех пор пока применяется данная инфраструктура, она может быть использована кем угодно и адаптирована к изменению потребностей и возникновению новых требований.

Еще более интересно, что, когда мы применяем эту модель к разработке программного обеспечения, все необходимые компоненты уже имеются в наличии в виде принципов и паттернов проектирования.

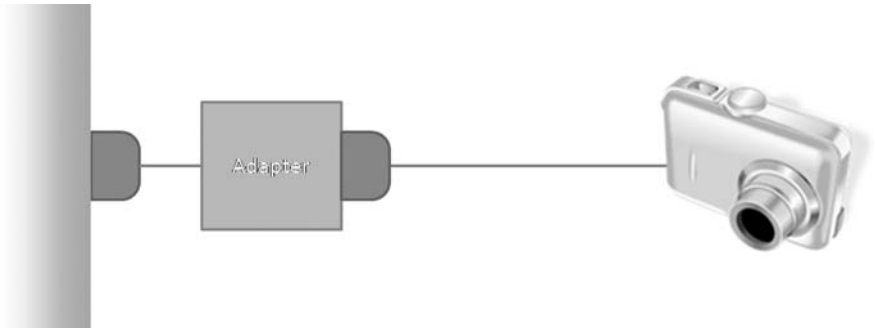


Рис. 1.10. Во время путешествий нам часто приходится использовать адаптер, чтобы включить устройство в розетку, соответствующую местным стандартам (например, для зарядки фотоаппарата). Эта ситуация соответствует паттерну проектирования «Адаптер»

Слабая связанность может значительно упростить поддержку кода.

Это легко понять. Программировать в соответствии с интерфейсами, а не с реализацией просто. Вопрос заключается в том, откуда берутся экземпляры? В определенном смысле этому вопросу посвящена вся эта книга.

Вы не можете создать новый экземпляр интерфейса — тем же способом, которым можно создать конкретный тип. Код, похожий на представленный ниже, не будет компилироваться:

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

Программируйте в соответствии с интерфейсом

Не компилируется

У интерфейсов не бывает конструкторов, поэтому компиляция невозможна. Объект `writer` должен быть создан при помощи другого механизма. Внедрение зависимостей решает эту проблему.

Теперь, после такого введения в внедрения зависимостей, полагаю, можно перейти к рассмотрению небольшого примера.

1.2. Hello DI (Привет, ВЗ)

В лучших традициях бесчисленного количества книг по программированию давайте рассмотрим простое консольное приложение, которое пишет текст `Hello DI` на экране. В данном разделе я продемонстрирую программный код и коротко,

не вдаваясь в детали, перечислю некоторые ключевые преимущества. В оставшейся части книги я объясню все подробнее.

1.2.1. Код для Hello DI

Возможно, вы видели примеры приложений `Hello World`, которые были написаны в одной строке кода. В нашем примере мы возьмем такой же предельно простой код и несколько его усложним. Почему? Мы постепенно подойдем к этому, но сначала посмотрим, на что будет похож наш `Hello World`, если он будет написан с использованием внедрения зависимостей.

Совместно работающие программные компоненты

Чтобы придать смысл структуре программы, мы начнем с рассмотрения кода метода `Main` консольного приложения, а затем я покажу вам сотрудничающие классы:

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

Программа должна вывести текст в консоль, поэтому она создает новый экземпляр класса `ConsoleMessageWriter`, который реализует именно такую функциональность. Программа передает созданный предназначенный для вывода сообщений компонент в класс `Salutation`, в результате объект `salutation` знает, куда писать сообщения. Поскольку взаимодействие компонентов организовано правильно, данная программа может быть выполнена, в результате чего сообщение будет выведено на экран.

Рисунок 1.11 демонстрирует взаимосвязи между совместно работающими компонентами приложения. Метод `Main` создает новые экземпляры как класса `ConsoleMessageWriter`, так и класса `Salutation`. Класс `ConsoleMessageWriter` реализует интерфейс `IMessageWriter`, который используется классом `Salutation`. Таким образом, класс `Salutation` применяет класс `ConsoleMessageWriter`, хотя эта непрямая связь и не отражена на схеме.

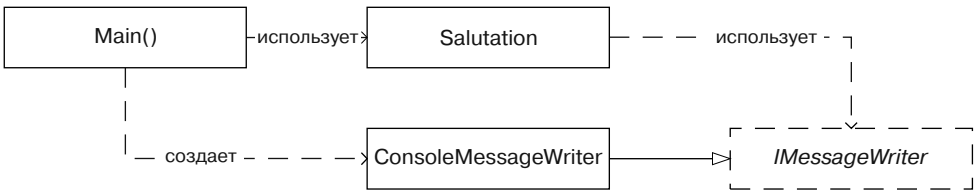
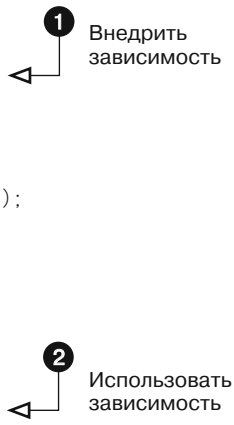


Рис. 1.11. Взаимосвязи между совместно работающими компонентами приложения

Основная логика приложения заключена в классе `Salutation`, код которого представлен в листинге 1.1.

Листинг 1.1. Класс Salutation

```
public class Salutation
{
    private readonly IMessageWriter writer;

    public Salutation(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }
        this.writer = writer;
    }

    public void Exclaim()
    {
        this.writer.Write("Hello DI!");
    }
}
```



Класс `Salutation` зависит от пользовательского интерфейса, называющегося `IMessageWriter`, и запрашивает экземпляр реализующего этот интерфейс типа в конструкторе ❶. Такой способ внедрения зависимости называется внедрением конструктора (Constructor Injection). Этот способ подробно описан в главе 4, где также содержится более подробный код похожего примера.

Экземпляр `IMessageWriter` затем используется в реализации метода `Exclaim` ❷, который записывает нужное сообщение в установленную зависимость.

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

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

Он может иметь и другие методы, но для нашего примера нам нужен только метод `Write`. Этот интерфейс реализуется классом `ConsoleMessageWriter`, который передается методом `Main` в класс `Salutation`:

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

Класс `ConsoleMessageWriter` реализует интерфейс `IMessageWriter` путем создания обертки (метод `Write`) для класса `Console`, входящего в базовую библиотеку классов (Base Class Library, BCL). Такая реализация представляет собой простое приложение паттерна проектирования «Адаптер», который обсуждался в подразделе 1.1.2.

Вы должны оценить преимущества, полученные от замены единственной строки кода двумя классами и интерфейсом, с общим количеством строк, равным 11. Таких преимуществ несколько.

1.2.2. Достоинства внедрения зависимостей

Почему представленный в предыдущем примере код является лучшим, чем единственная строка кода, используемая обычно для реализации приложения `Hello World` в C#? Что же, в данном примере использование внедрения зависимостей увеличило размер кода на 1100 %, но по мере того, как ваш код будет увеличиваться с одной строки до десятков тысяч строк, эти издержки будут уменьшаться, пока не станут совсем незаметными. В главе 2 приведен более сложный пример использования ВЗ, и хотя этот пример также является чрезмерно упрощенным по сравнению с реальными приложениями, вы, пожалуй, согласитесь, что издержки, связанные с использованием внедрения зависимостей, существенно уменьшились.

Я не виню вас, если вы считаете, что предыдущий пример использования внедрения зависимостей является совершенно надуманным, но примите во внимание следующее: классический пример приложения `Hello World` является по своей природе простой проблемой с ясно описанными и ограниченными требованиями. В реальном мире разработка программного обеспечения никогда не выполняется в таких условиях. Требования изменяются и зачастую являются нечеткими. Функции программ, которые вы должны реализовать, также имеют тенденцию усложняться. Внедрение зависимостей позволяет решить эти проблемы за счет использования слабого связывания. При этом мы получаем преимущества, перечисленные в табл. 1.1. Каждое из перечисленных преимуществ имеется всегда, но его важность оценивается по-разному в зависимости от способа применения внедрения зависимостей.

Таблица 1.1. Преимущества, получаемые от слабого связывания

Преимущество	Описание	Когда применяется
Динамическое связывание	Сервисы могут быть заменены на другие сервисы	Ценно в стандартном ПО, но, по-видимому, не так значимо в корпоративных приложениях, в которых среда времени исполнения, как правило, точно известна
Расширяемость	Код может быть дополнен и использован повторно способами, не требующими четкого определения	Ценно всегда
Параллельная разработка	Разработка кода может вестись параллельно	Ценна для крупных, сложных приложений; не так значима для малых простых приложений
Легкость в поддержке	Классы с точно определенными функциями легче поддерживать	Ценно всегда
Пригодность для тестирования	Классы могут тестироваться модульно	Ценно только, если вы осуществляете модульное тестирование (что вы обязательно должны делать)

Первым в табл. 1.1 я указал преимущество позднего связывания, поскольку, как подсказывает мой опыт, оно является важнейшим для большинства специалистов. По-видимому, основной причиной того, что архитекторы и разработчики не всегда могут понять преимущества слабого связывания, является то, что они никогда не рассматривали других вытекающих из этого преимуществ.

Динамическое связывание

Когда я объясняю преимущества программирования в соответствии с интерфейсами и использования внедрения зависимостей, наиболее существенным достоинством обычно считается возможность замены одного сервиса другим, поэтому они, как правило, сравнивают преимущества и недостатки только по этому пункту.

Помните, я предупреждал вас, что вы должны очистить свою память, прежде чем начнете учиться? Вы можете сказать, что вы знаете требования к вашему программному обеспечению настолько хорошо, что вам никогда не придется их менять, например менять ваш SQL Server на какой-либо другой. Однако требования со временем изменяются.

NOSQL, WINDOWS AZURE И АРГУМЕНТЫ В ПОЛЬЗУ СОЧЕТАЕМОСТИ

Несколько лет назад я часто оказывался в сложной ситуации, когда пытался убедить разработчиков и архитекторов в преимуществах внедрения зависимостей.

«Хорошо, итак, вы можете заменить ваш компонент доступа к реляционным данным на что-либо еще. На что? Существует ли какая-либо альтернатива реляционным базам данных?»

XML-файлы ранее не рассматривались в качестве убедительной альтернативы в высокопроизводительных сценариях для крупных корпоративных приложений. Эта ситуация значительно изменилась за последние годы.

Технология Windows Azure была анонсирована на PDC 2008 и предназначалась во многом для того, чтобы убедить даже организации, твердо придерживающиеся в области технологий принципа «только Microsoft», переосмыслить их позиции в отношении хранилищ данных. Сегодня Windows Azure — это уже реальная альтернатива реляционным базам данных, и мне остается только спросить, хотите ли вы, чтобы ваши приложения были «готовыми к облачным технологиям» (cloud-ready). Аргументация в пользу такой замены сегодня звучит намного сильнее.

Подобная тенденция просматривается и в так называемой NoSQL концепции, согласно которой приложения строятся вокруг ненормализованных данных — часто баз данных документов. Повышается степень важности и других концепций, таких как «порождение событий» (event sourcing)[1].

В подразделе 1.2.1 мы не использовали динамическое связывание, поскольку там вручную создавался новый экземпляр `IMessageWriter`. Это делалось путем создания нового экземпляра `ConsoleMessageWriter`. Однако можно ввести динамическое связывание, изменив всего один фрагмент кода. Речь идет об этой строке:

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

[1] *Martin Fowler* Event Sourcing, 2005: www.martinfowler.com/eaaDev/EventSourcing.htmls.

Чтобы добавить динамическое связывание, нужно заменить эту строку кода на что-то подобное следующему фрагменту:

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

Получая имя типа из конфигурационного файла приложения и создавая затем экземпляр типа `Type` с использованием этого полученного типа, вы можете задействовать механизмы класса `Reflection` для создания экземпляра `IMessageWriter`, не зная во время компиляции, какой конкретный тип будет применяться.

Выполнив перечисленные действия, вы сможете указывать требуемое имя типа для приложения `messageWriter`, задавая нужную настройку в конфигурационном файле приложения:

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

ВНИМАНИЕ

Этот пример использует некоторые сокращения, чтобы выделить главное. Фактически он соответствует антипаттерну «Ограниченное конструирование» (Constrained Construction), описанному в главе 5.

Слабое связывание допускает динамическое (позднее) связывание, поскольку имеется лишь единственное место, где создается экземпляр `IMessageWriter`. Так как класс `Salutation` использует только интерфейс `IMessageWriter`, разница не проявляется.

В коде примера `Hello DI` динамическое связывание должно обеспечить возможность записи сообщений в разные места, а не только на консоль — например, в базу данных или в файл. Можно добавлять такие возможности, даже если у вас нет четкого плана о том, как их использовать.

Расширяемость

Качественное программное обеспечение должно иметь возможность изменяться. Вам потребуется расширять уже реализованные функции и добавлять новые возможности. Слабое связывание позволяет эффективно пересобирать приложение подобно тому, как мы переподключаем электрические приборы при помощи розеток и вилок.

Пусть, например, вы хотите сделать код `Hello DI` более защищенным, чтобы он позволял отправлять сообщения только тем пользователям, которые обладают необходимыми правами. Листинг 1.2 показывает, как можно добавить такую возможность в код без изменения каких-либо уже существующих возможностей: вы просто добавляете новую реализацию интерфейса `IMessageWriter`.

Листинг 1.2. Расширение приложения Hello DI с добавлением функции безопасности

```
public class SecureMessageWriter : IMessageWriter
{
    private readonly IMessageWriter writer;
    public SecureMessageWriter(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }
        this.writer = writer;
    }

    public void Write(string message)
    {
        if (Thread.CurrentPrincipal.Identity
            .IsAuthenticated)
        {
            this.writer.Write(message);
        }
    }
}
```

❶ Проверка прав пользователя

❷ Запись сообщения

Класс `SecureMessageWriter` и реализует интерфейс `IMessageWriter` и одновременно использует его: он применяет внедрение конструктора для получения экземпляра `IMessageWriter`. Это стандартный способ использования шаблона проектирования «Декоратор», упоминавшегося в подразделе 1.1.2. Мы поговорим подробнее о «Декораторе» в главе 9.

В реализации метода `Write` прежде всего выполняется проверка, аутентифицирован ли текущий пользователь ❶. Только в этом случае задекорированное поле `writer` будет применяться для вывода сообщений ❷.

ПРИМЕЧАНИЕ

Метод Write в листинге 1.2 работает с текущим пользователем через окружающий контекст. Более гибкий, но и более сложный вариант — получать информацию о пользователе через еще одну внедрение конструктора.

Единственное место, в котором мы должны изменить существующий код, — это метод `Main`, поскольку существующие классы теперь используются по-другому:

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

Использовавшийся ранее экземпляр класса `ConsoleMessageWriter` заменен на новый класс `SecureMessageWriter`. Класс же `Salutation` не изменялся, поскольку он применяет только интерфейс `IMessageWriter`.

Слабое связывание позволяет писать код, который допускает расширяемость, но не позволяет вносить изменения. Такой подход называется принципом

открытости/закрытости (Open/Closed Principle). Единственное место, требующее модификации кода, — это стартовая точка приложения, называемая корнем сборки (Composition Root).

Класс `SecureMessageWriter` реализует связанную с безопасностью функциональность приложения, тогда как класс `ConsoleMessageWriter` реализует пользовательский интерфейс. Такое разделение позволяет обрабатывать эти аспекты независимо друг от друга и объединять их по мере необходимости.

Параллельная разработка

Разделение ответственности обеспечивает разработку кода в параллельно работающих командах. Когда проект по разработке программного обеспечения превышает некоторый размер, возникает необходимость разделения команды разработчиков на небольшие группы с целью удержания процесса под контролем. Каждая группа становится ответственной за определенный аспект приложения.

В соответствии с разделением зон ответственности каждая группа будет разрабатывать один или более модулей, которые затем будут собраны в готовое приложение. Если функции модулей, разрабатываемых каждой группой, будут определены нечетко, то, скорее всего, какие-то группы начнут зависеть от кода модулей, разработанных другими группами.

Поскольку в представленном выше примере классы `SecureMessageWriter` и `ConsoleMessageWriter` напрямую не зависят друг от друга, они могут разрабатываться параллельно работающими группами. Единственное, что эти группы должны согласовать, — это разделяемый интерфейс `IMessageWriter`.

Легкость в сопровождении

Если зона ответственности каждого класса в приложении является однозначно определенной и неизменяемой, степень сопровождаемости приложения повышается. Это известное достоинство так называемого принципа единичной ответственности, гласящего, что каждый класс должен иметь единственную обязанность.

Добавление новых возможностей в программу становится проще, так как ясно, какие изменения должны быть произведены. Чаще всего при этом даже не требуется изменять существующий код, а вместо этого нужно добавить новые классы и заново собрать приложение. Это еще одна демонстрация действия принципа открытости/закрытости.

Поиск и устранение неисправностей также упрощаются, поскольку область выявления дефектов сужается. При наличии ясно определенных обязанностей часто становится очевидным, в каком месте возникла проблема.

Пригодность к тестированию

Кого-то пригодность к тестированию вообще не заботит; для других она является абсолютным требованием. Лично я принадлежу ко второй категории: за свою карьеру я отклонил несколько предложений о работе, поскольку они предполагали работу с продуктами, принципиально непригодными для тестирования.

ОПРЕДЕЛЕНИЕ

Приложение признается пригодным для тестирования, если к нему применимо модульное тестирование.

Пригодность к тестированию является, по-видимому, наиболее спорным из перечисленных мною достоинств. Многие разработчики и архитекторы не практикуют модульное тестирование, поэтому в лучшем случае они считают данное достоинство не имеющим значения. Другие, как и я, полагают его весьма важным. Майкл Физерс даже ввел термин «унаследованное приложение» (Legacy Application), относящийся к любому приложению, которое не содержит модульных тестов[1].

Слабое связывание позволяет осуществлять модульное тестирование, поскольку потребители следуют принципу подстановки Лисков: они не определяют конкретные типы своих зависимостей. Это означает, что мы можем подставить в тестируемую систему (System Under Test, SUT) тестовые двойники рабочих модулей, как это показано далее в листинге 1.3.

Возможность осуществлять замену требуемых для реальной работы зависимостей на их аналоги, предназначенные для тестирования, является одним из свойств слабого связывания. Но я предпочитаю обозначать эту возможность как отдельное преимущество, поскольку результаты получаются различные.

ПРИГОДНОСТЬ К ТЕСТИРОВАНИЮ

Термин *пригодный к тестированию* является совершенно неточным. Тем не менее он используется самым широким образом в сообществе разработчиков программного обеспечения, как правило, теми, кто практикует модульное тестирование.

В принципе, любое приложение может быть протестировано, если пытаться вывести его из строя. Тесты могут быть выполнены людьми, работающими с приложением через его пользовательский интерфейс или через любой другой интерфейс, который предоставляет это приложение. Такое ручное тестирование занимает много времени и является сложным для реализации, поэтому гораздо более предпочтительным является тестирование автоматизированное.

Существуют различные типы автоматизированного тестирования, такие как модульное тестирование, интеграционное тестирование, тестирование производительности, стресс-тестирование и т. д. Поскольку модульное тестирование предъявляет невысокие требования к среде времени исполнения, оно оказывается наиболее эффективным и надежным типом тестирования; пригодность к тестированию часто оценивается именно в таком контексте.

Модульные тесты позволяют быстро оценить состояние приложения, но их разработка становится возможной только тогда, когда рассматриваемый модуль может быть как следует изолирован от его зависимостей. Бывает трудно правильно оценить уровень детализации тестового модуля, но нельзя не согласиться с тем, что один модуль наверняка не может включать в себя несколько модулей. Способность проводить тестирование программных модулей отдельно друг от друга весьма важна для модульного тестирования.

Только приложения, допускающие модульное тестирование, могут быть признаны пригодными для тестирования. Самый безопасный способ обеспечить пригодность для тестирования — вести разработку в соответствии с принципами разработки через тестирование.

[1] *Michael Feathers* Working Effectively with Legacy Code. — New York: Prentice Hall, 2004. — xvi.

Нельзя не отметить, что модульные тесты сами по себе не обеспечивают работоспособности приложения. Полное системное тестирование и промежуточное тестирование других типов по-прежнему должны быть проведены для выяснения ответа на вопрос: работает ли приложение так, как положено.

В зависимости от типа разрабатываемого приложения, я могу предусматривать или не предусматривать возможность динамического связывания, но я всегда забочусь о возможности протестировать приложение. Некоторые разработчики не уделяют пригодности для тестирования никакого внимания, но считают именно возможность позднего связывания ключевой для своих приложений.

ТЕСТОВЫЕ ДВОЙНИКИ

Существует общепринятая практика создавать такие реализации зависимостей, которые функционируют как дублеры уже разработанных или находящихся в стадии разработки реализаций. Такие реализации так и называются — «тестовые двойники» (Test Doubles), и они никогда не включаются в окончательную версию приложения. Они используются как заглушки для реальных зависимостей, когда последние еще недоступны или их применение по тем или иным причинам нежелательно.

Вокруг тестов-дублеров сформировался определенный набор терминов, и для них существует множество разновидностей, таких как заглушки (Stubs), подставные объекты (Mocks) и поддельные объекты (Fakes)[1].

Пример: модульное тестирование логики примера Hello

В подразделе 1.2.1 был представлен пример `Hello DI`. И хотя в нем был продемонстрирован окончательный вариант кода, в действительности я разрабатывал его по методике разработки через тестирование (TDD). В листинге 1.3 представлен самый важный модульный тест.

ПРИМЕЧАНИЕ

Не отчаивайтесь, если у вас нет опыта в модульном тестировании или в использовании динамических подставных объектов. Они будут периодически появляться в этой книге, но предварительных знаний, чтобы читать связанный с ними материал, не требуется.

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!"));
}
```

[1] *Gerard Meszaros* xUnit Test Patterns: Refactoring Test Code. —New York: Addison-Wesley, 2007. — 522.

Классу `Salutation` для работы нужен экземпляр интерфейса `IMessageWriter`, так что вы должны создать его. Для этого можно использовать любую реализацию, но в модульном тестировании может быть чрезвычайно полезен динамический подставной объект — в этом случае вам будет удобно использовать `Moq`[1], но вы можете применять и другие библиотеки или написать собственный код взамен. Важно использовать специальную тестовую реализацию `IMessageWriter`, чтобы гарантировать, что в каждый момент времени вы тестируете какой-то конкретный компонент приложения; в представленном варианте кода тестируется метод `Exclaim` класса `Salutation`, так что для выполнения этого теста вам не нужна будет никакая предназначенная для использования в готовом приложении реализация интерфейса `IMessageWriter`.

Для создания класса `Salutation` вы передаете в `Mock` экземпляр интерфейса `IMessageWriter`. Поскольку `writerMock` является экземпляром класса `Mock<IMessageWriter>`, то свойство `Object` представляет собой динамически созданный экземпляр интерфейса `IMessageWriter`. Внедрение требуемой зависимости через конструктор называется внедрением конструктора.

После подготовки испытываемой системы `Mock` может использоваться для проверки того, что метод `Write` был вызван с необходимым текстом. Вы применяете `Moq`, вызывая метод `Verify` с выражением, которое определяет контрольную фразу для проверяемого текста. Если метод `IMessageWriter.Write` будет вызван со строкой `"Hello DI!"`, то, соответственно, будет вызван метод `Verify`. Однако если метод `Write` не будет вызван или будет вызван с другим параметром (строкой), то метод `Verify` сгенерирует исключительную ситуацию и выполнение теста будет прекращено.

Слабое связывание обладает многими преимуществами: код становится более легким в разработке, поддержке и расширении, его также проще тестировать. Достичь этих результатов сравнительно несложно. Мы программируем с учетом интерфейсов, а не конкретных реализаций. Единственное серьезное препятствие — выяснить, как достать (получить) экземпляры этих интерфейсов. Внедрение зависимостей решает эту проблему путем внедрения зависимостей из внешнего окружения. Внедрение конструктора — это предпочтительный метод решения такой проблемы.

1.3. Что следует и что не следует внедрять

В предыдущем разделе я описывал побудительные причины, заставляющие нас в первую очередь обращаться к внедрению зависимостей. Если вы убедились в том, что применение слабого связывания весьма полезно, то, возможно, захотите реализовывать весь код своих приложений с применением этой технологии. Вообще-то это хорошая мысль. Когда вы будете принимать решение о способах разделения ваших модулей, слабое связывание особенно упростит вам работу.

Но вы не должны абстрагировать все на свете и делать абсолютно весь ваш код состоящим из подключаемых (pluggable) элементов. В данном разделе я продемонстрирую вам некоторые критерии, которые помогут вам принимать решения при проектировании ваших зависимостей.

---

[1] code.google.com/p/moq/.

Базовая библиотека классов .NET (Base Class Library, BCL) состоит из множества сборок (Assemblies). Всякий раз, когда вы пишете код, использующий тип из сборки BCL, вы добавляете в свой модуль зависимость. В предыдущем разделе обсуждался вопрос о важности слабого связывания и о том, что *программирование с упором на интерфейсы* является краеугольным камнем такого подхода.

Означает ли вышесказанное, что вы не можете использовать сборки BCL и включенные в них классы напрямую в своих приложениях? Что если вы захотите применять класс `XmlWriter`, определенный в сборке `System.Xml`?

Вы не должны оценивать все зависимости одним и тем же образом. Многие типы из BCL могут использоваться без риска изменения степени связанности приложения — но не все из них. Важно знать, как оценить разницу между типами, не представляющими опасности, и типами, которые могут повысить степень связанности приложения. Здесь рассматриваются в основном последние.

1.3.1. Швы

Всякий раз, когда мы собираемся программировать с учетом интерфейсов, а не конкретных классов, мы назовем это швом (Seam) в приложении. *Шов* — это место, в котором стыкуются составные части приложения, подобно тому, как сшиваются части одежды. Шов в программном обеспечении — это также место, в котором мы можем разделить приложение на части и затем работать с полученными модулями по отдельности.

Пример `Hello DI`, представленный в разделе 1.2, содержит шов между `Salutation` и `ConsoleMessageWriter`, как показано на рис. 1.12. Класс `Salutation` напрямую не зависит от класса `ConsoleMessageWriter`; чтобы написать сообщение, он использует интерфейс `IMessageWriter`. Вы можете разобрать приложение на части в этом месте и затем пересобрать его, используя другой способ вывода сообщений.

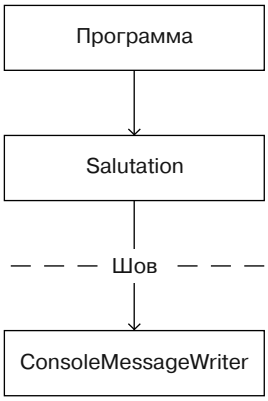


Рис. 1.12. Приложение Hello DI из раздела 1.2 содержит шов между классами Salutation и ConsoleMessageWriter, поскольку класс Salutation осуществляет вывод сообщения только через абстракцию интерфейса IMessageWriter

По мере изучения ВЗ будет полезно начать делить зависимости на *стабильные* и *нестабильные*, при этом принятие решений о месте размещения швов станет

для вас необходимостью. В следующих подразделах эти вопросы обсуждаются более подробно.

1.3.2. Стабильные зависимости

Многие модули из BCL никак не влияют на степень модульности приложения. Они содержат функциональность, которая допускает повторное использование и которую вы можете развить, чтобы сделать свой код более компактным.

Модули BCL всегда доступны в вашем приложении, поскольку для исполнения требуется наличие фреймворка .NET. Принцип параллельной разработки неприменим к этим модулям, поскольку они уже существуют, и вы всегда можете использовать эту же BCL-библиотеку в других приложениях.

По умолчанию вы можете считать большинство типов (но не все), включенных в BCL, безопасными, или, другими словами, стабильными зависимостями. Я называю их стабильными, поскольку они действительно являются таковыми, имеют тенденцию быть обратно-совместимыми и их вызовы всегда дают одинаковые вполне предсказуемые результаты.

Большинство стабильных зависимостей являются BCL-типами, но и другие зависимости могут быть стабильными. Важными критериями для признания зависимости стабильной являются следующие:

- класс или модуль уже существует;
- вы полагаете, что новые версии не будут содержать критических изменений;
- рассматриваемые типы содержат детерминированные алгоритмы;
- вы планируете никогда не заменять класс или модуль другим.

По иронии контейнеры внедрения зависимостей сами по себе имеют тенденцию к тому, чтобы быть стабильными зависимостями, так как они соответствуют всем перечисленным критериям. Когда вы принимаете решение разрабатывать свое приложение на основе того или иного контейнера зависимостей, вы рискуете оказаться ограниченным этим выбором для всего жизненного цикла приложения. Это еще одна причина для ограниченного использования контейнеров в корне сборки приложения.

Другие примеры могут включать специализированные библиотеки, реализующие алгоритмы, специфичные для вашего приложения. Если вы разрабатываете приложение, которое предназначено для химической отрасли, вы можете использовать библиотеки сторонних разработчиков, содержащие специфическую для химической промышленности функциональность.

В общем, зависимости могут считаться стабильными путем исключения: они являются стабильными, если не являются нестабильными.

1.3.3. Нестабильные зависимости

Добавлять швы в приложение не так просто, поэтому прибегать к такому варианту следует лишь при необходимости. Можно представить себе немало причин, чтобы скрывать зависимость за швом, но они все тесно связаны с преимуществами слабого связывания, обсуждавшимися в разделе 1.2.2.

Такие зависимости характеризуются негативным влиянием на одно или несколько преимуществ. Они не являются стабильными, так как не обеспечивают достаточно надежного фундамента приложениям, и я называю их *нестабильными* по этой причине. Зависимость признается нестабильной, если для нее справедливо хотя бы одно из следующих утверждений.

- Зависимость требует установки и конфигурирования среды времени исполнения. Такая ситуация, как правило, возникает с реляционными базами данных: если мы не скроем реляционную базу данных за швом, мы никогда не сможем заменить ее на какую-либо другую технологию. При этом трудности также возникают при подготовке и выполнении автоматизированного модульного тестирования.

  Базы данных являются хорошим примером типов из BCL, которые являются нестабильными зависимостями. Несмотря на то что LINQ to Entities — это включенная в BCL технология, ее применение требует наличия реляционной базы данных.

  Другие находящиеся вне процесса ресурсы, в частности очереди сообщений, веб-сервисы, и даже файловая система, также попадают в эту категорию. Обратите внимание, что эти типы являются нестабильными не потому, что они являются конкретными типами платформы .NET, а потому, что требуют наличия среды времени исполнения.

  Симптомы данного типа зависимости — потеря возможности динамического связывания и возможности расширения приложения, а также невозможность обеспечить пригодность приложения для тестирования.

- Зависимость еще не существует, а находится в процессе разработки. Характерный симптом такой ситуации — невозможность выполнения параллельной разработки.

- Зависимость не установлена на всех компьютерах в организации-разработчике. Такое может произойти в случае использования дорогостоящих библиотек сторонних разработчиков или зависимостей, которые не могут быть установлены на всех операционных системах. Наиболее общий симптом данной ситуации — невозможность обеспечения пригодности для тестирования.

- Зависимость содержит недетерминированное поведение. Это особенно важно при модульных тестах, поскольку все тесты должны быть детерминированными. Типичные источники недетерминизма — это случайные числа и алгоритмы, зависящие от текущих значений даты и времени.

  Распространенные источники недетерминизма, такие как `System.Random`, `System.Security.Cryptography.RandomNumberGenerator` или `System.DateTime.Now`, определяются в `mscorlib`, поэтому вы не можете обойтись без указания сборок, в которых они определены. Тем не менее вы должны рассматривать их как нестабильные зависимости, поскольку они могут нарушить тестопригодность приложения.

Нестабильные зависимости являются центральной концепцией ВЗ. Наличие в приложении нестабильных зависимостей в большей степени, чем наличие стабильных, требует обуславливает наличие швов в приложении. Соответственно, сборка приложения должна осуществляться при помощи внедрения зависимостей.

Теперь, когда вы познакомились с различиями между стабильными и нестабильными зависимостями, вы, вероятно, уже представляете границы применения технологии внедрения зависимостей. Слабое связывание является главным принципом разработки, поэтому внедрение зависимостей (как способствующий этому фактор) должно присутствовать повсюду в вашем коде. Нельзя провести четкую разграничительную линию между внедрением зависимостей и разработкой качественного программного обеспечения, но чтобы очертить круг проблем для оставшейся части книги, я кратко опишу область применения внедрения зависимостей.

1.4. Область применения внедрения зависимостей

Как было сказано в разделе 1.2, важным элементом внедрения зависимостей является разнесение различных обязанностей по разным классам. Единственная обязанность, которую мы выносим из классов, — это создание экземпляров зависимостей.

Когда класс отказывается от контроля за зависимостями, он уступает не просто решение о выборе конкретной реализации. Однако как разработчики мы получаем определенные преимущества.

ПРИМЕЧАНИЕ

Как разработчики мы получаем управление за счет удаления такого контроля из классов, которые используют зависимости. Здесь проявляется принцип единичной ответственности: такие классы должны действовать только в пределах своей области ответственности, без учета того, как именно создавались зависимости.

На первый взгляд, позволяя классу отказаться от контроля над тем, какие объекты будут создаваться, мы принимаем неудачное решение. Но как разработчики мы не теряем этот контроль, мы просто переносим его в другое место.

Однако композиция объектов — это не единственная сторона управления, от которой мы отказываемся, поскольку класс также теряет возможность управлять жизненным циклом объекта. Когда экземпляр зависимости внедряется в класс, потребитель зависимости не знает, ни когда он был создан, ни когда он станет недоступным. Как правило, это не волнует потребителя, но в каких-то случаях это может оказаться важным.

Внедрение зависимостей предоставляет нам возможность управления зависимостями унифицированным способом. Когда потребители напрямую создают и настраивают экземпляры зависимостей, каждый из них может выполнять это каким-то собственным способом, который может не согласовываться с тем, как это делают другие потребители. В данном случае у нас нет возможности централизованного управления зависимостями и нет простого способа реализовать сквозные аспекты приложения. Используя внедрение зависимостей, мы получаем возможность перехватывать каждый экземпляр зависимости и выполнять с ним какие-то действия, прежде чем он будет передан потребителю.

При применении внедрения зависимостей мы можем собирать приложения, имея в то же время возможность перехвата зависимостей и управления их жизненными циклами. Композиция объектов, перехват и управление жизненным циклом являются тремя измерениями внедрения зависимостей. Далее я опишу их кратко, а более детальное описание будет дано в части 3.

1.4.1. Композиция объектов

Чтобы воспользоваться преимуществами расширяемости, динамического связывания и параллельной разработки, необходимо иметь возможность собирать приложения из классов (рис. 1.13). Возможность композиции объектов часто является основной причиной введения внедрения зависимостей в приложение. Первоначально внедрение зависимостей являлось синонимом композиции объектов; она является единственным аспектом, обсуждаемым в оригинальной статье Мартина Фаулера[1].

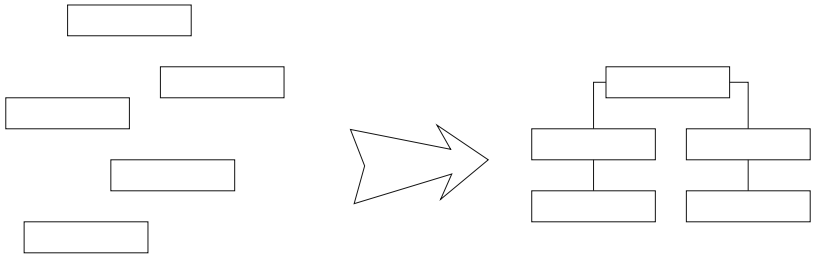


Рис. 1.13. Композиция объектов означает, что модули могут быть собраны в приложения

Сборка классов в приложения может осуществляться несколькими способами. Когда мы говорили о динамическом связывании, я использовал файл конфигурации и динамическое создание объекта, чтобы вручную собрать приложение из доступных модулей. Но я мог также использовать конфигурацию зависимости в коде (Code as Configuration) или контейнер внедрения зависимостей. Мы вернемся к обсуждению этих моментов в главе 7.

Несмотря на то что на момент появления этого термина (Code as Configuration) внедрение зависимостей было тесно связано с композицией объектов, другие ее аспекты также имеют важное значение.

1.4.2. Жизненный цикл объектов

Класс, передавший управление своими зависимостями, теряет более чем просто возможность выбирать конкретные реализации абстракций. Он также теряет возможность управления как моментом создания экземпляра, так и моментом, когда этот экземпляр становится недоступным.

---

[1] *Martin Fowler* Inversion of Control Containers and the Dependency Injection pattern, 2004: http://martinfowler.com/articles/injection.html.

В платформе .NET решением этих проблем занимается сборщик мусора (garbage collector). Потребитель может хранить зависимости, внедренные в этот потребитель, и использовать их в течение неограниченно долгого периода времени. Когда зависимости становятся ненужными, они могут быть удалены из области видимости потребителя. Если при этом на них не ссылаются другие классы, сборщик мусора утилизирует их.

Что произойдет, если два потребителя используют один и тот же тип зависимости? На рис. 1.14 изображено, что мы можем выбрать вариант внедрения разных экземпляров в каждого потребителя, тогда как рис. 1.15 показывает, что у нас есть и вариант разделения одного и того же экземпляра между несколькими потребителями. При этом, с точки зрения потребителя, разницы между этими вариантами нет. В соответствии с принципом подстановки Лисков потребитель должен оценивать все экземпляры данного интерфейса одинаково.

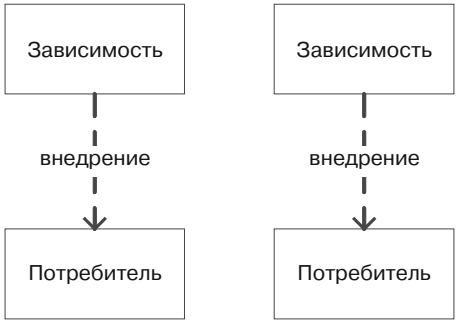


Рис. 1.14. Каждый из потребителей, разделяющих один и тот же тип зависимостей, внедряется собственным экземпляром

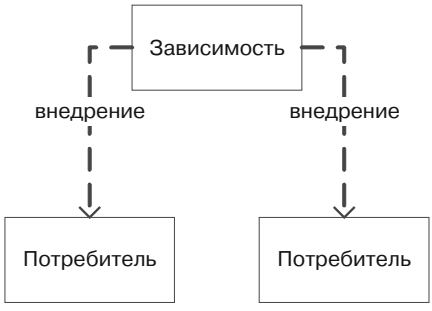


Рис. 1.15. Все потребители, использующие один и тот же тип зависимостей, внедряются одним и тем же совместно используемым экземпляром

Поскольку зависимости могут использоваться совместно, отдельно взятый потребитель, возможно, не сможет управлять их жизненным циклом. Это не составляет проблемы в случае, когда управляемый объект может выходить из области видимости потребителя и подпадать под сборку мусора, но если зависимости реализуют интерфейс `IDisposable`, проблемы могут возникнуть.

Таким образом, управление жизненным циклом — это отдельное измерение внедрения зависимостей, достаточно важное, чтобы я посвятил ему целую главу 8.

Потеря контроля над зависимостью означает также потерю управления его жизненным циклом; какой-то расположенный выше в стеке вызовов программный код должен управлять жизненным циклом зависимости.

1.4.3. Перехват

Когда мы делегируем управление третьей стороне посредством зависимостей, как показано на рис. 1.16, мы можем модифицировать зависимости перед передачей в классы, использующие их. Обозначенные точками стрелки показывают направление действия — зависимость направлена в противоположную сторону.

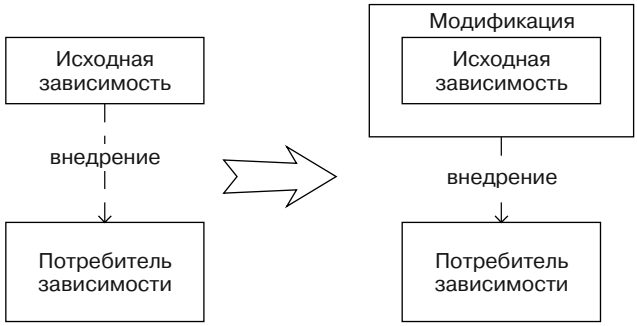


Рис. 1.16. Вместо того чтобы внедрять оригинал зависимости, мы можем предварительно модифицировать ее, оборачивая другой класс вокруг этого оригинала, прежде чем передать оригинал потребителю

В примере `Hello DI` я первоначально внедрил экземпляр `ConsoleMessageWriter` в экземпляр класса `Salutation`. Затем, изменив пример, я добавил в приложение функцию безопасности путем создания нового класса `SecureMessageWriter`, который разрешает дальнейшую обработку классу `ConsoleMessageWriter` только в случае, если пользователь имел необходимые права доступа, то есть был идентифицирован. Такой подход обеспечивает реализацию *принципа единичной ответственности*.

Все это может быть реализовано только в случае, если программирование ведется с упором на интерфейсы; вызовы зависимостей должны быть всегда *абстрактными*. Так, для класса `Salutation` не имеет значения, является ли переданный ему экземпляр `IMessageWriter` экземпляром класса `ConsoleMessageWriter` или `SecureMessageWriter`. Класс `SecureMessageWriter` является оберткой для `ConsoleMessageWriter`, выполняющего реальную работу.

ПРИМЕЧАНИЕ

Перехват — это способ применения шаблона проектирования «Декоратор». Не беспокойтесь, если вы не знакомы с ним — я сделаю необходимые пояснения в главе 9, которая будет целиком посвящена перехватам.

Возможность выполнения перехватов подводят нас к основам аспектно-ориентированного программирования (Aspect-Oriented Programming) — концепции,

тесно связанной с обсуждаемой здесь темой, которая, тем не менее, лежит за пределами этой книги. Используя перехваты, можно реализовывать сквозные аспекты приложения, такие как ведение логов, аудит, управление доступом, валидация и т. д., в четкой структуре, позволяющей поддерживать разделение аспектов.

1.4.4. Внедрение зависимостей в трех измерениях

Хотя внедрение зависимостей начинается как набор шаблонов, предназначенных для решения проблем композиции объектов, этот термин впоследствии стал подразумевать и жизненный цикл объектов, и перехват. В настоящее время я подразумеваю под внедрением зависимостей некое органичное смешение всех этих трех областей.

В этой триаде композиция объектов занимает главенствующее положение, поскольку без гибкой композиции объектов не будет никакого перехвата, да и управление жизненным циклом объектов вряд ли потребуется. Композиция объектов доминировала в этой главе и будет доминировать во всей книге, но мы не должны забывать про другие аспекты. Композиция объектов создает основу, управление жизненным циклом объектов устраняет ряд существенных сторонних эффектов, но все преимущества можно получить, только если реализуется еще и перехват.

В части 3 я отвожу по главе рассмотрению каждого из этих аспектов, но там же дан и общий обзор, поскольку требуется хорошее понимание того, что на практике внедрение зависимостей является более чем просто композицией объектов.

1.5. Резюме

Внедрение, или инъекция, зависимостей является не самоцелью, а средством достижения цели. Это наилучший способ реализации слабого связывания и важная часть кода, который легко и удобно поддерживать. Польза, которую мы получаем от применения слабого связывания, не всегда очевидна, но проявляется с течением времени, по мере нарастания размера и сложности разработанного кода. Сильно связанный код неизбежно будет запутываться, в то время как правильно спроектированный, слабо связанный код останется легко поддерживаемым и развиваемым. Безусловно, для получения по-настоящему гибкой архитектуры (Supple Design) требуется намного больше, чем простое наличие слабой связанности, но программирование с упором на интерфейсы является безусловным исходным требованием.

Внедрение зависимостей представляет собой не больше чем набор принципов и паттернов проектирования. Оно является в большей степени способом анализа и проектирования программного кода, чем набором инструментов и технологий. Это важно учитывать, работая со слабым связыванием и внедрением зависимостей. Чтобы быть эффективными, эти технологии должны быть реализованы во всем вашем коде, а не где-то в одном-единственном месте.

СОВЕТ

Внедрение зависимостей должно быть всеобъемлющим. Вы не сможете легко модифицировать ранее написанный код, чтобы реализовать слабое связывание.

Относительно внедрения зависимостей существует много заблуждений. Многие считают, что ВЗ позволяет решать только частные проблемы типа позднего связывания или модульного тестирования. Но хотя внедрения зависимостей особенно удобны при работе именно с этими аспектами программного обеспечения, диапазон их применения все же намного шире. Главная цель внедрения — обеспечение хорошей поддерживаемости кода.

В начале главы я указывал, что вы должны забыть все, что вы знаете о внедрении зависимостей, прежде чем приступите к его изучению. Это остается справедливым и для оставшейся части книги: вы должны освободить свой мозг. В одном великолепном посте своего блога Николас Блюмхарт пишет:

> *Словарь или ассоциативный массив* являются *одними из первых конструкций, с которыми мы знакомимся в разработке программного обеспечения. Легко провести аналогию между словарем и контейнером с инверсией управления, который собирает объекты, используя внедрение зависимостей*[1].

Основная идея внедрения зависимостей как сервиса, моделируемого через строки словаря, вытекает непосредственно из антипаттерна «Локатор сервисов». Именно поэтому я последовательно хочу избавить вас даже от самых простых, но неверных предположений. Учитывая все это, при обсуждении словарей мы говорим о вещах, которые можно отнести к «древнейшей истории программирования».

Назначение внедрения зависимостей — упростить поддержку кода. Код небольшого размера, как, например, код классического примера `Hello World`, легко поддерживается просто именно в силу своей компактности. Именно поэтому внедрение зависимостей кажется натянутым в небольших простых примерах. По мере увеличения размера кода польза от применения внедрения зависимостей становится все более и более ощутимой. Я выделил всю следующую главу для демонстрации более крупного и более сложного примера, чтобы вы более четко представляли себе такую пользу.

---

[1] *Blumhardt, Nicholas* Container-Managed Application Design, Prelude: Where does the Container Belong?, 2008: http://blogs.msdn.com/b/nblumhardt/archive/2008/12/27/container-managed-application-design-prelude-where-does-the-container-belong.aspx.

2 Детальный пример

Меню:

- детальный пример;
- неверная реализация;
- правильная реализация.

Знание того, что беарнский соус — это «эмульгированный соус, приготовленный из яичного желтка и масла», не наделит вас волшебным образом умением готовить его. Наилучший способ научиться этому — практика; при этом хороший пример может послужить мостом над пропастью между теорией и практикой. Будет полезно понаблюдать за тем, как готовит беарнский соус профессиональный повар, до того как вы начнете делать это сами.

Во введении во внедрение зависимостей в первой главе я сделал небольшой обзор, чтобы помочь вам разобраться с назначением этой технологии и ее основными принципами. Однако представленный там простой пример не позволяет полностью оценить внедрения зависимостей на практике. Внедрение зависимостей — это способ обеспечить слабое связывание, которое, в свою очередь, является основным способом решения проблем, связанных со сложностью разрабатываемых программных систем.

Большинство программ являются сложными в том смысле, что они должны одновременно решать множество проблем. Помимо проблем бизнес-логики (которые могут быть сложными и сами по себе), программное обеспечение должно решать задачи, связанные с безопасностью, диагностикой, рабочими процессами и расширяемостью. Вместо того чтобы смешивать все эти проблемы в один большой ком, слабое связывание рекомендует решать каждую проблему независимо от других. Решать проблемы независимо друг от друга просто, но в конечном итоге все равно придется соединять их вместе. Для лучшего понимания внедрения зависимостей рассмотрим более сложный пример. Эта глава содержит два варианта одного и того же примера (рис. 2.1). Сначала будет продемонстрировано, как просто написать сильно связанный код. Затем код этого же приложения будет переписан в соответствии с принципами слабого связывания. Оба примера содержат как собственно код, так и анализ соответствующего подхода. Если вы хотите побыстрее познакомиться со слабо связанным кодом, вы можете пропустить первый подраздел.

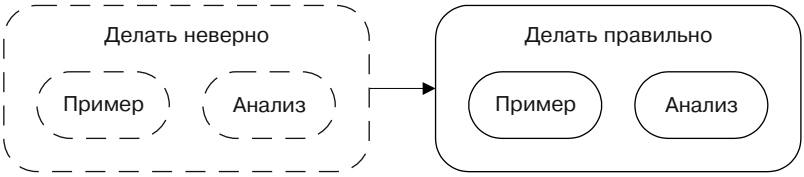


Рис. 2.1. Два варианта одного и того же примера

ПРЕДУПРЕЖДЕНИЕ

Основное назначение слабо связанного кода — эффективное решение проблем, связанных со сложностью кода, и нам понадобится сравнительно сложный пример, чтобы проиллюстрировать сложные концепции. Вы должны быть готовыми к тому, что большинство примеров в этой книге будут сложными и в них будет задействовано множество классов из множества библиотек. Без сложности здесь просто не обойтись.

Я выделил для рассмотрения этого комплексного примера целую главу. Мне представляется важным сравнить и противопоставить слабо связанный код и более традиционное сильно связанное решение, так что в данной главе вы найдете оба варианта реализации одних и тех же функций. Сначала я продемонстрировал, насколько просто писать сильно связанный код. Затем я реализовал ту же функциональность с использованием внедрения зависимостей. Вы можете пропустить пример с сильно связанным кодом, если вас интересует только работа со слабым связыванием. После изучения данной главы вы должны ясно понимать, как следует использовать внедрение зависимостей для получения слабо связанного кода.

2.1. Неверная реализация

Идея разработки слабо связанного кода не выглядит слишком неоднозначной, но существует огромная пропасть между целью и практикой. Прежде чем я продемонстрирую способы использования внедрения зависимостей для построения слабо связанного приложения, я хочу рассказать, как просто выполнить эту работу неправильным образом.

Распространенный подход при использовании слабо связанного кода — разработка трехуровневого приложения. Кто угодно может нарисовать его схему, и рис. 2.2 подтверждает, что сделать это могу и я. На рисунке представлена архитектура стандартного трехуровневого приложения. Это простейший часто встречающийся вариант *n*-уровневой архитектуры приложений, в которой приложение формируется из *n* слоев, каждый из которых состоит из одного или более модулей. Некоторые варианты *n*-уровневых схем будут содержать расположенные по вертикали прямоугольники, охватывающие несколько уровней приложения. Они часто используются для представления сквозных аспектов приложения, таких как безопасность или регистрация пользователей.

Разработка приложения с трехуровневой схемой выглядит обманчиво простой. Но заявляя, что вы нарисуете такую схему, вы словно сообщаете о том, что к вашему стейку будет подан беарнский соус. То есть вы ставите перед собой цель, в до-

стижении которой вовсе даже не уверены. Как вы скоро увидите, в результате можно получить нечто, весьма отличающееся от задуманного.

Рис. 2.2. Разработка трехуровневого приложения

2.1.1. Разработка сильно связанного приложения

Существует много способов разработки гибких и легко поддерживаемых сложных приложений[1], но *n*-уровневая архитектура обычно сводится к широко распространенному трехуровневому подходу. Сложность заключается в его правильной реализации.

Вооружившись трехуровневой схемой, представленной на рис. 2.2, вы можете начинать разработку приложения.

Познакомьтесь — Мэри Рован

Мэри Рован — профессиональный .NET-разработчик, работающая в компании — сертифицированном партнере Microsoft. В основном компания занята разработкой веб-приложений. Мэри 34 года, разработкой программного обеспечения она занимается 11 лет. Ее опыт позволяет ей считаться одним из наиболее квалифицированных разработчиков в компании, и она, помимо выполнения своих постоянных обязанностей ведущего разработчика, часто выступает в качестве наставника для молодых специалистов.

В целом Мэри нравится ее работа. Но ее раздражает то, что зачастую отсутствуют временные рамки для реализации проекта. В результате ей и ее коллегам приходится работать сверхурочно и в выходные, чтобы выполнить работу в срок. Она подозревает, что должны существовать более эффективные способы разработки. Чтобы узнать больше об этой эффективности, Мэри она покупает множество книг по программированию, но у нее редко находится время, чтобы прочитать их.

---

[1] В настоящее время наиболее многообещающей альтернативой n-уровневым приложениям является архитектурный стиль, связанный с шаблоном «разделение ответственности на команды и запросы» (CQRS). Дополнительную информацию можно получить из статьи Rinat Abdullin, “CQRS Starting Page,” http://abdullin.com/cqrs.

Большую часть своего редкого свободного времени Мэри проводит с мужем и двумя дочерьми. Мэри увлекается пешим горным туризмом. Еще она большая любительница готовить и, разумеется, знает, как готовить беарнский соус.

Мэри получила задание разработать новое приложение в области электронной коммерции по технологии ASP.NET MVC с использованием фреймворка Entity и СУБД SQL Server в качестве хранилища данных. Чтобы приложение было максимально модульным, оно должно быть трехуровневым.

Первая функция приложения, которая должна быть реализована, — это простой список предлагаемых продуктов, заполняемый из таблиц базы данных и отображаемый на веб-странице. Пример представлен на рис. 2.3. На рисунке показан простой список предлагаемых товаров и их цены (kr. — символ валюты для обозначения датских крон). Если просматривающий список пользователь обладает правом на скидки, цена всех представленных продуктов должна быть уменьшена на пять процентов.

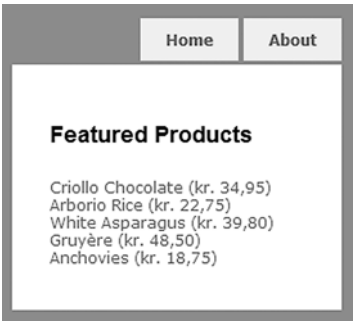


Рис. 2.3. Скриншот веб-приложения электронной коммерции, разрабатываемого Мэри

Давайте заглянем через плечо Мэри и посмотрим, как она реализовала эту первую функцию приложения.

Слой данных

Поскольку источником данных для приложения является база данных, Мэри решила начать разработку с реализации слоя данных. Первое, что она должна сделать, — определить саму таблицу базы данных. Для выполнения этой задачи Мэри использовала SQL Server Management Studio. Структура созданной таблицы показана на рис. 2.4. Для решения этой задачи также можно попробовать написать скрипт на языке T-SQL или создать таблицу вручную, без использования Visual Studio, а также можно попробовать какие-то другие инструменты.

Для реализации слоя доступа к данным (Data Access Layer) Мэри добавила в свое приложение новую библиотеку. Она использовала входящий в состав Visual Studio инструмент Мастер сущностной модели данных (Entity Data Model Wizard) для генерации модели (схемы) из базы данных, которую она только что создала. Чтобы привести модель к окончательному виду, она изменила в ней несколько имен, как показано на рис. 2.5. Мэри изменила название колонки `Featured` на `IsFeatured`, а также изменила несколько имен в сгенерированном объекте `ObjectContext` (не показан на рисунке).

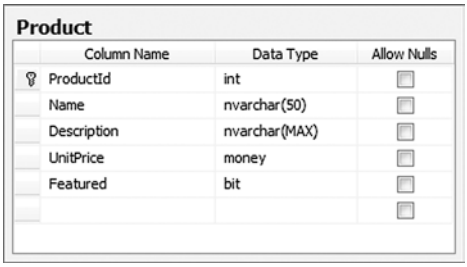


Рис. 2.4. Мэри создала таблицу Product, используя SQL Server Management Studio

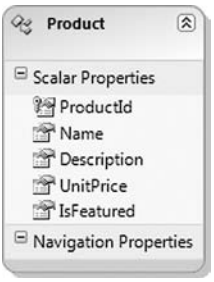


Рис. 2.5. Сущность Product, сгенерированная из таблицы Product базы данных, показанной на рис. 2.4

ПРИМЕЧАНИЕ

Не волнуйтесь, если вы не знакомы с фреймворком Entity. Детали реализации доступа к данным в контексте нашего разговора неважны, так что вы можете продолжать рассмотрение примера, даже если вам лучше знакома какая-то другая технология доступа к данным.

Сгенерированные `ObjectContext` и сущность `Product` являются общедоступными (Public) типами, находящимися в одной сборке. Мэри знает, что позднее она будет добавлять в свое приложение другие функции, но компонент доступа к данным, требуемый для реализации начальной функциональности, уже сформирован.

На рис. 2.6 показано, насколько продвинулась Мэри в реализации многоуровневой архитектуры, представленной на рис. 2.2. Мэри реализовала в своем приложении слой доступа к данным (Data Access Layer). Чтобы полностью реализовать необходимую функциональность, ей еще нужно запрограммировать слой бизнес-логики (Domain Logic Layer) и слой пользовательского интерфейса (User Interface Layer).

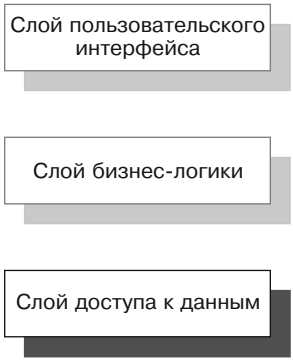


Рис. 2.6. Реализован слой доступа к данным

Поскольку слой доступа к данным уже реализован, логично будет далее реализовать слой бизнес-логики.

Слой бизнес-логики

Поскольку никакие специальные бизнес-требования для данного примера не определены, список продуктов (`Products`), формируемый сгенерированным компонентом

`ObjectContext`, может быть использован непосредственно в слое пользовательского интерфейса.

ВНИМАНИЕ

Практически во всех случаях, кроме приложений, которые выполняют только генерацию отчетов данных, бизнес-логика в программах присутствует. Ее можно не реализовывать в самом начале разработки, но, по мере расширения функциональности, в приложение будет постепенно добавляться код, определяющий основные и дополнительные бизнес-правила и допущения. Попытка реализовать данную логику в слое пользовательского интерфейса или в слое доступа к данным приведет к значительному усложнению процесса разработки. Наилучшим решением в таком случае является создание отдельного слоя бизнес-логики.

В требованиях к разрабатываемому Мэри приложению указывается, что привилегированные пользователи должны видеть цены на продукты с пятипроцентной скидкой. Мэри не знает, каким образом определяется привилегированность пользователей, поэтому она консультируется у своего коллеги Йенса.

Мэри: Мне нужно реализовать бизнес-логику, в соответствии с которой привилегированные пользователи имеют скидку в пять процентов.

Йенс: Это просто. Умножай на 0,95.

Мэри: Спасибо, но я хотела узнать не это. Я хотела спросить, как определяются привилегированные пользователи?

Йенс: Ах, вон что. А это веб-приложение или настольное приложение?

Мэри: Это веб-приложение.

Йенс: Тогда ты можешь создать профиль для каждого пользователя и определить в нем свойство IsPreferredCustomer. Профиль можно определить в HttpContext.

Мэри: Помедленнее, Йенс. Этот код должен находиться в слое бизнес-логики. Это библиотека. И в ней нет HttpContext.

Йенс: Так [задумывается]. Я считаю, ты можешь воспользоваться функциональностью Profile из ASP.NET, чтобы получить данное свойство пользователя. Затем ты можешь передать это свойство в свою бизнес-логику как булево (Boolean) значение.

Мэри: Ну не знаю...

Йенс: Такой подход обеспечит тебе еще и хорошее разделение ответственности, так как твоя бизнес-логика не должна быть связана с безопасностью. Ты же знаешь: Принцип Единственной Ответственности! Это гибкий способ получения требуемого результата!

Мэри: Я думаю, ты попал в точку.

ВНИМАНИЕ

Йенс делает выводы на основе своих технических знаний технологии ASP.NET. Но по мере того, как дискуссия уводит его от этой комфортной для него области, он просто засыпает Мэри набором пустых слов. Он не понимает того, о чем говорит: он неверно использует концепцию разделения проблем, совершенно ошибается, говоря о принципе единственной ответственности, и упоминает гибкость (Agile) только потому, что недавно слышал, как кто-то с большим энтузиазмом говорил о ней.

Вооружившись советами Йенса, Мэри создает новый проект библиотеки C# и добавляет класс, называемый ProductService, код которого представлен в листинге 2.1. Чтобы скомпилировать класс ProductService, она добавила ссылку на свою библиотеку доступа к данным, в которой определен класс CommerceObjectContext.

Листинг 2.1. Разработанный Мэри класс ProductService

```
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 радуется, что ей удалось соединить технологию доступа к данным (LINQ to Entities), конфигурирование и бизнес-логику в классе ProductService. Она передала вопрос оценки привилегированности пользователя вызывающей программе, определив в написанном методе параметр isCustomerPreferred, значение которого она использует при вычислении скидки на все свои продукты.

Написанный код можно улучшить, заменив константное значение скидки (0.95) на конфигурируемое число, но в данный момент это не требуется. Мэри почти закончила работу — ей осталось только реализовать интерфейс пользователя. Эту работу Мэри решила отложить до следующего дня.

Рисунок 2.7 показывает, насколько Мэри продвинулась в реализации представленной на рис. 2.2 архитектуры. По сравнению с рис. 2.6, добавился слой бизнес-логики. Остается реализовать только слой пользовательского интерфейса.

Слой пользовательского интерфейса

На следующий день Мэри продолжает работать с приложением электронной коммерции, добавляя новое приложение ASP.NET MVC к своему решению.

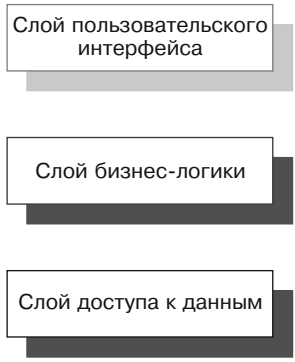


Рис. 2.7. На данный момент Мэри реализовала слой доступа к данным и слой бизнес-логики

ПРИМЕЧАНИЕ

Не беспокойтесь, если вы не знакомы с фреймворком ASP.NET MVC. Тонкости функционирования этого фреймворка не главная тема нашей книги. Основным является вопрос о том, как используются зависимости, что не так сильно обусловлено платформой.

КРАТКИЙ КУРС ASP.NET MVC

ASP.NET MVC получил свое название от паттерна проектирования «Модель — представление — контроллер» (Model-View-Controller). Если выделить только те аспекты этого шаблона, которые имеют отношение к предмету нашей беседы, то наиболее существенным будет то, что, когда система получает веб-запрос (Request), контроллер обрабатывает его, возможно, используя модель (или домен) для получения данных, и формирует ответ (Response), который в конечном итоге отображается представлением.

Контроллер, как правило, — это класс, являющийся наследником абстрактного класса `Controller`. У него имеется один или более методов, определяющих действия по обработке запроса; например, класс `HomeController` содержит метод `Index`, обрабатывающий запрос к странице, указанной по умолчанию (главной странице).

Когда обрабатывающий действие метод завершает свою работу, он передает полученную модель в представление, используя экземпляр класса `ViewResult`.

Листинг 2.2 демонстрирует реализацию метода `Index` в классе `HomeController` для выбора продуктов со скидками из базы данных и передачи их в представление. Для компиляции этого кода в него должны быть добавлены ссылки как на библиотеку доступа к данным, так и на библиотеку бизнес-логики, поскольку класс `ProductService` определен в библиотеке бизнес-логики, а класс `Product` — в библиотеке доступа к данным.

Листинг 2.2. Метод `Index` указанного по умолчанию класса-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();
}
```

Поскольку свойство `User` класса `HomeController` поддерживается жизненным циклом `ASP.NET MVC`, оно автоматически заполняется информацией о текущем пользователе, расположенной в соответствующем объекте. Поэтому Мэри использует это свойство, чтобы определить, является ли пользователь привилегированным. Имея данную информацию, она может вызвать бизнес-логику для получения списка продуктов со скидками. Через некоторое время я вернусь к этому вопросу, поскольку здесь имеется один подвох, но сейчас я позволю Мэри попытаться обнаружить это самостоятельно.

Полученный список продуктов должен быть отображен в представлении `Index`. Следующий листинг показывает разметку для этого представления.

Листинг 2.3. Разметка для представления `Index`

```
<h2>Featured Products</h2>
<div>
<% var products =
       (IEnumerable<Product>)this.ViewData["Products"];
   foreach (var product in products)
   { %>
   <div>
   <%= this.Html.Encode(product.Name) %>
   (<%= this.Html.Encode(product.UnitPrice.ToString("C")) %>)
   </div>
<% } %>
</div>
```

Получение продуктов, информацию о которых внес контроллер

Технология ASP.NET MVC позволяет создавать веб-страницы, на которых стандартные элементы HTML комбинируются с фрагментами необходимого ASP-кода, внедренного для доступа к объектам. Такие объекты которые создаются и передаются контроллером, создавшим данное представление. В данном случае метод `Index` класса `HomeController` передает список продуктов со скидками в переменную с именем `Products`, которую Мэри использует в представлении для генерации списка продуктов.

Рисунок 2.8 демонстрирует, как Мэри реализовала архитектуру, представленную на рис. 2.2. Этот рисунок идентичен рис. 2.2, но повторяется здесь для показа текущего состояния разрабатываемого Мэри приложения.

После того как все три слоя разработаны, приложение теоретически должно работать, но проверить это можно, только выполнив необходимые тесты.

2.1.2. Тест «На дым» (Smoke test)

Мэри реализовала все три слоя приложения, так что пришло время для проверки — работает ли оно. Она нажимает клавишу F5 и получает следующее сообщение:

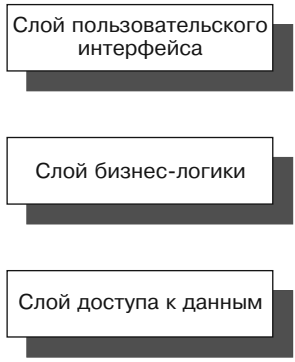


Рис. 2.8. Мэри реализовала все три слоя в своем приложении

The specified named connection is either not found in the configuration, not intended to be used with the EntityClient provider, or not valid. (Указанное именованное соединение или не найдено в текущей конфигурации, или не предназначено для работы с провайдером EntityClient, или некорректно.)

Мэри использует стандартный конструктор класса `CommerceObjectContext` (см. листинг 2.1), в котором подразумевается, что строка соединения (Connection String), называемая `CommerceObjectContext`, содержится в файле конфигурации `web.config`. Как я указывал при обсуждении листинга 2.2, данное обстоятельство имеет подвох. За ночь Мэри забыла детали реализации слоя бизнес-логики. Код компилируется, но сайт не работает.

В данном случае устранение ошибки не представляет особой проблемы. Мэри добавила необходимую строку соединения в файл `web.config`. Когда она снова запускает приложение, отображается веб-страница, показанная на рис. 2.3.

Итак, функция «Продукты со скидками» реализована, Мэри уверена в себе и готова к реализации следующей функции своего приложения. Ведь она использовала лучшие известные ей приемы и разработала трехуровневое приложение.

2.1.3. Оценка

Добилась ли Мэри успеха в разработке настоящего многоуровневого приложения? Нет, не добилась — хотя она, конечно же, действовала с наилучшими намерениями. Она создала три проекта в Visual Studio, совпадающих с тремя уровнями архитектуры ее проекта, что отражено на рис. 2.9. Случайному наблюдателю может показаться, что это и есть искомая трехуровневая архитектура, но, как мы увидим далее, полученный код является сильно связанным.

Рис. 2.9. Разработанное Мэри веб-приложение для электронной коммерции содержит по одному проекту Visual Studio на каждый уровень реализуемой архитектуры — но является ли оно настоящим трехуровневым приложением?

Пакет Visual Studio предназначен для облегчения и упрощения разработки решений и проектов следующим образом. Если нам требуется функциональность из другой библиотеки, мы можем легко добавить ссылку на нее и написать код, создающий новые экземпляры типов, определяемых в этих дополнительных библиотеках. Каждый раз, когда мы добавляем ссылку, мы создаем *зависимость*.

Схема зависимостей

При работе с решениями в Visual Studio легко потерять контроль над важными зависимостями, поскольку Visual Studio отображает их вместе со всеми другими ссылками, которые могут указывать на сборки из базовой библиотеки классов .NET.

Чтобы понять, как связаны модули в разрабатываемом Мэри приложении, мы можем нарисовать схему зависимостей (рис. 2.10).

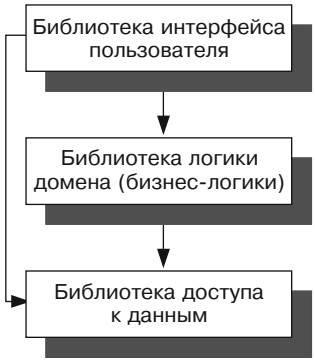


Рис. 2.10. Схема зависимостей для приложения Мэри показывает, как модули зависят один от другого. Стрелки на схеме направлены в сторону зависимостей

Самое ценное, что дает нам рис. 2.10, — понимание того, что библиотека слоя интерфейса пользователя зависит как от библиотеки слоя бизнес-логики, так и от библиотеки слоя доступа к данным. И похоже, что в некоторых случаях слой пользовательского интерфейса может обходиться без слоя бизнес-логики, так что требуется дальнейший анализ.

Оценка сочетаемости

Основная цель построения трехуровневых приложений — это разделение ответственности. Мы стремимся отделить бизнес-логику от доступа к данным и интерфейса пользователя, чтобы возникающие в них проблемы не влияли на бизнес-логику. В крупных системах особенно важно, чтобы коды различных программных компонент были изолированы друг от друга.

Для оценки реализованного Мэри кода мы можем задать простой вопрос.

ТЕСТ

Возможно ли использовать каждый модуль независимо (изолированно)?

Теоретически мы должны иметь возможность комбинировать модули в любых нужных нам вариантах. Нам может потребоваться написать новые модули, связав их с существующими модулями новыми и на настоящий момент неизвестными способами, и в идеале мы должны иметь возможность сделать это, не модифицируя существующий код.

ПРИМЕЧАНИЕ

Ниже обсуждается вопрос — какие модули могут быть заменены, но учитывайте, что эта технология применяется только для оценки сочетаемости. Даже если замена модулей никогда не потребуется, такой способ анализа раскрывает потенциальные проблемы, касающиеся связывания. Если будет установлено, что код является сильно связанным, то все преимущества слабого связывания будут потеряны.

Можем ли мы использовать модули из разработанного Мэри приложения новыми и необычными способами? Рассмотрим несколько возможных сценариев.

Новый интерфейс пользователя

Если написанное Мэри приложение окажется успешным, руководители проекта, возможно, захотят, чтобы она разработала версию с улучшенным пользовательским интерфейсом, используя технологию Windows Presentation Foundation (WPF). Можно ли сделать это, используя разработанные слой бизнес-логики и слой доступа к данным?

Если рассмотреть схему зависимостей, представленную на рис. 2.10, можно быстро установить, что ни один модуль приложения не зависит от разработанного пользовательского веб-интерфейса, поэтому можно легко удалить его и заменить на интерфейс, разработанный по технологии WPF.

В результате разработки нового WPF-интерфейса получается новое приложение, значительная часть которого взята из исходного веб-приложения. Рисунок 2.11 показывает, что новое WPF-приложение имеет те же зависимости, что и веб-приложение. Исходное веб-приложение при этом не изменяется. Первоначальный веб-интерфейс оставлен на схеме и подсвечен светло-серым цветом для иллюстрации того факта, что добавление нового интерфейса не требует удаления существующего.

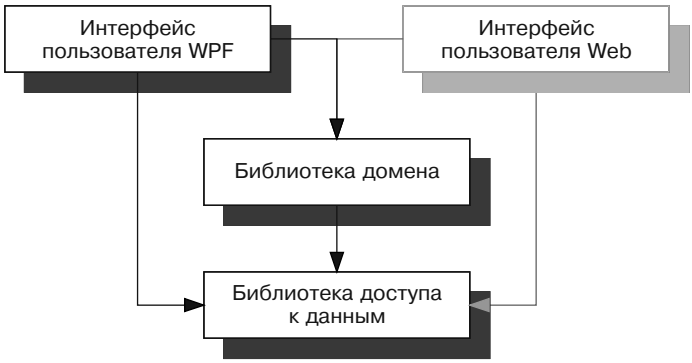


Рис. 2.11. Замена веб-интерфейса пользователя на WPF-интерфейс возможна, поскольку в приложении нет модулей, зависящих от веб-интерфейса

Замена слоя пользовательского интерфейса в реализованном Мэри приложении оказалась возможной, попробуем проверить другие возможные варианты композиции.

Новый слой доступа к данным

Представим, что анализ рынка выявил, что для оптимизации прибыли разработанное Мэри приложение должно иметь возможность работать как облачное, базирующееся на технологии Windows Azure. Windows Azure позволяет сохранять данные в хорошо масштабируемом Azure Table Storage Service (TSS, Сервис хранения таблиц Azure). Этот механизм хранения основан на гибких контейнерах для содержания неограниченных данных (Unconstrained Data). Сервис не использует отдельных схем баз данных, и в нем отсутствует ссылочная целостность.

Для взаимодействия с TSS используется протокол HTTP, а самая подходящая технология доступа к данным в .NET основана на ADO.NET Data Services.

Такой тип базы данных иногда называется *ассоциативной базой данных* (key-value database). Она отличается от реляционной базы данных, доступ к которой осуществляется через фреймворк Entity.

Чтобы обеспечить облачное функционирование приложения электронной коммерции, разработанная библиотека доступа к данным должна быть заменена модулем, использующим сервис для хранения таблиц Table Storage Service (TSS). Возможно ли это?

Из схемы зависимостей (см. рис. 2.10) понятно, что как библиотека пользовательского интерфейса, так и библиотека бизнес-логики зависят от построенной на базе фреймворка `Entity` библиотеки доступа к данным. Если удалить эту библиотеку, приложение больше не будет компилироваться, поскольку в нем отсутствуют требуемые зависимости (рис. 2.12).

Рис. 2.12. Попытка удаления библиотеки, обеспечивающей доступ к реляционным данным, разрушает приложение, так как все прочие модули зависят от нее. В существующем варианте кода не существует способа подключения к библиотеке бизнес-логики новую библиотеку доступа к данным Azure вместо первоначальной

В крупном приложении с большим количеством модулей можно также попытаться удалить модули, которые не компилируются. Это делается, чтобы определить,

что же в итоге остается в приложении. В разработанном Мэри приложении, очевидно, нам придется удалить все модули, не оставив в нем ничего.

И хотя можно разработать библиотеку для доступа к данным таблиц Azure (Azure Table Data Access), которая имитировала бы API исходной библиотеки доступа к данным, не существует способа внедрить ее в приложение.

Таким образом, разработанное приложение не является настолько гибким, насколько этого хотелось бы заказчикам проекта. Чтобы включить в приложение максимизирующие прибыль облачные возможности, требуется значительно видоизменить код, так как ни один из разработанных ранее модулей не может переиспользоваться.

Прочие комбинации

Можно было бы рассмотреть возможность сборки приложения и в других комбинациях модулей, но это дохлый номер, поскольку уже и так ясно, что приложение не реализует как минимум один важнейший сценарий.

Кроме того, не всякие комбинации модулей имеют смысл. Например, может быть задан вопрос — можно ли заменить модуль бизнес-логики на модуль с другой функциональностью. В большинстве случаев такой вопрос является как минимум странным, поскольку бизнес-модель является сердцем приложения. Без соответствующей бизнес-модели существование большинства приложений не имеет смысла.

2.1.4. Анализ

Почему приложение Мэри не обеспечивает требуемого уровня сочетаемости? Является ли это следствием того, что пользовательский интерфейс напрямую зависит от библиотеки доступа к данным? Рассмотрим этот вопрос детально.

Анализ схемы зависимостей

Почему пользовательский интерфейс зависит от библиотеки доступа к данным? Виновником является вот эта сигнатура метода бизнес-модели:

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

Показывает клиентам тип используемого доступа к данным

Метод `GetFeaturedProducts` возвращает набор продуктов, но класс `Product` определен в библиотеке доступа к данным. Любой клиент, использующий метод `GetFeaturedProducts`, должен содержать ссылку на библиотеку доступа к данным, чтобы быть скомпилированным.

Можно было бы изменить сигнатуру метода таким образом, чтобы он возвращал последовательность того типа, который был бы задан в бизнес-модели. Такой вариант был бы более корректным, но и он не решает проблему.

Предположим, нам удалось избавиться от зависимости между библиотеками пользовательского интерфейса и доступа к данным. Модифицированная схема зависимостей в этом случае будет выглядеть так, как показано на рис. 2.13.

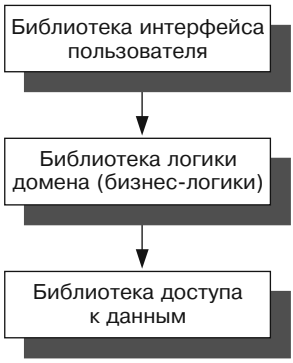


Рис. 2.13. Схема зависимостей для гипотетической ситуации, в которой отсутствует зависимость между пользовательским интерфейсом и библиотекой доступа к данным

Позволит ли такое изменение заменить библиотеку доступа к реляционным данным на библиотеку доступа к сервису Azure? К сожалению, нет, поскольку сохраняется зависимость библиотеки бизнес-логики от библиотеки доступа к данным. Пользовательский интерфейс, в свою очередь, все еще зависит от библиотеки бизнес-логики, так что если мы попытаемся удалить исходную библиотеку доступа к данным, то в приложении больше не останется модулей.

Значит, корень проблемы находится где-то в другом месте.

Анализ интерфейса доступа к данным

Бизнес-модель зависит от библиотеки доступа к данным, вся модель данных определена там. Класс `Product` был сгенерирован, когда Мэри использовала инструмент LINQ to Entities. Использование фреймворка Entity для реализации слоя доступа к данным может быть хорошим решением.

Однако же прямое использование его в бизнес-модели таковым не будет.

Вызывающий код может быть распределен в классе `ProductService`. Конструктор создает новый экземпляр класса `CommerceObjectContext` и связывает его с приватной переменной — участником класса:

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

При этом возникает сильная связь класса `ProductService` с библиотекой доступа к данным. Не существует приемлемого способа удалить эту часть кода и заменить ее каким-либо иным. Ссылка на библиотеку доступа к данным жестко запрограммирована в классе `ProductService`.

Реализация метода `GetFeaturedProducts` использует `CommerceObjectContext` для выбора объектов типа `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 к этому моменту вред уже нанесен. Нам нужен лучший способ соединения модулей, не имеющий такой сильной связи.

Прочие недостатки

Прежде чем приступить к описанию лучшей альтернативы, мне хотелось бы показать некоторые дополнительные недостатки в написанном Мэри коде, которые желательно устранить.

- Большая часть бизнес-логики реализована в библиотеке доступа к данным. То, что библиотека бизнес-модели ссылается на модель доступа к данным, является чисто *технической* проблемой. Однако то, что класс `Product`, определяется в библиотеке доступа к данным, является *концептуальной* проблемой, так как логически общедоступный класс `Product` относится к бизнес-модели.
- По совету Йенса Мэри решает реализовать код, определяющий, является ли пользователь привилегированным, в слое интерфейса пользователя. Однако на самом деле выяснение данного вопроса относится к уровню бизнес-логики, поэтому и реализовано оно должно быть в слое бизнес-логики.

  Аргументация Йенса относительно разделения проблем и принципа единственной ответственности не оправдывает помещения кода в неверное место. Следование принципу единственной ответственности в одной библиотеке полностью возможно — это предполагаемый подход.
- Класс `ProductService` использует XML-конфигурацию. Если вы помните, в какой-то момент Мэри забыла добавить фрагмент конфигурационного кода в файл `web.config`. Наличие возможности изменения конфигурации приложения без его перекомпиляции важно, но файлы конфигурации должны использоваться только самим приложением. Настройку конфигурации повторно применяемых библиотек предпочтительнее оставлять вызывающим их программным компонентам.

  На самом верхнем уровне все библиотеки вызываются самим приложением. Поэтому все необходимые конфигурационные параметры должны быть прочитаны приложением в начале его работы из файла конфигурации, а затем по мере необходимости переданы в вызываемые библиотеки.
- Представление (как показано в листинге 2.3) содержит слишком много функционала. Оно преобразует типы и форматирует строки. Эта функциональность должна быть передана в соответствующую модель.

В следующем разделе я продемонстрирую другой способ создания приложения с той же функциональностью, что и программа, разработанная Мэри. Одновременно я покажу, как устраняются перечисленные недостатки.

2.2. Правильная реализация

Для разрешения выявленных проблем может использоваться внедрение зависимостей. Поскольку разработка приложения с его использованием кардинально отличается от того способа, которым реализовала свое приложение Мэри, я не стану модифицировать написанный код, а перепишу приложение с нуля.

Это не означает, что принципиально невозможно выполнить рефакторинг ранее написанного кода с применением внедрения зависимостей. Это решаемая, хотя

и сложная задача. Мой опыт подсказывает, что для этого потребуется слишком масштабный рефакторинг[1].

ПРИМЕЧАНИЕ

Не беспокойтесь, если вы потеряете нить рассуждений во время знакомства с рассматриваемым примером. Внедрение зависимостей — сложная тема. Я выбрал рассматриваемый пример потому, что он соответствует реальному сценарию, но у него есть и недостаток — он более сложен, чем учебный пример, рассмотренный ранее. Позднее я опишу упомянутые здесь концепции и технологии более подробно. Во время чтения книги вы всегда можете вернуться к этому разделу и перечитать его.

Многие говорят о внедрения зависимостей как об *инверсии управления* (Inversion of Control, IoC). Эти два термина иногда используются как синонимы, но на самом деле внедрение зависимостей является только подмножеством IoC. В книге я повсеместно использую термин «внедрение зависимостей». Если где-то речь будет идти конкретно об IoC — инверсии управления, я буду оговаривать это специально.

ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ ИЛИ ИНВЕРСИЯ УПРАВЛЕНИЯ?

Термин «инверсия управления» (Inversion of Control, IoC) изначально означал некоторый способ программирования, при котором ход выполнения программы управляется либо фреймворком, выступающим в качестве технологической основы приложения, либо регулируется средой времени выполнения программы[2]. В соответствии с данным определением, большинство программного обеспечения, разработанного на платформе фреймворка .NET, использует IoC.

Когда вы пишете приложение ASP.NET, вы ограничены жизненным циклом страницы ASP.NET, но вы им не управляете — это делает ASP.NET.

Когда вы разрабатываете сервис WCF, вы реализуете интерфейсы, имеющие те или иные атрибуты. Вы пишете код сервиса, но в конечном итоге вы не управляете его исполнением — это делает WCF.

Сегодня фреймворки используются так часто, что это уже не является чем-то особенным, но все равно их применение отличается от полного управления порядком выполнения вашего кода. Это утверждение справедливо и для .NET-приложений, при работе с которыми данное явление наиболее характерно для простых приложений, запускаемых из командной строки. Как только вызывается `Main`, ваш код получает полный контроль над ходом выполнения программы. Он управляет последовательностью операций, жизненным циклом, то есть всеми аспектами. Не возникает особых событий и не вызываются переопределенные члены.

До того как внедрение зависимостей получило свое название, программисты начали называть фреймворки, управляющие зависимостями, контейнерами инверсии управления, и смысл термина IoC начал изменяться в сторону следующего определения: «инверсия управления через зависимости». Будучи последовательным классификатором, Мартин Фаулер ввел термин «внедрение зависимостей», чтобы точно определить инверсию

[1] На эту тему написана целая книга. См. *Michael Feathers* Working Effectively with Legacy Code. — New York: Prentice Hall, 2004.

[2] *Martin Fowler* Inversion Of Control, 2005: http://martinfowler.com/bliki/InversionOfControl.html.

управления в контексте управления зависимостями[1]. С той поры термин «внедрение зависимостей» (Dependency Injection) получил всеобщее одобрение как наиболее точный из всех возможных.

Таким образом, инверсия управления, IoC — это намного более широкий термин, включающий внедрение зависимостей, но не ограничивающийся ею.

В контексте управления зависимостями инверсия управления точно описывает, что мы пытаемся выполнить. В разработанном Мэри приложении код напрямую управляет своими зависимостями. Когда коду `ProductService` требуется новый экземпляр класса `CommerceObjectContext`, он просто создает этот экземпляр, используя оператор `new`. Аналогично, когда коду `HomeController` требуется новый экземпляр класса `ProductService`, он создает этот новый экземпляр. Таким образом, приложение является полностью управляемым. Возможно, это и звучит впечатляюще, но в действительности это ограничение. Я называю такой код антипаттерном «Диктатор» (Control Freak). Инверсия управления предписывает нам избавиться от такого управления и позволить какому-то другому компоненту программы управлять зависимостями.

2.2.1. Новая разработка приложения

Когда я пишу программы, я предпочитаю начинать с наиболее важного места. Очень часто таковым является интерфейс пользователя. К элементам интерфейса я добавляю все новую и новую функциональность, пока требуемые возможности модуля не будут реализованы, после чего я перехожу к разработке следующих функций. Такой метод — «снаружи-внутрь» — помогает мне сфокусироваться на реализуемых функциях без переработки приложения.

ПРИМЕЧАНИЕ

Метод «снаружи-внутрь» тесно связан с принципом YAGNI (You Aren't Gonna Need It, «Вам это не понадобится»). Этот принцип гласит, что реализованы должны быть только необходимые функции и их реализация должна быть максимально простой.

Поскольку я практикую методологию разработки через тестирование, я создаю модульные тесты сразу же, как только мне требуется создать новый класс. В процессе разработки этого примера я написал большое количество модульных тестов, но, поскольку TDD не требуется для реализации и использования внедрения зависимостей, я не включил эти тесты в текст книги. Если вам интересна эта тема, вы можете найти эти тесты в исходном коде, прилагающемся к изданию.

Интерфейс пользователя

В постановке задачи для списка предлагаемых продуктов говорится, что должно быть разработано приложение, получающее предлагаемые продукты из базы данных и показывающее их в списке, как представлено на рис. 2.3. Поскольку я знаю,

[1] Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 2004, http://martinfowler.com/articles/injection.html.

что руководители проекта в первую очередь заинтересованы в визуализации результатов, разработка пользовательского интерфейса выглядит хорошей начальной точкой.

Первое, что я делаю после запуска Visual Studio, — добавляю в мое решение новое приложение ASP.NET MVC. Поскольку список предлагаемых продуктов должен появляться на начальной странице, я начинаю работу с модификации файла `Index.aspx`, включая в него представленный в листинге 2.4 код разметки.

Листинг 2.4. Разметка представления Index

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

Обратите внимание, насколько более понятен код, представленный в листинге 2.4 по сравнению с листингом 2.3. Первое улучшение заключается в том, что исчезла необходимость приведения типа элемента словаря к последовательности продуктов перед выполнением итерации. Я легко добился этого, сделав страницу `Index.aspx` потомком `System.Web.Mvc.ViewPage<FeaturedProductsViewModel>` вместо `System.Web.Mvc.ViewPage`. Это означает, что свойство `Model` этой страницы имеет тип `FeaturedProductsViewModel`.

Полная строка описания продукта берется непосредственно из свойства `SummaryText` продукта.

Оба эти улучшения связаны с введением связанных с соответствующими представлениями моделей, инкапсулирующих поведение данных представлений. Такие модели являются Plain Old CLR[1] Objects (POCO, «старый добрый объект CLR»). Их структура представлена на рис. 2.14. Свойство `SummaryText` формируется из свойств `Name` и `UnitPrice` и инкапсулирует логику отображения.

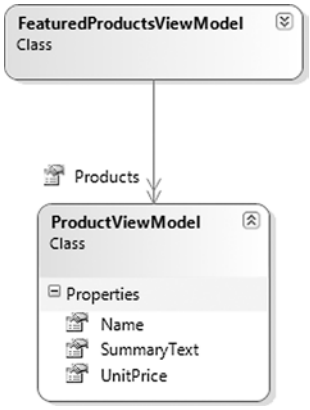


Рис. 2.14. Модель FeaturedProductsViewModel содержит список ProductViewModels. Они являются POCO, что делает их пригодными для модульного тестирования

[1] Common Language Runtime — общеязыковая среда времени выполнения.

HomeController должен вернуть *представление* с экземпляром FeaturedProductsViewModel, чтобы приведенный в листинге 2.4 код заработал. В качестве первого шага это можно реализовать в HomeController, например, в следующем коде:

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

Такое решение позволит запускать веб-приложение, не вызывая ошибки, но список предлагаемых продуктов всегда будет пустым. Формирование списка предлагаемых продуктов — это задача доменной модели.

Рисунок 2.15 показывает текущее состояние реализации архитектуры, изображенной на рис. 2.2. На данном этапе реализован только интерфейс пользователя, уровни же бизнес-логики и доступа к данным еще отсутствуют. Сопоставьте эту схему и схему, представленную на рис. 2.6, показывающем, какого прогресса добилась Мэри на этом же этапе. Одно из преимуществ, возникающее при разработке пользовательского интерфейса, заключается в том, что мы уже имеем код, который можно запускать и тестировать. Мэри же достигнет момента, когда ее приложение будет готово к исполнению, лишь на значительно более поздней стадии, показанной на рис. 2.8

Рис. 2.15. Текущее состояние реализации архитектуры

Хотя пользовательский интерфейс и существует, он сам по себе не представляет ничего интересного. Список предлагаемых продуктов всегда пуст, поэтому я должен приступить к реализации доменной логики, формирующей корректный список.

Доменная модель

Доменная модель — это библиотека C#, которую я добавил в свое решение. Эта библиотека будет содержать классы POCO и абстрактные типы. Классы POCO будут моделировать домен, тогда как абстрактные типы представляют абстракции, которые выступают в качестве основной точки входа в доменную модель.

Принцип, предписывающий программировать через интерфейсы, а не через конкретные классы, является основополагающим в внедрении зависимостей. Этот принцип позволяет заменять одну конкретную реализацию другой.

ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ?

Многие руководства по объектно-ориентированному проектированию фокусируются на интерфейсах как на основном механизме абстрагирования, в то время как в руководствах по проектированию для фреймворка .NET абстрактные классы предпочитаются интерфейсам. Что же следует выбрать — интерфейсы или абстрактные классы?

Применительно к внедрению зависимостей обнадеживающий ответ заключается в том, что это не имеет значения. Главным является то, что вы программируете, используя тот или иной вид абстракций.

Выбор между интерфейсами и абстрактными классами может оказаться важным в других случаях, но не в этом. Заметьте, что я использую данные термины как взаимозаменяемые; я также часто применяю термин «абстракция», подразумевающий в равной степени как интерфейсы, так и абстрактные классы.

Я все еще следую подходу «снаружи-внутрь», поэтому буду добавлять код в слой пользовательского интерфейса понемногу. Некоторый добавляемый мною код будет использовать типы из доменной модели. Это означает, что я добавлю ссылку на доменную модель из пользовательского интерфейса, подобно тому, как это делала Мэри. Это будет хорошо, но я отложу анализ графа зависимостей до подраздела 2.2.2, так что я смогу представить вам полную картину.

Широко используемая абстракция доступа к данным представлена в шаблоне «Репозиторий» (Repository), поэтому я определю абстрактный класс `ProductRepository` в библиотеке доменной модели:

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

Полномасштабный *репозиторий* будет содержать намного больше методов для поиска и модификации продуктов, но, следуя принципу «снаружи-внутрь», я определю сейчас только те классы и методы, которые потребуются мне для решения первоочередной задачи. Проще добавлять функциональность в код, чем удалять из него что-либо.

Класс `Product` также реализован с минимальным количеством участников, как показано на рис. 2.16. Класс `Product` содержит только свойства `Name` и `UnitPrice`, так как только они нужны для реализации требуемых функций приложения. `ApplyDiscountFor` применяет скидку (если она задана) для данного пользователя и возвращает экземпляр класса `DiscountedProduct`. Абстракция `GetFeaturedProducts` возвращает последовательность `Product`.

Метод `Index` в `HomeController` должен использовать экземпляр `ProductService` для формирования списка предлагаемых продуктов, применения скидок, преобразования экземпляров `Product` в экземпляры `ProductViewModel` и добавления их в `FeaturedProductsViewModel`. Поскольку класс `ProductService` получает в конструкторе экземпляр `ProductRepository`, важно передать туда корректный экземпляр. В процессе анализа реализованного Мэри приложения было установлено, что создание новых объектов для зависимостей[1] является очень серьезным недостатком. Как только я иду на это, я делаю используемый тип зависимости сильно связанным.

[1] Используя оператор `new`. — *Примеч. пер.*

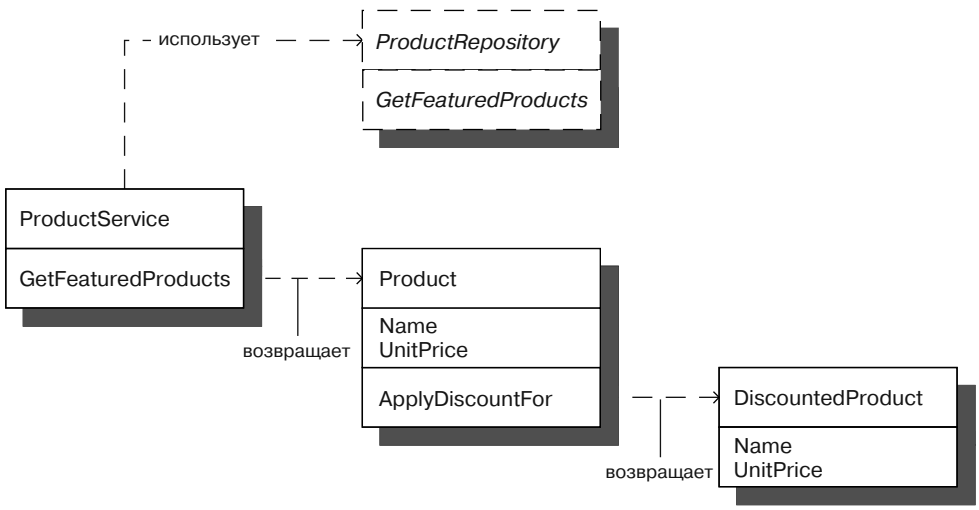


Рис. 2.16. Реализация класса Product

Я собираюсь отказаться от управления зависимостью `ProductRepository`. Как видно из листинга 2.5, я предпочту, чтобы некий механизм предоставлял бы моему коду нужный мне объект, используя конструктор класса `HomeController`. Такой шаблон называется «Внедрение конструктора» (Constructor Injection). При этом классу `HomeController` все равно, как и кем этот объект создается.

Листинг 2.5. Класс HomeController с внедрением конструктора

```
public partial class HomeController : Controller
{
    private readonly ProductRepository repository;

    public HomeController(ProductRepository repository)
    {
        if (repository == null)
        {
            throw new ArgumentNullException("repository");
        }
        this.repository = repository;
    }

    public ViewResult Index()
    {
        var productService =
            new ProductService(this.repository);

        var vm = new FeaturedProductsViewModel();

        var products =
            productService.GetFeaturedProducts(this.User);
```

1 Внедрение конструктора

Сохранение внедренной зависимости для дальнейшего использования

Передача внедренной зависимости

```
        foreach (var product in products)
        {
            var productVM = new ProductViewModel(product);
            vm.Products.Add(productVM);
        }
        return View(vm);
    }
}
```

Конструктор класса `HomeController` (1) определяет, что некий механизм, из которого кто-то хочет использовать этот класс, должен предоставить ему экземпляр класса `ProductRepository` (который, как вы должны помнить, является абстрактным классом). Контрольный оператор (Guard Clause) обеспечивает выполнение этого условия, порождая исключительную ситуацию, если требуемый экземпляр не передается (`null`). Инжектированная зависимость сохраняется для дальнейшего использования и может безопасно использоваться другими членами класса `HomeController`.

Когда я впервые услышал о внедрения конструктора, мне потребовалось некоторое время на осознание преимуществ этого подхода. Не перекладывает ли оно бремя управления зависимостью на некоторый другой класс?

Да, оно делает именно это — и это является основным достоинством. В *n*-уровневом приложении мы можем переложить данную заботу на верхний уровень приложения, в корень компоновки (Composition Root). Это централизованное место, в котором различные модули могут быть скомпонованы в единое приложение. Такая работа может быть выполнена вручную или делегирована контейнеру внедрения зависимостей.

Класс `HomeController` делегирует значительную часть своей работы классу `ProductService`, представленному в листинге 2.6. Класс `ProductService` соответствует написанному Мэри одноименному классу, но является чистым классом доменной модели.

Листинг 2.6. Класс ProductService

```
public class ProductService
{
    private readonly ProductRepository repository;

    public ProductService(ProductRepository repository)
    {
        if (repository == null)
        {
            throw new ArgumentNullException("repository");
        }
        this.repository = repository;
    }

    public IEnumerable<DiscountedProduct>
        GetFeaturedProducts(IPrincipal user)
    {
        if (user == null)
```

Снова внедрение конструктора

(1) Внедрение метода

```
        {
            throw new ArgumentNullException("user");
        }

        return from p in
                   this.repository.GetFeaturedProducts()
               select p.ApplyDiscountFor(user);
    }
}
```

Использование обеих внедренных зависимостей для реализации функционала

Метод `GetFeaturedProducts` ❶ получает экземпляр `IPrincipal`, представляющий текущего пользователя. Это другое отличие от реализации Мэри, отраженной в листинге 2.1 (напомню — там только передается булево значение, указывающее, является ли пользователь привилегированным). Однако поскольку определение — является ли пользователь привилегированным — должно выполняться логикой домена, то представляется более корректным сразу реализовывать актуального пользователя как зависимость. Мы должны всегда придерживаться принципа «программировать в соответствии с интерфейсом», но в данном случае мне не нужно ничего изобретать (как это было нужно в случае с `ProductRepository`), так как базовая библиотека классов .NET уже содержит интерфейс `IPrincipal`, обеспечивающий стандартный способ моделирования пользователей приложения.

Передача зависимости через параметр метода известна как внедрение метода. И в этом случае, как и в случае внедрения конструктора, управление зависимостью делегируется вызывающей стороне. И хотя детали в двух этих способах внедрения несколько отличаются, основные принципы для них одинаковы.

На данном этапе приложение не работает. В нем имеются две проблемы.

- Не существует конкретных реализаций `ProductRepository`. Эта проблема решается легко. В следующем разделе я реализую конкретный экземпляр `ProductRepository`, который будет выбирать предлагаемые продукты из базы данных.
- По умолчанию ASP.NET MVC предполагает, что все контроллеры используют конструкторы по умолчанию. Поскольку я ввел параметр в конструктор класса `HomeController`, фреймворк MVC «не знает», как ему создавать экземпляр `HomeController`. Эта проблема может быть решена путем создания пользовательского `IControllerFactory`. Рассмотрение процесса его создания выходит за рамки данной главы, но будет обсуждаться в главе 7. Сейчас для нас важно только то, что эта пользовательская фабрика будет создавать конкретный экземпляр `ProductRepository` и передавать его в конструктор класса `HomeController`.

В доменной модели я использую только типы, определенные в самой этой модели (и еще несколько типов из .NET BCL). Концепты доменной модели реализуются как POCO. На этой стадии представлен только один концепт, называемый `Product`. Доменная модель должна уметь взаимодействовать с окружением (в частности, с базами данных). Это требование реализуется в виде абстрактных классов (таких как `Repository`), которые нужно будет заменить на конкретные реализации, чтобы доменная модель стала полезной.

Рисунок 2.17 демонстрирует текущее состояние реализации архитектуры, представленной на рис. 2.2. Слои интерфейса пользователя и доменной логики в насто-

ящее время реализованы, тогда как слой доступа к данным еще требует реализации. Сопоставьте данную схему со схемой, показанной на рис. 2.7, которая показывает степень продвижения Мэри на аналогичной стадии работ.

Рис. 2.17. Реализация архитектуры на данном этапе

Доменная модель приложения еще не является полностью объектно-ориентированной[1]; чтобы сделать ее таковой, нужно реализовать единственную абстракцию `ProductRepository`.

Доступ к данным

Как и Мэри, я реализую свою библиотеку доступа к данным при помощи технологии LINQ to Entities, поэтому я выполню те же действия, что и в подразделе 2.1.1 для создания модели сущностей (Entity Model). Основное отличие заключается в том, что модель сущностей и `CommerceObjectContext` теперь являются только лишь деталями реализации; но, используя их, я могу реализовать `ProductRepository`, что и показано в листинге 2.7.

Листинг 2.7. Реализация класса ProductRepository на основе LINQ to Entities

```
public class SqlProductRepository : Domain.ProductRepository
{
    private readonly CommerceObjectContext context;

    public SqlProductRepository(string connString)
    {
        this.context =
            new CommerceObjectContext(connString);
    }

    public override IEnumerable<Domain.Product> GetFeaturedProducts()
    {
        var products = (from p in this.context.Products
                        where p.IsFeatured
```

[1] Мы называем это анемичной моделью (*Martin Fowler* AnemicDomainModel, 2003: http://www.martinfowler.com/bliki/AnemicDomainModel.html).

```
                        select p).AsEnumerable();
        return from p in products
               select p.ToDomainProduct();
    }
}
```

1 Конвертация в Domain Product

В приложении Мэри сгенерированная сущность `Product` использовалась как объект домена, хотя определена была в модуле работы с базой данных. Теперь это не так, поскольку я уже создал класс `Product` в доменной модели. Когда я генерировал мастер сущностной модели (Entity Model Wizard), я создал другой класс `Product`, и теперь необходимо преобразовать один класс в *другой* (1). Рисунок 2.18 иллюстрирует, как эти два класса определены в двух различных модулях. Класс `Product` для таблицы базы данных является просто частью реализации, и я могу легко сделать его внутренним, чтобы он стал более явным. Таким образом, как в библиотеке модели домена, так и в библиотеке доступа к данным определены классы, называющиеся `Product`. Класс `Product` домена — это важный класс, определяющий доменную концепцию продукта. Класс `Product`, определенный в модуле доступа к данным, есть всего-навсего продукт работы мастера фреймворка Entity. Он без проблем может быть переименован или сделан внутренним.

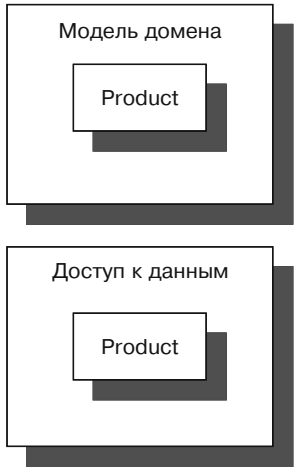


Рис. 2.18. Определение классов в двух различных модулях

ПРИМЕЧАНИЕ

Вы можете сказать, что то, что фреймворк Entity не поддерживает сущности, обеспечивающие независимость сохраняемости (Persistence Ignorance)[1] (как минимум, в версии .NET 3.5 SP1), является его специфическим недостатком. Однако это одно из многочисленных ограничений, с которыми вам придется сталкиваться в реальных проектах по разработке программного обеспечения.

[1] Хорошее введение в тему обеспечения независимости сохраняемости дано в работе *Jeremy Miller* Patterns in Practice: The Unit Of Work Pattern And Persistence Ignorance — MSDN Magazine, June 2009. Ее можно найти по адресу http://msdn.microsoft.com/en-us/magazine/dd882510.aspx.

Сущность `Product` определяет преобразование в тип `Product` домена. Это преобразование является обычным отображением значений свойств. И хотя оно не имеет непосредственного отношения к внедрения зависимостей, я включил его для большей наглядности:

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

После реализации `SqlProductRepository` я могу настроить ASP.NET MVC так, чтобы он выполнял внедрение экземпляра `SqlProductRepository` в экземпляр `HomeController`. Этот процесс будет обсуждаться в главе 7, поэтому я не описываю его сейчас.

Рисунок 2.19 показывает текущее состояние разработки приложения, архитектура которого соответствует рис. 2.2. Все три слоя в приложении реализованы в соответствии со схемой, представленной на рис. 2.2. Данный рисунок идентичен ему и повторен здесь для иллюстрации текущего состояния приложения. Рисунок аналогичен также и рис. 2.8, показывающему схему реализованного Мэри приложения

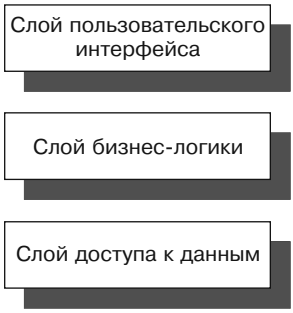


Рис. 2.19. Слои в приложении реализованы в соответствии со схемой рис. 2.2

Теперь, когда все правильно соединено друг с другом, я могу посмотреть начальную страницу приложения, и я должен получить страницу такую же, как показана на рис. 2.3.

2.2.2. Анализ слабо связанной реализации

Предыдущий подраздел содержит большое количество деталей, так что не отчаивайтесь, если вы потеряли по пути полное представление о картине в целом. В этом подразделе я попытаюсь прокомментировать полученные результаты.

Взаимодействие

Классы на каждом уровне взаимодействуют друг с другом либо напрямую, либо как абстракции. Они взаимодействуют через границы модулей, так что бывает трудно отследить процесс их взаимодействия. Рисунок 2.20 иллюстрирует, как соединяются зависимости. Обратите внимание, как экземпляр `SqlProductRepository` инжектируется в `HomeController` и затем через `HomeController` в `ProductService`, который использует его.

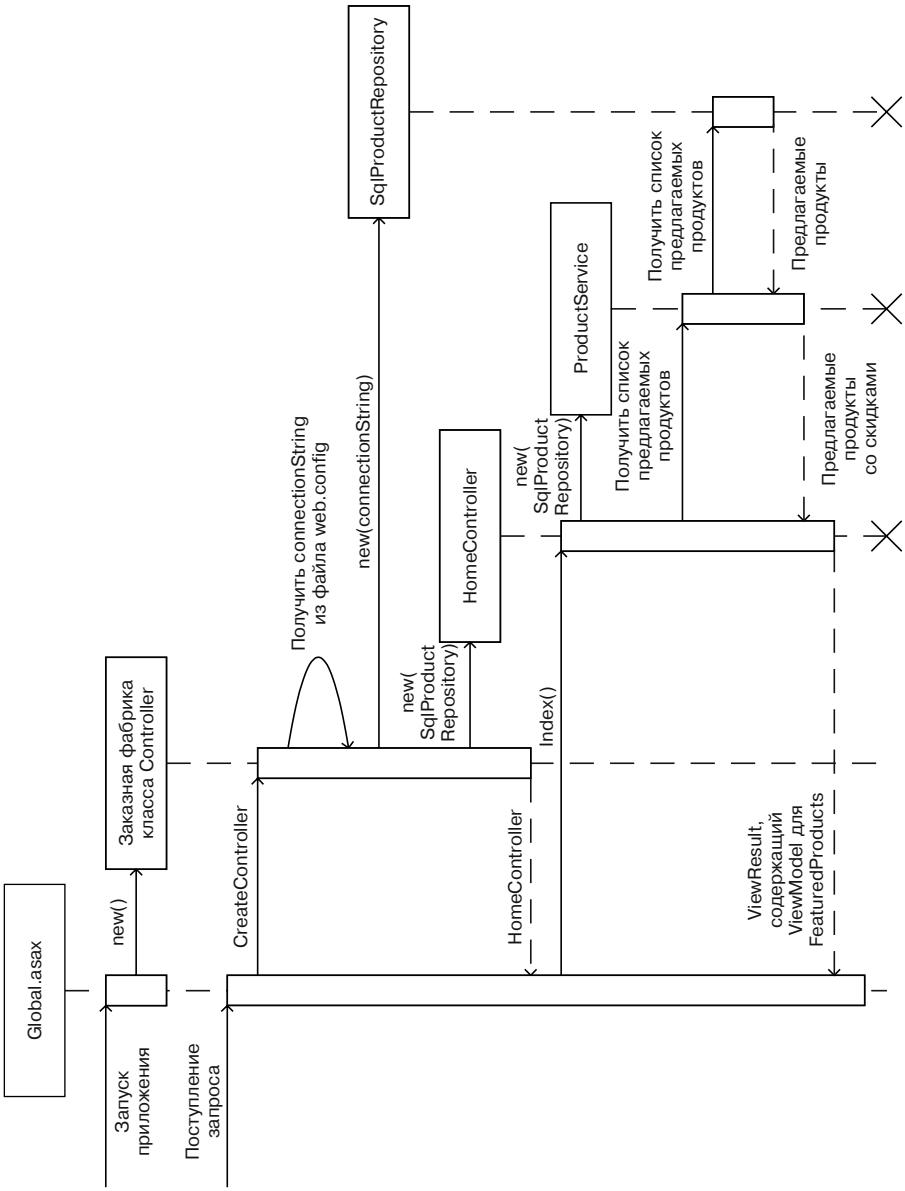


Рис. 2.20. Взаимодействие между элементами, вовлеченными во внедрение зависимостей в приложении Commerce

При запуске приложения код в Global.asax создает новую пользовательскую фабрику Controller. Приложение сохраняет ссылку на нее, так что когда поступает запрос к странице, приложение вызывает метод CreateController этой фабрики. Фабрика находит строку соединения (connection string) в файле web.config и передает ее новому экземпляру класса SqlProductRepository. Затем экземпляр класса SqlProductRepository внедряется в новый экземпляр класса HomeController и этот экземпляр (HomeController) возвращается.

Затем приложение вызывает метод Index экземпляра HomeController, который создает новый экземпляр ProductService, передавая экземпляр SqlProductRepository через конструктор. ProductService вызывает метод GetFeaturedProducts экземпляра SqlProductRepository.

В заключение возвращается ViewResult со сформированной моделью FeaturedProductsViewModel, и фреймворк ASP.NET MVC находит и отображает требуемую страницу.

Схема зависимостей

В подразделе 2.1.3 было продемонстрировано, как диаграмма зависимостей может помочь в анализе и оценке степени гибкости, обеспечиваемой реализацией приложения. Изменила ли внедрение зависимостей схему зависимостей приложения?

Рисунок 2.21 показывает, что схема зависимостей изменилась. Доменная модель больше не имеет никаких зависимостей и может функционировать как независимый модуль. С другой стороны, библиотека доступа к данным теперь содержит зависимость; в приложении Мэри этого не было. Схема зависимостей на рис. 2.21 показывает пример приложения Commerce, реализованного с помощью внедрения зависимостей. Наиболее важное отличие состоит в том, что библиотека домена больше не имеет никаких зависимостей. Светло-серые прямоугольники в темно-серых прямоугольниках показывают классы примера в каждой библиотеке, чтобы дать вам представление, какие классы берутся и откуда. Каждая библиотека в действительности содержит гораздо большее количество классов, чем представлено на диаграмме.

Это должно улучшить ситуацию с сочетаемостью компонентов приложения. Ответы на следующие вопросы будут положительными.

- Можем ли мы заменить разработанный по веб-технологии интерфейс пользователя на интерфейс, построенный на основе технологии WPF? Такая возможность существовала ранее, и она сохранилась и в новой версии приложения. Ни библиотека доменной модели, ни библиотека доступа к данным не зависят от веб-интерфейса пользователя, поэтому мы можем легко заменить его чем-либо иным.
- Можем ли мы заменить библиотеку доступа к реляционным данным на библиотеку, работающую с табличным сервисом Azure? В главе 3 я покажу, как приложение определяет и создает необходимый ProductRepository, сейчас примите как факт следующее: библиотека доступа к данным загружается динамически (позднее связывание), и название загружаемого типа задается как параметр приложения в файле конфигурации web.config. Вполне возможно отключить текущую библиотеку доступа к данным и инжектировать вместо нее другую, если только последняя является реализацией ProductRepository.

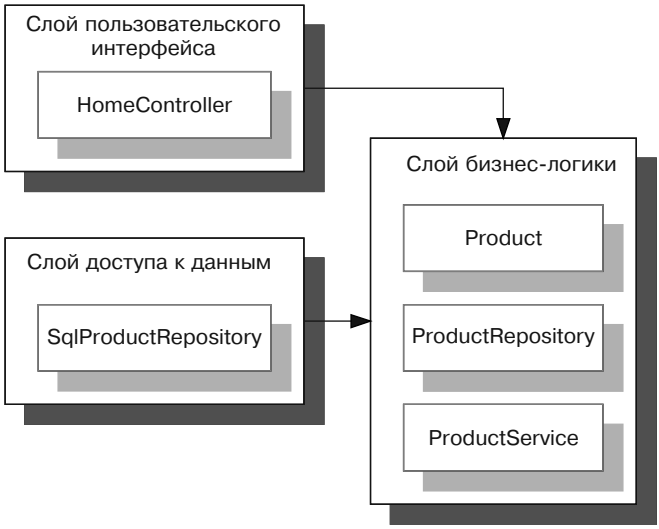


Рис. 2.21. Пример приложения Commerce, реализованного с помощью внедрения зависимостей

Больше не представляется возможным использовать текущую библиотеку доступа к данным в изоляции, так как теперь она зависит от доменной модели. Во многих видах приложений это не является проблемой, но если руководство проекта захочет иметь такую возможность, я могу решить проблему переходом на другой уровень косвенности: выделить интерфейс из класса `Product` (например, `IProduct`) и изменить класс `ProductRepository` таким образом, чтобы он работал с `IProduct`, а не с `Product`. Эти абстракции затем могут быть перемещены в отдельную библиотеку, совместно используемую библиотеками доступа к данным и библиотеками доменной модели. Это может потребовать дополнительных усилий, так как будет необходимо написать код для ассоциирования `Product` и `IProduct`, но это вполне решаемая задача.

Нужно построить решение, основанное на использовании внедрения зависимостей. В таком случае созданное изначально веб-приложение можно легко преобразовать в приложение типа «ПО + Сервисы» с использованием WPF-интерфейса и облачного хранилища данных. Единственный компонент исходного решения, применяемый и в этом случае, — это доменная модель, но это нас устраивает, так как она содержит важные бизнес-правила и, соответственно, является наиболее важным модулем.

Когда мы разрабатываем приложение, мы не можем предусмотреть все возможные направления развития продукта. Но это не составит проблемы в том случае, если мы будем поддерживать наши решения открытыми для изменений. Внедрение зависимостей помогает создавать слабо связанные приложения, так что мы можем использовать повторно или заменять отдельные модули по мере необходимости.

2.3. Расширение приложения-примера

Чтобы заложить основу для оставшейся части книги и полностью продемонстрировать различные аспекты внедрения зависимостей, мне понадобится расширить

данный образец приложения из области электронной коммерции. До сих пор я оставлял приложение максимально простым, чтобы только продемонстрировать некоторые фундаментальные концепции и принципы. Поскольку одна из главных задач внедрения зависимостей — это удерживать сложность приложения под контролем, нам потребуется более сложное приложение, чтобы в полном объеме продемонстрировать ее мощь.

Я буду расширять приложение в двух направлениях: выполню рефакторинг его архитектуры и добавлю новые возможности.

2.3.1. Архитектура

Демонстрационное приложение является трехуровневым, но сейчас я хочу поместить уровень модели представления (Presentation Model Layer) между интерфейсом пользователя и доменной моделью, как показано на рис. 2.22.

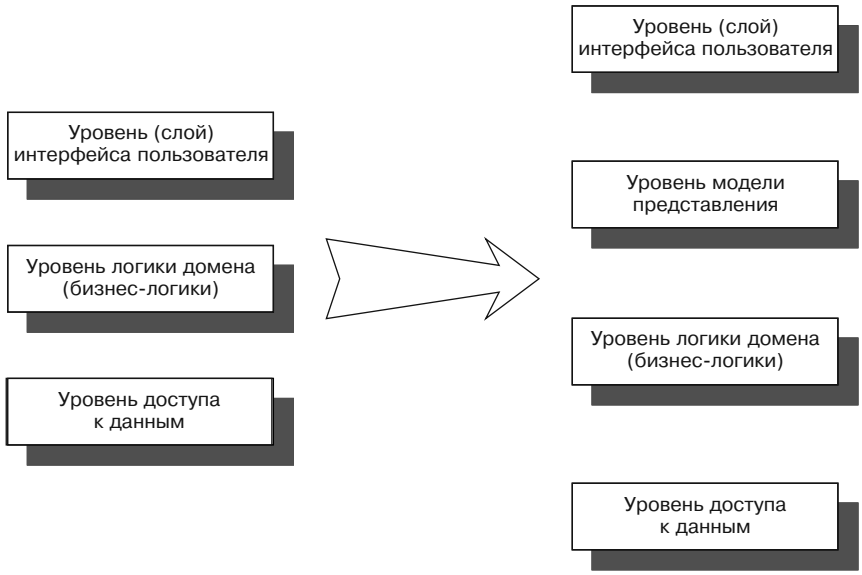


Рис. 2.22. Слой Presentation Model (модель представления) добавлен в демонстрационное приложение для отделения презентационной логики от корня компоновки

Я перемещу все контроллеры и модели представлений (`ViewModels`) из слоя пользовательского интерфейса в слой модели представления, оставив в слое пользовательского интерфейса только представления (Views, файлы с расширениями `.aspx` и `.ascx`) и корень компоновки.

Основная причина, почему я делаю это, — я хочу отделить корень компоновки от демонстрационной логики; таким образом я смогу показать различные вариации стилей конфигурирования, оставив без изменения максимально возможный объем кода приложения.

МИНИМАЛИСТИЧНЫЙ ОБЪЕКТ

Я разделил приложение, выделив в нем слой пользовательского интерфейса и слой демонстрационной модели не только в образовательных целях; я всегда делаю это для разрабатываемых мною приложений, если они имеют пользовательский интерфейс.

Такое деление обеспечивает четкое разграничение проблем между презентационной логикой (как *ведет* себя пользовательский интерфейс) и рендерингом (как *выглядит* пользовательский интерфейс). При этом вся логика помещается в слой, в котором она может быть подвергнута модульному тестированию, а вся разметка попадает в слой, в котором графический дизайнер может работать, не боясь испортить программный код.

Целью является максимально уменьшить количество кода в слое пользовательского интерфейса, оставив только самый необходимый, так как я не собираюсь писать модульные тесты для этого уровня.

Корень приложения (Application Root), который содержит самый минимум кода для начальной загрузки, после чего вся остальная работа делегируется модулям, которые могут быть протестированы, называется минималистичным объектом (Humble Object). В нашем случае он содержит только `представления` и код начальной загрузки и называется **корнем компоновки**.

Помимо данного архитектурного изменения, я хочу еще добавить более богатые функциональные возможности, чем те, что мы видели до сих пор.

2.3.2. Корзина покупок

Список предлагаемых продуктов только показывает их с ограниченным уровнем сложности: обслуживается единственный репозиторий, включенный в сценарий «только для чтения».

На следующем этапе целесообразно добавить в приложение корзину для покупок. Рисунок 2.23 представляет собой снимок экрана, на котором показана работающая корзина.

Рис. 2.23. Корзина покупок в модернизированном демонстрационном приложении электронной коммерции

Для предоставления корзины каждому пользователю мне нужны классы `Basket`, `BasketRepository` и вспомогательные классы. Если мы с вами мыслим схожими категориями, то вы прежде всего захотите увидеть класс `Basket`: На рис. 2.24 продемонстрирована корзина и список находящихся в ней продуктов.

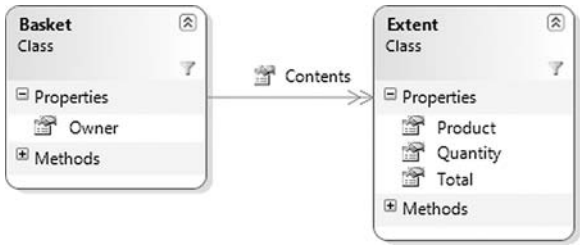


Рис. 2.24. Корзина (Basket) и ее содержимое (Contents), которое является списком типа Extent<Evaluated-Product>. Класс Extent дает количественную информацию по каждому продукту

С точки зрения внедрения зависимостей классы `Basket` и `Extent` не представляют интереса: оба они являются POCO-классами, не содержащими зависимостей. Намного более интересен класс `BasketService` и его вспомогательные классы, показанные на рис. 2.25. Класс `BasketService` может получать и обрабатывать корзину для заданного пользователя. Он использует класс `BasketRepository` для получения корзины и класс `BasketDiscountPolicy` для получения скидок (если таковые существуют для данного пользователя).

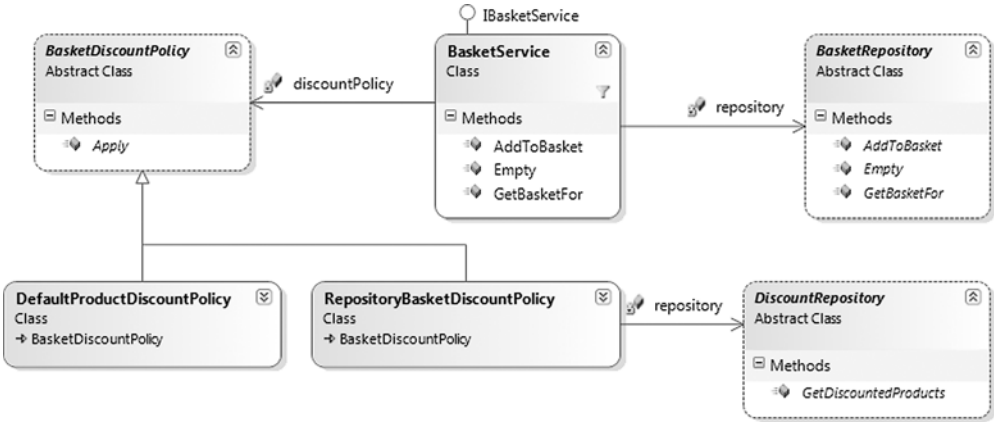


Рис. 2.25. Класс BasketService и его вспомогательные классы

Класс `BasketService` может использоваться для получения корзины для каждого пользователя и применения скидок, если они действуют для этого пользователя. Этот класс использует абстракцию `BasketRepository` для получения содержимого корзины и абстракцию `BasketDiscountPolicy` для получения скидок. Обе эти абстракции внедряются в `BasketService` посредством внедрения конструктора:

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

Класс `BasketDiscountPolicy` может быть простой реализацией с жестко запрограммированной политикой. Например, согласно политике, привилегированным

пользователям будет предоставляться пятипроцентная скидка — мы уже говорили об этом выше в даной главе. Эта политика реализована в классе `DefaultProductDiscountPolicy`, тогда как более сложная, управляемая данными реализация политики находится в классе `RepositoryBasketDiscountPolicy`. В свою очередь, этот класс использует абстракцию `DiscountRepository` для получения списка продуктов со скидками. Эта абстракция инжектируется в класс `RepositoryBasketDiscountPolicy` через внедрение конструктора:

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

Я могу использовать класс `BasketService` для управления операциями в корзине: добавления элементов, отображения содержимого, очистки корзины. Чтобы реализовать эти функции, `BasketService` должен использовать классы `BasketRepository` и `BasketDiscountPolicy`, которые передаются в него через конструктор:

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

Чтобы еще более усложнить ситуацию, мне потребуется контроллер ASP.NET MVC, называемый `BasketController` и являющийся оберткой для интерфейса `IBasketService`, который я снова инжектирую через конструктор:

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

Рисунок 2.25 показывает, что класс `BasketService` реализует `IBasketService`, так что мы воспользуемся именно этой реализацией.

`BasketController` создан фабрикой `IControllerFactory`, поэтому ей также понадобятся эти абстракции.

Если вы не успеваете следить за развитием событий, взгляните на рис. 2.26. Он помогает восстановить картину при помощи схемы, демонстрирующей, как зависимости включаются в разрабатываемое приложение. Каждый класс изолирует свое содержимое, и только корень компоновки содержит информацию о задействованных зависимостях.

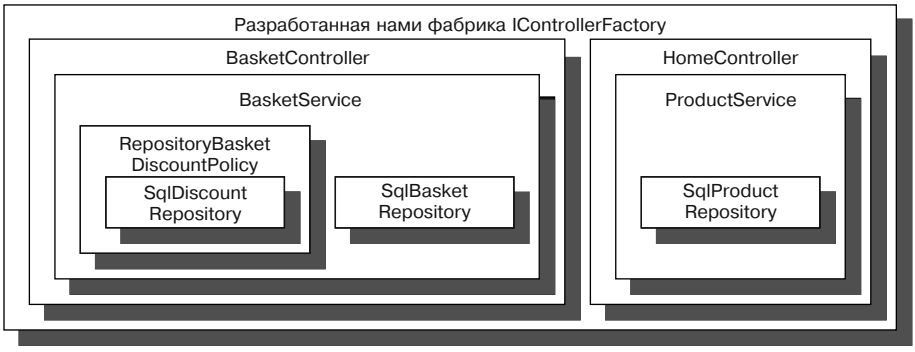


Рис. 2.26. Компоновка демонстрационного приложения для электронной коммерции с исходным списком предлагаемых продуктов на входной странице и добавленной корзиной покупок

Разработанная нами фабрика IControllerFactory создает экземпляры классов BasketController и HomeController, передавая им необходимые зависимости. Класс BasketService, например, использует экземпляр BasketDiscountPolicy для применения политики скидок для корзины:

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

Нет даже намека на то, что в данном случае передаваемый BasketDiscountPolicy является экземпляром RepositoryBasketDiscountPolicy, который в свою очередь является контейнером DiscountRepository.

Полученное расширенное демонстрационное приложение является основой для следующих примеров кода из нашей книги.

2.4. Резюме

Оказывается, что на удивление легко написать сильно связанный код. Хотя Мэри собиралась создать трехуровневое приложение, оно на деле превратилось в огромный монолитный кусок так называемого спагетти-кода (когда мы говорим о разбиении на слои, мы называем это «лазанья»).

Одна из многочисленных причин, почему так легко написать сильно связанный код, заключается в том, что как свойства языка, так и используемые инструменты уже подталкивают нас в этом направлении. Если нам требуется новый экземпляр объекта, мы можем использовать ключевое слово new, а если у нас нет ссылки на необходимую сборку, Visual Studio легко создаст ее.

Однако всякий раз, когда мы используем ключевое слово new, мы создаем сильное связывание.

Наилучший способ минимизировать применение оператора new — использовать вместо него паттерн проектирования «Внедрение конструктора» (Constructor Injection) всякий раз, когда нам понадобится экземпляр зависимости. Второй пример в этой главе показывает, как можно заново реализовать написанное Мэри приложение, работая с абстракциями, а не с конкретными классами.

Внедрение конструктора — это пример инверсии управления, здесь мы инвертируем управление через зависимости. Вместо того чтобы создавать экземпляры при помощи оператора new, мы делегируем ответственность третьей стороне. Как мы увидим в следующей главе, эта сторона называется корнем компоновки. Это место, где мы собираем все слабо связанные классы в единое приложение.

3 Контейнеры внедрения зависимостей

Меню:

- XML-конфигурация;
- конфигурирование в коде;
- авторегистрация;
- корень компоновки;
- «Регистрация, преобразование, высвобождение».

Когда я был ребенком, мы с мамой иногда делали мороженое. Это случалось нечасто, так как дело это долгое и сложное. Если вы никогда не делали мороженое, то можете узнать с помощью рис. 3.1, как это происходит.

Настоящее мороженое делается на основе заварного крема. Это легкий крем, изготавливаемый из сахара, яичных желтков и молока или сливок. Эта смесь свернется, если нагреть ее слишком сильно. Даже если здесь все пойдет хорошо, следующая стадия таит другие проблемы. Если оставить кремовую смесь в морозильнике, она замерзает и кристаллизуется, поэтому вам придется перемешивать ее через определенные промежутки времени, пока она не достигнет такой степени готовности, в которой кристаллизация уже станет невозможной. Только тогда вы получите качественное, приготовленное в домашних условиях мороженое.

Это медленный и трудоемкий процесс, но если у вас достаточно желания и есть все необходимые ингредиенты и оборудование, вы можете приготовить мороженое, руководствуясь полученной от меня инструкцией.

Сегодня, 30 лет спустя, моя теща готовит мороженое намного чаще, чем это делали мы с моей мамой много лет назад. И не потому, что она любит это занятие, а потому, что использует *технологию* для облегчения процесса. *Способ* приготовления мороженого остался тем же, но вместо того чтобы регулярно доставать мороженое из холодильника и взбивать его, она использует электрическую мороженицу, делающую эту работу за нее (рис. 3.2).

Внедрение зависимостей, прежде всего и в основном, является *методом*, но вы можете использовать *технологию* для упрощения работы с ней. В части 3 я описываю внедрение зависимостей как метод. Затем в части 4 мы рассмотрим технологии, которые могут применяться для реализации этого метода. Итак, речь пойдет о **контейнерах внедрения зависимостей**.

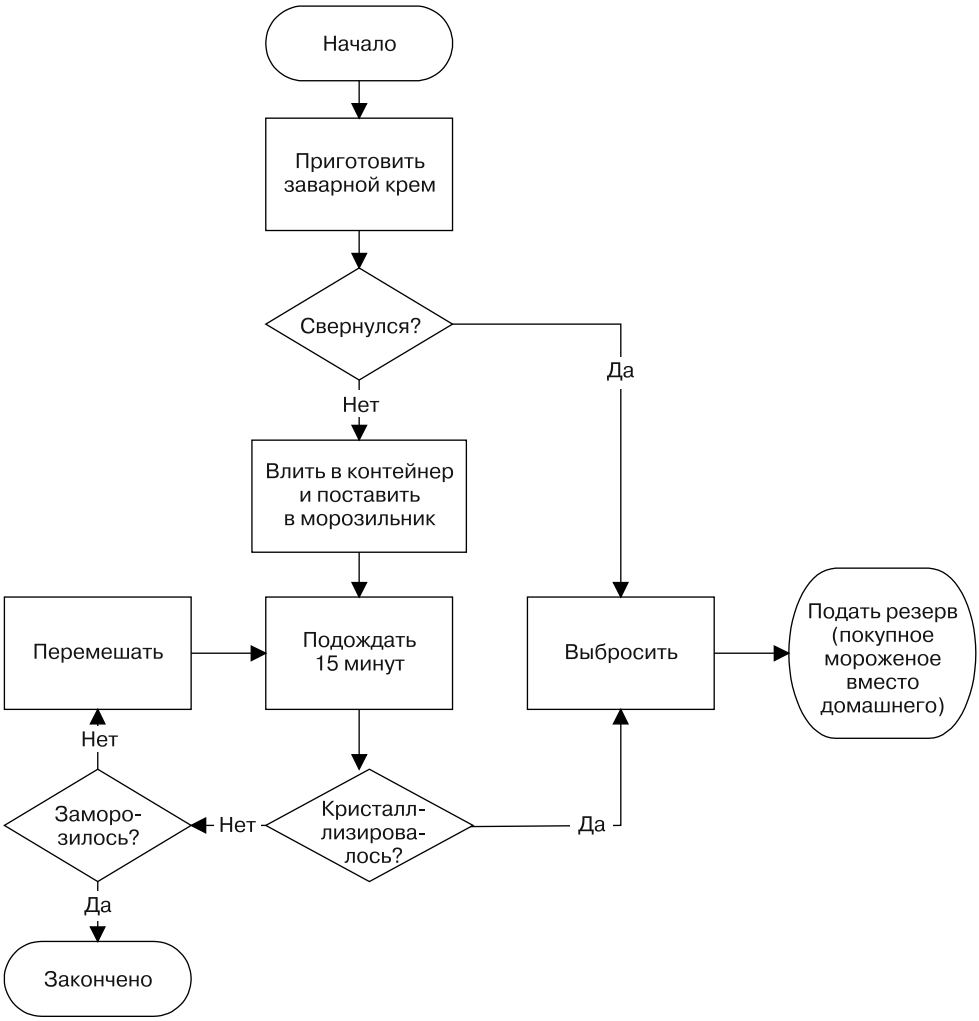


Рис. 3.1. Приготовление мороженого — это трудный процесс, любой повар не застрахован от ошибок

В этой главе мы изучим контейнеры внедрения зависимостей в концептуальном плане: выясним, насколько они соответствуют общей идеологии внедрения зависимостей, познакомимся с некоторыми шаблонами и приемами их использования, а также совершим небольшой экскурс в историю контейнеров платформы .NET. В процессе мы рассмотрим несколько примеров.

Общая структура главы показана на рис. 3.3. Она начинается с общего введения в контейнеры внедрения зависимостей, включая описание концепции, называемой автоподключением (Auto-Wiring). За введением следует раздел, описывающий различные конфигурационные варианты. Вы можете читать материал о каждом варианте конфигурирования независимо от других, но мне кажется, что будет полезнее

как минимум прочитать о конфигурировании в коде (Code As Configuration) до того, как вы начнете читать об авторегистрации (Auto-Registration).

Рис. 3.2. Итальянская мороженица моей тещи

Основная часть этой главы представляет собой мини-каталог шаблонов проектирования, которые можно использовать вместе с контейнерами внедрения зависимости. Хотя эта часть имеет формат каталога, описание шаблона «Регистрация, преобразование, высвобождение» (Register Resolve Release, RRR) относится к шаблону «Корень компоновки» (Composition Root), поэтому имеет смысл читать их последовательно. Вы можете пропустить часть, касающуюся конфигурирования, и перейти непосредственно к шаблонам, но лучше прочитать обе эти части.

Последний раздел отличается от других. Он содержит намного меньше технической информации и фокусируется на том, как контейнеры внедрения вписываются в экосистему .NET. Вы можете не читать этот раздел, если вам неинтересна эта тема.

Цель данной главы — дать вам достаточно полное представление о том, что такое контейнеры внедрения зависимостей и как они взаимодействуют с прочими шаблонами и принципами, описанными в книге. Эту главу можно считать введением в четвертую часть книги. Здесь мы обсудим лишь общие принципы использования контейнеров, а в части 4 мы поговорим о конкретных контейнерах и об их API.

Может показаться странным, что мы рассуждаем о контейнерах внедрения здесь, в главе 3, а затем почти не вспоминаем о них в следующих шести главах, но на то имеются свои причины. В данной части книги я хочу сделать лишь общий обзор внедрения зависимостей, и для вас будет важно понять, как контейнеры внедрения зависимостей

вписываются в общую схему. В разделах 3.2 и 3.3 я представлю несколько примеров, включающих контейнеры внедрения зависимостей, но в основном буду стараться дать лишь общее представление о предмете. Принципы и паттерны, описанные далее в книге, могут применяться ко всем контейнерам внедрения зависимостей.

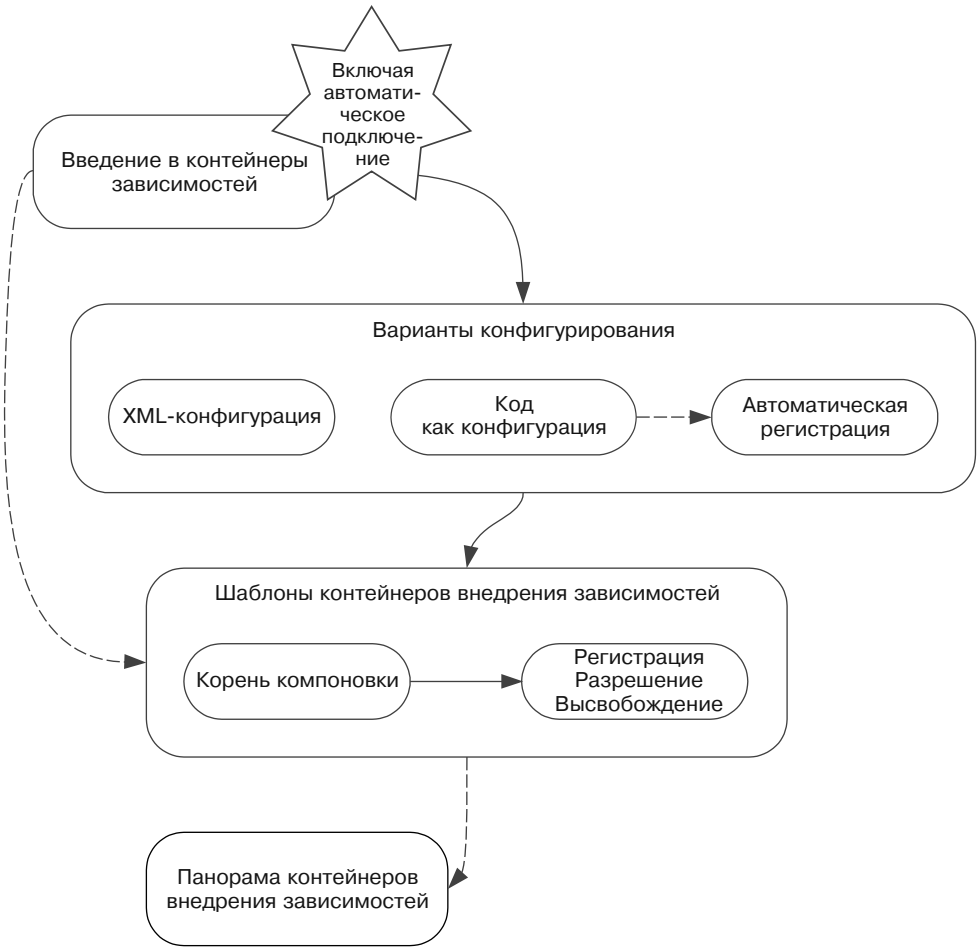


Рис. 3.3. Структура данной главы. Раздел «Панорама контейнеров внедрения» является необязательным

3.1. Введение в контейнеры внедрения зависимостей

Контейнер внедрения зависимостей (далее для краткости «контейнер внедрения» или просто «контейнер». — *Примеч. пер.*) — это программная библиотека, которая может автоматизировать многие задачи, выполняемые при компоновке объектов

и управлении их жизненным циклом. Хотя можно написать весь требуемый инфраструктурный код, используя подход «внедрение зависимостей для бедных» (когда приложение создается без использования контейнера, см. главу 1. — *Примеч. пер.*), это не увеличивает функциональную ценность приложения. С другой стороны, задача компоновки объектов достаточно универсальна и может быть решена всего один раз, сразу для всего приложения. В таком случае принято говорить о «неспециализированной подобласти» (Generic Subdomain[1]).

ОПРЕДЕЛЕНИЕ

Контейнер внедрения зависимостей — это библиотека, предоставляющая функционал внедрения зависимостей.

ПРИМЕЧАНИЕ

Контейнеры внедрения называются еще контейнерами инверсии управления (Inversion of Control (IoC) Containers) или (реже) упрощенными контейнерами (Lightweight Containers).

Хотя вы обязательно должны работать в соответствии с инфраструктурой приложения, этот этап работы никак не связан с выгодой пользователя. Поэтому бывает целесообразно использовать именно библиотеки общего назначения, а не контейнеры. Это похоже на реализацию доступа к данным. Журналирование данных приложения — это как раз такая проблема, которую лучше всего решать при помощи универсальной библиотеки для журналирования. Это справедливо и для компоновки графов объектов.

ВНИМАНИЕ

Не надейтесь, что контейнер внедрения волшебным образом превратит сильно связанный код в слабо связанный. Контейнер может повысить эффективность использования внедрения зависимостей, но упор в приложении должен быть сделан в первую очередь на использование паттернов и работу с внедрением зависимостей.

В данном разделе речь пойдет о том, как контейнеры внедрения выполняют компоновку графов объектов. Я покажу несколько примеров, чтобы сформировать у вас представление о том, в чем заключается использование контейнеров.

3.1.1. Привет, контейнер

Контейнер внедрения зависимостей — это программная библиотека, подобная любой другой программной библиотеке. Она представляет API, который вы можете использовать для компоновки объектов. Компоновка графа объектов заключается в вызове одного метода. Все контейнеры внедрения требуют конфигурирования до того, как они будут использоваться для компоновки, но я отложу рассмотрение этого вопроса до раздела 3.2.

Сейчас я продемонстрирую несколько примеров того, как контейнеры зависимостей могут преобразовывать графы объектов, на основе расширенного демонст-

[1] Eric Evans Domain-Driven Design: Tackling Complexity in the Heart of Software. — New York: Addison-Wesley, 2004. — 406.

рационного приложения, представленного в разделе 2.3. Для каждого запроса фреймворк ASP.NET MVC будет запрашивать экземпляр необходимого типа — потомка `IController`, поэтому вам нужно будет реализовать метод, использующий контейнер внедрения для компоновки соответствующего графа объекта.

СОВЕТ

Раздел 7.2 содержит подробную информацию о том, как компонуются приложения ASP.NET MVC.

Фреймворк MVC вызывает метод с параметром — экземпляром типа `Type`, идентифицирующим требуемый для данного случая подтип `IController` (например, `HomeController` или `BasketController`), и в результате должен быть возвращен экземпляр этого типа.

Такая функциональность реализуется всеми контейнерами внедрения, представленными в части 4, здесь же я представлю лишь несколько примеров.

Преобразование контроллеров при помощи различных контейнеров внедрения

Unity — это контейнер внедрения, API которого хорошо подходит для работы с шаблонами. Если у вас уже имеется экземпляр класса `UnityContainer`, входящего в состав Unity, то вы можете получить экземпляр `IController` из аргумента с именем `controllerType`, относящегося к типу `Type` (см. выше. — *Примеч. пер.*):

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

Вы передаете параметр `controllerType` в метод `Resolve` и получаете экземпляр требуемого типа, содержащий все нужные зависимости.

Поскольку слабо типизированный метод `Resolve` возвращает экземпляр типа `System.Object`, он должен быть явно преобразован в тип `IController`.

Если требуемый тип известен уже по время проектирования, имеется обобщенная (generic) версия метода `Resolve`.

Многие контейнеры внедрения имеют API, напоминающие API контейнера Unity. Функционально подобный код для контейнера `Castle Windsor` выглядит похожим на код Unity, хотя экземпляр контейнера в этом случае должен иметь тип `WindsorContainer`. Прочие контейнеры имеют немного различающиеся названия, например для `StructureMap` предыдущий код будет иметь следующий вид:

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

Единственное существенное различие заключается в том, что метод `Resolve` здесь называется `GetInstance`. Из приведенных примеров вы можете сформировать обобщенное представление о контейнерах внедрения.

Преобразование графов объектов при помощи контейнеров внедрения

Контейнер внедрения — это механизм, преобразующий графы объектов и управляющий ими. Хотя функции контейнеров внедрения зачастую не сводятся к преобразованию объектов, оно остается главным компонентом API любого

контейнера. Из представленных ранее примеров видно, что контейнеры имеют слабо типизированный метод, предназначенный для преобразования. Хотя для разных контейнеров существуют вариации в именах и сигнатурах, в общем виде он выглядит так:

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

Как видно из предыдущих примеров, поскольку возвращаемое значение имеет тип `System.Object`, нам, как правило, требуется явное приведение типа возвращаемого значения к требуемому типу, прежде чем это значение можно будет использовать.

Многие контейнеры зависимостей содержат обобщенную версию для тех случаев, когда уже во время компиляции точно известно, какой тип требуется. Зачастую эти методы выглядят подобно приведенному ниже:

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

В таком случае вместо передачи аргумента типа `Type` метод использует параметр типа (`T`), определяющий необходимый тип. Возвращает метод экземпляр типа `T`.

Большинство контейнеров порождают исключительную ситуацию, если они не могут преобразовать запрошенный тип.

ПРЕДУПРЕЖДЕНИЕ

Сигнатура метода Resolve исключительно универсальная и гибкая. Вы можете запросить экземпляр любого типа, и ваш код все равно будет скомпилирован. Фактически метод Resolve соответствует сигнатуре локатора сервисов[1], и вы должны будете позаботиться о том, чтобы не использовать ваш контейнер внедрения в качестве локатора сервисов.

Если мы рассмотрим метод `Resolve` отдельно от остального кода, он покажется нам едва ли не магическим. С точки зрения компилятора его можно вызывать для преобразования экземпляров произвольных типов. Как контейнер узнает, каким образом нужно компоновать требуемые типы, включая все зависимости?

Он не знает этого, и вы должны передать ему необходимую информацию, прежде чем он сможет работать. Осуществить это можно, используя *регистрацию* или *конфигурирование*, при этом вы соотносите абстракции с конкретными типами — я вернусь к этому вопросу в разделе 3.2. Если в контейнере не определена полная конфигурационная информация, необходимая для полного формирования запрошенного типа, то его нормальной реакцией должна быть генерация исключительной ситуации с достаточно полным ее описанием. Например, контейнер `Castle Windsor` выдает сообщения об исключительных ситуациях типа следующего:

Can't create component 'Ploeh.Samples.MenuModel.Mayonnaise' as it has dependencies to be satisfied.

Ploeh.Samples.MenuModel.Mayonnaise is waiting for the following dependencies:

[1] Mark Seemann Pattern Recognition: Abstract Factory or Service Locator?: http://blog.ploeh.dk/2010/11/01/PatternRecognitionAbstractFactoryOrServiceLocator.aspx.

Services:

- Ploeh.Samples.MenuModel.EggYolk which was not registered.

(Невозможно создать компонент 'Ploeh.Samples.MenuModel.Mayonnaise', так как он содержит зависимости, которые должны быть реализованы.

Ploeh.Samples.MenuModel.Mayonnaise ожидает следующих зависимостей:

Сервисы:

- Ploeh.Samples.MenuModel.EggYolk, который не был зарегистрирован.)

Из этого примера видно, что `Castle Windsor` не может преобразовать `Mayonnaise`, так как он не был сконфигурирован для работы с классом `EggYolk`.

Если контейнер сконфигурирован верно, он может преобразовать даже сложные графы объектов для запрошенного типа. Если что-либо отсутствует в конфигурации, контейнер может предоставлять подробную информацию об отсутствующих компонентах. В следующем подразделе мы подробнее рассмотрим, как это реализуется.

3.1.2. Автоподключение

Контейнеры внедрения прекрасно работают со статической информацией, компилируемой во все классы, которые используют внедрение конструктора. Применяя рефлексию, они могут анализировать структуру запрашиваемого класса и определять, какие зависимости требуются для него.

Некоторые контейнеры могут также работать с паттерном «Внедрение свойства» (Property Injection), но все они обязательно «понимают» внедрение конструктора и компонуют графы объектов, объединяя свою конфигурационную информацию с информацией, извлеченной из конструктора класса. Такой подход называется автоподключением.

ОПРЕДЕЛЕНИЕ

Автоподключение — это способность автоматически собирать граф объекта из соответствий между абстракциями и конкретными типами.

Рисунок 3.4 иллюстрирует обобщенный алгоритм, которому следует большинство контейнеров, выполняющих автоподключения графа объекта. Контейнер внедрения будет использовать свою конфигурационную информацию для поиска подходящего конкретного класса, соответствующего запрашиваемому типу. Затем он применяет рефлексию для проверки конструктора класса. Если задействуется конструктор по умолчанию (безаргументный. — *Примеч. пер.*), этот конструктор будет вызван и контейнер вернет созданный экземпляр.

Если же конструктор имеет аргументы, запускается рекурсивная процедура, в которой контейнер будет повторять процесс для каждого типа аргумента, пока все конструкторы не будут обработаны.

В разделе 3.2 будут подробнее рассмотрены способы конфигурирования контейнеров, сейчас же важно понять только, что основой конфигурирования является способ соотнесения различных абстракций и конкретных классов. Это кажется немного умозрительным (я уверен, что слово «абстракция» не поможет), поэтому мне кажется, что уместен будет небольшой пример.

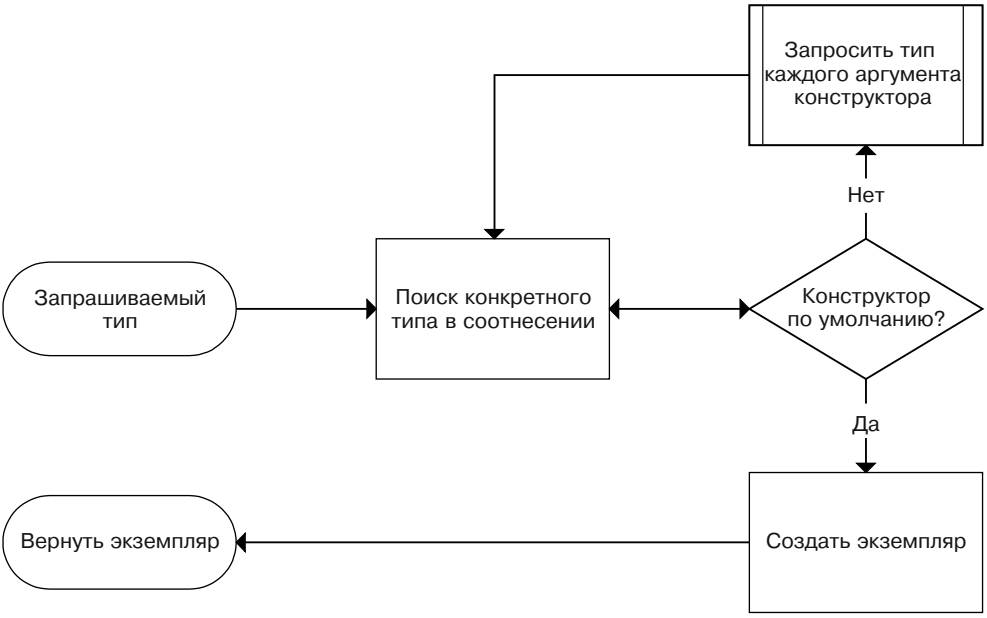


Рис. 3.4. Упрощенная схема действий при автоподключении. Контейнер внедрения зависимостей будет рекурсивно искать конкретные типы и проверять их конструкторы, пока не будет построено полное дерево объекта

Пример: автоподключение контроллера корзины

В этом примере я поясню принципы работы автоподключения. Пример не связан ни с каким конкретным контейнером, а представляет общий процесс сборки графа объектов контейнерами.

Представьте, что вы хотите преобразовать экземпляр класса `BasketController`. Вы делаете это, вызывая метод `Resolve` с типом `typeof(BasketController)`. После выполнения метода вы хотите получить экземпляр класса `BasketController`, скомпонованный так, как это показано на рис. 2.26. Чтобы добиться этого, вы прежде всего должны убедиться, что контейнер правильно сконфигурирован. Таблица 3.1 показывает, как эта конфигурация сопоставляет абстракции и конкретные типы. Дополнительно я добавил колонку, показывающую, является ли абстракция интерфейсом или абстрактным базовым классом — для контейнера это неважно, но может помочь понять суть происходящего процесса.

Таблица 3.1. Соотнесение типов для реализации автоподключения BasketController

Тип абстракции	Абстракция	Конкретный тип
Concrete	BasketController	BasketController
Interface	IBasketService	BasketService
Abstract class	BasketRepository	SqlBasketRepository
Abstract class	BasketDiscountPolicy	RepositoryBasketDiscountPolicy
Abstract class	DiscountRepository	SqlDiscountRepository
String	connString	"metadata=res://*/CommerceModel.csdl| [...]"

Когда контейнер внедрения получает запрос на тип BasketController, первым делом он попытается найти этот тип в своей конфигурации. BasketController — это конкретный класс, поэтому он соотносится сам с собой. Затем контейнер использует рефлексию для проверки конструктора класса BasketController. Из подраздела 2.3.2 вы должны помнить, что BasketController имеет единственный конструктор с такой сигнатурой:

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

Поскольку этот конструктор не задан по умолчанию, процесс будет повторен для аргумента конструктора — IBasketService, как показано на обобщенной схеме на рис. 3.4.

Контейнер находит IBasketService в своей конфигурации и определяет, что он соответствует конкретному классу BasketService. Единственный общедоступный (public) конструктор для BasketService имеет такую сигнатуру:

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

Это тоже не тот конструктор, который задан по умолчанию, и он имеет два аргумента, которые также должны быть обработаны. Контейнер обрабатывает их последовательно, начиная с абстрактного класса BasketRepository, который, в соответствии с конфигурационной информацией, соотносится с SqlBasketRepository.

SqlBasketRepository имеет public-конструктор с сигнатурой:

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

Единственный аргумент конструктора — это строковый параметр с именем connString, который конфигурируется в конкретное значение. Теперь, когда контейнер располагает необходимым значением, он может вызвать конструктор класса SqlBasketRepository. Это дает необходимый параметр repository для конструктора BasketService, но значение этого параметра должно быть сохранено, поскольку для выполнения конструктора требуется еще вычислить параметр discountPolicy.

В соответствии с заданной конфигурацией, BasketDiscountPolicy соотносится с конкретным классом RepositoryBasketDiscountPolicy, у которого имеется public конструктор следующего вида:

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

Произведя поиск DiscountRepository в конфигурации, контейнер обнаруживает, что он соотносится с классом SqlDiscountRepository, имеющим такой конструктор:

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

Повторяется ситуация, которая была для класса SqlBasketRepository. Аргумент connString соотносится с конкретной строкой соединения (Connection String), которую контейнер передает в конструктор.

Теперь новый экземпляр SqlDiscountRepository может быть передан в конструктор RepositoryBasketDiscountPolicy. Имея сформированный ранее параметр SqlBasketRepository, теперь можно сформировать конструктор BasketService

и вызвать его с помощью механизма рефлексии. В заключение созданный экземпляр `BasketService` передается в конструктор `BasketController`, после чего возвращается экземпляр `BasketController`.

Вот, собственно, в общих чертах и все, что необходимо сказать о работе автоподключения, хотя в действительности все обстоит несколько сложнее. Контейнеры внедрения должны брать на себя еще и функции **управления жизненным циклом**. Кроме того, вполне допустимо внедрение свойств, а также другие, более специфические требования. Характерно, что внедрение конструктора статически декларирует потребности класса в зависимостях и контейнеры зависимостей используют эту информацию, чтобы выполнить автоподключение сложных графов объектов.

Как показывает пример, контейнер должен быть сконфигурирован до того, как он начнет компоновку графов объектов. Регистрация компонентов может выполняться разными способами.

3.2. Конфигурирование контейнеров внедрения

Хотя действия совершаются именно в методе `Resolve`, вы должны быть готовы потратить больше времени на работу с конфигурационным API контейнера внедрения. Преобразование графа объекта, в конце концов, это единственный вызов метода.

Контейнеры внедрения обычно поддерживают два или три общеупотребительных способа конфигурирования, показанных на рис. 3.5. Лишь немногие из них не поддерживают авторегистрацию, и только один не поддерживает конфигурирование в коде, тогда как XML-конфигурирование поддерживается всеми контейнерами. В большинстве случаев разрешается смешивать разные методы в одном приложении.

Перечисленные три метода конфигурирования имеют различные характеристики, оправдывающие их использование в разных ситуациях. Как XML, так и конфигурирование в коде обычно реализуются явно, так как требуют отдельной регистрации каждого компонента. Авторегистрация, с другой стороны, является намного более неявной, так как она применяет соглашения о регистрации набора компонентов через одно правило.

Когда мы используем конфигурирование в коде, мы компилируем конфигурацию контейнера в сборку (Assembly), тогда как XML-конфигурирование позволяет нам использовать позднее (динамическое) связывание, при котором можно изменять конфигурацию без перекомпиляции приложения. В этом отношении авторегистрация занимает промежуточное положение, так как мы можем затребовать сканирование отдельной сборки, известной во время компиляции, или же сканировать все сборки в определенной папке.

В табл. 3.2 перечислены достоинства и недостатки каждого способа конфигурирования.

Первые контейнеры внедрения поддерживали только XML-конфигурирование, и это объясняет, почему все контейнеры поддерживают этот способ. Однако современная тенденция заключается в снижении популярности этого способа кон-

фигурирования по отношению к подходам, в большей степени ориентированным на соглашения (Conventions)[1]. И все же хотя авторегистрация является наиболее современным способом конфигурирования, изучение конфигурации мы начнем не с нее. Поскольку при авторегистрации конфигурация описывается неявно, этот способ может показаться более абстрактным, чем явные способы. Поэтому вместо того, чтобы описывать способы конфигурирования в обратной хронологической последовательности, начнем с XML-конфигурирования.

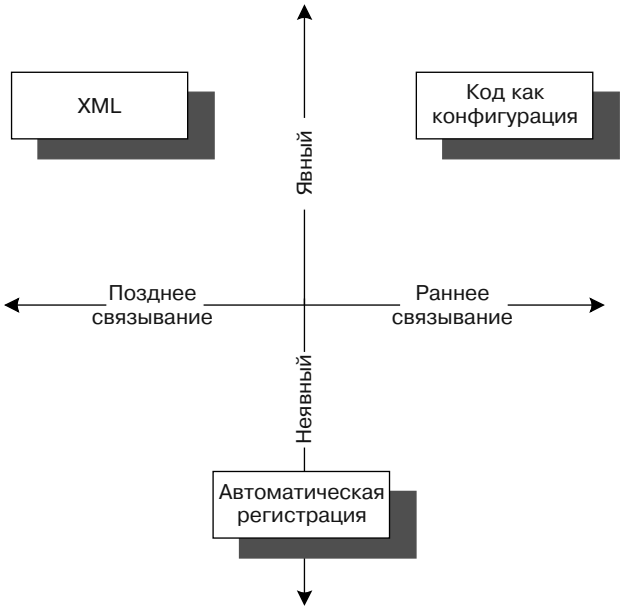


Рис. 3.5. Самые распространенные способы конфигурирования контейнера внедрения с точки зрения явности-неявности и степени связанности

Таблица 3.2. Способы конфигурирования

Способ	Описание	Преимущества	Недостатки
XML	Параметры конфигурации (часто находящиеся в файлах .config) определяют соотнесение классов	Поддерживает замену без перекомпиляции. Высокая степень контроля	Нет контроля во время компиляции. Код сравнительно пространный
Конфигурирование в коде	Код явно определяет соотнесение	Контроль во время компиляции. Высокая степень контроля	Отсутствует возможность изменения конфигурации без перекомпиляции

Продолжение ⇗

[1] Хорошая вводная статья на тему Convention over Configuration: *Jeremy Miller* Patterns in Practice: Convention Over Configuration. — MSDN Magazine, February 2009. Доступна в Интернете на сайте http://msdn.microsoft.com/en-us/magazine/dd419655.aspx.

Таблица 3.2 *(продолжение)*

Способ	Описание	Преимущества	Недостатки
Авторегистрация	Для поиска подходящих компонент и построения соотнесений используются правила	Поддерживает изменение конфигурации без перекомпиляции. Требуется меньшее количество усилий. Способствует соблюдению соглашений, чтобы повысить качество кода	Частичный контроль времени компиляции. Меньшая управляемость

3.2.1. Конфигурирование контейнеров с использованием XML

Когда контейнеры внедрения впервые появились (это было в начале 2000-х годов), они использовали в качестве механизма конфигурирования XML. Накопленный опыт работы с XML — как с механизмом конфигурирования — позднее показал, что он редко является лучшим вариантом.

Код XML обычно пространен и чувствителен к ошибкам. Когда вы конфигурируете контейнер внедрения при помощи XML, вы идентифицируете различные классы и интерфейсы, но у вас нет поддержки компилятора, который предупреждал бы вас, если вы где-нибудь сделаете орфографическую ошибку. И даже если имена классов написаны правильно, нет гарантии, что требуемая сборка будет находиться в пути приложения.

Преимущество XML-конфигурирования заключается в том, что вы можете изменить поведение приложения без перекомпиляции. Это ценное качество, если вы разрабатываете программу, продаваемую тысячам пользователей, так как такое приложение несложно настраивать. Однако если вы пишете приложение для внутреннего использования (то есть устанавливаемого небольшое количество раз. — *Примеч. пер.*) или создаете веб-сайт, для которого контролируете среду выполнения, то вам зачастую проще перекомпилировать и заново установить приложение, когда потребуется изменить его функциональность.

СОВЕТ

Используйте XML-конфигурирование только в тех случаях, когда необходимо обеспечить динамическое связывание. В других случаях выбирайте конфигурирование в коде или авторегистрацию.

Контейнер внедрения обычно использует XML-конфигурацию, содержащуюся в своем XML-файле, но иногда параметры конфигурации контейнера берутся из конфигурационного файла приложения. Этот вариант проиллюстрирован в следующем примере.

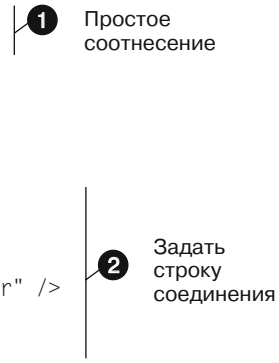
Пример: конфигурирование демонстрационного приложения с помощью xml

Поскольку контейнер Unity относится к числу наиболее XML-ориентированных контейнеров, представленных в этой книге, целесообразно использовать в примере с XML-конфигурированием именно его.

В данном примере будет конфигурироваться демонстрационное приложение для электронной коммерции из раздела 2.3. Значительная часть задачи состоит в применении конфигурации, представленной в табл. 3.1, но вы должны также реализовать схожую конфигурацию для обеспечения компоновки класса `HomeController`. Листинг 3.1 представляет конфигурацию, необходимую для установки и запуска приложения.

Листинг 3.1. Конфигурирование Unity при помощи XML

```
<register type="IBasketService"
          mapTo="BasketService" />
<register type="BasketDiscountPolicy"
          mapTo="RepositoryBasketDiscountPolicy" />
<register type="BasketRepository"
          mapTo="SqlBasketRepository">
  <constructor>
    <param name="connString">
      <value value="CommerceObjectContext"
             typeConverter="ConnectionStringConverter" />
    </param>
  </constructor>
</register>
<register type="DiscountRepository"
          mapTo="SqlDiscountRepository">
  <constructor>
    <param name="connString">
      <value value="CommerceObjectContext"
             typeConverter="ConnectionStringConverter" />
    </param>
    </constructor>
</register>
<register type="ProductRepository"
          mapTo="SqlProductRepository">
  <constructor>
    <param name="connString">
      <value value="CommerceObjectContext"
             typeConverter="ConnectionStringConverter" />
    </param>
  </constructor>
</register>
<register type="CurrencyProvider"
          mapTo="SqlCurrencyProvider">
  <constructor>
    <param name="connString">
      <value value="CommerceObjectContext"
             typeConverter="ConnectionStringConverter" />
    </param>
  </constructor>
</register>
```



Как видно из этого простого листинга, XML-конфигурация является очень пространной. В ней простые соотнесения вроде интерфейса `IBasketService` и класса `BasketService` ❶ легко выражаются простым элементом `register`.

Однако некоторые конкретные классы требуют для своей работы входной параметр — строку соединения, поэтому требуется указать, как будет находиться значение этой строки ❷. Если используется Unity, данная задача решается путем указания, что используется вспомогательный преобразователь типов с именем `ConnectionStringConverter`. Этот преобразователь (конвертер) будет выполнять поиск значения `CommerceObjectContext` среди стандартных строк соединения, определенных в файле `web.config`, и вернет строку соединения, имеющую искомое имя.

Оставшиеся элементы соответствуют двум этим образцам.

Поскольку Unity может автоматически преобразовывать запросы конкретных типов, даже если они явно не зарегистрированы, вам не нужно использовать XML-элементы для классов `HomeController` и `BasketController`.

Загрузка конфигурации в контейнер выполняется вызовом одного метода:

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

Метод `LoadConfiguration` загружает XML-конфигурацию, представленную в листинге 3.1, в контейнер. Контейнер, используя эту конфигурацию, может выполнить преобразования запросов для `HomeController`, и т. д.

Другие контейнеры внедрения также поддерживают XML-конфигурирование. Определяющие набор элементов варианты конфигурирования XML-схемы будут различными для каждого контейнера, но их логический смысл будет очень схожим.

ВНИМАНИЕ

По мере роста и усложнения приложения это же будет происходить с конфигурационным файлом, если компоновка будет основана на конфигурации. Этот файл будет расти, пока не превратится в настоящий камень преткновения. Ведь он описывает разнообразные концепции кодирования, такие как классы, параметры и тому подобное, но не обладает никакими преимуществами компиляции, возможностями отладки и т. д. Конфигурационные файлы будут все более уязвимыми при появлении ошибок, а обнаруживать такие ошибки будет все сложнее. Поэтому используйте такой подход, только если вам требуется динамическое связывание.

Из-за наличия таких недостатков, как пространность и чувствительность к ошибкам, следует отдавать предпочтение другим способам конфигурирования контейнеров. Конфигурирование в коде в общем и в частности напоминает XML-конфигурирование, но использует вместо XML обычный код.

3.2.2. Конфигурирование контейнеров при помощи кода

По-видимому, самым простым способом скомпоновать приложение будет написание соответствующего компоновочного кода. Может показаться, что это противоречит самому духу внедрения зависимостей, поскольку в коде жестко определяется, какие конкретные реализации будут использоваться для каждой абстракции. Однако если это будет выполнено в **корне компоновки**, то будет игнорироваться только одно из преимуществ, перечисленных в табл. 1.1.

Если зависимости будут закодированы жестко, будет потеряна возможность выполнять динамическое связывание, но, как отмечалось в главе 1, это может быть несущественным в приложениях некоторых типов. Если ваше приложение устанавливается на небольшом количестве устройств, да еще в контролируемой вами среде исполнения, то проще будет перекомпилировать и заново установить его, если понадобится замена модулей.

Я часто думаю, что люди чересчур сильно увлекаются определением файлов конфигурации. Зачастую язык программирования предоставляет простой и мощный механизм конфигурирования.

Мартин Фаулер[1]

Когда используется конфигурирование в коде, мы явно определяем отдельные соотнесения, как это делалось и при XML-конфигурации — только вместо XML применяется программный код.

Все контейнеры внедрения, за исключением Spring.NET, полностью поддерживают конфигурирование в коде — как альтернативу XML-конфигурированию. Фактически большинство контейнеров предоставляют данный механизм по умолчанию, тогда как XML-конфигурирование становится опциональным.

API, предназначенный для поддержки конфигурирования в коде, отличается в разных контейнерах, но конечная цель у них одинакова — определить отдельные ассоциации между абстракциями и конкретными типами.

СОВЕТ

Выбирайте конфигурирование в коде, а не XML-конфигурирование, если только вам не требуется динамическое связывание. Дополнительную помощь вам окажет компилятор, а система сборки Visual Studio будет автоматически копировать все требуемые зависимости в выходной каталог.

Многие конфигурационные API используют механизм обобщений (Generics) и «гибкие компоновщики» (Fluent Builders) для регистрации компонентов; `StructureMap` не является исключением.

Пример: конфигурирование демонстрационного приложения в коде

В подразделе 3.2.1 было показано, как осуществляется конфигурирование демонстрационного приложения для электронной коммерции посредством XML на примере контейнера Unity. Я мог бы применить Unity и для демонстрации конфигурирования в коде, но в этом примере воспользуюсь контейнером `StructureMap`. Он имеет API для тестирования, поэтому лучше вписывается в формат книги.

Используя конфигурационный API контейнера `StructureMap`, вы можете представить конфигурацию из листинга 3.1 более компактно, как показано в листинге 3.2.

Листинг 3.2. Конфигурирование StructureMap с использованием кода

```
c.For<IBasketService>().Use<BasketService>();
c.For<BasketDiscountPolicy>()
```

[1] Martin Fowler, “Inversion of Control Containers and the Dependency Injection pattern,” 2004, http://martinfowler.com/articles/injection.html.

```
    .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 представленным в листинге 3.1 и обратите внимание, насколько более компактным он является — хотя функционально аналогичен примеру листинга 3.1. Простое соотнесение типа представленного для `IBasketService` и `BasketService` осуществляется при помощи обобщенных методов `For` и `Use`. Переменная `c` представляет собой выражение `ConfigurationExpression`, но ее можно рассматривать как собственно контейнер.

Для поддержки классов, требующих строки соединения, вы должны вызвать за цепочкой вызовов `For/Use` метод `Ctor` и предоставить переменную для строки соединения. Метод `Ctor` ищет строковый параметр в конструкторе конкретного класса и использует найденное значение этого параметра.

Остальной код соответствует двум этим шаблонам.

Способ конфигурирования в коде не только намного компактнее XML-конфигурирования, но еще и обеспечивает поддержку компилятора. Аргументы типа, используемые в листинге 3.2, представляют реальные типы, контролируемые компилятором. API контейнера `StructureMap` даже поставляется с некоторыми обобщенными ограничениями, предписывающими компилятору осуществлять контроль совпадения типа, который определен в методе `Use`, с абстракциями, указанными в методе `For`. Если преобразование невозможно, код не будет компилироваться.

Хотя способ конфигурирование в коде является безопасным и практичным, он все же может потребовать большей поддержки, чем вам бы хотелось. Каждый раз, когда вы добавляете новый тип в приложение, вы должны не забыть зарегистрировать его, причем многие регистрации в конечном итоге оказываются похожими друг на друга. Авторегистрация устраняет эту проблему.

3.2.3. Конфигурирование контейнеров по соглашению

Заметили ли вы при изучении листинга 3.2, насколько схожи многие регистрации? Практически все основанные на использовании SQL-сервера компоненты доступа к данным следуют единому образцу: вы конфигурируете компонент соответствующей строкой соединения.

Многократное написание однотипного регистрационного кода нарушает принцип DRY[1]. Кроме того, написанный код похож на непродуктивный фрагмент инфраструктурного кода, не повышающего функциональной ценности приложения. Можно сэкономить время и силы, если автоматизировать регистрацию компонентов.

Набирающая популярность архитектурная модель представляет собой концепцию «Соглашение прежде конфигурации» (Convention over Configuration). Вместо того, чтобы писать и сопровождать большое количество конфигурационного кода, вы можете использовать систему соглашений, применяемых в коде.

Способ поиска контроллеров по именам, применяемый во фреймворке ASP.NET MVC, представляет собой великолепный пример простого соглашения.

1. Поступает запрос на `Controller`, называющийся `Home`.
2. Используемая по умолчанию фабрика контроллеров (Controller Factory) ищет в списке распространенных пространств имен (Namespaces) класс `HomeController`. Если она находит такой класс и он реализует `IController`, то соотношение выполнено.
3. Используемая по умолчанию фабрика контроллеров использует конструктор по умолчанию соотнесенного класса для создания экземпляра `Controller`.

В данном примере имеют место как минимум два соглашения: контроллер должен называться `[ControllerName]Controller` и он должен иметь конструктор, заданный по умолчанию.

Вы можете отклоняться от этих соглашений, реализуя собственную фабрику `IControllerFactory`, и это именно то, что сделал я, чтобы реализовать внедрение конструктора — более детально эта тема будет обсуждаться в главе 7.

Было бы хорошо, если бы удалось, воспользовавшись одним-двумя соглашениями, вообще обойтись без конфигурации контейнера. Ведь это довольно длительный процесс, к тому же чреватый возникновением ошибок. При работе с фабрикой `DefaultControllerFactory`, добавление нового `Controller` сводится к добавлению правильно названного класса в соответствующее пространство имен. Такое подспорье не помешало бы и при использовании внедрения конструктора.

Многие контейнеры внедрения допускают выполнение авторегистрации, что позволяет разработчикам устанавливать собственные соглашения.

ОПРЕДЕЛЕНИЕ

Авторегистрация — это способность к автоматической регистрации компонентов в контейнере путем сканирования одной или более сборок для поиска реализаций требуемых абстракций.

Соглашения могут применяться не только по отношению к контроллерам фреймворка ASP.NET MVC. Чем больше соглашений будет добавлено, тем в большей степени удастся автоматизировать процесс конфигурирования контейнера.

СОВЕТ

Преимущества использования принципа «Соглашение прежде конфигурации» не ограничиваются только поддержкой конфигурирования внедрения зависимостей. Такая концепция делает ваш код более стойким, поскольку он будет исправно функционировать, пока вы следуете установленным вами же соглашениям.

[1] Don't Repeat Yourself — не повторяй самого себя.

В действительности вам может потребоваться скомбинировать авторегистрацию и конфигурирование в коде или XML-конфигурирование, так как не всегда удается определить подходящие соглашения для всех используемых компонентов. Однако чем больше вашего кода будет подчиняться соглашениям, тем проще будет поддерживать такой код.

Пример: конфигурирование демонстрационного приложения с использованием авторегистрации

Контейнер `StructureMap` поддерживает авторегистрацию. Но я считаю, что будет интереснее использовать другой контейнер внедрения, чтобы продемонстрировать процесс конфигурирования демонстрационного приложения (для электронной коммерции) с применением соглашений. Я выбрал контейнер `Autofac`, так как у него имеется наглядный API для авторегистрации.

Если сравнить листинги 3.1 и 3.2, то несложно заметить, что регистрационные записи для различных компонентов доступа к данным в основном повторяются. Можно ли установить для них соглашения?

Все четыре конкретных типа обладают следующими сходными характеристиками.

- Все они определены в одной и той же сборке.
- Каждый из них наследует абстрактному базовому классу.
- Каждый имеет имя, начинающееся с `Sql`.
- Каждый имеет единственный общедоступный конструктор, принимающий строковый параметр с именем `connString`.

Сформированное корректное соглашение будет прямо определять эти характеристики для сканирования указанной в запросе сборки и регистрировать все классы, удовлетворяющие этому соглашению. Для контейнера `Autofac` такое соглашение будет иметь следующий вид:

```
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 соглашение будет сканировать сборку, содержащую компоненты доступа к данным. Существует много способов, которыми можно получить ссылку на эту сборку, но проще всего будет в качестве образца использовать имя типа, например `SqlProductRepository`, и получить сборку, в которой он находится. Можно также выбрать другой класс или поискать сборку по имени.

Теперь, когда у вас имеется сборка, вы можете сообщить контейнеру, что хотите сканировать ее. Метод `RegisterAssemblyTypes` свидетельствует о намерении регистрировать все типы в сборке, имена которых начинаются с `Sql`. Переменная

builder является экземпляром класса ContainerBuilder, но вы можете считать, что она представляет контейнер.

Каждый класс, обнаруженный при помощи фильтра Where, должен быть зарегистрирован со своим базовым классом. Например, поскольку базовым классом для класса SqlProductRepository является класс ProductRepository, он будет использоваться при ассоциировании ProductRepository с SqlProductRepository.

В заключение вы указываете, что конструктор каждого найденного класса должен иметь параметр connString и что значение этого параметра должно быть установлено из строки соединения, прочитанной из файла конфигурации.

Сравнение этого соглашения с четырьмя конкретными регистрациями из листинга 3.2 не является полностью правомерным, так как мы использовали два разных контейнера. Поэтому вы можете счесть, что полученные преимущества незначительны. Однако соглашение очень удобно при масштабировании.

Поскольку в рассматриваемый пример включены только четыре компонента доступа к данным, код удается сократить всего на несколько строк. Но после того как соглашение будет написано, оно с легкостью станет обрабатывать сотни компонентов.

Вы можете реализовать в виде соглашений и другие ассоциации из листингов 3.1 и 3.2, но на данном этапе это не слишком целесообразно. В качестве примера вы можете зарегистрировать через соглашение все сервисы:

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

Это соглашение сканирует указанную сборку на предмет наличия типов (классов), имена которых заканчиваются на Service, и регистрирует каждый тип вместе с интерфейсом, который этот тип реализует. Соглашение успешно регистрирует класс BasketService с интерфейсом IBasketService, и поскольку других соответствий для него в настоящий момент не определено, ничего больше не делается. Тем не менее имеет смысл все равно опубликовать это соглашение, чтобы разработчики следовали ему.

Авторегистрация — это мощная методика, потенциал которой позволяет сделать контейнер внедрения невидимым в приложении. После того как необходимые соглашения будут определены, модификация конфигурации контейнера потребуется лишь в очень редких случаях.

Итак, мы рассмотрели три различных подхода к конфигурированию контейнеров внедрения:

- XML;
- конфигурирование в коде;
- авторегистрация.

Ни один из них не исключает применения других. Вы можете совместить, например, авторегистрацию с явным соотнесением некоторых абстракций с конкретными типами или даже одновременно использовать все три подхода — часть

конфигурирования выполнять через авторегистрацию, часть — через конфигурирование в коде и часть — с помощью XML, чтобы иметь возможность динамического связывания.

Как правило, следует выбирать авторегистрацию как основной способ конфигурирования и дополнительно пользоваться конфигурированием в коде для обработки особых ситуаций. XML следует применять в случаях, когда вам требуется изменять реализацию без перекомпиляции приложения (что на самом деле необходимо реже, чем вы думаете).

Теперь, когда вы узнали, как конфигурируются контейнеры внедрения и как с их помощью преобразовываются графы объектов, подумаем, как их можно использовать. Но применять контейнеры внедрения — одно дело, а использовать их *правильно* — совсем другое.

3.3. Паттерны контейнеров внедрения

Контейнеры внедрения зависимостей — это мощный инструмент, но, как и любой инструмент, их можно использовать как правильно, так и неправильно. Хорошие повара обращаются со своими ножами аккуратно. Так и вы должны уметь правильно применять ваш контейнер — пальцев он вам не отрежет, но вам, возможно, не удастся воспользоваться теми достоинствами, которыми он обладает.

Главное, что необходимо понять, — где в архитектуре приложения находится то место, в котором должен использоваться контейнер внедрения. Но понять это — всего полдела. Нужно еще научиться это понимание использовать. Следующие два мини-паттерна дают ответ на этот вопрос.

3.3.1. Корень компоновки

Где мы должны компоновать графы объектов?

Максимально близко к точке входа (Entry Point) приложения.

Контейнер внедрения — это библиотека, которую вы потенциально можете использовать в любом месте приложения, где только пожелаете, но это не значит, что так следует поступать. Вы, конечно, *можете* применять контейнер так, чтобы он входил в состав многих ваших классов. Но лучше локализовать его в одном месте приложения. Это место называется корнем компоновки (Composition Root), и использовать контейнер следует только в нем (рис. 3.6).

Концепция корня компоновки не является специфичной для контейнеров внедрения зависимостей. Эта концепция применяется и в случае использования «внедрения зависимостей для бедных», но я считаю важным обсуждать ее в контексте контейнеров, поскольку понимание этого шаблона позволит вам использовать контейнеры правильно и эффективно. Прежде чем приступить к обсуждению вопроса влияния корня компоновки на применение контейнеров внедрения, я кратко опишу их основные характеристики.

Корень компоновки как универсальная концепция

Когда вы пишете слабо связанный код, вы создаете множество классов, которые нужно объединить, чтобы получилось готовое приложение. Возможно, покажется заманчивым компоновать классы постепенно, чтобы создавать маленькие подсистемы, но такое решение впоследствии ограничит возможность **перехватывать** эти системы для изменения их поведения. Следует компоновать все классы одновременно.

ОПРЕДЕЛЕНИЕ

Корень компоновки — это уникальное (желательно) место в приложении, где все модули соединяются вместе.

ПРИМЕЧАНИЕ

Корень компоновки может быть распределен по нескольким классам, если они находятся в одном модуле.

Если рассматривать внедрение конструктора изолированно, то может возникнуть вопрос: может быть, мы просто откладываем решение о выборе зависимости на потом? Да, это происходит, и это хорошо; такой перенос означает, что у вас появляется центральное место, в котором можно устанавливать непосредственную связь между взаимодействующими классами. Корень компоновки действует как посредник, соединяющий потребителей с их сервисами. Нат Прайс предпочитает термин «подключение через посредника» (Third-party Connect) термину ВЗ[1] именно по этой причине.

Чем на более позднее время вы откладываете принятие решения о способе соединения классов, тем дольше ваш код сохраняет возможность выбора варианта дальнейшей работы. Таким образом, корень компоновки должен размещаться как можно ближе к точке входа приложения (рис. 3.6).

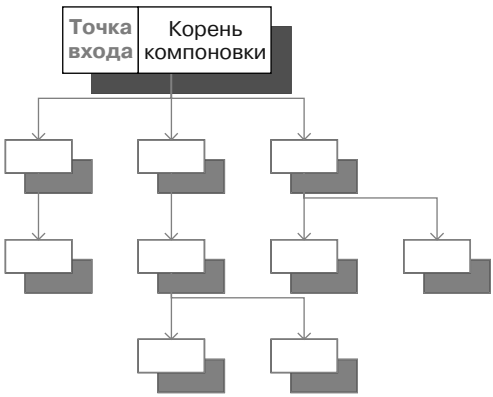


Рис. 3.6. Когда приложение собирается из нескольких слабо связанных классов, компоновка должна выполняться настолько близко к точке входа приложения, насколько это возможно. Корень компоновки компонует граф объекта, который затем выполняет реальную работу приложения

[1] *Nat Pryce* "Dependency Injection" Considered Harmful, 2011: http://www.natpryce.com/articles/000783.html.

ПРИМЕЧАНИЕ

Мне нравится считать корень компоновки архитектурным эквивалентом одной концепции метода экономной разработки программного обеспечения (Lean Software Development)[1]. Эта концепция называется «Последний ответственный момент» (Last Responsible Moment). Основная ее идея — откладывать принятие решений настолько, насколько это максимально возможно (но не дольше), так как предпочтительно держать возможные варианты действий открытыми как можно дольше и принимать решения на основе как можно большего объема информации. Когда речь идет о сборке приложений, мы можем таким же образом отложить решение о включении зависимостей в корень компоновки.

Даже модульное приложение, использующее слабо связанные классы и динамическое связывание для компоновки себя самого, имеет корень, содержащий точку входа в приложение. Некоторые примеры:

- консольное приложение — исполняемый файл (EXE), имеет метод `Main`;
- веб-приложение, разработанное на платформе `ASP.NET` — библиотека (DLL), которая содержит обработчик событий `Application_Start` в `Global.asax`;
- приложение платформы WPF — исполняемый файл (EXE) с файлом `App.xaml`;
- сервис на платформе WCF — библиотека (DLL), содержащая класс — потомок интерфейса сервиса (хотя используя `ServiceHostFactory`, можно создать еще и более низкоуровневую точку входа).

Существуют и другие технологии, общей чертой которых является то, что один модуль содержит точку входа в приложение: эта точка является корнем приложения, что проиллюстрировано на рис. 3.7. Корень компоновки должен находиться в корне приложения, что позволяет корректно собрать программу.

Не следует пытаться компоновать классы в каком-либо из модулей, так как такой подход ограничивает ваши возможности. Все классы в модулях приложения должны использовать внедрение конструктора (или в редких случаях один из других шаблонов из числа описанных в главе 4) и предоставить корню компоновки сборку графа объектов приложения. Любой применяемый контейнер внедрения должен быть ограничен корнем компоновки.

Использование контейнера внедрения в корне компоновки

Контейнер внедрения может ошибочно считаться локатором сервисов (Service Locator), но он должен использоваться только как механизм компоновки графов объектов. Если рассматривать контейнер внедрения с этой точки зрения, то имеет смысл ограничивать место его применения только корнем компоновки. Такой подход имеет важное достоинство, заключающееся в том, что он исключает любое связывание между контейнером внедрения и остальным кодом приложения.

СОВЕТ

Допускайте ссылки на контейнер внедрения только из корня компоновки. Никакие другие модули не должны иметь таких ссылок.

[1] Смотрите, например, *Mary Poppendieck, Tom Poppendieck* Implementing Lean Software Development: From Concept to Cash. —New York: Addison-Wesley, 2007.

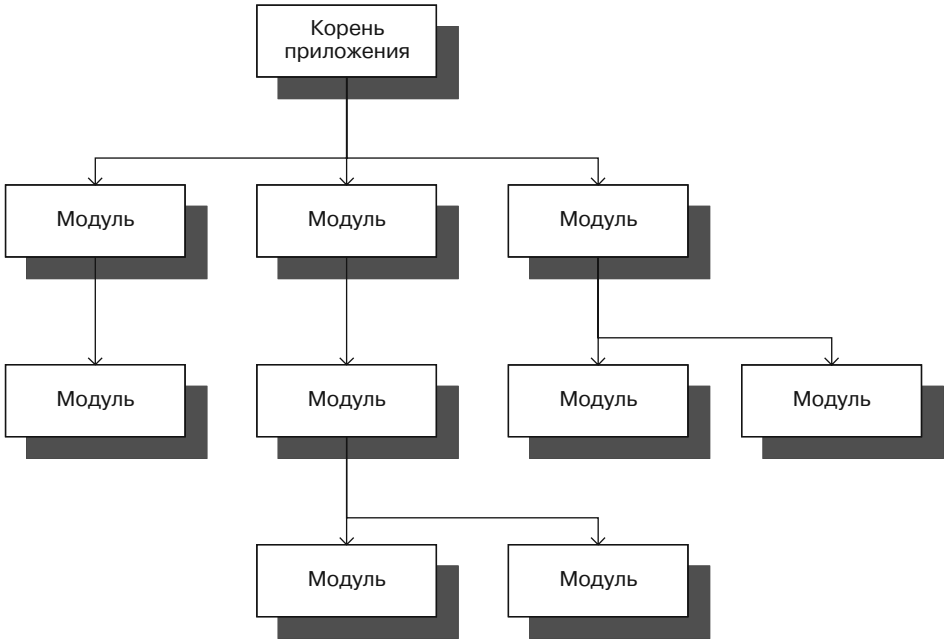


Рис. 3.7. Точка входа приложения лежит в корне модульного приложения. Явно или неявно корень объединяет прочие модули. Корень компоновки должен быть помещен в корень приложения — настолько близко к точке входа, насколько возможно

На рис. 3.8 видно, что только корень компоновки содержит ссылку на контейнер внедрения. Остальной код приложения не содержит ссылок на контейнер и использует шаблоны, описанные в главе 4. Контейнеры внедрения понимают эти шаблоны и применяют их для компоновки объектного графа приложения.

Корень компоновки может быть реализован при помощи контейнера внедрения. Это означает, что контейнер выполняет компоновку всего объектного графа приложения одним вызовом метода `Resolve`. Всякий раз, когда я обсуждал эту тему с разработчиками, я замечал, что такой подход кажется им неудобным, поскольку они убеждены, что он очень неэффективен и сильно снижает производительность. На самом деле о таких вещах можно не беспокоиться, так как такого почти никогда не случается, а в том очень небольшом количестве случаев, когда такое все же происходит, всегда имеются способы устранить эту проблему[1].

СОВЕТ

Не беспокойтесь о снижении производительности при использовании контейнеров внедрения для сборки больших графов объектов. Это почти никогда не является проблемой.

Когда такое случается в случае приложений, в основном обслуживающих запросы (Request-Based Applications), таких как веб-сайты и сервисы, вы конфигурируете

[1] *Mark Seemann* Compose object graphs with confidence, 2011: http://blog.ploeh.dk/2011/03/04/Com-poseObjectGraphsWithConfidence.aspx.

контейнер лишь однажды, но преобразуете граф объектов для каждого входящего запроса. Наше демонстрационное приложение является примером такой ситуации.

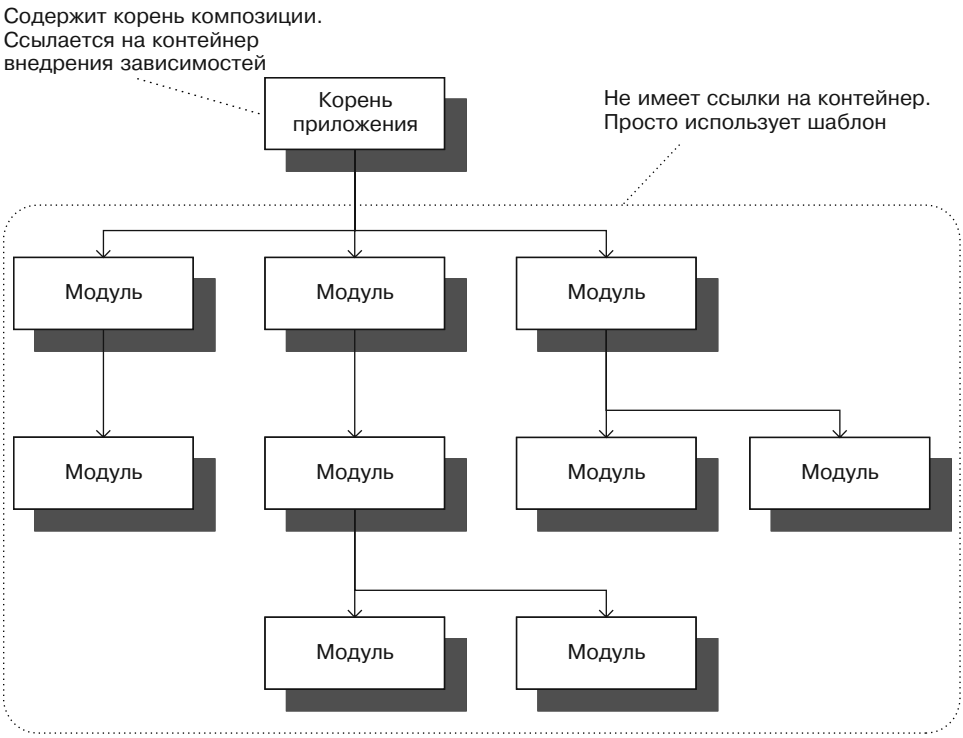


Рис. 3.8. Только корень компоновки, расположенный в корне приложения, может иметь ссылку на контейнер внедрения. Все прочие модули приложения должны использовать шаблоны внедрения зависимостей и не иметь ссылок на контейнер

Пример: реализация корня компоновки

Демонстрационное приложение, представленное в разделе 2.3, должно иметь корень компоновки для компоновки графов объектов для входящих HTTP-запросов. Как для любого веб-приложения, реализованного на платформе .NET, точкой входа является метод `Application_Start` в файле `Global.asax`.

В этом примере я использовал контейнер `Castle Windsor`, но получился код, похожий на код для любого другого контейнера. Метод `Application_Start` для контейнера `Castle Windsor` будет иметь следующий вид:

```
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);
}
```

Прежде чем вы сможете конфигурировать контейнер, вы должны создать новый экземпляр. Поскольку вся настройка приложения выполняется в классе, называемом `CommerceWindsorInstaller`, вы устанавливаете его в контейнер для конфигурирования. Код в `CommerceWindsorInstaller` реализован при помощи API контейнера `Castle Windsor`, но концептуально ситуация идентична примерам из раздела 3.2.

Чтобы разрешить контейнеру подключать контроллеры в приложение, вы должны использовать подходящий шов в модели ASP.NET MVC, называемый `IControllerFactory` (подробно обсуждается в разделе 7.2). Сейчас достаточно понять, что для интеграции с ASP.NET MVC вам необходимо создать адаптер вокруг контейнера и известить фреймворк о нем.

Поскольку метод `Application_Start` выполняется однократно, контейнер оказывается единственным экземпляром, инициализируемым единожды. Когда поступают запросы, экземпляр контейнера должен обслуживать их параллельно — но поскольку все контейнеры реализованы с использованием потокобезопасных методов `Resolve`, это не представляет проблемы.

Поскольку вы настраиваете `ASP.NET MVC` на работу с `WindsorControllerFactory`, он будет вызывать свой метод `GetControllerInstance` для каждого поступившего HTTP-запроса (подробно об этом можно прочитать в разделе 7.2). Реализация делегирует работу контейнеру:

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

Заметьте, что вы более или менее возвращаетесь к вводным примерам из подраздела 3.1.1. Метод `Resolve` компонует полный граф, который должен использоваться для управления данным отдельным запросом, и возвращает его. Это единственное место в приложении, где вызывается метод `Resolve`.

СОВЕТ

Код приложения должен содержать только один вызов метода Resolve.

Корень компоновки в этом примере распределен по нескольким классам, как показано на рис. 3.9. Это сделано намеренно — важно, что все классы находятся в одном модуле, который в данном случае является еще и корнем приложения.

Самое важное, что необходимо отметить в данном месте: используемые три класса являются единственными классами в приложении, ссылающимися на контейнер внедрения. Остальной код приложения применяет только шаблон «Внедрение конструктора»; если вы не верите мне, вернитесь назад и перечитайте главу 2.

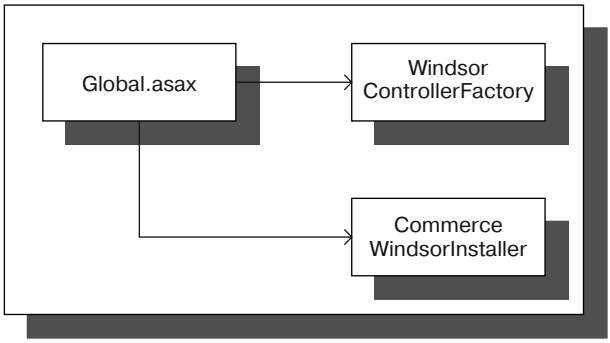


Рис. 3.9. Корень компоновки распределен по трем классам, но они определены в одном и том же модуле

СОВЕТ

Я хочу завершить содержащиеся в данном разделе инструкции, перефразировав известный голливудский принцип: не вызывай контейнер; он сам вызовет тебя[1].

COMMON SERVICE LOCATOR

Существует проект с открытым кодом, называемый Common Service Locator (http://commonservicelocator.codeplex.com), целью которого является обеспечение независимости кода приложения от конкретных контейнеров путем скрытия контейнеров за единым интерфейсом `IServiceLocator`.

Я надеюсь, что приведенные объяснения того, как корень компоновки фактически изолирует код приложения от контейнеров внедрения, помогут понять, почему вам не нужен Common Service Locator. Как я покажу в разделе 5.4, поскольку «Локатор сервисов» является антипаттерном, лучше им не пользоваться. В случае применения корня компоновки необходимость в нем вообще отпадает.

В главе 7 вы можете прочитать о специфических деталях реализации корней компоновки в разных фреймворках (включая ASP.NET MVC). В текущем контексте способ, *которым* вы делаете это, гораздо менее важен, чем место, *где* вы это выполняете. Как понятно из названия, корень компоновки является частью корня приложения, где вы компонуете все слабо связанные классы. Это справедливо независимо от того, используете ли вы контейнер внедрения или «ВЗ для бедных». Однако когда вы применяете контейнер внедрения, вы должны следовать шаблону «Регистрация, преобразование, высвобождение».

3.3.2. «Регистрация, преобразование, высвобождение»

Как нужно использовать контейнер внедрения?

Следуя строгой последовательности вызовов: регистрация, преобразование, высвобождение.

[1] Имеется в виду фраза Don't Call Me (I'll Call You). — *Примеч. ред.*

Паттерн «Корень компоновки» определяет, где должен использоваться контейнер внедрения. Однако он ничего не говорит о том, как его использовать. Паттерн «Регистрация, преобразование, высвобождение» (Register Resolve Release) отвечает на этот вопрос.

Контейнер внедрения должен применяться в три этапа, называемые «Регистрация» (Register), «Преобразование» (Resolve), и «Высвобождение» (Release). Таблица 3.3 представляет более подробную информацию об этих фазах.

Таблица 3.3. Фазы контейнера

Фаза	Что происходит на этой фазе?	Дополнительное чтение
Регистрация	Регистрация компонент в контейнере. Контейнер конфигурируется указанием, какие классы он может использовать, как он должен соотносить абстракции и конкретные типы и (необязательно) как отдельные классы должны быть соединены вместе	В разделе 3.2 уже обсуждался вопрос о том, как конфигурируется контейнер внедрения. В части 4 будут детально обсуждаться процессы конфигурирования шести конкретных контейнеров
Преобразование	Преобразование корневых компонент. Отдельный граф объекта преобразуется из запроса для одного типа	В разделе 3.1 показано, как преобразуются графы объектов в контейнере внедрения. В части 4 можно прочитать о API конкретных контейнеров
Высвобождение	Удаление компонент из контейнера. Все графы объектов, преобразованные в предыдущей фазе, должны быть удалены, когда исчезает потребность в них. Это служит контейнеру сигналом о том, что он может очистить граф объектов, что особенно важно, если некоторые из компонент являются одноразовыми	В главе 8 обсуждается управление жизненным циклом, включая вопрос важности очистки. Кроме того, в части 4 рассматриваются API для управления жизненным циклом конкретных контейнеров

ОПРЕДЕЛЕНИЕ

Шаблон «Регистрация, преобразование, высвобождение» определяет, что методы контейнера внедрения должны вызываться в строго определенной последовательности: Register, Resolve и Release (рис. 3.10).

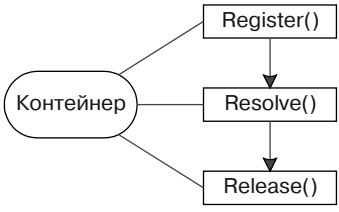


Рис. 3.10. Методы контейнера должны вызываться в строго определенном порядке: сначала метод Register, за ним метод Resolve и, наконец, метод Release

Эти фазы должны использоваться в определенном порядке. Вы не можете произвольно менять их местами. Например, вы не можете вернуться назад и переконфигурировать контейнер после начала преобразования графов объектов. Иногда

меня спрашивают, как добавить компоненты в контейнер после того, как начался процесс преобразования. Не делайте этого, а то не оберетесь проблем.

ПРИМЕЧАНИЕ

Некоторые контейнеры внедрения не поддерживают явное высвобождение графов объектов и вместо этого полагаются на сборщик мусора .NET. Когда используются такие контейнеры, следует применять модифицированный шаблон «Регистрация, высвобождение» (Register Resolve) и устранять потенциальные утечки ресурсов в реализациях ваших объектов. Подробности см. в главе 8.

Следующий пункт посвящен обсуждению методов `Register`, `Resolve` и `Release`, а также соответствующих фаз. Контейнер `Castle Windsor` имеет три метода с точно такими именами, и фазы названы по этим методам. Другие контейнеры внедрения могут использовать другие имена, но их концепции идентичны. Я применяю имена из `Castle Windsor` только потому, что они имеют удобную терминологию, а также представляют собой великолепную аллитерацию.

Статическая структура

В чистом виде паттерн «Регистрация, преобразование, высвобождение» требует, чтобы вы вызывали только один метод на каждой фазе. Кшиштоф Козьмич называет его «Паттерн трех вызовов» (Three Calls Pattern)[1], так как разрешено сделать три вызова методов.

С методами `Resolve` и `Release` все просто. В подразделе 3.3.1 я уже отмечал, что приложение должно содержать только *один* вызов метода `Resolve`. Как следствие, вы должны всегда высвобождать (`Release`) все, что вы преобразовали (`Resolve`).

СОВЕТ

Любой граф объектов, скомпонованный при помощи метода Resolve, должен удаляться методом Release.

Конфигурирование контейнера в одном вызове одного метода требует дополнительного разъяснения. Причина, по которой регистрация компонентов должна выполняться в одном вызове метода, состоит в том, что вы должны рассматривать конфигурирование контейнера как единую атомарную операцию. После того как конфигурирование будет выполнено, контейнер должен рассматриваться как «только для чтения».

Контейнер `Autofac` даже делает эту идею явной, передавая работу по конфигурированию контейнера в отдельный `ContainerBuilder`. Вы регистрируете (`Register`) компоненты при помощи `ContainerBuilder`, и когда вы закончите это, вы запросите его построить экземпляр контейнера из конфигурации. В случае с `Autofac` вы не конфигурируете контейнер напрямую.

Если рассматривать конфигурирование как единую атомарную операцию, то легче управлять кодом конфигурирования, так как становится очевидным, где он должен выполняться. Многие контейнеры внедрения используют этот принцип для замораживания конфигурации, если вы начали преобразование графов объектов для нее. Тогда преобразование выполняется лучше.

[1] *Krzysztof Kozrmic* How I use Inversion of Control containers, 2010: http://kozmic.pl/2010/06/20/how-i-use-inversion-of-control-containers.

Если вы помните листинг 3.2, то можете поспорить, утверждая, что он содержит более чем один вызов метода. Регистрация всегда требует выполнения нескольких операторов, но большинство контейнеров имеют механизм упаковки, позволяющий вам изолировать все эти операторы конфигурирования в единственный класс (возможно, скомпонованный из других классов). В Autofac они называются Modules, в StructureMap — Registries, а в Castle Windsor — Installers. Общим в них является то, что они могут использоваться для конфигурирования контейнера за один вызов метода. В подразделе 3.3.1 вы уже видели, как Castle Windsor использует Installer:

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

Для контейнеров внедрения, не имеющих механизма упаковки, вы можете всегда создать свой класс, который изолирует конфигурирование в одном методе.

Совет о том, что приложение должно иметь только по одной строке кода для фаз преобразования и уничтожения, должен быть воспринят всерьез, причем для фазы регистрации он должен быть осознан более глубоко. В этом вопросе важно то, что регистрация должна быть выполнена прежде, чем будет вызван метод Resolve. Рисунок 3.11 иллюстрирует вид последовательности вызовов, включая скрытие нескольких вызовов метода Register.

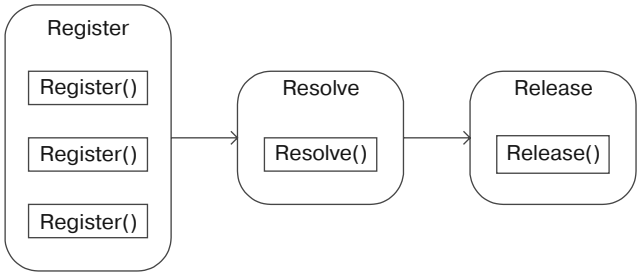


Рис. 3.11. Произвольное количество вызовов метода Register может осуществляться на фазе регистрации, но вы должны рассматривать их как атомарные операции. В фазах преобразования и высвобождения вы буквально должны вызвать соответствующие методы по одному разу

Распространенный источник заблуждений заключается в том, что «Шаблон трех вызовов» делает твердое утверждение о том, как часто каждый метод должен появляться в вашем коде, но ничего не говорит о том, сколько раз он может быть вызван.

Динамическое взаимодействие

Название «Паттерн трех вызовов» наводит на мысль, что каждый метод должен быть *вызван* только один раз. Источник такого заблуждения кроется в самом названии шаблона, и это одна из нескольких причин, почему я предпочитаю название «Регистрация, преобразование, высвобождение».

«Паттерн трех вызовов» предписывает, что в приложении должна быть только *одна строка кода* на вызов каждого метода. Однако, в зависимости от обстоятельств, некоторые методы могут *вызываться* более одного раза.

В однопоточном приложении — например, настольном приложении, утилите командной строки или пакетном задании — каждый метод действительно будет вызван лишь один раз, как показано на рис. 3.12. За конфигурированием контейнера сразу следует компоновка объектного графа приложения, который выполняет фактическую работу. Когда работа будет завершена, вызывается метод `Release`, затем приложение завершает свою работу.

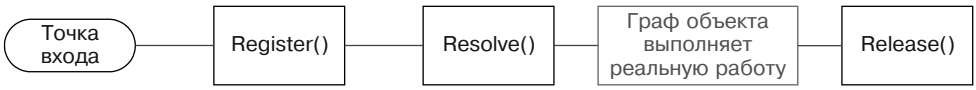


Рис. 3.12. В однопоточном приложении каждый метод будет вызываться лишь однажды

В обслуживающем запросы приложении, таком как веб-сайт или программа обработки асинхронных сообщений, корень компоновки собирает граф объекта для каждого поступившего запроса. В приложениях такого типа, как показано на рис. 3.13, метод `Register` также вызывается лишь один раз, но методы `Resolve` и `Release` вызываются попарно для каждого запроса — потенциально количество таких операций может быть огромным.

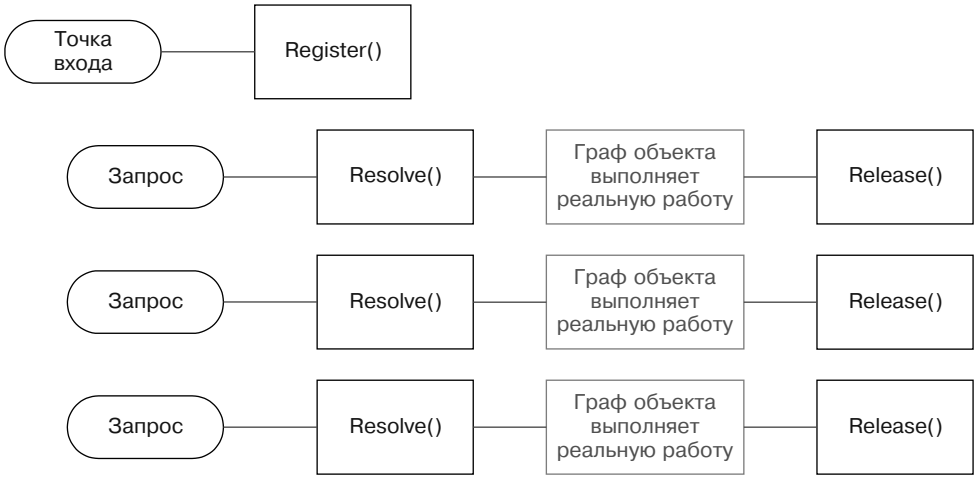


Рис. 3.13. В ориентированном на обслуживание запросов приложении метод Register вызывается лишь однажды, в то время как методы Resolve и Release вызываются многократно — по одному разу на запрос

Важно отметить, что контейнер конфигурируется только один раз. Контейнер — это совместно используемый экземпляр класса, применяемый для высвобождения многочисленных запросов, но сама конфигурация должна оставаться стабильной и неизменной.

Динамическая схема паттерна «Регистрация, преобразование, высвобождение» почти полностью противоположна схеме статической — сравните рис. 3.11 и 3.13. В статической схеме допускаются многочисленные строки кода, в которых вызы-

вается метод Register, а в динамической схеме этот блок кода должен быть вызван лишь однажды. С другой стороны, правило для статической схемы гласит, что вызов методов Resolve и Release должен выполняться однократно, тогда как в динамической системе они могут быть вызваны многократно.

Такая ситуация может показаться сложной и запутанной, но, как показывает следующий пример, в любом случае осуществляются вызовы только трех методов.

Пример: использование шаблона «Регистрация, преобразование, высвобождение»

В следующем примере будет реализован корень компоновки для приложения из раздела 2.3. При этом будет задействован контейнер Castle Windsor. Это тот же контейнер использовался в примере подраздела 3.3.1, поэтому данный пример может рассматриваться как продолжение предыдущего.

Точкой входа приложения является метод Application_Start, и, так как речь идет о веб-сайте, фаза регистрации изолируется от фаз преобразования и уничтожения, поскольку контейнер должен быть сконфигурирован только однажды. Код практически эквивалентен коду предыдущего примера, но я бы хотел расставить акценты по-иному:

```
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);
}
```

В соответствии с шаблоном «Регистрация, преобразование, высвобождение» вызов первого метода, осуществляемый в экземпляре контейнера, должен быть атомарным вызовом Register. В данном примере метод называется Install, и CommerceWindsorInstaller собирает отдельные регистрации в один класс. Листинг 3.3 содержит код реализации класса CommerceWindsorInstaller.

Листинг 3.3. Изоляция многочисленных регистраций

```
public class CommerceWindsorInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container,
        IConfigurationStore store)
    {
        container.Register(AllTypes
            .FromAssemblyContaining<HomeController>()
            .BasedOn<IController>()
            .Configure(r => r.LifeStyle.PerWebRequest));
```

```
        container.Register(AllTypes
            .FromAssemblyContaining<BasketService>()
            .Where(t => t.Name.EndsWith("Service"))
            .WithService
            .FirstInterface());
        container.Register(AllTypes
            .FromAssemblyContaining<BasketDiscountPolicy>()
            .Where(t => t.Name.EndsWith("Policy"))
            .WithService
            .Select((t, b) => new[] { t.BaseType }));
        string connectionString =
            ConfigurationManager.ConnectionStrings
            ["CommerceObjectContext"].ConnectionString;
        container.Register(AllTypes
            .FromAssemblyContaining<SqlProductRepository>()
            .Where(t => t.Name.StartsWith("Sql"))
            .WithService
            .Select((t, b) => new[] { t.BaseType })
            .Configure(r => r.LifeStyle.PerWebRequest
                .DependsOn((new
                {
                    connString = connectionString
                }))));
    }
}
```

1 Вызов метода Register

Класс `CommerceWindsorInstaller` выглядит сложным, но главное, что следует отметить, — он инкапсулирует четыре вызова метода `Register` (1). И это единственный способ, которым он взаимодействует с контейнером. Оставшаяся часть кода в настоящий момент неважна. Он использует соглашения для конфигурирования контейнера. Об API авторегистрации контейнера `Castle Windsor` можно прочитать в подразделе 10.1.2.

Поскольку демонстрационное приложение является веб-сайтом, `Resolve` и `Release` должны быть реализованы как единая пара. Для каждого HTTP-запроса вы должны преобразовывать граф объектов, который будет обрабатывать запрос, и когда запрос будет обработан, этот граф должен быть удален. Эти действия выполняются из класса, называемого `WindsorControllerFactory`, который является наследником класса `DefaultControllerFactory` из фреймворка ASP.NET MVC. Подробнее об этом рассказано в разделе 7.2, где описывается шов ASP.NET MVC.

Фреймворк ASP.NET MVC вызывает метод `GetControllerInstance` для преобразования `IControllers` и метод `ReleaseController`, когда запрос будет обработан. Методы `Resolve` и `Release` вызываются посредством вышеупомянутых методов:

```
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 сгенерированный граф объекта. После обработки запроса фреймворк ASP.NET MVC вызывает метод `ReleaseController`, передавая ему ранее созданный методом `GetControllerInstance` экземпляр `IController`, и этот экземпляр контроллера передается в метод `Release`.

Обратите внимание, что во всем коде приложения появляются только вызовы методов `Resolve` и `Release`.

Этот пример несколько более глубок, чем предыдущий, показывавший шаблон корня компоновки, но в основном это тот же код. Паттерн «Корень компоновки» определяет, где должны компоноваться графы объектов, а паттерн «Регистрация преобразование, высвобождение» задает способ использования контроллера внедрения в корне компоновки.

В следующей главе я рассмотрю и другие паттерны внедрения зависимостей, но сначала хотел бы сделать небольшое отступление и обсудить вопрос о том, как контейнеры внедрения зависимостей вписываются в общую структуру платформы .NET.

3.4. Панорама контейнеров внедрения

Теперь, после того как я описал, что такое контейнер внедрения зависимостей и как использовать его в корне компоновки, хочу сделать небольшое отступление и выполнить обзор современного состояния контейнеров внедрения в экосистеме .NET. Кроме того, приведу также исторический экскурс и рассмотрю вопрос, почему есть так много контейнеров с открытым кодом.

Поскольку существует большое количество контейнеров, из которых можно выбирать, я также приведу небольшое руководство о том, как следует выбирать контейнер.

3.4.1. Выбор контейнера внедрения зависимостей

Решение использовать ВЗ — как методологии — не должно зависеть от выбора конкретного контейнера. Внедрение зависимостей является прежде всего и в основе своей методологией, и я буду применять «Внедрение для бедных» в частях 2 и 3, чтобы подчеркнуть это.

Тем не менее применение контейнера внедрения упростит вам жизнь, так что используйте их везде, где только можно. При применении контейнеров в соответствии с паттернами, описанными в этой книге, возникают некоторые сложности, но кое-какие аспекты заслуживают рассмотрения.

Процесс принятия решений

Контейнер внедрения зависимостей является стабильной зависимостью, поэтому, с точки зрения внедрения зависимостей, его использование должно протекать без осложнений, но имеются другие, второстепенные проблемы.

- Добавление другой библиотеки всегда усложняет приложение — это касается не сопровождаемости самой программы, а относится к так называемой кривой обучения (которая характеризует процесс постепенного повышения квалификации. — *Примеч. пер.*). Новые разработчики должны будут не только изучать код приложения, но еще и осваивать API выбранного контейнера. В этой главе я надеюсь дать вам представление о том, что, изолируя использование контейнера в корне компоновки, вы сможете защитить контейнер от новичков. Если вы используете авторегистрацию, контейнер даже может взять на себя заботу об инфраструктуре без привлечения внимания к себе самому.
- За исключением работы с Managed Extensibility Framework (MEF), вы должны устанавливать сборки контейнеров вместе с вашим приложением. Это может быть связано с определенными юридическими нюансами, хотя вероятность этого и невелика. Все распространенные контейнеры с открытым кодом имеют разрешительные лицензии, но я не юрист, поэтому не полагайтесь всецело на мои слова: проконсультируйтесь с вашим юристом.
- Все контейнеры, за исключением MEF, являются библиотеками с открытым кодом. Для каждого из них вы можете решить, насколько вы доверяете людям или организациям, стоящим за ними.
- Между разными контейнерами существуют технические различия. Во введении к части 4 дана таблица, в которой перечисляются достоинства и недостатки каждого контейнера, описанного в этой книге. Вы можете использовать эту таблицу для общего знакомства с контейнерами внедрения и затем читать в главе только материал, касающийся заинтересовавшего вас контейнера.

Выбор контейнера не должен иметь большого значения. Выберите один из них для проверки, поработайте с ним и оцените, устраивает ли он вас — если нет, просто замените его на другой. Если вы разместили контейнер в корне компоновки, то заменить этот контейнер будет относительно легко.

Избранные контейнеры внедрения

Я не хочу указывать вам, какой контейнер выбрать. Выбор контейнера внедрения заключается не только в технической оценке. Вы должны также оценить тип лицензии, степень доверия разработчикам контейнера, насколько он соответствует стратегии вашей организации и учесть много других аспектов.

Большинство контейнеров внедрения относятся к числу проектов с открытым программным кодом (то есть свободно распространяются). Это следует иметь в виду, так как зачастую они лишены официальной поддержки и часто плохо документированы.

В табл. 3.4 перечислены контейнеры, описанные в части 4 книги. Их отбор производился по таким критериям, как релевантность, удельный вес на рынке и характерные особенности. Но каким бы длинным ни был этот список, он всегда ос-

танется субъективным и незавершенным. Ряд популярных контейнеров (например, `Ninject`) не включены в этот список, в основном из-за ограничений по срокам публикации и объему книги.

Таблица 3.4. Избранные контейнеры внедрения зависимостей

Название	Организация	Комментарии
Castle Windsor	С открытым кодом	Полнофункциональный и широко используемый
StructureMap	С открытым кодом	Полнофункциональный и широко используемый
Spring.NET	SpringSource	Полнофункциональный и широко используемый порт к контейнеру внедрения Spring, разработанному для использования с языком Java
Autofac	С открытым кодом	Более современный контейнер, разработанный с использованием возможностей языка C#
Unity	Microsoft patterns & practices	Первый опыт Microsoft в области внедрения зависимостей. Не является продуктом как таковым
Managed Extensibility Framework (MEF)	Microsoft	Поставляется в составе .NET 4, но на самом деле не является контейнером внедрения

В части 4 подробнее рассматриваются контейнеры Castle Windsor и StructureMap.

Обратите внимание, как доминируют в данной области проекты с открытым кодом и прочие некоммерческие проекты. Microsoft играет в этой сфере незначительную роль.

3.4.2. Microsoft и внедрение зависимостей

Несмотря на то что платформа .NET является продуктом Microsoft, другие организации (часто даже независимые программисты) играют намного более значительную роль в области внедрения зависимостей для этой платформы. В принципе, это может объясняться тем, что Microsoft не включает никакого контейнера внедрения в базовой библиотеке классов (BCL). Даже как независимое решение единственным разработанным Microsoft контейнером является относительно недавно появившийся контейнер Unity.

Необходимо сказать, что в течение нескольких первых лет существования фреймворка .NET Microsoft попросту игнорировала саму концепцию внедрения зависимостей. Нельзя утверждать почему, но я сомневаюсь, что это была осознанная стратегия.

Краткая история внедрения зависимостей в .NET

Здесь представлена моя субъективная попытка описать историю внедрения зависимостей в .NET, чтобы объяснить, почему Microsoft игнорировала ВЗ в течение такого длительного периода времени (рис. 3.14). Насколько мне известно, достоверного ответа на этот вопрос не существует.

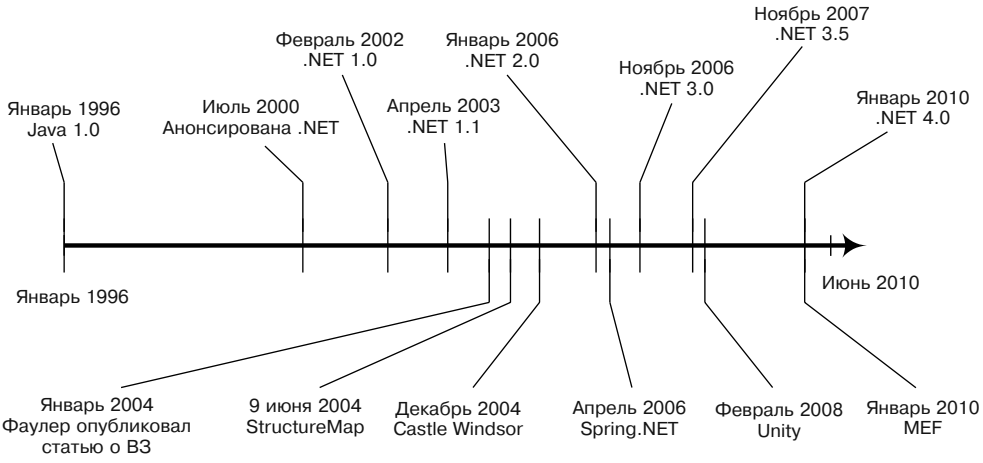


Рис. 3.14. Хронология реализаций выбранной платформы и контейнеров внедрения. Обратите внимание, насколько продвинутым был язык Java в 2004 году по сравнению с .NET

До того как ВЗ начали реализовывать в .NET, ВЗ начало развиваться в сообществе свободных разработчиков, использующих язык Java. Мартин Фаулер опубликовал свою статью о внедрениях зависимостей[1] в начале 2004 года. В то время актуальной версией .NET была 1.1, и Microsoft работала над версией .NET 2.0, тогда как Java приближался к своему десятилетнему юбилею. Мне кажется, что Microsoft просто занималась чем-то другим. Даже если корпорация и понимала всю перспективность внедрения зависимостей, я думаю, что для них более приоритетными тогда были другие функциональные возможности, такие, например, как дженерики (Generics).

В июне 2004 появился контейнер `StructureMap`, опередив `Castle Windsor` примерно на полгода.

В конце 2005 года Microsoft выпустила версию .NET 2.0 с поддержкой дженериков — как важнейшей новой возможности, и затем решила сконцентрироваться на WCF, WPF и позднее LINQ, чтобы включить их в следующий крупный релиз (Visual Studio 2008).

В это время внедрение зависимостей постепенно набирало популярность. В 2006 году появился Spring.NET.

До начала 2008 года, когда Microsoft patterns & practices выпустила контейнер Unity, казалось, что основная область исследовательских интересов Microsoft и близко не связана с внедрениями зависимостей. Этот четырехлетний период давал квалифицированным разработчикам хорошую возможность для старта, и в этот период популярность контейнеров внедрения, таких как `StructureMap` и `Castle Windsor`, возрастала.

Внедрение зависимостей — инициатива снизу

Из рис. 3.14 можно сделать интересное наблюдение, касающееся того, насколько быстро участники сообщества разработки .NET проникались идеями внедрения

[1] *Martin Fowler* Inversion of Control Containers and the Dependency Injection pattern, 2004: http://martinfowler.com/articles/injection.html.

зависимостей. Так, согласно домашней странице проекта `Castle Windsor`, его концепция зародилась еще до опубликования статьи Фаулера:

Castle возник из проекта Apache Avalon в середине 2003 года как попытка создать очень простой контейнер инверсии управления.

Веб-страница Castle от 01.08.2008[1]

В течение длительного времени контейнеры внедрения зависимостей для платформы .NET были «инициативой снизу» и многие ведущие участники симпатизировали методологии гибкой (Agile) разработки. Фактически, даже если модульная архитектура приложения обладает некоторыми достоинствами, остается первый и основной вопрос, касающийся *тестопригодности*, который мотивирует программистов разрабатывать и использовать контейнеры внедрения (что справедливо и для меня).

До недавнего времени официальная методология разработки в Microsoft была основана на фреймворке Microsoft Solutions Framework (MSF) 3.0 — каскадная модель, оставлявшая не так много возможностей для применения методов гибкой разработки. К методам гибкой разработки традиционно относят разработку через тестирование. Если говорить кратко, это была принципиально иная методология с идеологической точки зрения.

С течением времени гибкая разработка, TDD и внедрение зависимостей доказали свою эффективность и стали весьма популярными, а Microsoft медленно подошла к поддержке такого стиля программирования. В отличие от ситуации 2004 года, команды разработчиков теперь открыто обсуждают внедрение зависимостей, TDD и другие технологии, относящиеся к гибкой разработке.

Контейнеры внедрения разработки Microsoft

За прошедшее время команда patterns & practices (p&p) из Microsoft разработала большое количество доказательств правильности концепций из различных областей, связанных с .NET. Значительный опыт, полученный из этих проектов, был применен для формализации и определения границ дальнейшей разработки самого фреймворка .NET. Например, проект Updater Application Block дал богатый опыт, который позже использовался при разработке технологии ClickOnce.

В начале 2008 года p&p издала свой первый Community Technical Preview (CTP, «Предварительный технический обзор сообщества») своего нового контейнера внедрения Unity. Версия 1.0 этого контейнера была выпущена в апреле 2008 года. Контейнер Unity представляет собой полноценный контейнер внедрения, поддерживающий композицию объектов, управление жизненным циклом и перехват. Unity не является продуктом Microsoft, скорее это проект с незакрытым исходным кодом (Disclosed Source[2]), который просто был разработан в Microsoft.

---

1 http://www.castleproject.org/castle/history.html.

2 Поскольку исходный код Unity доступен свободно, Microsoft не предоставляет его исправлений. Вы можете просматривать исходный код, но не распространять его. В силу этого я посчитал термин «незакрытый исходный код» (Disclosed Source) наиболее подходящим, так как термин «открытый исходный код» (Open Source) обычно означает, что сотрудничество осуществляется в обе стороны. Однако его лицензия является разрешительной, так что вы можете использовать, модифицировать и распространять исходный код.

КОМПОНОВЩИК ОБЪЕКТОВ

Похоже, что имеет место небольшая путаница, возникшая, когда p&p анонсировала свой контейнер внедрения. Когда p&p в начале 2006 года представила свой проект Enterprise Library for .NET 2.0, он содержал модуль, называвшийся Object Builder (Компоновщик объектов) и использовавшийся для построения сложных объектов из заданных элементов.

Этот модуль управлялся атрибутами и работал лишь для классов, которые тесно интегрировались с самим `Object Builder`. Он никогда не позиционировался как контейнер внедрения, хотя было признано, что возможно построить контейнер внедрения на его основе.

Многие ошибочно полагают, что `Object Builder` был первым разработанным Microsoft контейнером внедрения, но это не так: первым был Unity.

В версию .NET 4.0 Microsoft включила фреймворк Managed Extensibility Framework (MEF), который реализует основные концепции контейнеров внедрения. В своей первой реализации MEF не является полнофункциональным контейнером внедрения, поддерживающим все аспекты внедрения зависимостей, скорее он является механизмом, предназначенным преимущественно для *композиции объектов*.

Команда разработчиков MEF великолепно разбирается в таких аспектах, как управление жизненным циклом и перехват, так что вовсе не выглядит невероятным, что мы не увидим полностью функциональный контейнер MEF в течение нескольких ближайших лет (пока я пишу эти строки, технические обзоры свидетельствуют, что так и происходит на самом деле).

3.5. Резюме

Контейнер внедрения может быть чрезвычайно полезным инструментом, если использовать его грамотно. Самое важное касательно контейнеров — понять, что применение внедрения зависимостей не зависит от использования контейнера внедрения. Приложение может быть построено из многих слабо связанных классов и модулей, и никакие из этих модулей не должны знать ничего о контейнере. Наилучший способ гарантировать, что код приложения ничего не знает о контейнере внедрения, — реализовать шаблон «Регистрация, преобразование, высвобождение» в корне компоновки. Это надежно защитит вас от случайного использования антишаблона «Локатор сервисов», так как такой подход ограничивает использование контейнера маленькой изолированной областью кода.

Используемый таким образом контейнер внедрения становится механизмом, управляющим большей частью инфраструктуры приложения. Он компонует графы объектов, используя их конфигурацию. Это дает особые преимущества, если вы применяете основанное на соглашениях конфигурирование — при правильной реализации контейнер берет на себя управление компоновкой графов объектов, а вы сможете сконцентрироваться на добавлении новых классов, реализующих новые возможности. Контейнер будет автоматически обнаруживать новые классы, соответствующие установленным соглашениям, и делать их доступными для использования.

В некоторых случаях вам может потребоваться более явное управление конфигурацией контейнера. Самым эффективным при этом будет конфигурирование в коде, а в случае, если потребуется возможность динамического связывания, вы можете использовать и XML.

Эта глава завершает часть 1 книги. Целью части 1 было дать представление о внедрении зависимостей. Предыдущие главы содержали общее введение во внедрение зависимостей, а в данной главе дано объяснение, как контейнеры внедрения связаны с собственно внедрениями зависимостей, а также приведены общие принципы построения приложений. Я счел уместным завершить главу историческим обзором контейнеров внедрения в .NET, чтобы дать более полное представление о различных контейнерах.

Данная глава вводит два связанных с контейнерами внедрения мини-шаблона — «Корень компоновки» и «Регистрация, преобразование, высвобождение». В следующей главе мы сконцентрируемся на паттернах проектирования.

ЧАСТЬ 2

Каталог паттернов внедрения зависимостей

В части 1 был дан общий обзор внедрения зависимостей. В ней обсуждались назначение и преимущества ВЗ, а также было разъяснено, как используются контейнеры внедрения зависимостей. Несмотря на то что глава 2 содержала развернутый пример, я уверен, что после прочтения первых глав у вас остались вопросы. В части 2 мы углубимся в тему, чтобы ответить на некоторые из них.

Как понятно из названия части 2, она представляет собой каталог паттернов, антипаттернов и приемов рефакторинга. Некоторые не любят паттерны проектирования, так как они могут показаться формальными или слишком абстрактными. Однако я люблю паттерны, поскольку они играют роль высокоуровневого языка, позволяющего более эффективно и кратко формулировать концепции при обсуждении вопросов проектирования ПО. Мне бы хотелось, чтобы предлагаемый в этой части каталог вы воспринимали как язык паттернов для внедрения зависимостей. Хотя описание паттерна должно содержать некоторые обобщения, я попытаюсь конкретизировать каждый паттерн путем использования примеров.

Вы можете читать все три главы по очереди, но каждый включенный в каталог элемент рассмотрен так, что его описание никак не зависит от других.

Глава 4 содержит мини-каталог паттернов проектирования внедрения зависимостей. В определенном смысле эти шаблоны являются нормативным руководством по способам реализации внедрения зависимостей, но вы должны понимать, что я не оцениваю их как имеющих одинаковую важность. «Внедрение конструктора» является намного более важным паттерном разработки, тогда как все остальные паттерны должны рассматриваться в качестве дополнительных технологий, применяемых в исключительных случаях. Так, например, паттерн «Окружающий контекст» используется настолько редко, что я даже думал не включать его в книгу (оставил его только потому, что рецензенты попросили сделать это).

В то время как глава 4 включает в себя набор обобщенных решений, глава 5 содержит каталог ситуаций, которых нужно избегать. Эти антипаттерны (или *дурно пахнущий код*) описывают распространенные, но неверные способы решить типичные проблемы, возникающие при внедрении зависимостей. Каждый антипаттерн показывает способы выявления и устранения проблем. Важно знать и понимать эти антишаблоны, чтобы избегать связанных с ними ловушек. Глава 5 описывает наиболее важный антипаттерн «Локатор сервисов», являющийся полной противоположностью внедрению зависимостей.

По мере того как вы начнете применять внедрение зависимостей в повседневной практике, вы начнете сталкиваться с определенными проблемами. Я думаю, у каждого из нас возникают моменты сомнений, когда мы полагаем, что понимаем некоторый инструмент или технологию, но в то же время думаем: *«Теоретически это должно работать, но мой случай — особенный...»* Когда такие мысли посещают меня, для меня это является сигналом, что пришла пора изучить какие-то новые материалы.

За время своей карьеры я обнаружил, что некоторые проблемы возникают снова и снова. Каждая из этих проблем имеет распространенное решение, которое вы

можете применить, чтобы приблизить код к одному из паттернов внедрения зависимостей, перечисленных в главе 4. В духе концепции «от *рефакторинга к* шаблонам» я назвал главу 6 «Рефакторинг внедрения зависимостей», так как она содержит каталог проблем и соответствующих решений.

Часть 2 представляет полный каталог паттернов, антипаттернов и методов рефакторинга. Мне кажется, что это самая полезная часть книги и самая обширная. Конечно, вы можете возвращаться к этим главам спустя месяцы или даже годы после того, как прочитаете их.

4 Паттерны внедрения зависимостей

Меню:

- «Внедрение конструктора»;
- «Внедрение свойства»;
- «Внедрение метода»;
- «Окружающий контекст».

Как и у прочих профессионалов, у поваров имеется собственный жаргон, позволяющий им обсуждать сложные вопросы приготовления пищи на языке, малопонятном для непосвященных. Даже если вы знаете, что большинство таких жаргонизмов заимствовано из французского языка, вам это не слишком поможет (если только вы не говорите по-французски).

Соусы — это как раз такая область, в которой жаргона хоть отбавляй. В главе 1 я немного рассказал о беарнском соусе, но я не вдавался в детали его классификации (рис. 4.1).

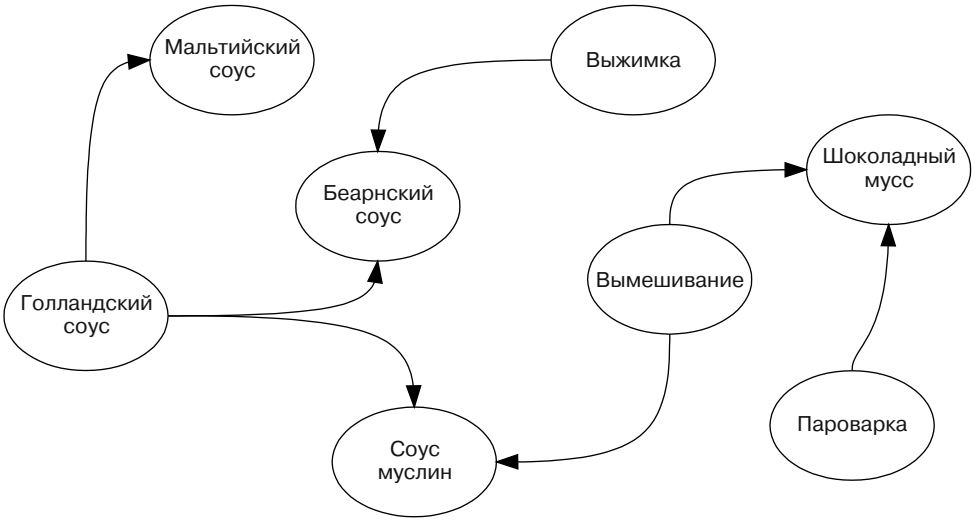


Рис. 4.1. Разные соусы основаны на голландском соусе

Итак, беарнский соус — это голландский соус, в котором лимонный сок заменен на выжимку из уксуса, лука-шалота, кервеля и полыни. Есть и другие соусы, осно-

вой которых является голландский соус — в том числе мой любимый соус муслин, для получения которого нужно *вымешать* взбитые сливки.

Обратили ли вы внимание на использованный жаргон? Вместо того чтобы сказать «осторожно введите взбитые сливки в соус, следя, чтобы он не свернулся», я использовал термин «вымешать». Если вы знаете, что он означает, его использование упрощает как рецепт, так и его понимание.

Термин «вымешать» используется не только в рецептах соусов — это популярный способ соединить что-то взбитое с другими ингредиентами. Например, при приготовлении классического шоколадного мусса я взбиваю взбитые яичные белки со смесью взбитых яичных желтков и растопленного шоколада.

При разработке программного обеспечения тоже используется сложный и недоступный для окружающих жаргон. Хотя вы можете не знать, что в кулинарии означает термин «пароварка», я уверен, что большинство поваров будут совершенно ошарашены, если вы скажете им, что «строки — это неизменяемые классы, представляющие последовательности символов Unicode».

Когда речь заходит о способах структурирования кода для решения тех или иных проблем, мы обращаемся к паттернам проектирования, которые дают названия популярным решениям. Так же как термины «голландский соус» и «вымешать» помогают нам кратко описать способ приготовления соуса муслин, паттерны помогают передать информацию о структуре кода. Система событий в .NET основана на паттерне проектирования, называемом «Наблюдатель» (Observer), а цикл `foreach` использует паттерн «Итератор» (Iterator).

В данной главе я представлю четыре основных паттерна проектирования, перечисленных на рис. 4.2. Поскольку структура главы ориентирована на представление каталога шаблонов, каждый паттерн описан так, чтобы его можно было изучать независимо от прочих. Однако нельзя не указать, что «Внедрение конструктора» является самым важным паттерном из четырех представленных.

Не беспокойтесь, если вы плохо знакомы с основами паттернов проектирования. Главное назначение паттерна — представить подробное и самодостаточное описание конкретного способа достижения цели, своего рода рецепт.

Рис. 4.2. Структура данной главы представляет собой каталог паттернов. Каждый паттерн может изучаться независимо от прочих

Для каждого паттерна я даю краткое описание, пример кода, сравниваю достоинства и недостатки и т. д. Вы можете читать описания паттернов последовательно или остановиться только на интересующих вас. Самым важным паттерном является «Внедрение конструктора», который следует использовать в большинстве случаев; другие паттерны следует применять в специальных случаях.

4.1. «Внедрение конструктора»

Как можно гарантировать, что требуемая зависимость будет всегда доступна разрабатываемому классу?

Это обеспечивается, если все вызывающие классы будут передавать зависимость как параметр конструктора.

Когда класс требует экземпляр зависимости, эта зависимость может быть передана через конструктор класса, который может сохранять ее для последующего (или немедленного) использования (рис. 4.3).

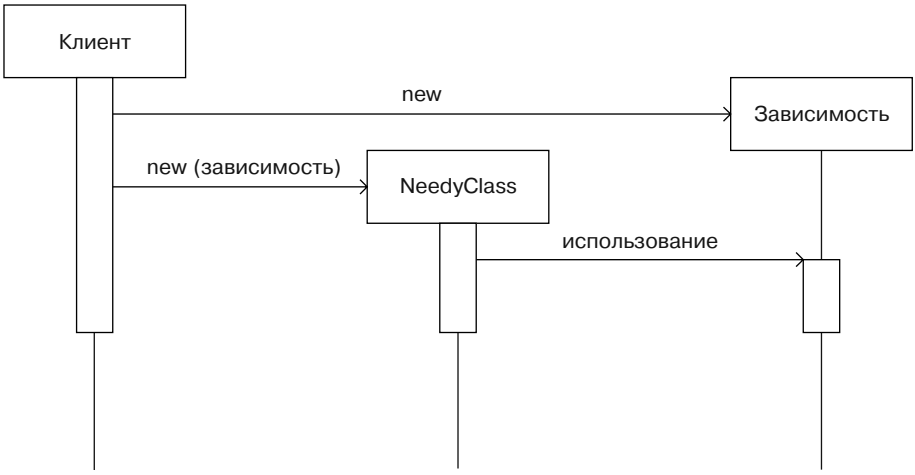


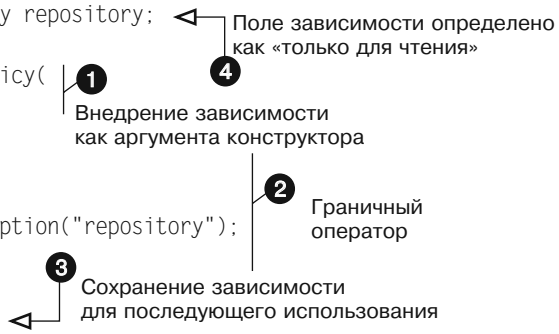
Рис. 4.3. Классу NeedyClass для работы нужен экземпляр Dependency, поэтому требуется, чтобы любой Client передавал экземпляр зависимости через конструктор NeedyClass. Так мы гарантируем, что экземпляр требуемой зависимости будет доступен классу NeedyClass

4.1.1. Как это работает

Класс, требующий зависимость, должен иметь конструктор с модификатором доступа `public` (общедоступный), который получает экземпляр требуемой зависимости в качестве аргумента конструктора (листинг 4.1). В большинстве случаев этот конструктор должен быть единственным в классе. Если необходимо более одной зависимости, могут использоваться дополнительные аргументы конструктора.

Листинг 4.1. Внедрение конструктора

```
private readonly DiscountRepository repository;

public RepositoryBasketDiscountPolicy(
    DiscountRepository repository)
{
    if (repository == null)
    {
        throw new ArgumentNullException("repository");
    }

    this.repository = repository;
}
```



Зависимость (в предыдущем листинге это абстрактный класс `DiscountRepository`) является обязательным аргументом конструктора ❶. Код любого клиента, который не предоставляет экземпляра зависимости, не может компилироваться. Однако поскольку как интерфейс, так и абстрактный класс являются ссылочными типами, вызывающий код может передать в аргумент специальное значение `null`, что делает приложение компилируемым; поэтому в разрабатываемый класс вводится граничный оператор ❷, защищающий класс от такого некорректного использования.

Поскольку совместная работа компилятора и блока защиты гарантирует, что аргумент конструктора является корректным (если не возникла исключительная ситуация (Exception)), конструктор может просто сохранить зависимость для будущего использования, не выясняя детали реальной реализации ❸.

Хорошей практикой является объявлять поле, хранящее значение зависимости, как «только для чтения» (Read-Only). Так мы гарантируем, что выполняется, причем только однажды, логика инициализации в конструкторе: поле не может быть модифицировано ❹. Это не нужно для реализации внедрения зависимостей, но таким образом код защищается от случайных модификаций поля (например, от установки его значения в `null`) в каком-то другом месте кода класса.

СОВЕТ

Не добавляйте в конструктор никакую другую логику. Принцип единственной ответственности подразумевает, что компоненты приложения должны реализовывать только одну функцию, и поскольку конструктор используется для внедрения зависимостей, предпочтительно будет не нагружать его другими задачами.

Считайте внедрение конструктора статическим объявлением зависимостей класса. Сигнатура конструктора компилируется с указанным типом, и ее могут «рассмотреть» все нуждающиеся в этом элементы. Она явно документирует, что требуемые зависимости класса должны быть представлены через конструктор.

Когда конструктор завершает свою работу, новый экземпляр зависимого класса находится в работоспособном состоянии, с корректным экземпляром зависимости, внедренным в него. Поскольку он хранит ссылку на зависимость, она может использоваться в любом из членов класса так часто, как это необходимо. Зависимость более нигде в классе не нужно проверять на `null`, поскольку конструктор гарантирует ее наличие.

4.1.2. Когда должно использоваться внедрение конструктора

Внедрение конструктора должно по умолчанию использоваться с внедрением зависимостей. Оно реализует наиболее популярный сценарий, когда классу *нужны* одна или более зависимостей, а в наличии не имеется подходящих **локальных умолчаний**.

Внедрение конструктора хорошо реализует данный сценарий, так как оно гарантирует существование зависимости. Если зависимый класс совершенно не может выполнять своих функций без наличия зависимости, такая гарантия является очень ценной.

СОВЕТ

При возможности ограничьте класс одним конструктором. Перегруженные конструкторы провоцируют неоднозначности: какой конструктор должно использовать внедрение зависимостей?

Когда в локальной библиотеке может быть найдена хорошая реализация, заданная по умолчанию, не исключено, что внедрение свойств может оказаться лучшим решением — но чаще всего таких ситуаций не возникает. В предыдущих главах я представил много примеров репозиториев, реализованных в виде зависимостей. Имеются хорошие примеры зависимостей, в которых локальная библиотека не содержит подходящей реализации по умолчанию, так как требуемые реализации относятся к специализированным библиотекам доступа к данным.

В табл. 4.1 представлены достоинства и недостатки внедрения конструктора.

Таблица 4.1. Достоинства и недостаток внедрения конструктора

Достоинства	Недостатки
Внедрение гарантировано. Простота реализации	В некоторых фреймворках сложно задействовать внедрение конструктора

Помимо уже рассмотренной выше гарантированности внедрения, паттерн «Внедрение конструктора» характеризуется простотой реализации. Но при работе с ним нужно следовать четырем этапам, перечисленным в листинге 4.1.

Основным недостатком внедрения конструктора является то, что для ее реализации вам потребуется модифицировать используемый фреймворк приложения. Большинство фреймворков предполагает, что классы будут иметь конструктор, заданный по умолчанию. Если такой конструктор отсутствует, то может потребоваться дополнительная помощь для создания экземпляров. В главе 7 я покажу, как обеспечить конструкцию внедрения для самых распространенных фреймворков приложений.

Очевидным недостатком внедрения конструктора является требование немедленной инициализации всего графа зависимости — зачастую уже при запуске приложения. Тем не менее, хотя и кажется, что этот недостаток снижает эффективность системы, на практике он редко становится проблемой. Ведь, в конце концов, даже для сложных графов объектов мы, как правило, говорим о создании дюжины экземпляров новых объектов, а создание экземпляра объекта — это действие, которое фреймворк .NET выполняет чрезвычайно быстро. Любое узкое место, свя-

занное с производительностью, будет возникать в другой части приложения, так что перестаньте беспокоиться об инициализации графов зависимостей.

В очень редких случаях эта проблема может оказаться действительно серьезной, но в главе 8 я опишу параметр жизненного цикла, называемый `Delayed` (отложенный), который вполне подходит для решения этой проблемы. Сейчас же я только отмечу, что (в крайних случаях) может возникнуть потенциальная проблема начальной загрузки.

4.1.3. Известные способы применения

Хотя внедрение конструктора является основной технологией в приложениях, реализующих внедрение зависимостей, оно не очень хорошо представлено в классах, входящих в базовую библиотеку .NET. Такое положение дел возникло преимущественно из-за того, что BCL — это набор библиотек, а не полноценное приложение.

Два примера, демонстрирующих применение внедрения конструктора в BCL, используют классы `System.IO.StreamReader` и `System.IO.StreamWriter`. Оба они получают экземпляр класса `System.IO.Stream` в конструктор. Оба этих класса содержат также несколько перегруженных конструкторов, использующих путь к файлу вместо экземпляра `Stream`, но эти конструкторы введены просто для удобства и они внутри себя все равно создают экземпляр `FileStream`, применяя для этого переданный путь файла. Ниже представлены конструкторы только для `StreamWriter`, так как конструкторы `StreamReader` похожи на них:

```
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` – это абстрактный класс, выступающий в роли той абстракции, с помощью которой выполняют свои задачи `StreamWriter` и `StreamReader`. Вы можете передать любую реализацию класса `Stream` в их конструкторы, и они будут использовать ее. Но если вы попытаетесь передать в конструктор в качестве `Stream` значение `null`, будет генерироваться `ArgumentNullExceptions`.

Несмотря на то что BCL содержит образцы использования внедрения конструктора, всегда полезно рассмотреть конкретный пример. Следующий подраздел содержит полный пример реализации.

4.1.4. Пример: добавление сервиса конвертации валют в покупательскую корзину

Я хочу добавить новую возможность в демоприложение, рассмотренное ранее в главе 2. Это будет возможность конвертации валют. Данный пример рассматривается

на протяжении всей главы, чтобы продемонстрировать в действии разные шаблоны внедрения зависимостей. В итоге домашняя страница будет выглядеть так, как показано на рис. 4.4. В данной версии пользователь может выбрать одну из трех валют, при этом как цены продуктов, так и окончательная сумма к расчету (представленная на странице корзины) будут показаны в этой валюте.

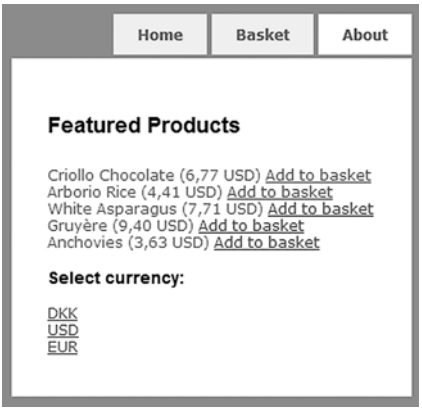


Рис. 4.4. Демоприложение электронной коммерции с подключенной функцией конвертации валют

Одно из первых требований к классу `CurrencyProvider` – наличие зависимости, обеспечивающей приложение информацией о запрошенной валюте. Это реализуется следующим образом:

```
public abstract class CurrencyProvider
{
    public abstract Currency GetCurrency(string currencyCode);
}
```

Класс `Currency` — это еще один абстрактный класс, предоставляющий курсы конвертации между текущей валютой и прочими валютами:

```
public abstract class Currency
{
    public abstract string Code { get; }

    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

Возможность конвертации валют должна присутствовать на всех страницах, отображающих цены — как в `HomeController`, так и в `BasketController`. Поскольку две эти реализации очень похожи, я рассмотрю только `BasketController`.

`CurrencyProvider` чаще всего представляет не участвующие в рассматриваемом процессе ресурсы, такие как веб-сервисы или базы данных. Эти компоненты непосредственно предоставляют курсы конвертации. Это значит, что конкретный поставщик `CurrencyProvider` лучше всего реализовывать в виде отдельного проекта (такого, на-

пример, как библиотека доступа к данным). Следовательно, в таком решении не будет приемлемого локального умолчания. Однако класс BasketController требует, чтобы CurrencyProvider имелся в наличии. Внедрение конструктора является удачным решением данной проблемы. Листинг 4.2 показывает, как зависимость CurrencyProvider внедряется в BasketController.

Листинг 4.2. Внедрение CurrencyProvider в BasketController

```
private readonly IBasketService basketService;
private readonly CurrencyProvider currencyProvider;

public BasketController(IBasketService basketService,
    CurrencyProvider currencyProvider)
{
    if (basketService == null)
    {
        throw new
            ArgumentNullException("basketService");
    }
    if (currencyProvider == null)
    {
        throw new
            ArgumentNullException("currencyProvider")
    }

    this.basketService = basketService;
    this.currencyProvider = currencyProvider;
}
```

4 Поля зависимостей отмечены как «только для чтения»

1 Зависимости внедряются как аргументы конструктора

2 Блок защиты

3 Сохранение зависимостей для последующего использования

Поскольку класс BasketController уже содержит зависимость, определяемую интерфейсом IBasketService, новая зависимость CurrencyProvider добавляется как второй аргумент конструктора 1, после чего следует та же последовательность кода, что и в листинге 4.1. Граничные операторы гарантируют, что в качестве зависимостей не передан null 2, после чего зависимости сохраняются как неизменяемые значения 4 для последующего использования 3.

Теперь, поскольку гарантируется наличие CurrencyProvider в BasketController, этот контроллер может использоваться в коде класса — например, в методе Index:

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);
    // …
}
```

Мы пока не рассматривали CurrencyProfileService, и сейчас скажу только, что он предоставляет код предпочтительной валюты для текущего пользователя. В подразделе 4.2.4 CurrencyProfileService обсуждается в деталях.

При наличии кода валюты `CurrencyProvider` может быть вызван, чтобы вернуть объект типа `Currency`, соответствующий этому коду. Следует отметить, что вы можете использовать в коде класса поле `currencyProvider` без дополнительных проверок, так как гарантируется, что оно содержит корректное значение.

После того как будет получен объект `Currency`, можно будет выполнить оставшуюся работу в коде метода `Index`; помните, что я еще не показывал его реализацию. Далее в этой главе я построю этот метод и при этом добавлю дополнительную функциональность, связанную с конвертацией валют.

4.1.5. Родственные паттерны

«Внедрение конструктора» — это наиболее часто используемый паттерн внедрения зависимостей, а также простейший способ обеспечить корректную реализацию. Он применяется, если требуется зависимость.

Если зависимость должна быть опциональной, внедрение конструктора может быть заменено на внедрение свойства. При этом в наличии должно быть подходящее локальное умолчание.

Когда зависимость представляет собой сквозную функциональность приложения, которая может быть потенциально доступна любому модулю приложения, вместо внедрения конструктора может использоваться окружающий контекст.

Следующий паттерн, обсуждаемый в этой главе, — это «Внедрение свойства», тесно связанное с внедрением конструктора. Единственный параметр, по которому требуется принять решение, — будет зависимость опциональной или нет.

4.2. «Внедрение свойства»

Как можно разрешить внедрение зависимостей как опцию в классе, если имеется подходящее локальное умолчание?

Использованием записываемого свойства, что позволяет вызывающей стороне устанавливать его значение, если она хочет заменить поведение, применяемое по умолчанию.

Когда класс имеет подходящее локальное умолчание, но требуется оставить возможность замены его для обеспечения расширяемости, можно представить записываемое свойство, что позволяет клиенту передать другую реализацию зависимости класса, заменив таким образом реализацию, действовавшую по умолчанию.

ПРИМЕЧАНИЕ

«Внедрение свойства» известно также как «Внедрение установщика» (Setter Injection).

Как показано на рис. 4.5, клиенты, желающие задействовать `SomeClass` «как есть», могут просто создать и затем использовать новый экземпляр класса, тогда как клиенты, желающие изменить поведение класса, могут сделать это, присвоив свойству `Dependency` другую реализацию `ISomeInterface`.

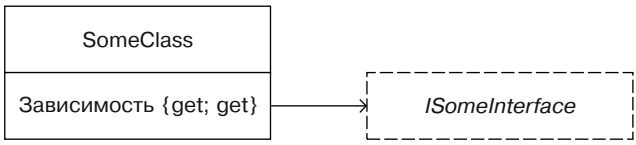


Рис. 4.5. SomeClass включает опциональную зависимость типа ISomeInterface. Вызывающая сторона может и не предоставлять экземпляр зависимости. Этой стороне предоставляется выбор — устанавливать или нет эту зависимость через свойство

4.2.1. Как это работает

Класс, использующий зависимость, должен иметь записываемое свойство с модификатором `public`: тип этого свойства должен соответствовать типу зависимости. В базовой реализации это можно сделать с помощью очень простого кода, похожего на представленный в листинге 4.3.

Листинг 4.3. Внедрение свойства

```
public partial class SomeClass
{
    public ISomeInterface Dependency { get; set; }
}
```

Здесь `SomeClass` зависит от `ISomeInterface`. Клиенты могут передавать реализации интерфейса `ISomeInterface` через свойство `Dependency`. Обратите внимание, что, в противоположность внедрению конструктора, вы не можете отметить поле свойства `Dependency` как «только для чтения», так как вызывающей стороне позволено изменять значение этого свойства в любой момент жизненного цикла класса `SomeClass`.

Прочие члены зависимого класса могут использовать инжектированную зависимость для выполнения своих функций, например:

```
public string DoSomething(string message)
{
    return this.Dependency.DoStuff(message);
}
```

Однако такая реализация является ненадежной, поскольку свойство `Dependency` не гарантирует возврата экземпляра `ISomeInterface`. Например, код, показанный ниже, будет генерировать `NullReferenceException`, так как значение свойства `Dependency` есть `null`:

```
var mc = new SomeClass();
mc.DoSomething("Ploeh");
```

Данная проблема может быть устранена установкой в конструкторе экземпляра зависимости по умолчанию для свойства, скомбинированной с добавлением блока защиты в метод — установщик свойства.

Еще одна трудность возникает, если клиентам будет позволено менять значение зависимости в течение жизненного цикла класса. Для устранения этой проблемы

можно добавить внутренний флаг, который позволит клиенту установить значение зависимости лишь однажды[1].

Пример в подразделе 4.2.4 показывает, как можно избавиться от этих трудностей, но прежде чем мы перейдем к примеру, я расскажу, когда имеет смысл применять внедрение свойства.

4.2.2. Когда следует применять внедрение свойства

Внедрение свойства следует применять только в случае, когда для разрабатываемого класса имеется подходящее локальное умолчание, но при этом вы хотели бы оставить вызывающей стороне возможность использовать другую реализацию типа зависимости.

Внедрение свойства лучше всего применять, если зависимость опциональна.

ПРИМЕЧАНИЕ

Спорным является вопрос, означает ли внедрение свойства наличие опциональной зависимости. В соответствии с основным принципом разработки API, следует считать свойства являются опциональными. Ведь легко забыть присвоить им значение, и компилятор никак не отреагирует на это. Если вы следуете этому принципу в общем, вы должны принять его и относительно внедрения зависимостей.

ЛОКАЛЬНОЕ УМОЛЧАНИЕ

Когда вы разрабатываете класс, имеющий зависимость, вы, вероятно, ориентируетесь на некую конкретную реализацию этой зависимости. Если вы пишете сервис домена, который имеет доступ к некоторому репозиторию, вы, вероятнее всего, планируете разработать реализацию этого репозитория, использующую реляционную базу данных.

Может показаться заманчивым задать эту реализацию по умолчанию для данного класса во время разработки. Однако если такое заблаговременное умолчание реализуется в другой сборке (Assembly), использование ее таким способом неизбежно вызовет создание неизменяемой ссылки на нее, что сведет на нет многие преимущества слабого связывания, описанного в главе 1.

И наоборот, если предполагаемая к использованию в качестве умолчания реализация определяется в той же библиотеке, что и применяющий ее класс, этой проблемы не возникает. Такой подход вряд ли будет хорош в случае работы с репозиторием, но такие локальные умолчания оказываются весьма полезными в качестве стратегий.

Пример в данном подразделе содержит пример локального умолчания.

В главе 1 обсуждалось много веских причин для написания слабо связанного кода и изоляции модулей друг от друга. Но слабое связывание может также успешно применяться с классами, находящимися в одном модуле. Это часто достигается

[1] Эрик Липперт называет это явление «эскимошной неизменяемостью» (Popsicle Immutability) — *Eric Lippert* Immutability in C# Part One: Kinds of Immutability, 2007: http://blogs.msdn.com/ericlippert/archive/2007/11/13/immutability-in-c-part-one-kinds-of-immutability.aspx.

путем создания абстракций в одном модуле и организацией взаимодействия классов через эти абстракции, а не через неизменяемые сильные связи друг с другом.

Рисунок 4.6 показывает, что абстракции могут быть определены, реализованы и использованы в одном и том же модуле. Это делается в основном для повышения степени расширяемости классов.

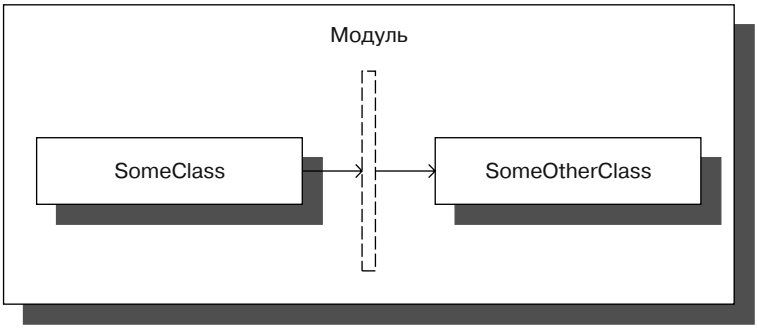


Рис. 4.6. Даже в отдельном модуле можно ввести абстракции (представлены вертикальным прямоугольником), помогающими понизить степень связанности класса в рамках данного модуля

ПРИМЕЧАНИЕ

Концепция повышения степени расширяемости класса берет начало в принципе открытости/закрытости[1], который в общем сводится к тому, что класс должен быть открыт для расширяемости, но закрыт для модификаций.

Когда классы реализуются на основе принципа открытости/закрытости, можно ориентироваться на использование локального умолчания, но при этом сохраняется возможность расширения класса клиентами путем замены локального умолчания на что-либо еще.

ПРИМЕЧАНИЕ

Внедрение свойств — это лишь один из многих возможных способов использования принципа открытости/закрытости.

СОВЕТ

Иногда вы хотите только повысить расширяемость приложения, но оставляете локальное умолчание как заглушку. В таком случае вы можете использовать паттерн «Нулевой объект» (Null Object) для реализации локального умолчания.

Иногда бывает целесообразно сохранить возможность работы с локальным умолчанием, но при этом иметь возможность для *добавления* других реализаций. Такого результата можно достичь, используя для моделирования зависимости либо паттерн «Наблюдатель» (Observer), либо паттерн «Составной» (Composite).

[1] Хорошее ориентированное на .NET введение в принцип открытости/закрытости содержится в статье *Jeremy Miller* Patterns in Practice: The Open Closed Principle.— MSDN Magazine, June 2008. Онлайн-версию статьи можно прочитать по адресу http://msdn.microsoft.com/en-us/magazine/cc546578.aspx.

Итак, к настоящему моменту я не привел никаких примеров внедрения свойств, поскольку этот паттерн применяется достаточно редко.

В табл. 4.2 представлены достоинства и недостатки внедрения свойств.

Таблица 4.2. Достоинства и недостатки внедрения свойств

Достоинства	Недостатки
Легкость для понимания	Ограниченная применимость. Сложности при обеспечении надежности

Основное достоинство внедрения свойств — легкость понимания ее концепции. Я часто встречаю у разработчиков именно этот паттерн при первой попытке реализации внедрения зависимостей.

В данном случае внешность обманчива и внедрение свойства оказывается довольно проблемным. Его сложно реализовать надежным образом. Клиенты могу «забыть» (или не захотеть) задать значение зависимости или могут ошибочно передать `null` в качестве значения. Кроме того, что должно произойти, если клиент попытается изменить значение зависимости в течение жизненного цикла класса? Следствием этого может быть противоречивое или неожиданное поведение класса, поэтому лучше защититься от такого поворота событий.

Используя внедрение конструктора, вы можете защитить класс от таких инцидентов, создав поле, хранящее значение зависимости, и определив его с ключевым словом `readonly`. Но такой прием невозможен, когда вы работаете с зависимостью через изменяемое (Writable) свойство. Во многих случаях внедрение конструктора является намного более простым и надежным способом, но существуют ситуации, когда вполне может быть применимо и внедрение свойства. Так, его можно использовать, когда передача значения зависимости не требуется, поскольку применяется локальное умолчание.

Существование подходящего локального умолчания частично зависит от детализации модулей. Базовая библиотека классов .NET довольно велика; и пока значение по умолчанию находится внутри BCL, оно может рассматриваться как локальное. В следующем подразделе я кратко затрону эту тему.

4.2.3. Известные способы применения

В .NET BCL внедрение свойства используется чаще, чем внедрение конструктора — возможно, в силу того, что подходящие локальные умолчания определены во многих местах.

Интерфейс System.ComponentModel.IComponent имеет изменяемое свойство `Site`, позволяющее определить экземпляр типа `ISite`. Это главным образом используется в сценариях времени разработки (например, в Visual Studio) для изменения или совершенствования компонента, когда он находится в редакторе.

Другой пример, в большей степени показывающий, как мы должны представлять себе внедрение зависимостей, находится в Windows Workflow Foundation (WF). Класс `WorkflowRuntime` предоставляет возможность добавлять, получать и удалять сервисы. Это не является в полном смысле внедрением свойств, поскольк-

ку API указанного класса позволяет добавлять ноль или множество не имеющих типа сервисов, используя один и тот же API общего назначения:

```
public void AddService(object service)
public T GetService<T>()
public object GetService(Type serviceType)
public void RemoveService(object service)
```

И хотя метод AddService будет порождать ArgumentNullException, если передаваемое значение сервиса есть null, не существует гарантии того, что вы получите сервис заданного типа, поскольку он может быть никогда не добавлен в используемый экземпляр WorkflowRuntime (фактически это происходит из-за того, что метод GetService является локатором сервисов).

С другой стороны, WorkflowRuntime поступает со множеством локальных умолчаний, определенных для каждого сервиса, которые могут понадобиться, и эти умолчания даже именуются с использованием префикса Default, например DefaultWorkflowSchedulerService и DefaultWorkflowLoaderService. Если, к примеру, альтернативный экземпляр класса WorkflowSchedulerService не добавляется ни через метод AddService, ни через конфигурационный файл приложения, то используется класс DefaultWorkflowSchedulerService.

После рассмотрения этих кратких примеров, взятых из библиотеки BCL, можно приступить к изучению более серьезного примера реализации и использования внедрения свойств.

4.2.4. Пример: реализация сервиса профиля валюты для BasketController

В подразделе 4.1.4 я начал добавлять функциональность конвертации валют в демонстрационное приложение и коротко описал некоторые детали реализации метода Index класса BasketController – но скрыл то, что относилось к CurrencyProfileService. Вот описание этого сервиса.

Приложение должно знать, в какой валюте пользователь хочет видеть цены и стоимость заказа. Если вы вернетесь к рис. 4.4, то увидите ссылки на некоторые валюты внизу экрана. Когда пользователь щелкает на одной из этих ссылок, приложение должно где-то сохранить выбранную валюту и связать этот выбор с пользователем. Сервис CurrencyProfileService обеспечивает сохранение и получение выбранной пользователем валюты:

```
public abstract class CurrencyProfileService
{
    public abstract string GetCurrencyCode();

    public abstract void UpdateCurrencyCode(string currencyCode);
}
```

Этот код представляет собой абстракцию, определяющую действия по применению и возврату выбранной текущим пользователем валюты.

В ASP.NET MVC (как и в ASP.NET в целом) в вашем распоряжении имеется популярный компонент, реализующий описанный сценарий: сервис `Profile`. Великолепно реализованное локальное умолчание `CurrencyProfileService` является одной из многочисленных оберток сервиса ASP.NET Profile, предоставляющее требуемую функциональность, которая определена в методах `GetCurrencyCode` и `UpdateCurrencyCode`. `BasketController` будет использовать `DefaultCurrencyProfileService` как умолчание, и в то же время в нем определяется свойство, которое позволит вызывающей стороне заменить его на что-то другое (листинг 4.4).

Листинг 4.4. Определение свойства CurrencyProfileService

```
private CurrencyProfileService currencyProfileService;

public CurrencyProfileService CurrencyProfileService
{
    get
    {
        if (this.currencyProfileService == null)
        {
            this.CurrencyProfileService =
                new DefaultCurrencyProfileService(
                    this.HttpContext);
        }
        return this.currencyProfileService;
    }
    set
    {
        if (value == null)
        {
            throw new ArgumentNullException("value");
        }
        if (this.currencyProfileService != null)
        {
            throw new InvalidOperationException();
        }
        this.currencyProfileService = value;
    }
}
```

1 Отложенная инициализация локального умолчания

2 Допускается лишь однократное определение зависимости

`DefaultCurrencyProfileService` сам по себе использует внедрение конструктора, так как ему требуется доступ к `HttpContext`, и поскольку `HttpContext` недоступен для `BasketController` во время разработки, создание `DefaultCurrencyProfileService` может быть отложено до момента, пока свойство не будет востребовано в первый раз. В таком случае требуется отложенная инициализация (1), но в других случаях локальное умолчание должно быть назначено в конструкторе. Обратите внимание, что локальное умолчание назначается через сеттер с модификатором `public`, что обеспечивает выполнение всех защитных блоков.

Первый блок защиты гарантирует, что зависимость не есть `null`. Следующий защитный блок (2) отвечает за то, чтобы зависимость была установлена только

один раз. В данном приложении я предпочитаю, чтобы CurrencyProfileService не мог быть изменен после того, как его значение будет установлено, поскольку в противном случае поведение сервиса станет непредсказуемым, ведь выбранная пользователем валюта будет сначала сохраняться в одном CurrencyProfileService, а выбираться затем из другого. И это, вероятнее всего, приведет к возврату другого, отличного от первоначально сохраняемого, значения.

Вы можете также заметить, что, поскольку для отложенной инициализации (1) используется сеттер, зависимость будет блокирована после того, как свойство будет прочитано. Опять же, это сделано для защиты клиентов от ситуаций, когда зависимость позднее изменяется без каких-либо извещений.

Если все блоки защиты пройдены успешно, экземпляр зависимости может быть сохранен для последующего использования.

По сравнению с внедрением конструктора, объем написанного кода намного больше. Внедрение свойства может показаться простым, если рассматривать его в виде блок-схемы, как показано в листинге 4.3. Но на практике она оказывается намного более сложной. В данном случае я даже проигнорировал проблему безопасности потоков.

Когда CurrencyProfileService готов к использованию, начало метода Index класса BasketController теперь может использовать его для получения предпочтительной валюты пользователя:

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);
    // …
}
```

Это тот же фрагмент кода, что и показанный в подразделе 4.1.4. CurrencyProfileService используется для возврата выбранной пользователем валюты, а CurrencyProvider применяется для получения самой этой валюты.

В подразделе 4.3.4 я вернусь к методу Index, чтобы показать, что случится далее.

4.2.5. Родственные паттерны

Внедрение свойства используется, когда зависимость является опциональной, так как у вас имеется подходящее локальное умолчание. Если же локального умолчания нет, следует применять не внедрение свойства, а внедрение конструктора.

Когда зависимость представляет сквозные функции приложения, которые должны быть доступны во всех модулях программы, она может быть реализована как окружающий контекст.

Но прежде чем мы перейдем к обсуждению этой темы, рассмотрим описанное в следующем разделе внедрение конструктора, которое рассматривается под несколько иным углом. Дело в том, что оно обычно применяется в ситуации, когда

уже имеется зависимость, которую мы хотим передать нескольким вызываемым нами взаимодействующим частям кода.

4.3. Внедрение метода

Как можно внедрить зависимости в класс, если они различны для каждой операции?

Путем передачи в качестве параметра метода.

Если при каждом вызове метода используется другая зависимость, вы можете передать ее (зависимость) через параметр метода (рис. 4.7).

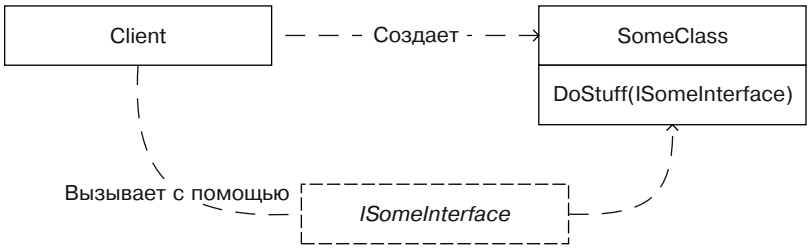


Рис. 4.7. Client создает экземпляр класса SomeClass, но сначала внедряет экземпляр зависимости ISomeInterface в каждом вызове метода

4.3.1. Как это работает

Вызывающая сторона передает зависимость как параметр метода при каждом его вызове. Эта процедура не сложнее, чем сигнатура представленного ниже метода:

```
public void DoStuff(ISomeInterface dependency)
```

Часто зависимость будет представлять некоторый вид контекста для операции, передаваемого как соответствующее значение:

```
public string DoStuff(SomeValue value, ISomeContext context)
```

В данном примере параметр `value` представляет значение, с которым должен работать метод, тогда как параметр `context` содержит информацию о текущем контексте операции. Вызывающая сторона передает зависимость в метод, а тот использует или игнорирует переданную зависимость.

Если сервис использует зависимость, в нем прежде всего должна выполняться проверка на `null`-ссылки, как показано в листинге 4.5.

Листинг 4.5. Проверка параметра метода на null перед его использованием

```
public string DoStuff(SomeValue value, ISomeContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException("context");
```

```
    }
    return context.Name;
}
```

Граничный оператор гарантирует, что контекст определен и доступен остальному коду в теле метода. В данном примере метод использует имя контекста для возврата значения, поэтому наличие объекта контекста является принципиально важным.

Если метод не применяет переданную зависимость, он может не содержать граничный оператор. Такая ситуация представляется немного странной, так как если параметр не задействуется, то зачем он нужен вообще? Однако может потребоваться сохранить параметр в сигнатуре метода, если метод является частью реализации интерфейса.

4.3.2. Когда следует использовать внедрение метода

Внедрение метода лучше всего использовать, когда при каждом вызове методу задается другая зависимость. Это может происходить в случае, когда зависимость сама по себе представляет некоторое значение. Кроме того, такой прием часто встречается, когда вызывающая сторона передает вызываемому методу информацию о контексте, в котором должна выполняться требуемая операция.

Это часто происходит в сценариях расширений, когда расширению передается информация о контексте времени исполнения через параметр метода. В таких случаях расширение должно реализовывать интерфейс, который определяет метод или методы, использующие внедрение.

Рассмотрим интерфейс расширения, имеющий следующую структуру:

```
public interface IAddIn
{
    string DoStuff(SomeValue value, ISomeContext context);
}
```

Любой класс, реализующий этот интерфейс, может использоваться в качестве такого расширения. Некоторые классы могут игнорировать переданный контекст, другие же применяют его. Клиент может использовать список расширений, вызывая каждое из них и передавая им обрабатываемое значение и контекст, чтобы получить агрегированный результат. Пример такого подхода дан в листинге 4.6.

Листинг 4.6. Пример клиента расширения

```
public SomeValue DoStuff(SomeValue value)
{
    if (value == null)
    {
        throw new ArgumentNullException("value");
    }

    var returnValue = new SomeValue();
```

```
        returnValue.Message = value.Message;

        foreach (var addIn in this.addIns)
        {
            returnValue.Message =
                addIn.DoStuff(returnValue, this.context);
        }

        return returnValue;
    }
```

❶ Передать контекст в расширение

Закрытое (Private) поле `addIns` содержит список экземпляров `IAddIn`, что позволяет клиенту обработать этот список в цикле, вызывая для каждого расширения метод `DoStuff`. Каждый раз при вызове метода `DoStuff` контекст операции, представленный полем `context`, передается как параметр метода ❶.

ПРИМЕЧАНИЕ

Внедрение метода тесно связано с использованием абстрактных фабрик (Abstract Factories), описанных в разделе 6.1. Любая абстрактная фабрика, получающая на входе абстракцию (то есть интерфейс или абстрактный класс. — *Примеч. пер.*), может рассматриваться как одна из форм внедрения метода.

Иногда обрабатываемое значение и контекст операции объединяются в одну абстракцию, работающую как их комбинация.

В табл. 4.3 показаны достоинство и недостаток внедрения метода.

Таблица 4.3. Достоинство и недостатки внедрения метода

Достоинство	Недостатки
Позволяет вызывающей стороне передать контекст, специфичный для конкретной операции	Ограниченная применимость

Внедрение метода отличается от других типов внедрения зависимостей тем, что внедрение не происходит в корне композиции, а скорее производится динамически во время вызова. Это позволяет вызывающей стороне передать специфический для операции контекст, что является распространенным механизмом расширения, используемым в .NET BCL.

4.3.3. Известные способы применения

Библиотека .NET BCL содержит много примеров внедрения метода, особенно в пространстве имен `System.ComponentModel`.

Интерфейс System.ComponentModel.Design.IDesigner используется для реализации пользовательской функциональности времени исполнения для компонентов. В нем определен метод `Initialize`, получающий экземпляр типа `IComponent`, поэтому реализация точно знает, какому компоненту она в данный момент оказывает помощь в создании. Дизайнеры (Designers) — это реализации, созданные `IDesignerHost`, который также получает экземпляр `IComponent` как параметр для создания дизайнеров:

```
IDesigner GetDesigner(IComponent component);
```

Это хороший пример сценария, в котором параметр сам по себе несет информацию: компонент несет информацию о том, какой IDesigner должен быть создан, но в то же время он является и компонентом, с которым будет работать созданный дизайнер.

Еще один пример из пространства имен System.ComponentModel дает класс TypeConverter. Некоторые из его методов получают экземпляр типа ITypeDescriptorContext, который, как понятно из названия, содержит информацию о контексте текущей операции. Поскольку таких методов существует много, я не стану перечислять их все, а приведу лишь один наглядный пример:

```
public virtual object ConvertTo(ITypeDescriptorContext context,
    CultureInfo culture, object value, Type destinationType)
```

В этом методе контекст операции явно представлен параметром context, тогда как значение для конвертации и соответствующий тип заданы отдельными параметрами. Те, кто будет реализовывать этот метод, могут использовать либо игнорировать параметр context по своему усмотрению.

Фреймворк ASP.NET MVC содержит также и несколько примеров собственно внедрения метода. Интерфейс IModelBinder может использоваться для преобразования данных, переданных через механизмы HTTP GET или POST в сильно типизированные объекты. Его единственный метод:

```
object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext);
```

В методе BindModel параметр controllerContext содержит информацию о контексте операции (среди прочих данных HttpContext), тогда как параметр bindingContext имеет информацию о значениях, полученных от браузера.

Когда я рекомендовал, чтобы основным используемым вами паттерном был бы «Внедрение конструктора», я подразумевал, что вы в основном разрабатываете приложения, базируясь на том или ином фреймворке. С другой стороны, если вы разрабатываете фреймворк, «Внедрение метода» может часто оказаться полезным, поскольку оно обеспечивает передачу информации о контексте расширениям фреймворка. Это основная причина такого частого использования внедрения метода в BCL.

4.3.4. Пример: конвертация валют в корзине

В предыдущих примерах мы рассматривали, как BasketController в демонстрационном приложении получал предпочтительную валюту пользователя (см. подразделы 4.1.4 и 4.2.4). Сейчас я завершу пример конвертации валют, приведя представленные в Basket цены к выбранной пользователем валюте.

Currency – это абстракция, моделирующая валюту (листинг 4.7).

Листинг 4.7. Абстрактный класс Currency

```
public abstract class Currency
{
```

```
    public abstract string Code { get; }

    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

Свойство `Code` возвращает (`get`) код валюты для данного экземпляра `Currency`. Предполагается, что в качестве кодов валют используются международные коды. Например, для датской кроны применяется код валюты `DKK`, а для долларов США — код `USD`.

Метод `GetExchangeRateFor` возвращает курс конвертации между экземпляром `Currency` и любой другой валютой. Заметьте, что этот метод — абстрактный. Это означает, что я работаю без учета конкретного способа вычисления курса конвертации, оставляя это полностью на усмотрение того, кто будет реализовывать данный метод.

Далее мы рассмотрим, как экземпляры `Currency` используются для конвертации цен и как данная абстракция может быть реализована и включена в приложение. Так мы сможем конвертировать цены в такие валюты, как доллары США или евро.

Внедрение валюты

Мы будем использовать абстракцию `Currency` как зависимость с информационной нагрузкой, необходимую, чтобы выполнить валютные конвертации в корзинах. Поэтому в класс `Basket` нужно добавить метод `ConvertTo`:

```
public Basket ConvertTo(Currency currency)
```

Этот метод будет перебирать все элементы в корзине и преобразовывать их рассчитанные цены в заданную валюту, возвращая новый экземпляр класса `Basket` с конвертированными ценами элементов заказа. Используя цепочку вызовов методов, реализация в итоге вернет класс `Money`, как показано в листинге 4.8.

Листинг 4.8. Конвертация Money в другую валюту

```
public Money ConvertTo(Currency currency)
{
    if (currency == null)
    {
        throw new ArgumentNullException("currency");
    }
    var exchangeRate =
        currency.GetExchangeRateFor(this.CurrencyCode);
    return new Money(this.Amount * exchangeRate,
        currency.Code);
}
```

Внедрение валюты как параметр метода (1)

`Currency` внедряет в метод `ConvertTo` через параметр `currency` (1) и проверяется в граничном операторе, который гарантирует, что экземпляр валюты всегда доступен в остальном коде метода.

Курс конвертации для текущей валюты (представленной через `CurrencyCode`) получается из текущей валюты и используется для вычисления возвращаемого нового экземпляра `Money`.

Имея реализации метода ConvertTo, вы можете окончательно дописать код метода Index в BasketController, как показано в листинге 4.9.

Листинг 4.9. Конвертация валюты корзины

```
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);

    var basket = this.basketService
        .GetBasketFor(this.User)
        .ConvertTo(currency);
    if (basket.Contents.Count == 0)
    {
        return this.View("Empty");
    }

    var vm = new BasketViewModel(basket);
    return this.View(vm);
}
```

1 Конвертация корзины в выбранную валюту

BasketController использует экземпляр IBasketService для получения корзины пользователя Basket. В главе 2 говорилось, что зависимость IBasketService передается в BasketController внедрением конструктора. После того как будет получен экземпляр Basket, его можно конвертировать в заданную валюту, используя метод ConvertTo, которому передается экземпляр currency (1).

В данном случае мы использовали внедрение метода, поскольку абстракция Currency несет в себе информацию, но в зависимости от контекста может отличаться (по выбору пользователя). Вы можете реализовать тип Currency как конкретный класс, но это ограничит ваши возможности при определении способов вычисления курсов конвертации.

После того как мы рассмотрели способы использования класса Currency, можно разобраться, как он может быть реализован.

Реализация класса Currency

До сих пор я не обсуждал способы реализации класса Currency, так как это совершенно непринципиально с точки зрения внедрения метода. Как говорилось в подразделе 4.1.4 и как показано в листинге 4.9, экземпляр Currency создается экземпляром CurrencyProvider, который был внедрен в класс BasketController путем внедрения конструктора.

Чтобы не усложнять пример, я покажу, как могут быть реализованы CurrencyProvider и Currency, если использовать сервер баз данных SQLServer и технологию LINQ to Entities. Такой подход предполагает, что база данных содержит таблицу с курсами конвертации, данные в которую были занесены каким-то внешним механизмом.

Другим способом получения курсов конвертации может быть использование независимого внешнего веб-сервиса.

Реализация CurrencyProvider передает соединительную строку реализации Currency, которая использует эту информацию для создания ObjectContext. В основе всего лежит реализация метода GetExchangeRateFor, показанного в листинге 4.10.

Листинг 4.10. Основанная на SQL Server реализация класса Currency

```
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, — получить курсы конвертации из базы данных. Таблица содержит курсы, определенные относительно одной базовой валюты (DKK), поэтому вам нужны два курса для корректной конвертации между двумя произвольными валютами. Выбранные валюты будут проиндексированы по коду, так что их можно будет легко использовать на заключительном шаге вычислений.

Такая реализация потенциально должна активно обращаться к базе данных. Метод ConvertTo класса Basket в итоге вызывает данный метод в цикле, и обращения к базе данных при каждом его вызове снижают производительность. В следующем разделе я вернусь к обсуждению этой проблемы.

4.3.5. Родственные паттерны

В отличие от других паттернов внедрения зависимостей, рассматриваемых в этой главе, внедрение метода используется главным образом в тех случаях, когда у нас уже имеется экземпляр зависимости, которую мы хотели бы передать взаимодействующим программным компонентам. Но при этом нам во время разработки не известен конкретный тип этих взаимодействующих компонентов (что, например, имеет место при применении расширений (Add-Ins)).

При использовании внедрения метода мы как бы находимся «по другую сторону рампы» от прочих шаблонов внедрения зависимостей: мы не потребляем зависимость, скорее предоставляем ее. Типы, которым мы предоставляем зависимость, не могут выбирать, ни как представлять внедрение зависимостей, ни нужна ли она им вообще. Они только могут использовать или игнорировать ее в зависимости от своих потребностей.

4.4. «Окружающий контекст»

Как можем мы сделать зависимость доступной в каждом модуле без включения в каждый API компонента сквозных аспектов приложения?

Сделав ее доступной через статический доступ.

По-настоящему универсальная сквозная функциональность приложения потенциально может загрязнить значительную часть API-приложения, если вы будете передавать свой экземпляр объекта каждому соисполнителю. Альтернативой является определение контекста, доступного каждому элементу, который нуждается в нем. Такой контекст может быть проигнорирован всеми остальными (рис. 4.8).

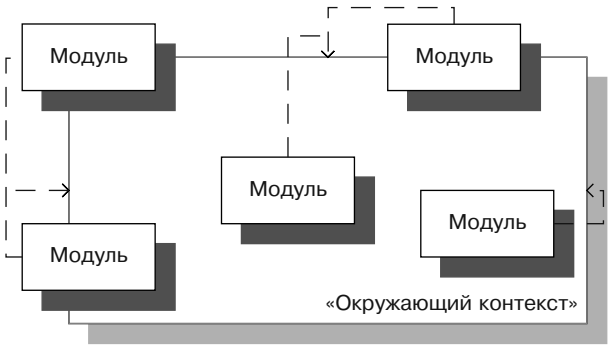


Рис. 4.8. Каждый модуль имеет доступ к окружающему контенту, если ему требуется такой доступ

4.4.1. Как это работает

«Окружающий контекст» доступен любому потребителю через статическое свойство или метод. Потребляющий класс может использовать его следующим образом:

```
public string GetMessage()
{
    return SomeContext.Current.SomeValue;
}
```

В данном случае контекст имеет статическое свойство `Current`, доступное пользователю. Это поле может быть по-настоящему статическим или может быть ассоциировано с выполняющимся в данный момент времени потоком.

Чтобы быть полезным в сценариях внедрения зависимостей, контекст сам по себе должен являться абстракцией. При этом должна иметься возможность его модификации извне — в предыдущем примере это означало, что свойство `Current` должно разрешать запись значений (быть `writable`). Контекст может быть реализован, например, так, как показано в листинге 4.11.

Листинг 4.11. Окружающий контекст

```
public abstract class SomeContext
{
    public static SomeContext Current
    {
        get
        {
```

```
            var ctx =
                Thread.GetData(
                Thread.GetNamedDataSlot("SomeContext"))
                as SomeContext;
            if (ctx == null)
            {
                ctx = SomeContext.Default;
                Thread.SetData(
                    Thread.GetNamedDataSlot("SomeContext"),
                    ctx);
            }
            return ctx;
        }
        set
        {
            Thread.SetData(
                Thread.GetNamedDataSlot("SomeContext"),
                value);
        }
    }

    public static SomeContext Default =
        new DefaultContext();

    public abstract string SomeValue { get; }
}
```

❶ Получить текущий контекст из TLS

❷ Сохранить текущий контекст в TLS

❸ Значение, предоставляемое контекстом

В данном примере контекст — это абстрактный класс, позволяющий заменять установленный контекст на другую реализацию контекста во время исполнения.

В данном примере свойство `Current` хранит текущий контент в локальной памяти потока (Thread Local Storage. TLS) ❶. Это означает, что каждый поток имеет собственный контекст, не зависящий от контекста любого другого потока. Для случаев, когда никто еще не устанавливал значение контекста в TLS, используется реализация, заданная по умолчанию. Важно гарантировать, что ни один пользователь не столкнется с исключением `NullReferenceException` при попытке получить доступ к свойству `Current`, поэтому должно быть разработано адекватное локальное умолчание. Заметьте, что при этом свойство `Default` используется всеми потоками. В данном примере это возможно, поскольку `DefaultContext` (класс-наследник класса `SomeContext`) определен как неизменяемый. Если бы применяемый по умолчанию контекст изменялся, требовалось бы назначать отдельный экземпляр контекста каждому потоку для предотвращения влияния потоков друг на друга.

Внешние клиенты могут устанавливать новый контекст в TLS ❷. Обратите внимание, что оставлена возможность присваивания контексту значения `null`, но в таком случае следующая попытка чтения автоматически переустановит свойство в контекст по умолчанию.

Основной смысл использования окружающего контекста заключается в возможности взаимодействия с ним. В данном примере такое взаимодействие пред-

ставлено одиночным абстрактным строковым свойством ❸, но определяющий контекст класс может быть при необходимости как весьма простым, так и очень сложным.

ПРЕДУПРЕЖДЕНИЕ

Для простоты я не рассматривал в коде листинга 4.11 вопросы безопасности потока. Если вы решите реализовать окружающий контекст на основе TLS, убедитесь, что полностью представляете себе весь предстоящий процесс.

СОВЕТ

Пример в листинге 4.11 использует TLS, но вы можете применять CallContext, чтобы получить похожий результат[1].

ПРИМЕЧАНИЕ

Окружающий контекст не требует ассоциации с контекстом потока или вызова. Иногда имеет смысл сделать его доступным во всем приложении, сделав его статическим (Static).

Когда вы хотите заменить установленный по умолчанию контекст на контекст пользователя, вы можете создать пользовательскую реализацию, основанную на контексте, и затем установить ее в нужный момент времени:

```
SomeContext.Current = new MyContext();
```

Для созданных на основе TLS контекстов нужно назначать пользовательский экземпляр при запуске нового потока, тогда как для контента, действующего в масштабе всего приложения, его можно установить в корне компоновки.

4.4.2. Когда следует использовать окружающий контекст

Окружающий контекст следует использовать как можно реже. В большинстве случаев намного более подходящими технологиями являются внедрение конструктора или внедрение свойства. При работе с настоящим сквозным аспектом приложения не исключено, что он загрязнит все API в вашем приложении, если вы будете передавать его во все сервисы.

ВНИМАНИЕ

Окружающий контент по структуре похож на антипаттерн «Локатор сервисов», который будет описан в главе 5. Различие между ними состоит в том, что окружающий контекст предоставляет только экземпляр единственной, сильно типизированной зависимости, тогда как «Локатор сервисов» предоставляет экземпляры для любой запрошенной вами зависимости. Различия эти тонкие, поэтому убедитесь, что вы полностью понимаете, когда следует применять окружающий контент, прежде чем сделать это. Если имеются сомнения, используйте другой паттерн внедрения зависимостей.

В подразделе 4.4.4 я реализую `TimeProvider`, который может использоваться для определения текущего времени, а также поясню, почему я предпочитаю такой подход

[1] См. *Mark Seemann* Ambient Context," 2007: http://blogs.msdn.com/ploeh/archive/2007/07/23/AmbientContext.aspx.

использованию статических членов (Members) типа DateTime. Текущее время является настоящим сквозным аспектом приложения, поскольку вы не можете заранее предсказать, каким классам и в каких модулях оно может потребоваться. Можно предположить, что текущее время будет применяться большинством классов, но лишь малая их часть использует его в действительности.

Потенциально это может вылиться в написание большого количества кода с дополнительным параметром TimeProvider, так как вы никогда не знаете, где такой параметр может потребоваться:

```
public string GetSomething(SomeService service,
    TimeProvider timeProvider)
{
    return service.GetStuff("Foo", timeProvider);
}
```

Представленный метод передает параметр TimeProvider сервису. Это может выглядеть безопасным, но когда мы рассмотрим метод GetStuff, мы обнаружим, что параметр никогда не используется:

```
public string GetStuff(string s, TimeProvider timeProvider)
{
    return this.Stuff(s);
}
```

Здесь параметр TimeProvider передается в качестве дополнительной нагрузки только потому, что он может понадобиться в будущем. Фактически мы просто зря засоряем API.

Использование окружающего контекста может оказаться подходящим решением, если выполняются условия, перечисленные в табл. 4.4.

Таблица 4.4. Условия применения окружающего контекста

Условие	Описание
Нужна возможность делать запросы к контексту	Если вам требуется только записывать какие-то данные (то есть все методы в контексте будут возвращать void), перехват (Interception) будет наилучшим решением. Может показаться, что такие случаи редки, но такой подход является распространенным: логирование (журналирование) происходящих событий, запись параметров производительности, гарантирование, что контекст безопасности не нарушается — все эти действия являются типичными операторами подтверждения отсутствия ошибок (Assertions), которые лучше всего реализуются через перехваты. Окружающий контекст следует использовать только в случаях, когда вам требуется запрашивать у него какие-либо данные (например, о текущем времени)
Определено подходящее локальное умолчание	Существование окружающего контекста является неявным, поэтому важно, чтобы контекст функционировал корректно — даже в случаях, когда он явно не устанавливался
Необходима гарантированная доступность	Даже если для контекста определено подходящее локальное умолчание, необходимо обеспечить его защиту от установки в значение null, что может нарушить его функциональность — в таком случае все клиенты будут получать NullReferenceExceptions. Листинг 4.11 демонстрирует некоторые подходы, которые могут использоваться для решения этой проблемы

В большинстве случаев достоинства окружающего контекста не компенсируют его недостатков (табл. 4.5), поэтому убедитесь, что все перечисленные условия выполняются; в противном случае выберите другую альтернативу.

Таблица 4.5. Достоинства и недостатки окружающего контекста

Достоинства	Недостатки
Не загрязняет API. Всегда доступен	Неявность. Непросто добиться корректной реализации. Может неправильно работать в некоторых средах исполнения

Самым значительным недостатком окружающего контекста является его неявность. Кроме того, как показывает листинг 4.11, его корректная реализация может оказаться затрудненной, а также могут возникнуть проблемы с некоторыми средами исполнения (ASP.NET).

Далее мы подробно рассмотрим все недостатки, перечисленные в табл. 4.5.

Неявность

При работе с окружающим контекстом вы не сможете, просто взглянув на интерфейс, с уверенностью сказать, используется ли данный контекст конкретным классом.

Рассмотрим класс, представленный на рис. 4.9: у него нет признаков использования окружающего контекста, тогда как реализованный в нем метод `GetMessage` делает это:

```
public string GetMessage()
{
    return SomeContext.Current.SomeValue;
}
```

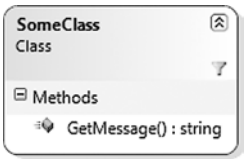


Рис. 4.9. Ни класс, ни его метод GetMessage не имеют явных указаний на то, что они используют окружающий контекст, тогда как на самом деле они его действительно применяют

Когда окружающий контекст реализован правильно, вы можете быть уверены, как минимум, что не будет сгенерировано исключительных ситуаций. Но в данном примере контекст влияет на поведение метода, поскольку он определяет возвращаемое значение. Если контекст изменится, поведение метода также может измениться, и вы не всегда сможете быстро понять, почему это произошло.

ПРИМЕЧАНИЕ

В своей работе Domain-Driven Design Эрик Эванс обсуждает так называемые информативные интерфейсы (Intention-Revealing Interfaces). В этом исследовании развивается мысль о том, что API должен сообщать о своих обязанностях через свой же публичный интерфейс. Когда класс использует окружающий контекст, он делает прямо противоположное: узнать о том, что здесь задействован такой контекст — прочесть документацию или исследовать исходный код.

Помимо потенциальной возможности возникновения заковыристых ошибок, неявность также усложняет определение мест возможного расширения класса. Окружающий контекст позволяет вам внедрять пользовательское поведение в любой класс, использующий такой контекст, но это неочевидно. Вы можете определить наличие такой возможности, только читая документацию или изучая реализацию намного более детально, чем вам бы этого хотелось.

Сложность реализации

Корректной реализации окружающего контекста может быть достаточно. Как минимум, вы должны гарантировать, что контекст всегда находится в пригодном для использования состоянии — то есть при запросах к нему не должно возникать никаких исключительных ситуаций типа `NullReferenceExceptions` только из-за того, что одна реализация контекста была удалена без замены ее на другую.

Чтобы гарантировать это, вы должны задать подходящее локальное умолчание, которое может использоваться, если явно не определено никакой другой реализации контекста. В листинге 4.11 я применял отложенную инициализацию свойства `Current`, поскольку C# не допускает статических инициализаций в потоке.

Когда окружающий контекст представляет собой настоящую универсальную концепцию типа времени, вы можете реализовать контекст как простой «Синглтон» (Singleton) с возможностью записи. «Синглтон» — это единственный экземпляр, совместно используемый во всем приложении. Я продемонстрирую пример такой реализации в подразделе 4.4.4.

Окружающий контекст может также представлять контекст, варьирующийся в зависимости от стека вызова. В частности, вариант может зависеть от того, какой элемент является инициатором запроса. Часто в веб-приложениях и веб-сервисах один и тот же код выполняется в контексте различных пользователей — каждый раз в своем собственном потоке. В таком случае окружающий контекст может быть тесно связанным с текущим потоком и сохраняться в TLS, как было сделано в листинге 4.11, но такое решение приводит к возникновению других проблем, особенно в ASP.NET.

Проблемы с выполнением в ASP.NET

Если окружающий контекст использует TLS, могут возникнуть проблемы при запуске приложения в ASP.NET, поскольку появляется вероятность изменения потоков в определенные моменты жизненного цикла интернет-страниц. При этом не гарантировано, что сохраненные в TLS данные будут скопированы из старого потока в новый.

В такой ситуации для хранения специфических данных запроса нужно использовать текущий `HttpContext`, а не TLS.

Подобное поведение, связанное с переключением потоков, не представляет проблем, когда окружающий контекст — это универсальный совместно используемый экземпляр, поскольку `Singleton` разделяется между всеми потоками в приложении.

4.4.3. Известные способы применения

Фреймворк .NET BCL включает несколько реализаций, использующих окружающий контекст.

Безопасность реализуется через интерфейс System.Security.Principal.IPrincipal, ассоциированный с каждым потоком. Вы можете получить (get) или установить (set) текущего владельца потока, используя методы доступа из Thread.CurrentPrincipal.

Еще один окружающий контекст, основанный на TLS, моделирует текущую культуру потока. Thread.CurrentCulture и Thread.CurrentUICulture позволяют получить доступ и модифицировать культурный контекст текущей операции. Многие предназначенные для форматирования API, такие как синтаксические анализаторы и конверторы типов, неявно используют текущую культуру, если никакая другая явно не установлена.

Наконец, трассировка является примером универсального окружающего контента. Класс Trace не связан с конкретным потоком, а совместно используется по всему приложению. Задействуя метод Trace.Write, вы можете записать трассировочное сообщение из любого места, при этом оно будет занесено в множество приемников TraceListeners, которые определяются конфигурированием свойства Trace.Listeners.

4.4.4. Пример: кэширование валюты

Абстракция Currency в демоприложении, описанном в предыдущих разделах, обращается к интерфейсу очень часто. Всякий раз, когда вы хотите конвертировать валюту, вы вызываете метод GetExchangeRateFor, который находит курс конвертации в какой-то внешней системе. Такое решение является примером гибкой реализации API, поскольку вы можете искать курс конвертации на текущее время, если вам это нужно, но в большинстве случаев такое требование не является необходимым и очень часто может негативно влиять на производительность системы.

Основанная на SQLServer реализация, представленная в листинге 4.10, выполняет запросы к базе данных каждый раз, когда ей требуется получить курс конвертации. Когда приложение отображает покупательскую корзину, каждый элемент в корзине конвертируется, что приводит к выдаче запросов к базе данных для каждого элемента в корзине, даже если используемый курс конвертации одинаков для всех элементов корзины. Лучше всего было бы кэшировать курс обмена на некоторое время. Так можно было бы избавить приложение от необходимости многократно запрашивать базу данных об одном и том же курсе в течение относительно небольшого периода времени.

В зависимости от того, насколько важно иметь именно текущие значения курса, время обновления (задержки) для кэша может быть небольшим или, наоборот, значительным. Например, задержка может составлять несколько секунд или несколько часов. Должна быть возможность конфигурирования значения задержки.

Чтобы определить, не устарело ли значение кэшированной валюты, нужно знать, сколько времени прошло с момента ее кэширования, что требует знания текущего времени. DateTime.UtcNow похож на встроенный окружающий контекст,

но он таковым не является, поскольку вы не можете установить время — можете только запросить его.

Невозможность переустановить текущее время редко является проблемой в рабочем приложении, но может оказаться проблемой при выполнении модульного тестирования.

ЭМУЛЯЦИЯ ВРЕМЕНИ

Как правило, веб-приложения не требуют возможности модификации текущего времени, другие же приложения могут значительно выиграть, если будут обладать такой возможностью.

Однажды мне пришлось писать достаточно сложное приложение-эмулятор, которое сильно зависело от текущего времени. Поскольку я всегда использую разработку через тестирование, я применил абстракцию текущего времени. Я внедрил экземпляры `DateTime`, которые отличались от реального машинного времени. Такой подход оправдал себя позднее, когда мне потребовалось ускорить время в эмуляторе. Это делалось путем задания нескольких новых значений. Все, что мне потребовалось выполнить, — зарегистрировать поставщик времени, который ускорял время, и все приложение немедленно ускорялось.

Если вы хотите понаблюдать подобную функцию в действии, вы можете обратиться к клиентскому приложению Всемирного телескопа (WorldWide Telescope[1]), которое позволит вам моделировать ночное время в ускоренном режиме. Представленный ниже рисунок — это скриншот пульта управления, позволяющего вам ускорять или замедлять ход времени с разной скоростью. Я не могу сказать, каким именно способом разработчики реализовали данную функцию, но это похоже на то, что пришлось сделать мне.

Всемирный телескоп позволяет останавливать время или двигаться во времени вперед и назад с разной скоростью. Так моделируется вид ночного неба в разные моменты.

В случае нашего демоприложения я хочу иметь возможность управления временем при проведении модульного тестирования, чтобы я мог удостоверяться, что кэшированные курсы конвертации валют устаревают в таком темпе, как следует.

Провайдер времени

Время — это абсолютная концепция (даже если время идет с разной скоростью в разных уголках Вселенной), поэтому я буду моделировать его как абсолютно универсальный общий ресурс. Поскольку нет причин иметь отдельный провайдер

[1] http://www.worldwidetelescope.org.

времени в каждом потоке, окружающий контекст TimeProvider реализован как синглтон с возможностью записи. Это показано в листинге 4.12.

Листинг 4.12. Окружающий контекст TimeProvider

```
public abstract class TimeProvider
{
    private static TimeProvider current;

    static TimeProvider()
    {
        TimeProvider.current =
            new DefaultTimeProvider();
    }

    public static TimeProvider Current
    {
        get { return TimeProvider.current; }
        set
        {
            if (value == null)
            {
                throw new ArgumentNullException("value");
            }
            TimeProvider.current = value;
        }
    }

    public abstract DateTime UtcNow { get; }

    public static void ResetToDefault()
    {
        TimeProvider.current =
            new DefaultTimeProvider();
    }
}
```

1 Инициализация TimeProvider по умолчанию

2 Граничный оператор

3 Важная часть

Назначение класса TimeProvider заключается в обеспечении управления взаимодействием времени с клиентами. Как описано в табл. 4.4, наличие локального умолчания является важным фактором, поэтому класс инициализируется статически, чтобы можно было использовать класс DefaultTimeProvider (я скоро это покажу) 1.

Еще одно указанное в табл. 4.4 условие гласит: вы должны гарантировать, что TimeProvider никогда не будет установлен в неработоспособное состояние. Поле current никогда не должно устанавливаться в null, что обеспечивается использованием граничного оператора 2.

Все описанные здесь операции являются только подготовкой для того, чтобы сделать TimeProvider легко доступным из любого места приложения. Смысл его применения объясняется его способностью обслуживать экземпляры DateTime, сообщая этим экземплярам текущее время 3. Я намереваюсь полностью повторить имя

и сигнатуру абстрактного свойства из DateTime.UtcNow. Если будет необходимо, я смогу добавить еще такие абстрактные свойства, как Now и Today, но для данного примера они не нужны.

Наличие корректного локального умолчания является важным условием, и, к счастью, его несложно реализовать в данном примере, так как оно просто возвращает текущее время. Это значит, что до тех пор, пока вы явно не назначите другой TimeProvider, любой клиент, использующий TimeProvider.Current.UtcNow, будет получать актуальное текущее время.

Реализация DefaultTimeProvider приведена в листинге 4.13.

Листинг 4.13. Используемый по умолчанию Провайдер времени

```
public class DefaultTimeProvider : TimeProvider
{
    public override DateTime UtcNow
    {
        get { return DateTime.UtcNow; }
    }
}
```

Класс DefaultTimeProvider является наследником класса TimeProvider и возвращает действительное время всякий раз, когда клиент читает значение свойства UtcNow.

Когда CachingCurrency использует TimeProvider для получения текущего времени, он будет получать актуальное текущее время, если только вы специально не установите в приложении другой TimeProvider — и я планирую сделать это только в модульных тестах.

Кэширование валют

Для реализации кэшированных валют задействуем шаблон «Декоратор», который модифицирует реализацию Currency.

ПРИМЕЧАНИЕ

Паттерн проектирования «Декоратор» очень важен для технологии перехвата, что подробно будет обсуждаться в главе 9.

Вместо того чтобы модифицировать имеющуюся реализацию Currency, базирующуюся на технологии SQL Server и продемонстрированную в листинге 4.10, мы *обернем* ее в кэш и вызовем реальную реализацию только при условии, что время актуальности кэша истечет или окажется, что он по каким-то причинам не содержит элементов.

Как указывалось в подразделе 4.1.4, CurrencyProvider — это абстрактный класс, возвращающий экземпляры Currency. Класс CachingCurrencyProvider реализует такой же базовый класс и охватывает функционал попадающего внутрь такой обертки класса CurrencyProvider. Будучи запрошенным о значении Currency, он возвращает экземпляр Currency, созданный классом CurrencyProvider, но обернутый в класс CachingCurrency (рис. 4.10).

СОВЕТ

Шаблон «Декоратор» — один из лучших способов обеспечить разделение ответственности.

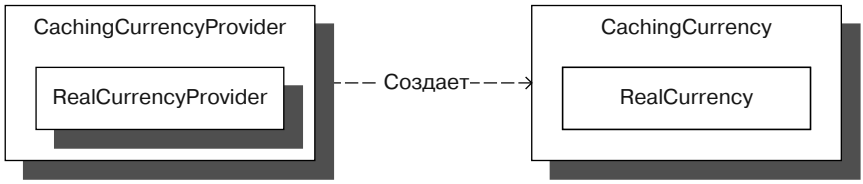


Рис. 4.10. Класс CachingCurrencyProvider обертывает «реальный» CurrencyProvider и возвращает экземпляры CachingCurrency, обертывающие «реальные» экземпляры Currency

Такое решение позволяет мне кэшировать любую реализацию валюты, а не только такую, которая базируется на технологии SQL Server, как это есть в настоящее время. Рисунок 4.11 показывает схему класса `CachingCurrency`.

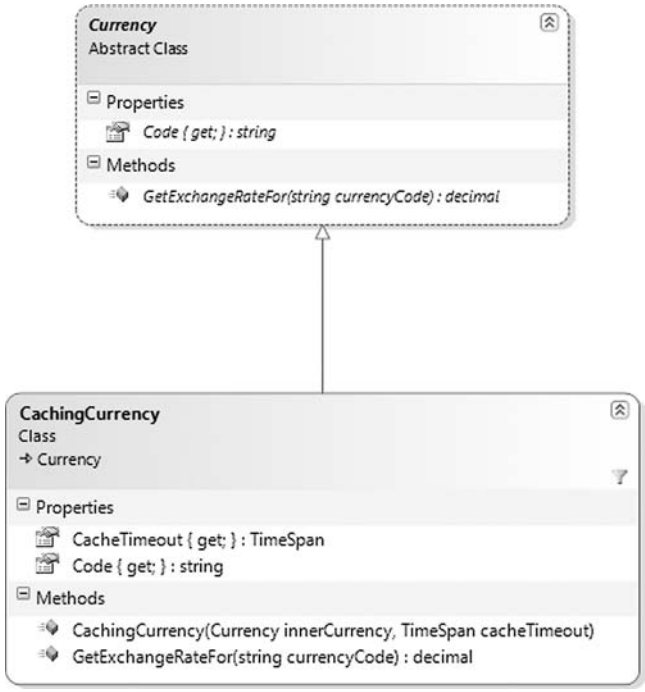


Рис. 4.11. Класс CachingCurrency получает внутреннюю валюту и задержку кэша в конструкторе, а также обертывает функциональность внутренней валюты

Класс `CachingCurrency` использует внедрение конструктора, чтобы получить «реальный» экземпляр, курсы конвертации которого должны кэшироваться. Например, `CachingCurrency` делегирует свое свойство `Code` свойству `Code` внутреннего класса `Currency`.

Интересная часть реализации `CachingCurrency` — его метод `GetExchangeRateFor`, показанный в листинге 4.14.

Листинг 4.14. Кэширование курса конвертации

```
private readonly Dictionary<string, CurrencyCacheEntry> cache;

public override decimal GetExchangeRateFor(string currencyCode)
{
    CurrencyCacheEntry cacheEntry;
    if ((this.cache.TryGetValue(currencyCode,
        out cacheEntry))
        && (!cacheEntry.IsExpired))
    {
        return cacheEntry.ExchangeRate;
    }
    var exchangeRate =
        this.innerCurrency
        .GetExchangeRateFor(currencyCode);

    var expiration =
        TimeProvider.Current.UtcNow + this.CacheTimeout;
    this.cache[currencyCode] =
        new CurrencyCacheEntry(exchangeRate, expiration);

    return exchangeRate;
}
```

❶ Возвращает кэшированный курс, если таковой имеется

❷ Кэшировать курс

Когда клиент запрашивает курс конверсии, вы прежде всего делаете запрос на поиск кода валюты в кэше. Если в кэше имеется актуальный код запрошенной валюты, вы возвращаете кэшированный курс конвертации, остальная часть кода пропускается ❶. Чуть позже я вернусь к вопросу о том, как определить, истек ли срок действия кода.

Внутренний класс `Currency` запрашивается о значении курса, только если в кэше нет непросроченного искомого значения. Прежде чем вернуть новый курс клиенту, его необходимо кэшировать. Первый шаг здесь — вычисление времени истечения срока действия кода, и именно в этом месте используется окружающий контекст `TimeProvider` вместо более традиционного `DateTime.Now`. Когда время истечения будет рассчитано, можно кэшировать курс ❷, прежде чем вернуть результат.

Определение того, истекло ли время актуальности кэшированного курса, также выполняется с применением окружающего контента `TimeProvider`:

```
return TimeProvider.Current.UtcNow >= this.expiration;
```

Класс `CachingCurrency` использует `TimeProvider` во всех местах, где ему требуется текущее время, поэтому возможно написание модульного теста, который бы точно управлял временем.

Модификация времени

Когда выполняется модульное тестирование класса `CachingCurrency`, вы можете управлять ходом времени в вашей системе. Оно будет идти совершенно независимо от реальных системных часов. Это позволит вам написать детерминированные

модульные тесты, несмотря на даже то, что тестируемая система зависит от концепции текущего времени. Следующий листинг содержит тест, проверяющий, что курс конверсии запрашивается у тестируемой системы четыре раза, при этом внутренний класс валюты вызывается только дважды: при первом вызове и еще раз — когда время актуальности сохраненных в кэше значений истечет. Код, показанный в листинге 4.15, тестирует: 1) что валюта корректно кэшируется; 2) что время ее актуальности истекает.

Листинг 4.15. Модуль, тестирующий корректное кэширование валюты и истечение времени ее актуальности

```
[Fact]
public void InnerCurrencyIsInvokedAgainWhenCacheExpires()
{
    // Настройка тестовой конфигурации
    var currencyCode = "CHF";
    var cacheTimeout = TimeSpan.FromHours(1);

    var startTime = new DateTime(2009, 8, 29);
    var timeProviderStub = new Mock<TimeProvider>();
    timeProviderStub
        .SetupGet(tp => tp.UtcNow)
        .Returns(startTime);
    TimeProvider.Current = timeProviderStub.Object;

    var innerCurrencyMock = new Mock<Currency>();
    innerCurrencyMock
        .Setup(c => c.GetExchangeRateFor(currencyCode))
        .Returns(4.911m)
        .Verifiable();

    var sut =
        new CachingCurrency(innerCurrencyMock.Object,
            cacheTimeout);
    sut.GetExchangeRateFor(currencyCode);
    sut.GetExchangeRateFor(currencyCode);
    sut.GetExchangeRateFor(currencyCode);

    timeProviderStub
        .SetupGet(tp => tp.UtcNow)
        .Returns(startTime + cacheTimeout);
    // Пробная система
    sut.GetExchangeRateFor(currencyCode);
    // Проверка результатов
    innerCurrencyMock.Verify(
        c => c.GetExchangeRateFor(currencyCode),
        Times.Exactly(2));
    // Разъединение (неявное)
}
```

1 Установка контекста TimeProvider

2 Вызов внутренней валюты

3 Должно быть кэшировано

4 Передача параметра времени, вызывающего задержку

5 Должна быть вызвана внутренняя валюта

6 Проверка того, что внутренняя валюта была вызвана корректно

ОСТОРОЖНО, ЖАРГОН

Следующий ниже текст содержит термины, используемые в модульном тестировании, — они выделены ***полужирным курсивом***, но поскольку эта книга не о модульном тестировании, я просто отошлю вас к книге xUnit Test Patterns[1], в которой определены используемые здесь термины.

Прежде всего в этом тесте устанавливается тестовый двойник (Test Double) для `TimeProvider`, который будет возвращать экземпляры `DateTime` вместо использования системных часов. В данном тесте я использую динамический фреймворк, называемый Moq[2], чтобы указать, что свойство `UtcNow` должно возвращать один и тот же `DateTime`, пока не будет указано обратное. После определения данная заглушка (***Stub***) инжектируется в окружающий контекст (1).

Первый вызов метода `GetExchangeRateFor` должен вызывать вложенный класс `Currency` класса `CachingCurrency`, так как на этот момент кэшированные данные отсутствуют (2), а вот два следующих вызова должны возвращать кэшированное значение (3), поскольку время никуда не передавалось в соответствии с заглушкой `TimeProvider`.

После того как будет выполнено несколько вызовов, наступает момент модификации времени. Заглушка `TimeProvider` модифицируется так, чтобы вернуть экземпляр `DateTime`, который свидетельствует о задержке кэша (4). Это приводит к еще одному вызову метода `GetExchangeRateFor` (5), из которого выполняется повторный запрос класса `Currency`. В результате должны быть возвращены новые данные о курсе, так как изначально сохраненные в кэше данные потеряли актуальность (устарели).

Поскольку предполагается, что вложенный класс `Currency` будет вызван дважды, вы можете в заключение проверить, что именно это и имело место путем указания внутреннему подставному объекту `Currency` ***Mock***, что метод `GetExchangeRateFor` должен быть вызван ровно два раза (6).

Одна из многочисленных опасностей, связанных с окружающим контекстом, заключается в том, что, будучи однажды установленным, он остается в таком состоянии, пока не будет модифицирован снова. Но в силу его неявной природы этот шаг легко упустить. В модульном тесте, например, поведение, заданное тестом в листинге 4.15, остается неизменным, пока не будет явно переустановлено (что я делаю на этапе разрыва (Teardown)). Это может вызвать сложные для выявления ошибки (пока только в тестовом коде), поскольку они могут перетекать из теста в тест и влиять на тесты, которые будут выполняться впоследствии.

Окружающий контекст выглядит обманчиво простым с точки зрения реализации и использования, но может обусловливать множество трудно идентифицируемых ошибок. Бывает целесообразно использовать окружающий контекст, но я рекомендую прибегать к нему лишь в тех случаях, когда нет лучшей альтернативы. Представьте себе хрен: он очень вкусен в некоторых блюдах, но никто же не будет питаться одним только хреном.

---

1 *Gerard Meszaros* xUnit Test Patterns: Refactoring Test Code. — New York: Addison-Wesley, 2007.

2 http://code.google.com/p/moq/.

4.4.5. Родственные паттерны

Окружающий контекст может использоваться для моделирования сквозных аспектов приложения, хотя он и требует наличия локального умолчания.

Если выяснится, что зависимость не является сквозным аспектом приложения, следует изменить стратегию использования внедрения зависимостей. Если по-прежнему имеется локальное умолчание, можно перейти к внедрению свойства, но чаще оказывается необходимым использовать внедрение конструктора.

4.5. Резюме

Представленные в этой главе шаблоны являются основной частью внедрения зависимостей. Имея в распоряжении корень компоновки и подходящий набор шаблонов внедрения зависимостей, вы сможете реализовать подход «Внедрение зависимостей для бедных».

При применении внедрения зависимостей существует множество нюансов и тонкостей, но именно паттерны определяют основные механизмы, дающие ответ на вопрос: «Как я могу инжектировать мои зависимости?»

Эти паттерны не являются взаимозаменяемыми. В большинстве случаев следует выбирать внедрение конструктора, но встречаются ситуации, когда более подходящим оказывается один из других подходов. Рисунок 4.12 описывает алгоритм принятия решения, который может облегчить вам принятие решения о выборе подходящего паттерна. А если этот алгоритм не поможет, выбирайте внедрение конструктора — вы в этом случае никогда не совершите ужасных ошибок.

Первое, что нужно выяснить в соответствии с данным алгоритмом, — является ли зависимость чем-то необходимым, или, возможно, вы уже располагаете ею и хотели бы использовать для организации взаимодействия с другим компонентом программы. В большинстве случаев зависимость вам, по-видимому, требуется, но в сценариях расширений вы, возможно, захотите передать текущий контекст в расширение. При этом зависимость, скорее всего, будет отличаться от операции к операции. В таком случае хорошим кандидатом на реализацию будет внедрение метода.

Когда зависимость представляет сквозной аспект приложения, выбор наилучшего шаблона зависит от направления коммуникации. Если вам нужно только записать какие-то данные (например, продолжительность операции или какие данные были переданы), перехват (мы рассмотрим его в главе 9) подходит лучше всего. Он также хорошо работает в случае, если ответ, который вам нужен от него, уже включен в определение интерфейса. Кэширование — это отличный пример такого использования перехвата.

Если вам нужно запросить зависимость сквозного аспекта на получение ответа, не включенного в оригинальный интерфейс, вы можете применять окружающий контекст, при условии, что для него имеется локальное умолчание. Так вы сможете объединять собственно контекст с поведением, заданным по умолчанию, которое используется всеми клиентами без явного конфигурирования.

Когда зависимость не представляет сквозной аспект приложения, наличие локального умолчания становится важным фактором для принятия решения, так как оно может сделать явную установку зависимости опциональной. Умолчание

используется, если отсутствует явно заданная реализация. Такой сценарий эффективно реализуется внедрением метода.

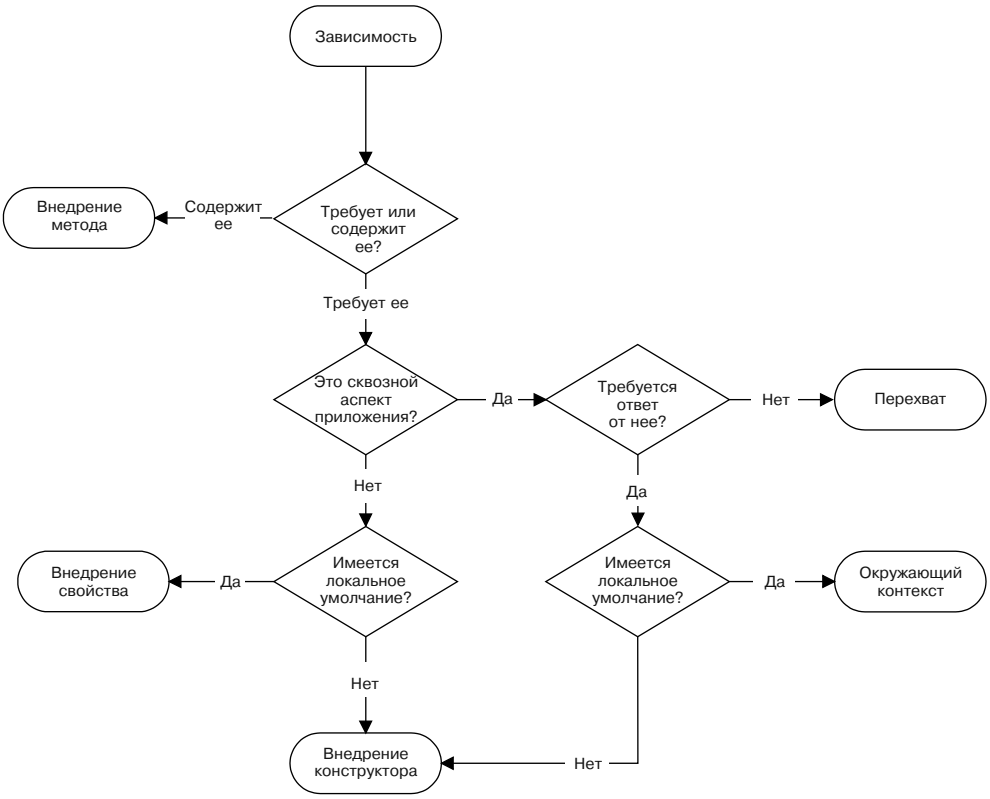


Рис. 4.12. В большинстве случаев следует выбирать внедрение конструктора, но в некоторых ситуациях более подходящими будут другие паттерны внедрения зависимостей

Во всех других случаях применяется внедрение конструктора. Из представленной на рис. 4.12 схемы может показаться, что внедрение конструктора является чем-то вроде шаблона последней надежды, который используется, только если все остальные окажутся непригодными. Это справедливо только отчасти, поскольку в большинстве случаев специализированные паттерны не применяются и внедрение конструктора выступает в роли паттерна, используемого по умолчанию. Такое внедрение просто для понимания, и его намного легче корректно реализовать, чем другие паттерны внедрения зависимостей. Вы можете разработать все приложение, используя только внедрение конструктора, но понимание других паттернов может помочь вам сделать правильный выбор в тех редких случаях, когда внедрение конструктора оказывается не совсем подходящим вариантом.

Эта глава содержит систематизированный каталог, поясняющий, как вы должны инжектировать зависимости в свои классы. В следующей главе внедрение зависимостей рассматривается с противоположной точки зрения. В ней описывается, как не следует реализовывать внедрения.

5 Антипаттерны внедрения зависимостей

Меню:

- «Диктатор»;
- «Гибридное внедрение»;
- «Ограниченное конструирование»;
- «Локатор сервисов».

С гастрономической точки зрения еще в 70-е годы прошлого века Дания была развивающейся страной — я жил там, но я никогда не чувствовал себя обделенным, потому что не знал ничего лучшего. Основными продуктами питания были мясо и картошка, но иностранные веяния постепенно проникали в страну — я думаю, отчасти из-за того, что это была пора массового туризма.

Датчане все больше путешествовали по европейским странам, и самые большие авантюристы из них решались попробовать местную кухню. Они возвращались домой, и макароны начали приобретать большую популярность у молодого поколения, хотя итальянцы так и не признали датскую версию болонского соуса.

И вот я вообразил себе такую ситуацию. Какой-то предприимчивой датской туристке настолько понравилась лапша тальятелле по-болонски, что она решила попробовать приготовить ее, когда вернется домой (я считаю, что это была женщина, потому что мужчина вряд ли решился бы на такое). Она попыталась запомнить, что входило в состав соуса, но это было нелегко во время долгого возвращения в Данию на автобусе.

Как часто происходит с ингредиентами, панчетта и красное вино были забыты еще до того, как наша героиня покинула Италию, бульон и куриная печенка потерялись где-то в Австрии или Швейцарии, ну и большинство овощей пропали на долгом пути через (Западную) Германию. Когда она пересекла датскую границу, все, что осталось от оригинального рецепта, — это нарезанный лук и фарш, приготовленные с единственным видом макаронных изделий, доступных в то время в Дании: спагетти.

Мы ели это кушанье многие годы, и оно нам нравилось. Где-то в 1980-е годы в рецепт были добавлены томатная паста и орегано, что сделало продукт более похожим на свой прототип. Я использовал этот рецепт в течение где-то десяти лет, пока кто-то не подсказал мне, что его можно значительно улучшить, добавив овощи, куриную печенку, красное вино и другие компоненты.

Мораль этой истории такова: я думал, что готовлю лапшу по-болонски, тогда как в действительности моя стряпня ее даже отдаленно не напоминала. У меня никогда не возникало даже мысли о том, чтобы проверить свой рецепт, поскольку я вырос с ним. И хотя подобие вкуса не является обязательным требованием, лапша, приготовленная по оригинальному рецепту, намного вкуснее, и я совершенно не собираюсь возвращаться к своим заблуждениям.

В предыдущей главе я сравнил паттерны проектирования с рецептами. Паттерн можно считать общим языком, который можно использовать для краткого описания сложных концепций, таких как лапша по-болонски. С другой стороны, когда концепция (или скорее ее реализация) искажается, мы имеем дело с антипаттерном.

ОПРЕДЕЛЕНИЕ

Антипаттерн — это описание распространенного решения проблемы, из-за которого возникают серьезные негативные последствия.

Антипаттерны часто возникают из-за игнорирования чего-либо (как в случае с моим болонским соусом) и их однозначно следует избегать. Но в данном случае врага нужно не только знать в лицо, но и понимать, почему такой антипаттерн возникает. Антипаттерны представляют собой более или менее формализованный способ описания распространенных ошибок, которые люди совершают снова и снова, независимо друг от друга.

В данной главе я опишу некоторые распространенные антипаттерны, связанные с внедрением зависимостей. За свою карьеру я сталкивался с ними всеми в той или иной форме и хочу повиниться в том, что и сам использовал некоторые из них. Во многих случаях они проистекают из самых благих намерений реализовать внедрение зависимостей в приложении; но реализации, выполненные без хорошего понимания основ внедрения зависимостей, превращаются в горе-решения, которых лучше избегать.

Знакомство с описанными здесь антипаттернами должно подсказать, каких ловушек следует избегать при реализации вашего первого проекта с внедрениями зависимостей. Ваши ошибки вряд ли будут в точности такими, которые делал я, или такими, которые описаны в предлагаемых здесь примерах, но данная глава продемонстрирует вам признаки имеющейся опасности.

От антипаттернов можно избавиться, выполнив рефакторинг кода, опираясь на один из описанных в главе 4 паттернов внедрения зависимостей. Сложность исправления конкретной ошибки будет зависеть от деталей реализации, но для каждого антипаттерна я даю некое обобщенное руководство по его устранению путем рефакторинга на основе какого-либо паттерна.

СОВЕТ

Мои руководства по рефакторингу от антипаттернов к паттернам ограничены объемами данной главы, так как это не является основной темой книги. Если вам интересна тема модификации существующих приложений в направлении использования внедрения зависимостей, вы можете начать с чтения вот этой книги, целиком посвященной вопросам такого рефакторинга: Working Effectively with Legacy Code[1]. Хотя она не посвящена конкретно внедрением зависимостей, в ней описывается много тех же концепций, которые я использую здесь.

[1] *Michael Feathers* Working Effectively with Legacy Code. — New York: Prentice Hall, 2004.

Антипаттерны, которым посвящена эта глава, перечислены в табл. 5.1. Рисунок 5.1 иллюстрирует структуру главы.

Рис. 5.1. Эта глава является каталогом антипаттернов. Каждый антипаттерн описан так, чтобы его можно было изучать независимо от других

Таблица 5.1. Антипаттерны внедрения зависимостей

Антипаттерн	Описание
«Диктатор»	Зависимости управляются напрямую, что является противоположностью «Инверсии управления»
«Гибридное внедрение»	Внешние умолчания задаются в качестве стандартных значений для зависимостей
«Ограниченное конструирование»	Предполагается, что конструкторы имеют конкретные сигнатуры
«Локатор сервисов»	Неявный сервис может предоставлять зависимости потребителям, но это не гарантируется

ПРЕДУПРЕЖДЕНИЕ

Эта глава отличается от других глав книги, потому что включенный в нее код дает в основном примеры того, как не следует реализовывать внедрение зависимостей. Не пытайтесь проделать это в домашних условиях!

Как я уже говорил, внедрение конструктора является наиболее важным паттерном внедрения зависимостей, «Диктатор» — доминирующий антипаттерн. Он фактически не позволяет вам использовать любые верные варианты применения внедрения зависимостей, поэтому его нужно хорошо изучить, прежде чем приступать к знакомству с прочими шаблонами. С другой стороны, антипаттерн «Локатор сервисов» является самым опасным из всех, поскольку складывается впечатление, что он действительно решает проблему.

Остальная часть главы детально описывает каждый антипаттерн. Вы можете читать ее с начала и до конца либо ограничиться только тем антипаттерном, который интересен вам. Каждому антипаттерну посвящен независимый и подробный раздел. Однако, если вы решите познакомиться только с одним антипаттерном, вы должны сконцентрироваться на «Диктаторе».

5.1. «Диктатор»

Что является противоположностью инверсии управления? Первоначально термин «инверсия управления» использовался для идентификации противоположности нормального хода дел, но мы не можем говорить об инерции мышления как о некоем универсальном антипаттерне. Итак, я хорошо подумал и предложил термин «Диктатор» (Control Freak), чтобы обозначить класс, который не хочет отдавать управление своими зависимостями.

Такая ситуация возникает всякий раз, когда мы создаем новый экземпляр какого-либо типа с помощью ключевого слова `new`. Тем самым мы явно заявляем, что собираемся сами управлять жизненным циклом данного экземпляра, и ни у кого не будет никаких шансов перехватить этот конкретный объект.

СОВЕТ

Количество появлений ключевого слова new в коде является великолепным индикатором того, насколько сильно он связан.

Антипаттерн «Диктатор» возникает всякий раз, когда мы создаем экземпляр зависимости и при этом явно или неявно используем ключевое слово `new` в любом месте, кроме корня компоновки.

ПРИМЕЧАНИЕ

Хотя ключевое слово new является признаком плохого кода в случае нестабильных зависимостей, нет ничего плохого в его применении со стабильными зависимостями. Ключевое слово new совсем не является запретным, но им лучше не пользоваться для получения экземпляров нестабильных зависимостей.

Наиболее вопиющий пример использования «Диктатора» — случай, когда мы не пытаемся использовать абстракции в своем коде. Вы видели несколько таких примеров в главе 2, когда Мэри реализовывала свое коммерческое приложение (см. подраздел 2.1.1). Такой подход обходится без всякого внедрения зависимостей. Но даже когда разработчик имеет представление о внедрении зависимостей и сочетаемости, антипаттерн «Диктатор» часто встречается в том или ином виде.

В следующих разделах я покажу примеры кода, похожего на тот, что я встречал в уже эксплуатируемых системах. В каждом случае разработчики имели наилучшие намерения, пытаясь программировать с использованием интерфейсов, но ни в одном случае программист просто не разобрался с реальными требованиями и обоснованиями.

5.1.1. Пример: создание новых экземпляров зависимостей

Многие разработчики слышали о принципе программирования на основе интерфейсов, но не всегда понимают принципиальные достоинства такого подхода. Пытаясь сделать все правильно или следовать передовым методологиям, они пишут довольно малограмотный код.

В главе 2 вы видели пример ProductService, где используется экземпляр абстрактного класса ProductRepository (см. листинг 2.6) для получения списка продуктов со скидками.

В качестве напоминания здесь приведен один метод из того кода, имеющий отношение к теме настоящего разговора:

```
public IEnumerable<Product> GetFeaturedProducts(IPrincipal user)
{
    return from p in this.repository.GetFeaturedProducts()
           select p.ApplyDiscountFor(user);
}
```

Если сравнить этот код с примером из листинга 2.6, то заметно, что я не вставлял в этот пример граничный оператор. Но главное изменение заключается в том, что переменная repository представляет абстрактный класс. В главе 2 вы видели, как поле repository может получить значение через внедрение конструктора, но сейчас я покажу другой способ (листинг 5.1).

Листинг 5.1. Создание объекта ProductRepository

```
private readonly ProductRepository repository;

public ProductService()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    this.repository =
        new SqlProductRepository(connectionString);   // (1) Прямое создание нового экземпляра
}
```

Поле repository объявлено как имеющее тип абстрактного класса ProductRepository, поэтому любой участник класса ProductService (такой, например, как метод GetFeaturedProducts) будет программироваться через интерфейс. Хотя это выглядит как правильное решение, оно приносит немного пользы, потому что во время исполнения тип всегда будет SqlProductRepository (1). Не существует способа, которым вы могли бы перехватить или изменить значение переменной репозитория, кроме как изменение исходного кода и перекомпиляция.

Таким образом, вы не получите существенной пользы, если просто объявите переменную как имеющую абстрактный тип или тип интерфейса, а затем жестко кодируете ее как имеющую значение лишь одного заданного конкретного типа. Единственное незначительное достоинство такого решения в том, что данный конкретный тип определяется только в одном или в совсем небольшом количестве мест вашего кода, так что замена одной конкретной реализации на другую не потребует значительного рефакторинга. В приведенном примере вам потребуется лишь создать экземпляр другой реализации типа ProductRepository в конструкторе (1), оставив прочий код класса ProductService без изменения.

Непосредственное создание новых объектов зависимостей — это только один пример антипаттерна «Диктатор». Прежде чем я перейду к анализу и описанию возможных способов устранения проблем, связанных с «Диктатором», рассмотрим еще несколько примеров, чтобы у вас сформировалось лучшее представление о контексте проблемы и распространенных неудачных попытках устранения некоторых возникающих неприятностей.

В данном конкретном примере очевидно, что найденное решение неоптимально. Многие разработчики в такой ситуации попытаются исправить ситуацию, как это будет продемонстрировано в следующем примере.

5.1.2. Пример: фабрика

Самый распространенный (и ошибочный) способ устранения проблем, возникающих из-за создания нового конкретного экземпляра зависимости (с использованием ключевого слова `new`), заключается в попытке применения фабрики какого-то вида. При попытке использовать фабрики существует несколько возможных путей, и я опишу следующие из них:

- «Конкретная фабрика» (Concrete Factory);
- «Абстрактная фабрика» (Abstract Factory);
- «Статическая фабрика» (Static Factory).

Если бы вы сказали Мэри, что она может использовать только абстрактный класс `ProductRepository`, она (из главы 2) попробовала бы задействовать фабрику `ProductRepositoryFactory`, которая генерирует экземпляры, необходимые для работы. Давайте послушаем, как Мэри обсуждает данный подход со своим коллегой Йенсом. Их разговор коснется вариантов фабрик, которые я перечислил.

Мэри: Нам нужен экземпляр ProductRepository в классе ProductService. Но ProductRepository — это абстрактный класс, поэтому мы не можем просто создать новый экземпляр, и наш консультант сказал, что мы не сможем создать экземпляры другого типа, кроме SqlProductRepository.

Йенс: А что насчет какой-нибудь фабрики?

Мэри: Да, я думала об этом, но я не очень хорошо представляю себе, как это реализовать. Кроме того, я не понимаю, как этот подход решит нашу проблему. Смотри сюда...

Мэри начинает писать фрагмент кода, чтобы продемонстрировать возникшую проблему.

«Конкретная фабрика»

Вот какой код написала Мэри:

```
public class ProductRepositoryFactory
{
    public ProductRepository Create()
    {
        string connectionString =
                 ConfigurationManager.ConnectionStrings
```

```
                ["CommerceObjectContext"].ConnectionString;
        return new SqlProductRepository(connectionString);
    }
}
```

Мэри: Эта ProductRepositoryFactory инкапсулирует информацию о том, как создавать экземпляры ProductRepository, но она не решает проблему. Ведь придется использовать ее в ProductService примерно таким образом:

```
var factory = new ProductRepositoryFactory();
this.repository = factory.Create();
```

Видишь? Мы создали экземпляр класса ProductRepositoryFactory в ProductService, но по-прежнему жестко задаем генерацию SqlProductRepository. Все, чего мы добились, — это перенесли проблему в другой класс.

Йенс: Да, я вижу... А не можем ли мы решить проблему, используя вместо конкретной фабрики абстрактную?

Давайте ненадолго прервем беседу Мэри и Йенса и оценим, что произошло. Мэри абсолютно права, утверждая, что класс «Конкретной фабрики» не решает проблемы, провоцируемых «Диктатором», а всего лишь откладывает их. Такой подход усложняет код, не повышая при этом его эксплуатационную ценность. Класс `ProductService` теперь напрямую управляет жизненным циклом фабрики, а фабрика напрямую управляет жизненным циклом `ProductRepository`, так что мы по-прежнему не можем перехватить или заменить экземпляр репозитория во время исполнения.

ПРИМЕЧАНИЕ

Я не хочу сказать, что являюсь принципиальным противником использования классов «Конкретных фабрик». «Конкретная фабрика» может решить другие проблемы, такие как повторяемость кода, путем инкапсуляции сложной логики генерации объектов. Но она бесполезна при реализации внедрения зависимостей. Используйте конкретные фабрики, только если это действительно целесообразно.

Совершенно очевидно, что конкретные фабрики не решают никаких проблем, связанных с внедрением зависимостей, и я не думаю, что они вообще используются в данной сфере. Комментарий Йенса насчет «Абстрактной фабрики» выглядит более многообещающим.

«Абстрактная фабрика»

Вернемся к разговору Мэри и Йенса и послушаем, что Йенс скажет относительно «Абстрактных фабрик».

Йенс: Что если мы сделаем фабрику абстрактной, например вот так?

```
public abstract class ProductRepositoryFactory
{
    public abstract ProductRepository Create();
}
```

Это значит, что мы нигде не кодируем жестко никаких ссылок на SqlProductRepository и теперь можем использовать фабрику в ProductService, чтобы получить экземпляры ProductRepository.

Мэри: Но ведь фабрика абстрактная, как же мы создадим новый экземпляр ее самой?

Йенс: Мы создадим ее реализацию, которая будет возвращать экземпляры SqlProductService.

Мэри: Да, но как же мы будем создавать ее экземпляр?

Йенс: Мы просто используем new в ProductService... Ох. Подожди...

Мэри: И это вернет нас туда, откуда мы начали.

Мэри и Йенс быстро осознали, что «Абстрактная фабрика» не изменяет ситуацию. Их первоначальная проблема заключалась в том, что им был нужен экземпляр абстрактного класса ProductRepository, теперь же вместо этого им нужен экземпляр абстрактной фабрики ProductRepositoryFactory.

«АБСТРАКТНАЯ ФАБРИКА»

«Абстрактная фабрика» — это один из шаблонов проектирования, представленных в оригинальной книге Design Patterns[1]. Она приносит пользу в отношении внедрения зависимостей, так как она может инкапсулировать сложную логику, создающую другие зависимости.

Она представляет собой хорошую альтернативу полной передаче управления, возникающей при реализации полной инверсии управления, поскольку частично позволяет пользователю управлять жизненным циклом зависимостей, создаваемых фабрикой. Фабрика управляет лишь тем, что именно создается, а также самим процессом создания.

Шаблон «Абстрактная фабрика» является более распространенным, чем вы можете предположить, — имена используемых классов часто могут завуалировать этот факт. Например, класс `CurrencyProvider`, используемый в подразделе 4.1.4, на самом деле является абстрактной фабрикой с другим именем: это абстрактный класс, создающий экземпляры другого абстрактного класса (`Currency`).

В разделе 6.1 мы вернемся к шаблону «Абстрактная фабрика», чтобы рассмотреть, как он может помочь разрешить некоторые проблемы, зачастую возникающие при реализации внедрения зависимостей.

По иронии судьбы Мэри и Йенс отклонили единственную реализацию фабрик, которая была хотя бы безвредной. С другой стороны, она не может решить их проблему; и поскольку не предполагается, что логика создания экземпляров `ProductRepository` будет сколько-нибудь сложной, использование абстрактной фабрики не принесет никакой пользы.

Теперь, после того как Мэри и Йенс отвергли единственную безопасную реализацию, нерассмотренным остался только один, причем несущий опасность, вариант.

«Статическая фабрика»

Мэри и Йенс продолжают искать решение. Посмотрим, что они говорят о подходе, который, как им кажется, даст желаемый результат:

---

[1] *Erich Gamma* Design Patterns: Elements of Reusable Object-Oriented Software. — New York: Addison-Wesley, 1994. — 87.

Мэри: А давай создадим статическую фабрику. Я покажу тебе:

```
public static class ProductRepositoryFactory
{
    public static ProductRepository Create()
    {
        string connectionString =
              ConfigurationManager.ConnectionStrings
              ["CommerceObjectContext"].ConnectionString;
        return new SqlProductRepository(connectionString);
    }
}
```

Теперь, поскольку класс сделан статическим, нам не нужно думать о том, как создавать его.

Йенс: Но у нас остается жестко закодированным создание экземпляров SqlProduct-Repository, поэтому будет ли он полезен нам?

Мэри: Мы можем работать с ним, используя параметры конфигурации, которые будут определять, какой тип ProductRepository должен быть создан. Что-то вроде:

```
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("...");
    }
}
```

Видишь? Таким образом, мы можем определить, на чем должна быть основана наша реализация: на технологии SQL Server или на Windows Azure, и нам не потребуется заново компилировать приложение, чтобы производить смену одного на другое.

Йенс: Здорово! Это то, что нам нужно. Твой консультант сейчас должен быть просто счастлив!

Имеется несколько причин, по которым такая статическая фабрика не может считаться удовлетворительным решением изначальной цели, то есть не подходит для программирования через интерфейсы. Рассмотрим граф зависимостей, представленный на рис. 5.2.

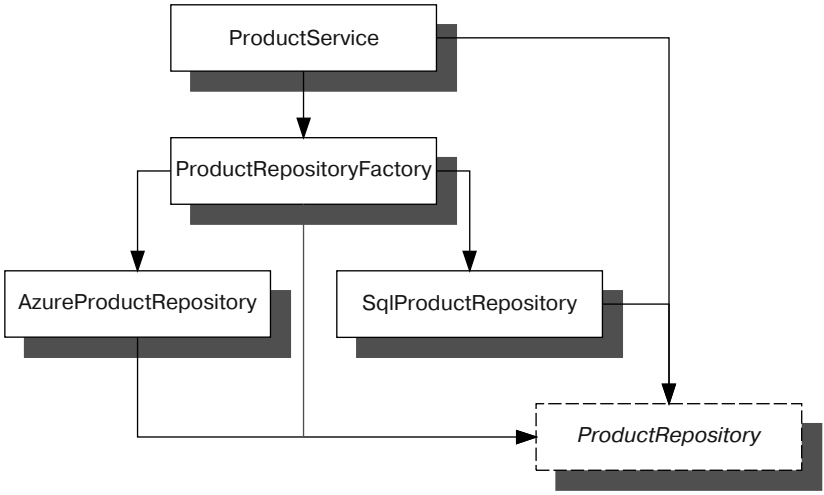


Рис. 5.2. Граф зависимостей для предлагаемого решения: статическая фабрика ProductRepositoryFactory используется для создания экземпляров ProductRepository

Я НЕ СПРАВИЛСЯ С ЭТИМ

Если бы я был консультантом в рассмотренном примере, я вовсе не был бы счастлив. Фактически именно такое «решение» было принято в проекте, участником которого я был, и результатом моего труда был 14-страничный документ, описывавший, почему это не может работать и что должно быть сделано вместо этого.

Это был по-настоящему большой проект, касавшийся основного бизнеса компании, входящей в список лучших по версии Fortune 500, так что верное членение приложения на модули было важно из-за высокой сложности приложения. К сожалению, я подключился к проекту слишком поздно и мои предложения были отклонены, поскольку они требовали радикальных изменений в уже написанном программном коде.

Я перешел на другой проект, но позднее узнал, что, несмотря на достаточно солидный объем работы, выполненной командой (проект мог даже показаться завершенным), его тем не менее признали проваленным и руководство было заменено.

У меня нет оснований утверждать, что проект провалился из-за того, что в нем не было реализовано внедрение зависимостей, но выбранный для проекта подход имел все признаки отсутствия хорошего проектирования. Я не могу сказать, что крах проекта меня удивил.

Вот список файлов, которые должны иметь ссылки на абстрактный класс `ProductRepository`:

- `ProductService`, поскольку он использует экземпляры `ProductRepository`;
- `ProductRepositoryFactory`, поскольку он создает экземпляры `ProductRepository`;
- `AzureProductRepository` и `SqlProductRepository`, так как они реализуют `ProductRepository`.

`ProductRepositoryFactory` зависит и от класса `AzureProductRepository`, и от класса `SqlProductRepository`. Так как `ProductService` напрямую зависит от `ProductRepositoryFactory`, он также оказывается зависимым и от этих двух конкретных реализаций.

КОЛЛАПС ЗАВИСИМОСТЕЙ

Ситуация, когда абстракция `ProductRepository` и использующий ее сервис `ProductService` определены в одной и той же сборке (а именно так обстоит дело с реализациями, создававшимися в книге до сих пор), является вырожденным случаем. Давайте предположим, что такая сборка называется `Domain Model`. В таком случае и `ProductRepositoryFactory` тоже должна находиться в этой же сборке — в противном случае мы получим циклическую ссылку, что является недопустимым.

Однако фабрика содержит ссылки на обе реализации, и они имеют ссылки на сборку `Domain Model`, так как реализуют класс `ProductRepository`. Поэтому, чтобы избежать циклических ссылок, необходимо поместить конкретные реализации в эту же сборку.

Но наличие реализаций `AzureProductRepository` и `SqlProductRepository` в сборке `Domain Model` полностью противоречит принципу разделения обязанностей. По существу мы получаем монолитное приложение.

Единственный способ разрешения данной проблемы заключается в том, чтобы поместить абстракцию `ProductRepository` в отдельную сборку. Это решение является неплохим и по ряду других соображений, но тем не менее мы не можем признать «Статическую фабрику» пригодной для реализации внедрения зависимостей.

Вместо того чтобы получить слабо связанные реализации `ProductRepository`, Мэри и Йенс сделали сильно связанные модули. Хуже того, фабрика будет всегда вовлекаться во все реализации — даже в те, в которых она вовсе не нужна.

Если Мэри и Йенсу когда-нибудь понадобится третий тип `ProductRepository`, им придется изменить фабрику и перекомпилировать приложение. Хотя их решение может конфигурироваться, оно не является расширяемым.

Наконец, невозможно заменить конкретные реализации `ProductRepository` на такие, которые будут пригодны для тестирования. Например, здесь не подойдут автоматически сгенерированные динамические подстановочные объекты (Dynamic Mocks), поскольку это потребует задания экземпляра `ProductRepository` во время исполнения вместо статического использования конфигурационного файла во время разработки.

ПРИМЕЧАНИЕ

Динамические подстановочные объекты (Dynamic Mocks) не являются темой книги, но я немного касался их, когда говорил о пригодности для тестирования в главе 1 (см. подраздел 1.2.2).

Таким образом, может показаться, что «Статическая фабрика» решает проблему, хотя на самом деле она лишь усугубляет ее. В лучшем случае с ней вы придете к использованию нестабильных зависимостей.

Теперь, когда вы рассмотрели достаточное количество примеров «Диктатора», я надеюсь, вы понимаете, что нужно искать: места, где ключевое слово `new` используется вместе с зависимостями. Так вы сможете обойти большинство известных ловушек; но если вы хотите избавиться от уже имеющегося в вашем коде образцов этого антипаттерна, следующий подраздел описывает, как это делается.

5.1.3. Анализ

«Диктатор» — это антипод инверсии управления. Когда мы напрямую управляем созданием нестабильных зависимостей, мы получаем в результате сильно связанный

код, в котором отсутствуют многие (если не все) достоинства слабого связывания, перечисленные в главе 1.

Влияние

В сильно связанном коде, который возникает при применении «Диктатора», теряются многие преимущества модульного дизайна.

- Хотя у нас остается возможность конфигурирования приложения для применения одной из многих заранее сконфигурированных зависимостей, мы не можем заменить ни одну из них. Невозможно подключить реализацию, созданную после компиляции приложения, и, как правило, нельзя использовать конкретные экземпляры как реализации.
- Становится трудно повторно применять пользовательские модули, поскольку они переносятся вместе со своими зависимостями, которые могут оказаться нежелательными в новом контексте.
- Параллельная разработка становится более трудной, поскольку пользовательское приложение сильно связано со всеми реализациями своих зависимостей.
- Пригодность к тестированию снижается, поскольку динамические подстановочные объекты не годятся в качестве замены зависимостей.

Если проектирование осуществляется тщательно, у нас все еще остается возможность реализовать сильно связанные приложения с четко ограниченными зонами ответственности, так что возможность поддержки не ухудшается. Но даже в этом случае цена такого решения оказывается слишком высокой. Нам нужно уйти от «Диктатора» и добиться корректной реализации внедрения зависимостей.

Рефакторинг в направлении внедрения зависимостей

Чтобы избавиться от «Диктатора», необходимо произвести рефакторинг кода с ориентацией на один из шаблонов внедрения зависимостей, представленных в главе 4. Прежде всего нужно воспользоваться следующим руководством, чтобы определить, какой из шаблонов подходит больше всего. В большинстве случаев это будет внедрение конструктора. Шаги рефакторинга описаны ниже.

1. Убедитесь, что вы программируете в соответствии с интерфейсом. В приведенных примерах это было заложено изначально; но в других ситуациях вам может понадобиться выделить интерфейсы и поменять объявления переменных.
2. Если вы создаете независимые реализации одной и той же зависимости в разных местах, переместите их все в один-единственный метод создания зависимости. Убедитесь, что данный метод возвращает абстракцию, а не конкретный тип.
3. Теперь, когда у вас имеется единственное место, где создается экземпляр, выведите его из пользовательского класса, реализуя один из шаблонов внедрения зависимостей, такой как внедрение конструктора.

В случае примеров `ProductService` в предыдущих разделах внедрение конструктора является самым лучшим решением:

```
private readonly ProductRepository repository;

public ProductService(ProductRepository repository)
{
    if (repository == null)
    {
        throw new ArgumentNullException("repository");
    }

    this.repository = repository;
}
```

В некоторых случаях оригинальный код использует сложную логику для определения того, как следует создавать экземпляры зависимостей. В таких случаях эта сложная логика может быть реализована в фабрике, и мы можем затем использовать интерфейс этой фабрики для создания «Абстрактной фабрики». Это означает, что зависимость изменяется, и теперь зависимостью становится «Абстрактная фабрика» вместо первоначальной абстракции, и та же логика рефакторинга может теперь быть применена к фабрике. В большинстве случаев дело закончится внедрением фабрики в потребляющий класс через его конструктор.

«Диктатор» — это самый распространенный антипаттерн внедрения зависимостей. Он представляет собой используемый по умолчанию способ создания экземпляров в большинстве языков программирования, так что его можно встретить и в приложениях, разработчики которых никогда и не слышали о внедрении зависимостей. Это настолько естественный и глубоко укоренившийся способ создания новых объектов, что многим разработчикам тяжело расстаться с ним. Даже когда разработчики начинают интересоваться внедрением зависимостей, у многих начинается тяжелая ломка стереотипов и приходит понимание того, что они почему-то должны сами контролировать, когда и где создаются экземпляры. Такая ломка может протекать не без труда; но даже если она вам удастся, вас могут ожидать другие, хотя и более мелкие, ловушки.

«Диктатор» — наиболее разрушительный антипаттерн, но даже если вам удастся удержать его под контролем, могут возникнуть более тонкие проблемы. В следующих разделах рассматриваются другие антипаттерны. И поскольку они не так опасны, как «Диктатор», разрешать их тоже не так сложно. Будьте бдительны и устраняйте их сразу, как только обнаружите.

5.2. «Гибридное внедрение»

Перегрузка конструктора часто встречается в кодах .NET (в том числе в библиотеке BCL). Часто код перегрузки обеспечивает наличие подходящих значений по умолчанию для одного или двух полнофункциональных конструкторов, получающих на входе все определенные параметры.

Когда речь заходит о внедрении зависимостей, мы можем встретить и другие способы использования перегрузки. Так, антипаттерн сплошь и рядом определяет предназначенный для использования при тестировании перегруженный конструктор,

позволяющий явно определить зависимость, тогда как конструктор по умолчанию используется в рабочем коде.

Это может оказаться вредным, когда используемая по умолчанию реализация зависимости представляет внешнее, а не локальное умолчание.

Когда мы полностью задействуем внедрение зависимостей, такие перегруженные конструкторы становятся в лучшем случае избыточными. Если учитывать возможность возникновения негативных последствий, их лучше избегать.

5.2.1. Пример: ProductService с внешним умолчанием

Когда Мэри реализовала первую версию своего класса `ProductService` (в главе 2), она предполагала наличие лишь одной зависимости: реализацию, работающую с SQL Server. Изначально предполагалось, что класс `SqlProductRepository` останется единственной реализацией класса `ProductRepository`, поэтому предполагалось использовать его по умолчанию.

ВНЕШНЕЕ УМОЛЧАНИЕ

Внешнее умолчание является противоположностью локального умолчания. Это такая реализация зависимости, которая используется как значение по умолчанию, даже если оно определено в другом модуле, чем применяющий его класс.

В качестве примера рассмотрим реализации класса `Repository`, которые встречались нам в демоприложении в предыдущих главах. Сервис, такой как `ProductService`, требует для работы наличия экземпляра `ProductRepository`. Во многих случаях, когда разрабатываются такие приложения, у нас имеется на примете подходящая реализация: это обычно та, что реализует необходимую функциональность путем чтения данных и записи их в реляционную базу. Обычно такое решение хорошо подходит на роль реализации по умолчанию.

Проблема заключается в том, что реализация по умолчанию, которую мы собираемся использовать (`SqlProductRepository`), определяется в другом модуле, не в том, где находится `ProductService`. Это приводит к созданию нежелательной зависимости в модуле `CommerceSqlDataAccess`, как показано ниже.

ProductService использует SqlProductRepository в качестве реализации по умолчанию, из-за чего мы вводим жесткую ссылку на модуль CommerceSqlDataAccess, хотя это нежелательно

Необходимость переноса в другие приложения нежелательных модулей лишает нас многих преимуществ слабого связывания, которое обсуждалось в главе 1. Повторное использование модуля `CommerceDomain` становится более сложным, так как это вынуждает включать в приложение еще и модуль `CommerceSqlDataAccess`, а мы, возможно, не хотели бы его появления в другом контексте. Усложняется также и параллельная разработка, поскольку класс `ProductService` теперь напрямую зависит от класса `SqlProductRepository`.

Это основные причины, по которым следует избегать внешних умолчаний, если это возможно.

Мэри еще не готова использовать внедрение конструктора, так как ей сложно выяснить, где будет совершаться компоновка объектов. Она также должна учитывать концепцию корня компоновки.

Приглашенный консультант порекомендовал ей применить внедрение конструктора в `ProductService`, но Мэри все еще считает, что она должна создавать новый экземпляр этого класса классическим способом:

```
var productService = new ProductService();
```

Чтобы выполнить эту задачу, Мэри добавила следующий код в класс `ProductService` (листинг 5.2).

Листинг 5.2. `ProductService` с гибридным внедрением

```
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);
}
```

1 Конструктор по умолчанию

2 Внедрение конструктора

Класс `ProductService` теперь имеет конструктор по умолчанию (1), вызывающий другой конструктор, используя внешнее умолчание.

Этот другой конструктор корректно реализует шаблон «Внедрение конструктора», используя блок защиты и затем сохраняя инжектированный экземпляр `ProductRepository` в поле, объявленное «только для чтения» (2). Конструктор по умолчанию вызывает этот конструктор, применяя внешнее умолчание, созданное приватным методом `CreateDefaultRepository`. Класс `SqlProductRepository` выступает в качестве внешнего умолчания, поскольку он определен не в той сборке, что класс `ProductService`. Это тесно связывает сборку, содержащую класс `ProductService`, со сборкой, которая включает в себя класс `SqlProductRepository`.

ProductService может быть повторно использован с другими типами-наследниками ProductRepository путем передачи этого сервиса через конструктор с более гибкой перегрузкой. Мэри не сможет перехватывать экземпляр ProductRepository в своем приложении, если она по-прежнему будет применять только конструктор по умолчанию.

5.2.2. Анализ

Гибридное внедрение чаще всего возникает в тех случаях, когда разработчики пытаются сделать свои классы пригодными для тестирования, не понимая всех тонкостей внедрения зависимостей. При написании модульного теста для класса оказывается важным, можете ли вы заменить нестабильную зависимость на ***тестовый двойник***, чтобы можно было корректно изолировать тестируемую систему (System Under Test, SUT) от ее зависимостей; внедрение зависимостей позволяет нам делать именно это.

Хотя «Гибридное внедрение» оставляет приложение пригодным для тестирования, оно вызывает некоторые нежелательные последствия.

КОНКРЕТНЫЙ ПРИМЕР: ASP.NET MVC

Когда вы создаете новый проект ASP.NET MVC, автоматически создается несколько шаблонных классов типа Controller. Одни из них — класс AccountController, использующий гибридное внедрение. Это поясняется в комментариях к коду:[1]

```
// Этот конструктор используется фреймворком MVC для создания
// экземпляра контроллера с применением используемых по умолчанию форм
// провайдеров аутентификации и членства.
public AccountController()
    : this(null, null)
{
}
// Этот конструктор не используется фреймворком MVC, а определен
// для облегчения модульного тестирования.
// Для получения дополнительной информации см. комментарии
// в конце этого файла.
public AccountController(IFormsAuthentication formsAuth,
    IMembershipService service)
{
    this.FormsAuth =
        formsAuth ?? new FormsAuthenticationService();
    this.MembershipService =
        service ?? new AccountMembershipService();
}
```

[1] Я переформатировал данный код так, чтобы длина строк согласовывалась с форматом книги. Кроме того, я еще добавил ключевое слово this, чтобы повысить наглядность кода. Больше я не вносил никаких изменений. Вы можете найти данный код в загружаемых материалах для этой книги — там я оставил первоначальный код неизменным.

Я утверждаю здесь, что гибридное внедрение плохо, но ведь сама компания Microsoft использует и одобряет его! В данном случае причина кроется исключительно в желании обеспечить пригодность для тестирования, и гибридное внедрение адекватно решает эту задачу. Оно просто не решает другие задачи обеспечения модульности, такие как способность заменять и повторно использовать модули и вести параллельную разработку.

Другие специалисты придерживаются такой же точки зрения. Айенд Рахьен разместил в блоге следующую заметку, касающуюся приложения, разработанного на платформе фреймворка ASP.NET MVC:

«По моему мнению, если вы хотите реализовать "инверсию управления для бедных", делайте это. Но, пожалуйста, не создавайте этого несчастного гибрида (Bastard Child)[1]*».*

Именно поэтому я назвал такой антипаттерн «Гибридным внедрением».

Влияние

Основная проблема, связанная с гибридным внедрением, — это использование внешнего умолчания. Хотя пригодность для тестирования обеспечивается, мы не можем в дальнейшем свободно переиспользовать такой класс, поскольку он тянет за собой нежелательные зависимости. Усложняется также параллельная разработка, поскольку класс сильно зависит от своих зависимостей.

Кроме последствий, которые гибридное внедрение вызывает для модульности приложения, следует отдельно остановиться на существовании нескольких конструкторов, что также представляет проблему. Когда имеется только один конструктор, контейнер внедрения зависимостей может автоматически подключить все зависимости, поскольку в такой ситуации всегда ясно, какой конструктор использовать.

Когда же существует несколько конструкторов, выбор между ними становится неоднозначным. Контейнер внедрения зависимостей должен использовать какую-то эвристику, чтобы принять решение о выборе конструктора — или он может отказаться. По иронии судьбы это один из немногих сценариев, который почти не оказывает негативного влияния на «внедрение зависимостей для бедных», поскольку мы можем решить сами, когда следует вручную подключить зависимости.

Среди различных антипаттернов внедрения зависимостей «Гибридное внедрение» не настолько разрушительно, как «Диктатор», и от него намного проще избавиться.

Рефакторинг в направлении внедрения зависимостей

Гибридное внедрение часто возникает как результат неправильной попытки реализовать внедрение зависимостей. Его достоинством является то, что оно основано на программировании через интерфейсы, поэтому его легко исправить, выполняя рефакторинг в направлении выбранного паттерна внедрения зависимостей.

---

[1] *Ayende Rahien* Reviewing NerdDinner, 2009: http://ayende.com/Blog/archive/2009/07/30/reviewing-nerddinner.aspx.

СОВЕТ

Даже если вы думаете, что влияние гибридного внедрения вполне безобидно, все же следует выполнить его рефакторинг в направлении корректного шаблона внедрения зависимостей. Это настолько просто, что оставлять гибридные внедрения без внимания просто преступно.

Первый шаг состоит в выборе наиболее подходящего шаблона внедрения зависимостей в качестве цели. Рисунок 5.3 иллюстрирует простой процесс принятия решений. Если применяемое значение по умолчанию является внешним, то лучше всего воспользоваться внедрением конструктора. В противном случае хорошей альтернативой будет внедрение свойства.

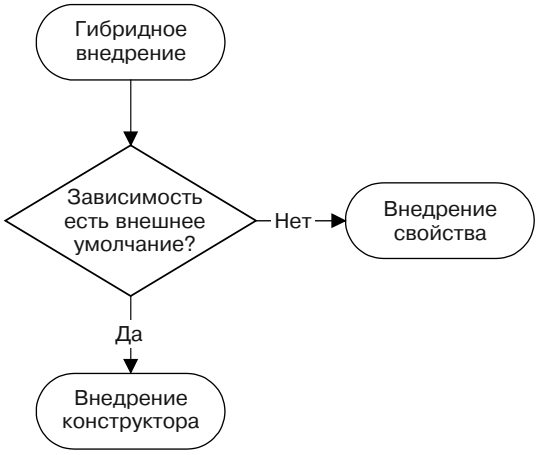


Рис. 5.3. Когда производится рефакторинг гибридного внедрения, для принятия решения следует определить, является ли зависимость локальным или внешним умолчанием

Во многих случаях значение умолчания, используемое конструктором по умолчанию, является внешним. В таких случаях лучше всего задействовать внедрение конструктора, потому что оно простое в реализации хорошо подходит для использования с зависимостями любого вида. Конструктор, получающий зависимость как параметр, уже существует, так что единственное изменение, которое придется сделать в классе-клиенте, — убрать применяемый по умолчанию конструктор.

Из-за этого, несомненно, возникнут ошибки компиляции, но в этой точке мы можем положиться на компилятор и переместить весь код, создающий заявленный в запросе класс, в корень компоновки.

Когда используемое по умолчанию значение представляет локальное умолчание, ситуация очень напоминает основной сценарий, применяемый во внедрении свойства. Хотя механизмы отличаются, структура одинакова: в обоих случаях имеются подходящие локальные умолчания, но мы еще хотим, чтобы оставалась возможность расширения класса-клиента.

Это вырожденный случай гибридного внедрения, в котором его влияние оказывается намного менее тяжелым. Поскольку значение по умолчанию является локальным умолчанием, степень сочетаемости класса не снижается; единственное

негативное последствие состоит в том, что наличие нескольких конструкторов делает автоподключение более сложным.

В таком случае мы должны реализовать внедрение свойства, убрав конструктор, получающий зависимость как параметр, и заменив ее на свойство с возможностью записи. Если такие изменения приведут к возникновению ошибок компиляции, мы можем опять положиться на компилятор и переместить код создания в корень компоновки.

В корне компоновки имеется множество способов подключения зависимостей — включая некоторые, очень далекие от идеала, как, например, следующий антипаттерн.

5.3. «Ограниченное конструирование»

Наибольшей проблемой при правильной реализации внедрения свойств является перемещение всех классов, имеющих зависимости, в корень композиции. Когда это будет выполнено, мы сможем сказать, что уже прошли длинную дорогу.

И даже в этом случае останутся еще ловушки, которые нужно будет обнаружить. Распространенная ошибка — требование, чтобы все зависимости имели конструктор с определенной сигнатурой. Это вытекает из желания обеспечить динамическое связывание, чтобы зависимости можно было бы определять во внешнем файле конфигурации и, следовательно, изменять без перекомпиляции приложения.

ПРИМЕЧАНИЕ

Так называемый шаблон Provider[1], используемый в ASP.NET, является примером ограниченного конструирования, поскольку провайдеры должны иметь конструктор по умолчанию. Обычно эта ситуация усугубляется из-за попыток конструктора провайдера читать информацию из файла конфигурации приложения. Часто конструктор порождает исключительную ситуацию, если требуемый раздел конфигурации оказывается недоступным.

Этот раздел касается только таких сценариев, в которых требуется позднее связывание. В сценариях, где мы прямо создаем ссылки на все зависимости в корне компоновки, такой проблемы не возникает — но в данной ситуации у нас нет возможности при необходимости заменять зависимости без перекомпиляции.

В главе 3 мы кратко затрагивали эту тему. В данном разделе она раскрывается более подробно.

5.3.1. Пример: динамическое связывание ProductRepository

В нашем демоприложении некоторые классы зависят от абстрактного класса `ProductRepository`. Это означает, что для создания этих классов прежде всего требуется создать экземпляр `ProductRepository`. Вы уже знаете, что корень компоновки является самым подходящим местом для этого. В приложениях ASP.NET в качестве

[1] *Rob Howard* Provider Model Design Pattern and Specification, Part 1, 2004: http://msdn.microsoft.com/en-us/library/ms972319.aspx.

корня компоновки выступает файл Global.asax; в листинге 5.3 представлен фрагмент кода, в котором создается экземпляр ProductRepository.

Листинг 5.3. Явные ограничения в конструкторе ProductRepository

```
string connectionString =
    ConfigurationManager.ConnectionStrings
    ["CommerceObjectContext"].ConnectionString;

string productRepositoryTypeName =
    ConfigurationManager.AppSettings
    ["ProductRepositoryType"];
var productRepositoryType =
    Type.GetType(productRepositoryTypeName, true);
var repository =
    (ProductRepository)Activator.CreateInstance(
    productRepositoryType, connectionString);
```

❶ Создание экземпляра конкретного типа

Первое, что вызывает здесь подозрение, — тот факт, что строка соединения читается из файла web.config. Зачем нужна строка соединения, если планируется, что ProductRepository будет абстракцией? Хотя это и маловероятно, но вы можете захотеть реализовать ProductRepository для работы со встроенной базой данных или с XML-файлом. Выбор веб-сервиса RESTful (с передачей состояния представления), такого как Windows Azure Table Storage Service, представляется более вероятной альтернативой, но все же наиболее реальным кандидатом остается реляционная база данных. Базы данных очень легко доступны, и поэтому вполне можно забыть, что строка соединения явно определяет вариант базы данных, необходимый для данной реализации.

Чтобы обеспечить динамическое связывание ProductRepository, нужно также определить, какой тип будет выбран в качестве реализации. Это можно сделать путем чтения определенного с учетом сборки (квалифицированного на уровне сборки) имени типа из файла web.config и созданием экземпляра Type из этого имени.

Сам по себе этот шаг несложен — трудности возникают, когда вам требуется создать экземпляр такого типа.

Имея Type, вы можете создать его экземпляр с использованием класса Activator. Метод CreateInstance вызывает конструктор типа, поэтому необходимо передать корректные параметры конструктора, чтобы предотвратить генерацию исключительного события. Для этого применяется строка соединения ❶.

Если вы ничего не знаете о приложении, кроме фрагмента кода из листинга 5.3, вы должны сейчас удивиться тому, что строка соединения передается как аргумент конструктора в неизвестный тип. Это оказывается бессмысленным, если реализация основана на веб-сервисе REST или на XML-файле.

И в самом деле, это не имеет смысла, поскольку представляет несущественное ограничение конструктора зависимости. В таком случае вы имеете неявное требование, чтобы любая реализация ProductRepository имела конструктор содним строковым параметром на входе. Это помимо явного ограничения, что класс должен наследоваться из ProductRepository.

ПРИМЕЧАНИЕ

Неявное ограничение, что конструктор должен иметь один строковый параметр, оставляет нам огромную степень свободы, так как мы можем закодировать в строке различную информацию и декодировать ее позже для использования в самых разных целях. Представьте, наоборот, что параметры конструктора ограничены типами TimeSpan и «число», и можете себе вообразить, насколько узки будут возможности вашей реализации.

Вы можете возразить, что `ProductRepository`, использующий XML-файл, тоже может потребовать строку (`string`) в качестве параметра конструктора, хотя эта строка может определять имя файла, а не строку соединения. Однако концептуально это остается странным, поскольку вы можете определить это имя файла в элементе `connectionStrings` файла `web.config` (и в моем случае, я думаю, что гипотетический `XmlProductRepository` должен принимать `XmlReader` в качестве аргумента конструктора вместо имени файла).

Моделирование конструкторов зависимостей с использованием преимущественно явных ограничений (интерфейса или родительского класса) является гораздо более выгодным и гибким вариантом.

5.3.2. Анализ

В предыдущем примере неявное ограничение требовало, чтобы реализации имели конструктор с одним параметром типа `string`. Более общее ограничение состоит в том, что все реализации должны иметь конструктор по умолчанию, так что простейшая форма `Activator.CreateInstance` будет работать:

```
var dep = (ISomeDependency)Activator.CreateInstance(type);
```

И хотя можно сказать, что это простейшее подходящее решение, мы слишком много теряем в отношении гибкости, и нас это не устраивает.

Влияние

Независимо от того, каким образом мы ограничиваем конструирование объекта, мы теряем гибкость. Соблазнительно было бы утверждать, что все реализации зависимостей должны иметь конструкторы по умолчанию — они могут выполнять свою инициализацию внутри, например читая конфигурационные данные, такие как строки конфигурации, прямо из файла `.config`. Однако это ограничит нас с другой стороны, поскольку мы, возможно, захотим создать приложение в виде многоуровневой структуры, где в одних экземплярах будут инкапсулированы другие экземпляры. В некоторых случаях мы, возможно, захотим совместно применять и разделять экземпляр между разными пользователями, что проиллюстрировано на рис. 5.4.

Если одна и та же зависимость требуется в нескольких классах, мы, возможно, захотим использовать один экземпляр зависимости между всеми требующими ее классами. Это возможно, только если мы внедрим этот экземпляр извне. Хотя мы можем написать в каждом из этих классов код для чтения информации о типе из файла конфигурации и применения `Activator.CreateInstance` для создания экземпляра корректного типа, мы так и не сможем совместно использовать экземпляр.

Вместо этого мы получим несколько экземпляров одного и того же класса, что приводит к расходу лишней памяти.

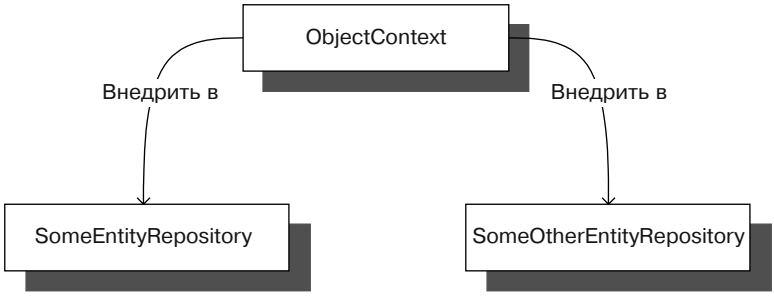


Рис. 5.4. В этом примере мы хотим создать единственный экземпляр класса ObjectContext и внедрить один и тот же экземпляр в оба Repository. Это будет возможно, только если мы можем внедрить экземпляр извне

ПРИМЕЧАНИЕ

То, что внедрение зависимостей позволяет нам совместно использовать один экземпляр между несколькими потребителями, не означает, что мы всегда должны так поступать. Разделение экземпляра сохраняет память, но может привести к появлению проблем взаимодействия, в частности, проблем с потоками. Вопрос — захотим ли мы разделять экземпляр — тесно связан с концепцией жизненного цикла объекта, обсуждаемой в главе 8.

Вместо введения неявных ограничений на способы создания объектов, следует реализовать корень компоновки таким образом, чтобы он мог работать с любым конструктором или методом фабрики, который мы поместим в него.

Рефакторинг в направлении внедрения зависимостей

Что мы должны будем сделать с конструкторами компонентов, не имеющими ограничений, если нам необходимо динамическое связывание? Может показаться заманчивым использовать абстрактную фабрику, которая может применяться для создания экземпляров требуемой абстракции, и затем выставить требование, чтобы реализации такой абстрактной фабрики имели конструкторы по умолчанию. Но, скорее всего, так мы только отложим существующие проблемы, не решая их.

ПРЕДУПРЕЖДЕНИЕ

Хотя абстрактные фабрики могут использоваться для успешной реализации динамического связывания, реализация такого подхода потребует строгой дисциплины. Как правило, выбирается решение на основе контейнера внедрения зависимостей; но я все же обрисую, как реализовать это более трудным способом.

Кратко рассмотрим такой подход. Представьте, что у вас имеется абстракция некоторого воображаемого сервиса, пусть его имя будет `ISomeService`. Согласно схеме абстрактной фабрики, вам необходим также интерфейс `ISomeServiceFactory`. Рисунок 5.5 иллюстрирует данную структуру. `ISomeService` представляет реальную зависимость. Однако, чтобы сделать реализующие ее механизмы независимыми от

неявных ограничений, вы пытаетесь решить проблему динамического связывания путем введения фабрики ISomeServiceFactory, которая будет использоваться для создания экземпляров ISomeService. При этом остается требование, чтобы любая фабрика имела конструктор по умолчанию.

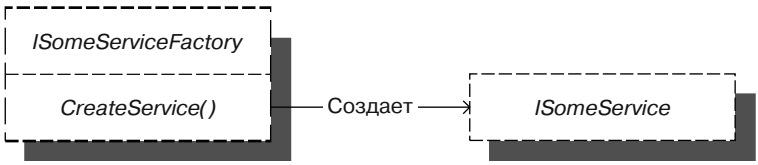


Рис. 5.5. ISomeService представляет реальную зависимость

Теперь предположим, что вы хотите использовать реализацию ISomeService, что, в свою очередь, требует для работы наличия работоспособного экземпляра ISomeRepository, что продемонстрировано в листинге 5.4.

Листинг 5.4. SomeService, использующий 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, ISomeServiceFactory оказывается весьма уместной.

Сейчас вы хотите использовать реализацию ISomeRepository, построенную на основе фреймворка Entity. Вы назвали эту реализацию SomeEntityRepository, и она реализована не в той же сборке, что SomeService.

Поскольку вы не хотите втягивать в будущем в другие реализации ссылку на библиотеку EntityDataAccess вместе с SomeService, единственным решением будет реализовать SomeServiceFactory в иной сборке, нежели SomeService, что и показано на рис. 5.6.

Хотя ISomeService и ISomeServiceFactory выглядят как связанная пара, важно реализовать их в двух различных сборках, так как фабрика должна иметь ссылки на все зависимости, чтобы иметь возможность корректно подключить их вместе.

По соглашению, реализация ISomeServiceFactory имеет конструктор по умолчанию, так что вы можете задать привязанное к сборке (Assembly-Qualified) имя типа в файле .config и использовать Activator.CreateInstance для создания экземпляра. Всякий раз, когда вам потребуется подключить вместе новую комбинацию зависимостей, вы должны будете реализовать новый экземпляр ISomeServiceFactory, связывающий именно такую комбинацию, и затем сконфигурировать приложение, чтобы оно использовало эту фабрику вместо определенной ранее. Это означает, что вы не сможете определить произвольные комбинации зависимостей без написания

и компиляции нового кода, но вы можете делать это, не перекомпилируя повторно все приложение в целом.

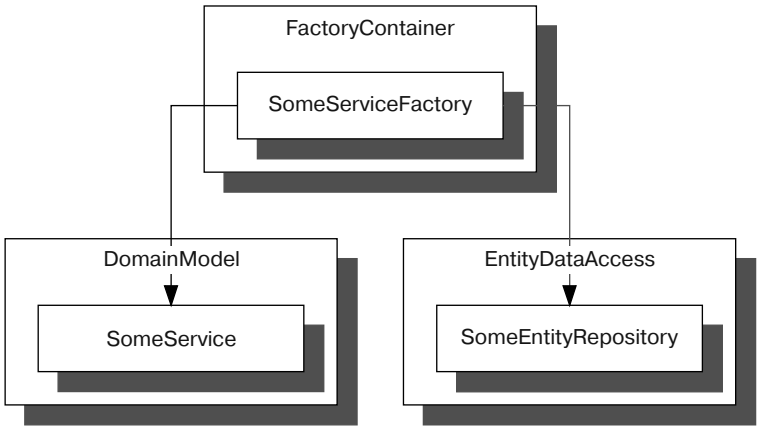


Рис. 5.6. Класс SomeServiceFactory должен быть реализован в другой сборке по сравнению с SomeService, чтобы избежать связывания библиотеки DomainModel с библиотекой EntityDataAccess

По существу, абстрактная фабрика таким образом превращается в абстрактный корень компоновки, который определяется в сборке отдельно от основного приложения. Хотя, в общем-то, это жизнеспособный подход, обычно оказывается, что намного проще использовать универсальный контейнер внедрения, который может делать то же самое, но без использования файлов конфигурации.

Антипаттерн «Ограниченное конструирование» реально применяется, только когда реализуется динамическое связывание, поскольку когда реализуется раннее (статическое) связывание, сам компилятор гарантирует, что мы никогда не сможем использовать неявные ограничения способов конструирования компонентов.

Последний рассматриваемый здесь паттерн используется гораздо более широко — многие даже считают его не антипаттерном, а вполне безобидным законным приемом.

5.4. «Локатор сервисов»

Возможно, кому-то будет тяжело отказаться от идеи прямого управления зависимостями, поэтому многие разработчики выводят статические фабрики (описанные в подразделе 5.1.2) на новые уровни. Так образуется антипаттерн «Локатор сервисов» (Service Locator).

ПРЕДУПРЕЖДЕНИЕ

Отнесение «Локатора сервисов» к категории антипаттернов является спорным вопросом. Некоторые разработчики считают его нормальным паттерном, тогда как другие (включая меня) относят

его к антипаттернам[1]. В этой книге я решил описать его как антипаттерн, потому что считаю его недостатки более весомыми, чем его достоинства; но не удивляйтесь, если у других авторов вы найдете хвалебные отзывы о нем. Важно разобраться в его достоинствах и недостатках настолько, чтобы уметь самостоятельно принять решение.

«Локатор сервисов» был описан как шаблон проектирования Мартином Фаулером в 2004-м[2], поэтому объявление его антипаттерном — довольно смелый шаг. Если говорить кратко, он представляет статическую фабрику, в которую добавлена возможность внедрять сервисы.

ПРИМЕЧАНИЕ

Термин «сервис» в контексте данного раздела полностью эквивалентен термину «зависимость».

В наиболее часто встречающемся варианте реализации «Локатор сервисов» является статической фабрикой[3], которая может быть сконфигурирована с конкретными сервисами до того, как первый пользователь начнет использовать ее (рис. 5.7). Предполагаемым местом конфигурирования является корень компоновки. В зависимости от конкретной реализации, «Локатор сервисов» может конфигурироваться в коде, из файла конфигурации или любой комбинацией этих способов. Первоочередной задачей «Локатора сервисов» является предоставление экземпляров сервисов по запросам пользователей. Consumer использует интерфейс IService и запрашивает экземпляр у «Локатора сервисов», который возвращает экземпляр той конкретной реализации, на возврат которой он сконфигурирован.

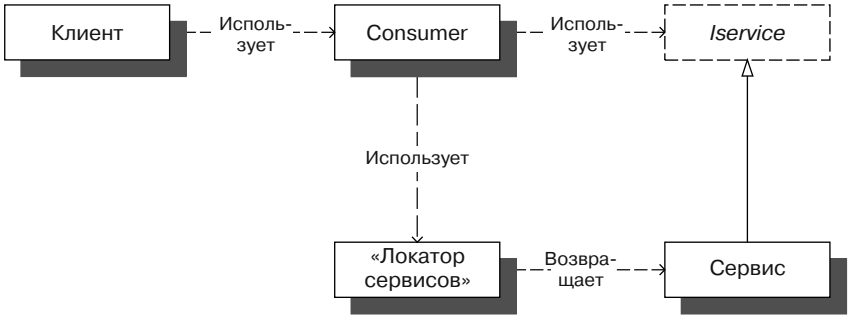


Рис. 5.7. Реализация «Локатора сервисов»

[1] *Daniel Cazzulino* What is all the fuzz about the new common IServiceLocator? 2008: http://www.clariusconsulting.net/blogs/kzu/archive/2008/10/03/WhatisallthefuzzaboutthenewcommonIServiceLocator.aspx.

Nicholas Blumhardt Container-Managed Application Design, Prelude: Where does the Container Belong? 2008: http://blogs.msdn.com/b/nblumhardt/archive/2008/12/27/container-managed-application-design-prelude-where-does-the-container-belong.aspx.

[2] *Martin Fowler* Inversion of Control Containers and the Dependency Injection pattern, 2004: http://martinfowler.com/articles/injection.html.

[3] Дополнительная информация: *Mark Seemann* Service Locator is an Anti-Pattern, 2010: http://blog.ploeh.dk/2010/02/03/ServiceLocatorIsAnAntiPattern.aspx.

МОЙ ОПЫТ РАБОТЫ С «ЛОКАТОРОМ СЕРВИСОВ»

Я использовал «Локатор сервисов» в течение нескольких лет, прежде чем отказался от него. Хотя я не помню, когда впервые прочитал статью Фаулера, она подсказала мне потенциальное решение проблемы, над которой я бился уже некоторое время: как внедрить зависимости.

Мне показалось, что шаблон «Локатор сервисов» решает все мои проблемы, и я быстро приступил к разработке «Локатора сервисов» для первой версии Microsoft patterns & practices Enterprise Library. Она размещалась на теперь уже не существующем сайте GotDotNet. Хотя у меня сохранился исходный код, моя история релизов была утеряна, когда GotDotNet прекратил свое существование; поэтому я не могу сказать с уверенностью, но припоминаю, что первая версия была выложена в середине 2005 года.

В 2007-м я реализовал полностью переделанную версию, названную Enterprise Library 2. Она доступна и сейчас на CodePlex, но давно уже заброшена, потому что вскоре после этого я пришел к убеждению, что «Локатор сервисов» в действительности является антипаттерном.

Как вы можете убедиться, я несколько лет интенсивно использовал этот шаблон, прежде чем осознал его недостатки и понял, что имеются лучшие альтернативы. Поэтому я хорошо понимаю, почему так много разработчиков находят его привлекательным, несмотря на его недостатки. Описанные в главе 4 паттерны предоставляют великолепные альтернативы, но пока вы не изучите их, «Локатор сервисов» не начнет казаться вам непривлекательным.

ВНИМАНИЕ

Если вы будете рассматривать только статическую структуру классов, контейнеры внедрения зависимостей покажутся вам очень похожими на локаторы сервисов. Разница между ними очень мала и заключается не в деталях механизмов реализации, а в способах их использования. По существу, запрос контейнеру или локатору на формирование полного графа зависимостей, выполненный из корня компоновки, является корректным способом их применения. Тот же запрос, но сделанный из любого другого места, приводит к тому, что «Локатор сервисов» придется рассматривать как антипаттерн.

Изучим пример, в котором «Локатор сервисов» конфигурируется в коде.

5.4.1. Пример: ProductService, использующий «Локатор сервисов»

В качестве примера реализуем возврат в наш проверенный и надежный `ProductService`, требующий экземпляр абстрактного класса `ProductRepository`. В этом случае `ProductService` может использовать статический метод `GetService`, чтобы получить требуемый экземпляр:

```
this.repository = Locator.GetService<ProductRepository>();
```

В этом примере я реализовал методы, применяя обобщенные параметры типа, чтобы указать тип запрошенного сервиса, но можно было бы использовать экземпляр типа `Type`, чтобы задать тип, если такой вариант покажется более предпочтительным.

Как видно из листинга 5.5, реализация класса `Locator` является настолько минималистичной, насколько это вообще возможно. Я мог бы добавить граничные операторы и обработку ошибок, но я хотел сделать наглядным основной код. В него

можно также включить возможность загрузки конфигурации из файла .config, но это вы можете сделать и сами в качестве самостоятельной работы.

Листинг 5.5. Минималистичная реализация "Локатора сервисов"

```
public static class Locator
{
    private readonly static Dictionary<Type, object> services
        = new Dictionary<Type, object>();

    public static T GetService<T>()
    {
        return (T)Locator.services[typeof(T)];
    }

    public static void Register<T>(T service)
    {
        Locator.services[typeof(T)] = service;
    }

    public static void Reset()
    {
        Locator.services.Clear();
    }
}
```

1 Вернуть сервис

Locator — это класс, содержащий только статические члены, поэтому его можно явно определить как статический класс. Он содержит все конфигурируемые сервисы во внутреннем словаре, который соотносит абстрактные типы с их конкретными экземплярами.

Клиенты, такие как ProductService, могут использовать метод GetService (1) для запроса экземпляра абстрактного типа T. Поскольку код этого примера не содержит граничных операторов и кода обработки ошибок, данный метод будет генерировать критическую исключительную ситуацию KeyNotFoundException, если в словаре не находится соответствие запрашиваемому типу. Но вы легко можете добавить код, чтобы сгенерировать более информативное исключительное событие.

Метод GetService может вернуть экземпляр запрашиваемого типа только в случае, если этот экземпляр был предварительно помещен во внутренний словарь. Это осуществляется методом Register. И снова код примера не содержит блока защиты, поэтому имеется возможность зарегистрировать значение null, но более качественный код должен иметь защиту от этого.

В некоторых случаях (особенно при проведении модульного тестирования) важно иметь возможность переустановки локатора сервисов. Данная функциональность реализуется в методе Reset, очищающем внутренний словарь.

Классы, подобные ProductService, рассчитывают, что необходимый им сервис будет предоставлен локатором сервисов, поэтому важно, чтобы локатор был сконфигурирован до начала его использования. В модульном тестировании это может быть реализовано через ***тестовый двойник***, реализованный посредством библиотеки

динамически генерируемых подставных объектов (Dynamic Mock Library), такой как ***Moq***[1], что показано в следующем примере:

```
var stub = new Mock<ProductRepository>().Object;
Locator.Register<ProductRepository>(stub);
```

Прежде всего создается заглушка абстрактного класса `ProductRepository`, затем используется статический метод `Register` для конфигурирования локатора сервисов (заполнения его экземплярами). Если этот шаг реализуется до того, как `ProductService` будет использован в первый раз, `ProductService` будет применять сконфигурированную заглушку `Stub` для работы с `ProductRepository`. В полной рабочей версии приложения локатор сервисов будет конфигурироваться с корректной реализацией `ProductRepository` в корне компоновки.

Такой способ поиска зависимостей из класса `ProductService` определенно работает, если единственным критерием оценки успеха является то, что зависимость может быть использована и при желании заменена, но у него имеются и достаточно серьезные недостатки.

5.4.2. Анализ

«Локатор сервисов» является опасным паттерном, так как он почти всегда работает. Мы можем осуществлять поиск зависимостей из классов-клиентов и можем заменять эти зависимости на другие реализации — даже когда применяются тестовые двойники при проведении модульного тестирования.

Когда описанная в главе 1 модель анализа применяется для оценки, обладает ли «Локатор сервисов» преимуществами модульной архитектуры приложений, обнаруживается, что он удовлетворяет большинству требований.

- Динамическое связывание возможно за счет возможности изменения регистрации.
- Возможна параллельная разработка кода, поскольку программирование ведется в терминах интерфейсов и модули при желании могут заменяться.
- Возможно получение хорошего разделения ответственности, поэтому ничто не мешает писать удобный в сопровождении код. Но реализация этого на практике становится намного более сложным делом.
- Зависимости могут заменяться на тестовые двойники, что обеспечивает пригодность приложения для тестирования.

Существует лишь одна область, в которой использование «Локатора сервисов» быстро дает отрицательный результат.

Влияние

Основная проблема, возникающая при использовании «Локатора сервисов», заключается в том, что он сильно влияет на возможность переиспользования классов, являющихся его клиентами. Эта проблема проявляется в виде двух симптомов:

---

[1] http://code.google.com/p/moq/.

- модуль может быть перемещен только вместе с ненужной зависимостью;
- неочевидно, что используется внедрение зависимостей.

Прежде всего рассмотрим граф зависимостей для `ProductService` из примера подраздела 5.4.1 (рис. 5.8). Помимо ожидаемой связи с абстрактным классом `ProductRepository`, `ProductService` зависит также еще и от класса `Locator`.

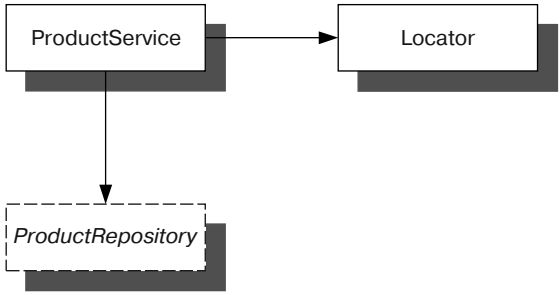


Рис. 5.8. Граф зависимостей для реализации ProductService, использующего «Локатор сервисов» для получения экземпляров абстрактного класса ProductRepository

Это означает, что для переиспользования класса `ProductService` нам придется включить в дистрибутив не только его и соответствующую ему зависимость `ProductRepository`, но и зависимость `Locator`, которая нужна только из технических соображений. Если класс `Locator` будет определен не в том же модуле, в котором определены `ProductService` и `ProductRepository`, новые приложения, которые захотят повторно использовать `ProductService`, должны будут использовать также и модуль класса `Locator`.

Чтобы найти компромисс, предположим, что новое приложение, в котором предполагается повторное применение класса `ProductService`, уже реализует другую стратегию, основанную на внедрении конструктора. `ProductService` не удовлетворяет этой стратегии, зато реализует собственную стратегию, фактически загрязняющую архитектуру внедрения зависимостей в новом приложении. Чтобы задействовать ее, разработчики должны согласиться с существованием «Локатора сервисов». После того как он будет введен в приложение, неопытные разработчики, возможно, будут использовать его не по назначению — в случаях, когда могут найтись и более целесообразные альтернативы.

Наверное, можно даже согласиться с существованием избыточной зависимости от класса `Locator`, если это окажется совершенно необходимо для функционирования внедрения зависимостей; нам придется смириться с ее наличием — как с платой за получение других преимуществ.

Тем не менее имеются лучшие варианты (такие, как внедрение конструктора), поэтому данная зависимость все же является избыточной.

И наконец, немного негатива. Ни данная избыточная зависимость, ни ее противоположность — `ProductRepository` не будут явно видны разработчикам, решившим использовать класс `ProductService`. Рисунок 5.9 демонстрирует, что Visual Studio не предоставляет помощи при работе с этим классом.

Рис. 5.9. Единственная информация, которую нам может предоставить IntelliSense о классе ProductService, заключается в том, что у него имеется конструктор по умолчанию. Его зависимости не видны

Если мы хотим создать новый экземпляр класса `ProductService`, Visual Studio подскажет нам только, что этот класс имеет конструктор по умолчанию. Однако если мы затем попытаемся выполнить код, который только что написали, мы получим ошибку времени исполнения, если забудем зарегистрировать экземпляр класса `ProductRepository` в классе `Locator`. Вероятность, что так и произойдет, весьма высока, если мы недостаточно хорошо знакомы с классом `ProductService`.

СОВЕТ

Представьте, что написанный нами код попадает в недокументированную, обфусцированную .dll. Просто ли будет кому бы то ни было использовать его? Можно разрабатывать API, очень близкий к самодокументированному коду, и, хотя для этого и требуется определенный навык, к этому следует стремиться.

ПРИМЕЧАНИЕ

Проблема с «Локатором сервисов» заключается в том, что любой клиент, использующий его, вводится в заблуждение относительно уровня его сложности. Выглядит он обманчиво простым, на что указывает его известный API, но на поверку оказывается весьма сложным, и мы ничего не знаем о связанных с ним проблемах, пока не получаем ошибку времени исполнения.

Проблема с классом `ProductService` заключается в том, что он далек от самодокументированности: мы не можем сказать, какие зависимости должны быть представлены до того, как он начнет работать. Фактически разработчики `ProductService` могут даже решить добавить другие зависимости в будущие версии. В таком случае код, работающий в текущей версии, может давать ошибку в следующей версии, а мы даже не можем получить ошибку компиляции, которая могла бы предупредить нас об этом. При применении локатора сервисов становится проще внести в код изменения, нарушающие его работоспособность.

ВНИМАНИЕ

Использование дженериков может создать обманчивое впечатление, что «Локатор сервисов» является сильно типизированным. Однако даже код, похожий на пример из листинга 5.5, является слабо типизированным, поскольку запрашивать можно любой тип. Наличие возможности компилировать код, вызывающий метод GetService<T>, не гарантирует, что не будут генерироваться исключения во время исполнения.

ПРИМЕЧАНИЕ

При проведении модульного тестирования возникает дополнительная проблема, связанная с тем, что тестовый двойник, зарегистрированный в одном наборе тестов, будет вызывать ***взаимозависимые тесты***, поскольку будет оставаться в памяти, когда начнет выполняться другой набор тестов. Следовательно, нужно выполнять метод Fixture Teardown после выполнения каждого теста путем вызова Locator.Reset(); о необходимости этого действия нужно помнить разработчику, а забыть об этом очень легко.

Это уже по-настоящему плохо. «Локатор сервисов» может казаться безобидным, но способен порождать совершенно любые ошибки времени исполнения. Как можно было бы обойти эти проблемы?

Рефакторинг в направлении внедрения зависимостей

Когда мы решаем избавиться от «Локатора сервисов», нам потребуется найти способ сделать это. Как обычно, целью по умолчанию является внедрение конструктора, если только один из других представленных в главе 4 шаблонов внедрения зависимостей не будет подходить лучше для конкретной ситуации.

ВНИМАНИЕ

Когда мы рассматриваем структуру «Локатора сервисов», он оказывается похожим на «Окружающий контекст». Оба неявно используют синглтоны, различие же между ними состоит в возможности применения локальных умолчаний. Окружающий контекст гарантирует, что он всегда может предоставить подходящий экземпляр требуемого сервиса (как правило — только одного). «Локатор сервисов» не обеспечивает такой гарантии, поскольку он фактически является слабо типизированным контейнером сервисов, о которых отсутствует встроенная информация.

Во многих случаях класс, использующий «Локатор сервисов», может содержать несколько вызовов, распределенных по всему коду. В таких случаях он действует как замена оператору `new`.

Когда ситуация именно такова, первый этап рефакторинга заключается в консолидации создания каждой зависимости в единственном методе.

Если в нашем распоряжении нет поля элемента (Member Field) для сохранения экземпляра зависимости, мы можем добавить такое поле и реализовать остальной код таким образом, чтобы он использовал это поле, когда коду требуется зависимость. Такое поле следует сделать с атрибутом «только для чтения», чтобы гарантировать, что оно не будет модифицироваться за пределами конструктора. Если поле будет создано как «только для чтения», придется устанавливать его значение в конструкторе, используя «Локатор сервисов». Затем можно добавить параметр в конструктор, через который будет устанавливаться данное поле, и избавиться от «Локатора сервисов», который теперь может быть удален. Вводя в конструктор параметр, необходимый для представления зависимости, мы, вполне возможно, спровоцируем ошибки при работе клиентов, использующих ранее разработанный код. Это потребует перемещения всех подключений зависимостей в корень компоновки.

Выполнение рефакторинга класса, применяющего «Локатор сервисов», похоже на рефакторинг класса, использующего антипаттерн «Диктатор», поскольку «Локатор сервисов» по сути является своеобразной разновидностью паттерна «Диктатор». Подраздел 5.1.3 содержит подробное описание процесса рефакторинга реализаций антипаттерна «Диктатор» с целью перехода к внедрению зависимостей.

На первый взгляд «Локатор сервисов» выглядит как настоящий паттерн внедрения зависимостей, но не дайте себя одурачить: он может явно обеспечить слабое связывание, но при этом порождает другие проблемы. Представленные в главе 4 паттерны являются значительно более выгодными альтернативами с меньшим количеством недостатков. Это справедливо по отношению к антипаттерну «Локатор

сервисов» в той же мере, что и к другим антипаттернам, представленным в этой главе. Несмотря на все их различия, все они имеют одну сходную черту — все они могут быть исправлены путем перехода к одному из паттернов внедрения зависимостей, представленных в главе 4.

5.5. Резюме

Поскольку внедрение зависимостей представляет собой набор паттернов и приемов программирования, не существует единственного инструмента, который обеспечил бы механическую проверку того, насколько правильно оно реализовано. В главе 4 мы познакомились с паттернами, определяющими способы корректной реализации внедрения зависимостей, но это лишь одна сторона медали. Помимо этого важно понять, где можно совершить ошибку, даже если вы допустите ее из наилучших побуждений. Существуют важные уроки, которые следует выучить, чтобы избежать провалов, но мы не должны всегда учиться на своих ошибках — иногда кое-чему можно научиться и на ошибках других.

В этой главе я описал наиболее часто встречающиеся ошибки, имеющие форму антипаттернов. Я встречал много проявлений таких ошибок в реальной жизни и сам себе признался в грехе их применения. К счастью, я давно отбросил все эти привычки — я чист уже многие годы.

Первая и самая главная привычка, от которой нужно избавиться, — это мнимая необходимость прямого управления зависимостями. Легко обнаружить места использования шаблона «Диктатор»: везде, где вы применяете ключевое слово `new` (в C# как минимум) для создания экземпляра нестабильной зависимости, вы прибегаете к «Диктатору». Неважно, сколько слоев фабрик вы используете, чтобы скрыть этот факт. Единственное место, в котором можно создавать зависимости с помощью `new`, — это корень композиции.

Избавление от антипаттерна «Диктатор» является самой важной задачей. Начинайте заниматься устранением других антипаттернов только после того, как избавитесь от «Диктатора» — все остальные намного менее опасны.

СОВЕТ

«Диктатор» не позволяет вам реализовать динамическое связывание; другие антипаттерны просто делают динамическое связывание более неуклюжим. Поэтому в первую очередь разберитесь с «Диктатором».

«Гибридное внедрение» допускает внедрение зависимостей, но все портится из-за необходимости перетаскивания из кода в код ненужных зависимостей при переиспользовании. К счастью, «Гибридное внедрение» легко поддается рефакторингу через внедрение конструктора, так что даже если и кажется, что его можно оставить, не следует делать этого. Мы больше выиграем, чем проиграем, перейдя к корректному решению — фактически мы потеряем лишь время, затраченное на рефакторинг.

«Ограниченное конструирование» налагает искусственные ограничения на типы, используемые для реализации абстракций. В большинстве случаев это вы-

ражается в необходимости для всех реализаций иметь конструктор по умолчанию, но в других случаях может потребоваться, чтобы конструкторы имели определенный параметр для инициализации компонентов.

Вы должны отменить эти ограничения и использовать контейнер внедрения или ручную сборку для связывания всех объектов с любыми требуемыми ими зависимостями. Если у вас имеется сценарий, согласно которому требуется инициализировать определенные компоненты информации о текущем контексте, самым подходящим паттерном будет «Внедрение метода».

«Локатор сервисов» может показаться вполне нормальным паттерном, но я отношу его к антипаттернам, хотя кому-то это может показаться спорным. Несмотря на то что он решает некоторые связанные с внедрением зависимостей проблемы, он порождает другие, перевешивающие его достоинства. Нет надобности мириться с этими недостатками, поскольку представленные в главе 4 паттерны являются лучшей альтернативой. Это справедливо при замене всех антипаттернов, описанных в этой главе: представленные в главе 4 паттерны позволяют решать проблемы, порождаемые антипаттернами.

Теперь вы знаете, чего следует избегать и что следует делать, но остаются вопросы, касающиеся легкости устранения проблем. В следующей главе обсуждается эта тема и описываются способы решения.

6 Рефакторинг внедрения зависимостей

Меню:

- соотнесение значения времени исполнения с абстракциями;
- использование краткосрочных зависимостей;
- устранение циклических зависимостей;
- использование сверхвнедрения конструктора;
- мониторинг связанности.

Итак, вы уже знаете, что я обожаю беарнский, или в целом голландский, соус. Первая причина — его превосходный вкус; другая — его не так просто готовить. Помимо трудностей в приготовлении, с беарнским соусом связана еще одна проблема: его нужно подавать на стол свежеприготовленным (ну или я так думал).

Когда я принимаю гостей, возникает далеко не идеальная ситуация. Вместо того чтобы лично их встретить и помочь им освоиться и расслабиться, я бешено верчусь на кухне, готовя соус и предоставив гостей самим себе.

После того как такая ситуация повторилась несколько раз, моя очень общительная жена решила взять дело в свои руки. Прямо напротив нашего дома есть ресторан, и однажды она задала поварам вопрос — какой прием можно было бы использовать, чтобы я мог готовить отличный голландский соус заранее. Оказалось, что такой способ имеется, и теперь я готовлю деликатесный соус для гостей, не напрягая их при этом своим отсутствием.

У каждого мастера имеются свои особые приемы работы. Это верно как для разработки программного обеспечения в целом, так и для реализации внедрения зависимостей в частности. Существуют часто возникающие проблемы, и во многих случаях есть известные способы их ликвидации.

В течение многих лет я наблюдал усилия людей, осваивающих внедрение зависимостей, и для меня очевидно, что многие проблемы имеют схожий характер. В этой главе мы рассмотрим проблемы, которые наиболее часто возникают при внедрении зависимостей в программный код, и способы их устранения. Дочитав главу, вы научитесь лучше распознавать и обрабатывать ситуации, в которых они происходят.

Как и две предыдущие главы в этой части книги, данная глава организована в виде каталога — на этот раз каталога проблем и их решений (или, если вы предпочитаете, методов рефакторинга). Рисунок 6.1 представляет структуру главы.

Рис. 6.1. Эта глава представляет собой каталог методов рефакторинга и устранения распространенных проблем внедрения зависимостей

В каждом разделе я описываю одну распространенную проблему и способы ее ликвидации, включая пример. Вы можете читать разделы независимо друг от друга или последовательно. Назначение каждого раздела — познакомить вас с решением распространенной проблемы, чтобы вы были во всеоружии, если такая ситуация возникнет.

6.1. Соотнесение значения времени исполнения с абстракциями

Когда вы только начинаете использовать внедрение зависимостей, одна из первых сложных ситуаций, с которой, по-видимому, придется столкнуться, возникнет, когда абстракции зависят от значений времени исполнения. Например, сайт, представляющий собой онлайновую карту, может предлагать в качестве услуги расчет расстояния между двумя точками. При этом могут быть предложены разные варианты расчета: нужен ли вам кратчайший путь? Или самый быстрый путь — рассчитанный на основании известных параметров расписания транспорта? А может, нужен наиболее живописный путь?

За каждым вариантом расчета стоит свой алгоритм, и приложение может рассматривать каждый алгоритм расчета пути как абстракцию. Тогда оно может работать с этими абстракциями унифицированным образом. Для расчета пути приложению требуется специфический алгоритм, но с точки зрения использования этого алгоритма приложению неважно, какой именно. Мы должны сообщить приложению, какой именно алгоритм должен быть применен, но мы можем сделать это только во время работы приложения, так как выбор алгоритма производится на основании решения, принятого пользователем.

Данный раздел показывает, как мы можем решать проблемы такого рода. Прежде чем перейти к рассмотрению примера, мы коротко обсудим проблему в целом. Когда мы закончим, вашей автоматической реакцией на проблему такого вида должно стать применение «Абстрактной фабрики».

6.1.1. Абстракции с зависимостями времени исполнения

Когда используется внедрение конструктора, то подразумевается, что во время исполнения зависимость будет однозначно определена. Пусть сигнатура конструктора имеет следующий вид:

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

Неоднозначность запрещена!

Такой вариант никогда не будет работать, если во время исполнения не будет понятно, какая именно реализация `DiscountRepository` должна быть использована. Во время разработки мы можем трактовать зависимость как абстракцию и следовать принципу подстановки Лисков; но во время исполнения решение о том, какая реализация `DiscountRepository` будет использована, должно быть принято до того, как `RepositoryBasketDiscountPolicy` будет создаваться. Поскольку зависимость запрашивается через конструктор, мы не можем отложить принятие решения на более поздний момент.

Вот что это означает: при работе с классом `RepositoryBasketDiscountPolicy` никакой неоднозначности с `DiscountRepository` быть не может. Другие потребители могут также запрашивать экземпляры `DiscountRepository`, при этом не имеет особого значения — используют ли они все один и тот же экземпляр или каждый из них получает собственный. Такие зависимости часто представляют собою сервисы (Services), а не объекты домена (Domain Objects). На практике существует лишь единственный экземпляр данного сервиса.

ПРИМЕЧАНИЕ

Как будет показано в главе 9, одновременно могут использоваться несколько реализаций одной и той же абстракции. Однако, с точки зрения потребителя, существует только одна.

Сервисы относятся к распространенной группе зависимостей, но временами зависимость представляет собой настоящий объект домена. В частности, ситуация именно такова, если речь идет об изменяющих поведение абстракциях, таких как стратегии (Strategies). Упомянутый ранее алгоритм расчета пути представляет собой такой пример. Еще один пример — набор растровых эффектов в графическом редакторе: каждый эффект выполняет конкретное преобразование растра, но приложение рассматривает их всех как абстракции — такое решение одновременно представляет собой архитектуру, позволяющую использование расширений (плагинов, Add-Ins).

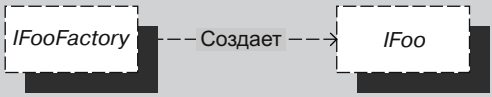
В подобных случаях мы не можем запросить зависимость через конструктор, поскольку компоновщик не может знать, какую реализацию он должен передать. В различные моменты жизненного цикла приложения может существовать ноль (то есть ни одного), один или много экземпляров. Зависимость является неоднозначной во время разработки.

Как всегда происходит в разработке программного обеспечения, решение находится на другом уровне косвенности: в нашем случае это шаблон проектирования «Абстрактная фабрика» (Abstract Factory).

«АБСТРАКТНАЯ ФАБРИКА»

Шаблон проектирования «Абстрактная фабрика» решает проблему, возникающую, когда требуется обеспечить возможность запрашивать по желанию тот или иной экземпляр абстракции. «Абстрактная фабрика» представляет собой перемычку между абстракциями и конкретными значениями, определенными во время исполнения, что дает нам возможность передавать значение времени выполнения в зависимость.

Следующий рисунок показывает, как это работает. При этом вводится новая абстракция, которая создает экземпляры первоначально требуемой абстракции.



Если нам нужна возможность создания экземпляров `IFoo` по запросу, то нам необходим и способ сделать это. «Абстрактная фабрика» представляет собой еще одну абстракцию, которую мы можем использовать для создания требуемых экземпляров.

«Абстрактная фабрика» сама по себе — это абстракция, единственным назначением которой является создание экземпляров первоначально требуемой абстракции. Если нам нужна возможность создавать экземпляры `IFoo` из конкретных экземпляров `Bar`, соответствующая «Абстрактная фабрика» может выглядеть, например, так:

```
public interface IFooFactory
{
    IFoo Create(Bar bar);
}
```

В данном случае `Bar` — это конкретный класс. `IFooFactory` позволяет нам преобразовывать экземпляр конкретного класса `Bar` в экземпляр абстрактного `IFoo`. Реализация `IFoo` может содержать экземпляр `bar` или использовать какую-то вводимую информацию в качестве руководства по выбору конкретного варианта экземпляра `IFoo`.

В вырожденном случае «Абстрактная фабрика» может не иметь никаких входных параметров:

```
public interface IFooFactory
{
    IFoo Create();
}
```

В таком случае «Абстрактная фабрика» становится чистой фабрикой, при этом отпадает необходимость преобразования.

«Абстрактная фабрика» — это один из самых полезных паттернов проектирования. Имейте это в виду, поскольку с ее помощью можно решить многие проблемы, связанные с внедрением зависимостей.

СОВЕТ

Когда один или несколько аргументов «Абстрактной фабрики» в свою очередь являются абстракциями, данная технология становится еще и образцом внедрения метода.

«Абстрактная фабрика» — это универсальное решение, если имеется необходимость создания зависимостей из значений времени исполнения.

Соображения по проектированию

Часто «Абстрактная фабрика» должна применяться с ограничениями. Зависимости, созданные «Абстрактной фабрикой», по природе своей требуют наличия значения времени исполнения. Преобразование из значения времени исполнения в абстракцию должно быть принципиально целесообразным. Если вы хотите использовать «Абстрактную фабрику», чтобы иметь возможность создавать экземпляры конкретной реализации, вы можете вместо этого получить протекающую абстракцию (Leaky Abstraction).

ПРОТЕКАЮЩИЕ АБСТРАКЦИИ

Как вы помните, разработка через тестирование обеспечивает пригодность для тестирования. Соответственно, при разработке программного обеспечения лучше сначала определить интерфейсы, а затем программировать, ориентируясь на них. Но даже если процесс осуществляется именно так, могут возникнуть ситуации, в которых уже существует конкретный тип, и на его основе нужно определить интерфейс.

Когда мы делаем это, мы должны позаботиться о том, чтобы лежащая в основе реализация не имела утечек. Одна из причин, по которой такое может произойти, заключается в том, что интерфейс строится на основе только одного конкретного типа, а все типы параметров и возвращаемые типы остаются конкретными типами, которые определены в той же библиотеке.

К формированию интерфейса следует подходить рекурсивно, добиваясь, чтобы все типы, обнаруженные в корневом интерфейсе, в свою очередь были определены как интерфейсами. Я называю это *глубоким выделением* (Deep Extraction), в результате чего получаются *глубокие интерфейсы* (Deep Interfaces).

ASP.NET MVC содержит несколько примеров глубоких интерфейсов. Например, `HttpContextBase` имеет свойство `Request` типа `HttpRequestBase` и т. д. Данная абстракция была рекурсивно получена из `System.Web.HttpContext`.

Всегда следует гарантировать, что данная абстракция может быть применена к другим реализациям, а не только к одной первоначальной. Если это не выполняется, вам следует перепроектировать свое приложение.

Абстрактные фабрики могут выглядеть по-разному, и не всегда очевидно, что вы имеете дело именно с такой фабрикой.

ПРИМЕЧАНИЕ

Абстракция, создающая экземпляры других абстракций, является абстрактной фабрикой. Она вовсе необязательно должна иметь имя, заканчивающееся на Factory.

Рассмотрим несколько примеров: сначала простой, специфичный для языка, а затем более сложный пример, в котором абстрактная фабрика скрыта под другим именем.

6.1.2. Пример: выбор алгоритма расчета пути

Во введении к данному разделу кратко обсуждался сайт онлайновой карты, на котором посетитель может выбрать один из нескольких алгоритмов расчета пути. В этом разделе мы рассмотрим способ применения абстрактной фабрики для реализации этого требования.

В веб-приложении можно только передавать примитивные типы[1] из браузера на сервер, поэтому когда пользователь выбирает алгоритм расчета пути из раскрывающегося списка, вы должны передать его как номер соответствующей строки этого списка. *Перечисление* (Enum) — это на самом деле просто число, поэтому на сервере значение выбора может быть представлено типом `RouteType`:

```
public enum RouteType
{
    Shortest = 0,
    Fastest,
    Scenic
}
```

Но вам нужен экземпляр `IRouteAlgorithm`, который будет вычислять путь. Для преобразования из значения времени исполнения типа `RouteType` в `IRouteAlgorithm` можно определить абстрактную фабрику:

```
public interface IRouteAlgorithmFactory
{
    IRouteAlgorithm CreateAlgorithm(RouteType routeType);
}
```

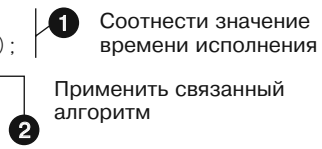
Так мы сможем реализовать метод `GetRoute` для `RouteController`, внедряя `IRouteAlgorithmFactory` и используя ее затем для преобразования значения времени исполнения в нужную нам зависимость: `IRouteAlgorithm`. Листинг 6.1 показывает этот процесс.

Листинг 6.1. Использование IRouteAlgorithmFactory

```
public class RouteController
{
    private readonly IRouteAlgorithmFactory factory;

    public RouteController(IRouteAlgorithmFactory factory)
    {
        if (factory == null)
        {
            throw new ArgumentNullException("factory");
        }

        this.factory = factory;
```

[1] Если быть уж совсем точным, можно передавать только строки, но большинство используемых в веб-приложениях фреймворков поддерживают преобразование типов для примитивных типов.

```
    }

    public IRoute GetRoute(RouteSpecification spec,
        RouteType routeType)
    {
        IRouteAlgorithm algorithm =
            this.factory.CreateAlgorithm(routeType);
        return algorithm.CalculateRoute(spec);
    }
}
```



Задачей класса `RouteController` является обработка веб-запросов. Метод `GetRoute` получает определенную посетителем спецификацию маршрута, определяющую его начальный и конечный пункты, а также выбранный тип маршрута `RouteType`. Вам нужна абстрактная фабрика, чтобы соотнести значение времени исполнения `RouteType` с экземпляром `IRouteAlgorithm`, поэтому вы запрашиваете экземпляр `IRouteAlgorithmFactory`, используя стандартную внедрение конструктора.

В методе `GetRoute` можно использовать фабрику для соотнесения переменной `routeType` с `IRouteAlgorithm` ❶. После получения требуемого экземпляра вы можете использовать его для вычисления маршрута ❷, после чего вернуть рассчитанный результат.

ПРИМЕЧАНИЕ

Для краткости я опустил контрольный оператор в методе GetRoute. Но переданная сущность RouteSpecification может иметь значение null, поэтому более надежная реализация метода должна содержать такую проверку.

Наиболее очевидная реализация `IRouteAlgorithmFactory` должна содержать простой оператор `switch` и возвращать три различные реализации `IRouteAlgorithm` на основе полученного значения типа расчета. Я оставляю это вам в качестве упражнения.

Приведенный пример демонстрирует соотнесение значения времени исполнения и зависимости, выполненное с помощью абстрактной фабрики в ее простейшем виде. Следующий пример демонстрирует более сложный вариант, в котором на первый взгляд абстрактная фабрика даже не используется.

6.1.3. Пример: использование CurrencyProvider

В главе 4 мы в основном говорили о реализации конвертации валют на основе ASP.NET MVC Controller. Тип `Currency` — это абстрактный класс, воспроизведенный здесь, чтобы вам не нужно было возвращаться к подразделу 4.1.4:

```
public abstract partial class Currency
{
    public abstract string Code { get; }

    public abstract decimal GetExchangeRateFor(
        string currencyCode);
}
```

На первый взгляд немного странной кажется трактовка валюты как абстракции, поскольку она выглядит более похожей на *объект-значение*. Однако отмечу, что метод `GetExchangeRateFor` позволяет нам запрашивать практически неограниченный набор курсов конверсии. Если мы собираемся использовать 100 коэффициентов конверсии, каждый экземпляр `Currency` будет занимать более 2 Кбайт памяти. Кажется, это не много, но это может потребовать проведения оптимизации, например, путем использования паттерна проектирования «Приспособленец» (Flyweight).

Еще одна характерная проблема, быстро проявляющаяся в связи с курсами конверсии, со сроками действия курса обмена: другими словами, речь идет об актуальности информации. Такие приложения, как торговые площадки для валютных рынков, требуют, чтобы обменные курсы обновлялись несколько раз в секунду, тогда как международный коммерческий сайт за все время своего существования может потребовать всего нескольких обновлений для устойчивых валют. Такие приложения могут также включать стратегии наценок или округления, увеличивая потенциальную сложность реализации типа `Currency`. Учитывая все это, определение класса `Currency` как абстрактного начинает выглядеть вполне обоснованным.

Когда потребитель типа ASP.NET MVC Controller нуждается в конвертации цен, он требует `Currency` — как зависимости для выполнения такой конверсии. В демоприложении, используемом в этой книге, класс `Money`, применяемый для представления цен, содержит такой метод конвертации:

```
public Money ConvertTo(Currency currency)
```

Потребитель, такой как `Controller`, может передать экземпляр `Currency` во все экземпляры цен, чтобы конвертировать их, но встает вопрос — какой именно экземпляр `Currency`?

Выбор целевой валюты типа `Currency` основан на значении времени исполнения: то есть на валюте, предпочтительной для посетителя. Это означает, что мы не можем передать единый для всех объект `Currency` через внедрение конструктора, поскольку компоновщик не знает, какую `Currency` применять.

Как вы видели в подразделе 4.1.1, решение заключается во внедрении `CurrencyProvider` вместо единственной `Currency`:

```
public abstract class CurrencyProvider
{
    public abstract Currency GetCurrency(string currencyCode);
}
```

На рис. 6.2 показано, как `Controller` получает код предпочтительной для посетителя валюты из профиля и использует внедренный `CurrencyProvider` для создания экземпляра требуемой `Currency`.

Несмотря на то что `CurrencyProvider` имеет другое имя, он является абстрактной фабрикой, помогающей создать связующее звено между значением времени исполнения и зависимостью времени исполнения. `Currency` принципиально зависит от кода валюты, поэтому мы можем быть уверены в том, что при введении `CurrencyProvider` не будет создана протекающая абстракция.

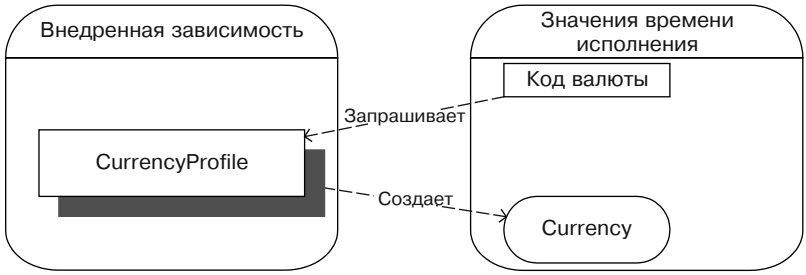


Рис. 6.2. Внедренный CurrencyProvider используется для связывания имеющего примитивный тип значения времени исполнения (код валюты типа string) и зависимости времени исполнения (экземпляр Currency)

Другой взятый из главы 4 пример демонстрирует исключительный случай, когда отсутствует начальный входной параметр. В подразделе 4.2.4 вы видели, что абстрактный класс `CurrencyProfileService` имеет метод `GetCurrencyCode`, возвращающий код валюты текущего пользователя:

```
public abstract string GetCurrencyCode();
```

Хотя метод `GetCurrencyCode` возвращает строку вместо абстракции, вы можете рассматривать `CurrencyProfileService` как вариант абстрактной фабрики.

В классе `HomeController` комбинируются оба варианта для определения предпочтительного значения `Currency` посетителя:

```
var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
var currency =
        this.currencyProvider.GetCurrency(currencyCode);
```

Как `CurrencyProfileService`, так и `currencyProvider` являются внедренными абстрактными фабриками, доступными любому участнику класса `HomeController`. Подразделы 4.1.4 и 4.2.4 демонстрируют, как они внедряются.

Всякий раз когда нам нужно значение времени исполнения и мы хотим иметь возможность изменять способы, которыми мы передаем это значение независимо от потребителя, мы можем внедрить абстрактную фабрику. Это обычно не хранящий состояние (Stateless) сервис, так что он в большей степени соответствует нашей обычной трактовке зависимостей, и мы можем использовать внедрение конструктора или свойства для передачи фабрики потребителю.

Существует и другой тип сценария, в котором абстрактная фабрика также обеспечивает хорошее решение. Речь идет о случаях, когда нам нужно работать с краткосрочными зависимостями.

6.2. Использование краткосрочных зависимостей

Некоторые зависимости в принципе имеют короткий жизненный цикл. Обычно они представляют соединения к внешним ресурсам, таким как базы данных или

веб-сервисы. Такие соединения должны быть закрыты, иначе возникнет утечка ресурсов. В данном разделе мы рассмотрим наилучший способ решить эту проблему.

Как в предыдущем разделе, мы начнем с изучения общего случая, а затем разберем конкретный пример. В конце вы должны будете понимать две вещи:

- вы можете моделировать такие соединения с помощью абстрактной фабрики, создающей одноразовые экземпляры;
- вы должны стараться скрыть этот шаблон за не сохраняющей состояние (Stateless) абстракцией.

Прежде чем мы перейдем к примеру, рассмотрим, что заставило меня сказать это.

6.2.1. Закрытие соединений с помощью абстракций

Свойством слабого связывания и принципа подстановки Лисков является то, что зависимость может быть реализована несколькими способами. Даже когда вы ограничиваетесь одним-единственным решением, радикально противоположное ему решение может потенциально возникнуть в будущем.

Некоторые зависимости определяют доступ к внешним ресурсам, и на такие зависимости распространяются проблемы, связанные с использованием ресурсов. Конечно же, я говорю о соединениях разного вида.

Большинство разработчиков .NET знают, что они должны открывать соединения ADO.NET непосредственно перед их использованием и закрывать их по возможности сразу после выполнения работы. Современные API типа LINQ to SQL или LINQ to Entities делают это автоматически, что избавляет разработчика от необходимости делать это вручную.

Правильный способ использования соединений в ADO.NET известен достаточно широко, но намного менее известно, что такой же порядок действий должен применяться и к клиентам WCF. Они должны быть закрыты как можно скорее после завершения выполнения необходимого набора операций, так как в противном случае они могут оставить брошенными ресурсы на стороне сервера.

WCF-СЕРВИСЫ И СОСТОЯНИЯ

Основное правило при использовании сервисов — сервис не должен хранить состояния. Если это правило выполняется, клиент WCF, несомненно, не должен оставлять неиспользуемые ресурсы на стороне сервера.

Поразительно, но это может оказаться неверным. Даже если мы разрабатываем сервис, который не хранит состояния, WCF может не соответствовать этому. Все зависит от способа связывания.

Один из иллюстрирующих данное утверждение примеров относится к области безопасности. Диагностические (Message-Based) системы безопасности влияют на производительность. Дело в том, что асимметричные ключи требуют больших объемов вычислений, и это тем более актуально для безопасности федеральных систем, поскольку они работают с огромным количеством сообщений. Реализованное в качестве умолчания поведение WCF заключается в организации безопасных сеансов связи путем обмена асимметричными ключами. Клиент и сервер используют асимметричное защищенное рукопожатие для обмена специальным симметричным ключом, применяемым для шифрования всех будущих сообщений, которые являются частью сессии.

Но такое поведение требует, чтобы обе стороны держали разделяемый секретный ключ в памяти. Клиент должен отписаться от сервера, когда он завершает сессию или же симметричный ключ останется на сервере. Он будет уничтожен после истечения тайм-аута, но до этого момента он остается в памяти. Чтобы не расходовать ресурсы сервера впустую, клиент должен явно закрыть соединение после завершения работы.

Хотя сказанное не является верным для всех WCF-соединений, оно все же справедливо для столь большого их количества, что мы должны применять его в своих клиентах WCF.

Как мы можем согласовать закрытие WCF-соединения, чтобы не создать при этом абстракцию с утечками? Эта проблема может быть разрешена двумя способами:

- реализацией всей логики управления соединением в виде абстракции;
- имитацией открытия и закрытия соединений на более детальном уровне.

Первый вариант является предпочтительным, но иногда должен применяться второй. Оба варианта могут комбинироваться для получения наилучшего результата.

Скрытие управления соединением под абстракцией

Введение зависимостей не оправдывает написания протекающих приложений, поэтому мы должны иметь возможность закрывать соединения так скоро, насколько это возможно. С другой стороны, любая зависимость может представлять или не представлять находящееся вне процесса соединение, поэтому она может оказаться протекающей абстракцией, если мы моделировали абстракцию для включения метода `Close`.

Некоторые разработчики создают зависимости на основе `IDisposable`. Но метод `Dispose` — это тот же метод `Close` с другим именем, так что данный подход не позволяет решить описанную проблему.

Решение позволяют найти технологии доступа к данным, такие как LINQ to SQL и LINQ to Entities. В обоих случаях мы работаем с данными, используя *контекст*, содержащий соединение. Когда бы мы ни связывались с базой, контекст при необходимости автоматически открывает и закрывает соединения, полностью избавляя разработчика от участия в этом процессе.

Логично предположить, что мы сразу попробуем сделать то же самое. Рисунок 6.3 показывает, как определить абстракцию на достаточно обобщенном уровне, что позволит приложению открывать и закрывать соединения по мере необходимости. Можно спроектировать интерфейс таким образом, что он будет в достаточной степени крупно-модульным, чтобы каждый метод инкапсулировал все взаимодействия с внешним ресурсом в одном пакете заданий. Класс `Consumer` вызывает метод интерфейса `IResource`. Реализация этого метода должна открывать соединение и вызывать несколько методов, работающих с внешним ресурсом, после чего соединение должно быть закрыто и результат возвращен потребителю.

Потребитель никогда не знает, будут ли реализации открывать и закрывать соединения для него.

Рис. 6.3. Определение абстракции на обобщенном уровне

Мы должны по мере возможности стараться проектировать зависимости потребителя таким образом, чтобы нам никогда не пришлось бы явно вмешиваться в жизненный цикл зависимости на этом уровне. Но существуют тем не менее экземпляры, в которых мы не можем добиться этого.

Открытие и закрытие зависимостей

Проблема с крупно-модульными API такова, что они не обладают достаточной гибкостью. Иногда нам просто нужна абстракция, которая позволила бы явно смоделировать жизненный цикл зависимостей, которые в противном случае приведут к утечкам памяти.

ВНИМАНИЕ

Ликвидация одной утечки порождает другую. Мы избавляемся от утечек памяти и получаем протекающие абстракции.

Самый распространенный жизненный цикл, который мы должны смоделировать, представлен на рис. 6.4. Самый распространенный жизненный цикл соединения состоит из его создания, использования и закрытия после окончания работы. Именно такой жизненный цикл приходится моделировать в большинстве случаев.

Рис. 6.4. Распространенный жизненный цикл соединения

В разделе 6.1 мы видели, как можно использовать абстрактные фабрики для создания зависимостей, и теперь нужно подобрать идиому кодирования, которая

могла бы быть использована для закрытия соединения. Как подсказывает рис. 6.4, мы можем использовать шаблон `IDisposable` (Dispose — уничтожить. — *Примеч. ред.*) для освобождения зависимостей, занимающих соединения.

ВНИМАНИЕ

Одноразовые зависимости — это плохой стиль проектирования. Используйте их, только если у вас нет иных вариантов. Подробнее об этом говорится в разделе 8.2.

Иными словами, мы можем моделировать практически любое взаимодействие, соответствующее представленному на рис. 6.4 жизненному циклу, путем использования «Абстрактной фабрики», создающей одноразовые зависимости (рис. 6.5). Представленный на рис. 6.5 шаблон применения обычно лучше всего реализуется с помощью ключевого слова языка C# `using` (или аналогичной конструкции в других языках). Мы можем моделировать управление соединениями и подобные ему жизненные циклы путем передачи зависимости в абстрактную фабрику. Такова показанная здесь `IFooFactory`. Каждый раз, когда потребителю оказывается нужен экземпляр `IFoo`, он создается фабрикой `IFooFactory`, но потребитель должен не забыть освободить его.

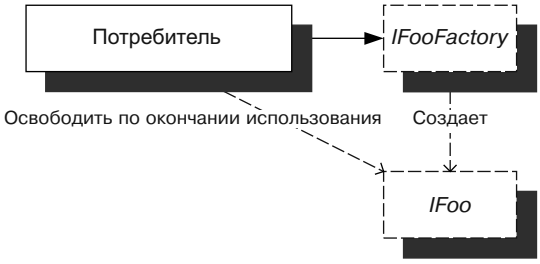


Рис. 6.5. Шаблон применения

Как будет видно из следующего примера, имеет смысл комбинировать оба подхода, рассмотренных выше. Доступ к ресурсу моделируется в виде крупно-модульной абстракции, позволяющей потребителю явно управлять жизненным циклом, тогда как реализация использует описанную комбинацию абстрактной фабрики и одноразовых зависимостей. Рассмотрим, как это работает.

6.2.2. Пример: обращение к сервису управления продуктами

Представим себе разработанное по технологии Windows Presentation Foundation (WPF) приложение, имеющее пользовательский интерфейс с широкими функциональными возможностями (насыщенный интерфейс пользователя) для управления каталогом продуктов. Такой клиент должен взаимодействовать с серверной стороной через WCF-сервис, реализующий необходимые операции по управлению каталогом продуктов.

Рисунок 6.6 показывает, как реализация комбинирует оба подхода, описанных в предыдущем разделе. Класс `MainWindowViewModel` использует интерфейс `IProduct-`

ManagementAgent. Это крупномодульный интерфейс, который предоставляет потребителю необходимые методы, которые тот может вызывать. С точки зрения MainWindowViewModel никакого управления соединением не производится. Когда приложение запущено, класс WcfProductManagementAgent предоставляет реализацию этого крупномодульного интерфейса. Он делает это с помощью абстрактной фабрики IProductChannelFactory, которая создает одноразовые экземпляры. Интерфейс IProductManagementServiceChannel является наследником IDisposable, что позволяет WcfProductManagementAgent освобождать WCF-клиент после успешного вызова операций.

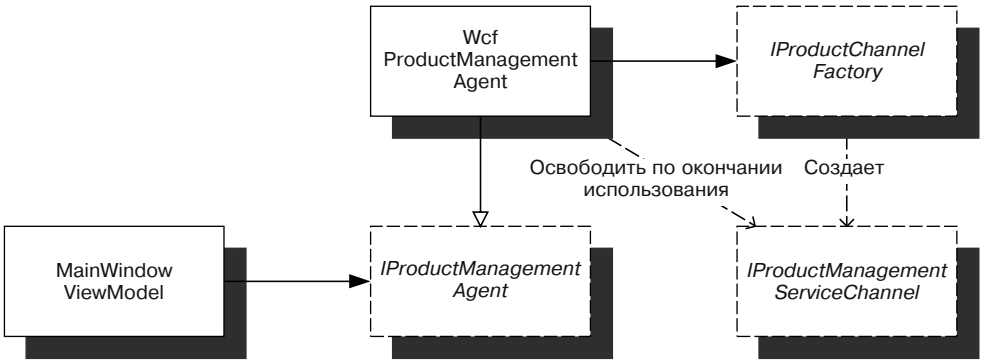


Рис. 6.6. Комбинирование двух подходов

ПРИМЕЧАНИЕ

Мы вернемся к этому WPF-приложению в подразделах 6.3.2 и 7.4.2.

Потребитель в данном случае избавлен от необходимости управлять соединениями, что становится частью реализации WcfProductManagementAgent.

Когда классу MainWindowViewModel требуется вызвать служебную операцию, он использует зависимость IProductManagementAgent. Это совершенно нормальная зависимость, внедренная через внедрение конструктора. Вот, к примеру, как выглядит операция удаления продукта:

```
this.agent.DeleteProduct(productId);
```

В этом примере this.agent — это внедренная зависимость IProductManagementAgent. Как видите, здесь нет прямого управления соединением; но если вы рассмотрите реализацию класса WcfProductManagementAgent, то 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, которого вы могли бы использовать для вызова служебной операции, так как вам нужно закрыть клиент сразу после завершения работы с ним и это требование исключает возможность повторного использования WCF-каналов. Вместо этого в вашем распоряжении имеется внедренная

абстрактная фабрика, которую вы используете для создания новых каналов. Поскольку данная операция ограничена окружающей областью видимости, выход из этой области видимости приведет к закрытию канала.

Зависимость, представляющая фабрику, является экземпляром интерфейса `IProductChannelFactory`. Это специальный (Custom) интерфейс, разработанный для реализации конкретной потребности:

```
public interface IProductChannelFactory
{
    IProductManagementServiceChannel CreateChannel();
}
```

В то же время `IProductManagementServiceChannel` является автоматически генерируемым интерфейсом, создаваемым вместе с другими типами-посредниками WCF. Всякий раз, когда мы создаем ссылку на сервис в Visual Studio или используем `svcutil.exe`, данный интерфейс создается наравне с другими типами. Привлекательной особенностью такого автоматически сгенерированного интерфейса является то, что он уже реализует `IDisposable` наряду с прочими служебными операциями.

WCF распознает этот тип, так что реализовывать `IProductChannelFactory` оказывается несложно, поскольку для создания экземпляров можно использовать `System.ServiceModel.ChannelFactory<TChannel>`.

Основным принципом, которого я придерживаюсь в разработке, является создание не сохраняющего состояние крупномодульного интерфейса наподобие `IproductManagementAgent` для скрытия деталей реализации от потребителей. Хотя мы должны рассматривать одноразовые зависимости как разновидность протекающих абстракций, утечка может находиться внутри конкретной реализации; реализовав такую структуру, мы добьемся пригодности для тестирования без нанесения ущерба остальному коду.

«Абстрактная фабрика» является чрезвычайно полезным паттерном проектирования. Она помогает реализовать зависимости времени исполнения и краткосрочные зависимости. Ее также можно включить в число инструментов разрешения циклических зависимостей, но в этом отношении ее можно считать периферийным инструментом.

6.3. Устранение циклических зависимостей

Время от времени реализации зависимостей оказываются циклическими. Это означает, что реализация требует другую зависимость, внедрение которой требует первую абстракцию.

Такой граф зависимостей не может быть воплощен. При реализации важно, чтобы абстракции сами по себе были нециклическими, хотя конкретная реализация и может содержать цикл. Рисунок 6.7 демонстрирует, как такое может произойти. В данном примере каждая реализация реализует свой интерфейс, но также использует и зависимость. Поскольку `ConcreteC` требует `IA`, но единственной реализацией `IA` является `ConcreteA` со своей зависимостью от `IB` и т. д, мы имеем цикл, который не может быть разрешен.

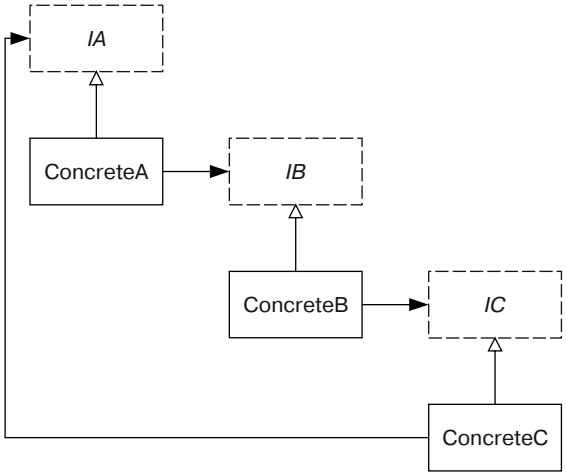


Рис. 6.7. Циклы в графе зависимостей могут возникать, даже если абстракции не имеют прямых связей друг с другом

Пока в вашей программе имеются циклы, не существует возможности разрешить все зависимости и приложение будет невозможно запустить. Понятно, что что-то нужно сделать, но что?

В данном разделе мы рассмотрим проблемы, касающиеся циклических зависимостей, в том числе пример. Когда мы изучим этот вопрос, вашей первой реакцией будет попытка перепроектировать имеющиеся у вас зависимости. Если это окажется невозможным, мы сможем разорвать цикл, используя рефакторинг от внедрения конструктора к внедрения метода. Так ослабляется инвариантности классов, поэтому к данному вопросу не следует относиться легкомысленно.

6.3.1. Устранение циклов зависимостей

Когда я обнаруживаю цикл зависимостей, моим первым вопросом является: «Где я ошибся?»

СОВЕТ

Цикл зависимостей — это пример плохого проектирования. Если такое происходит, вам следует серьезно переделать ваше приложение.

При обнаружении циклической зависимости нужно немедленно произвести поиск и тщательную оценку главной причины ее возникновения. Часто цикл возникает или из-за неверных допущений или даже серьезного нарушения правил однонаправленных зависимостей. В многоуровневом приложении классы должны взаимодействовать только с классами своего уровня и уровня, находящегося непосредственно под ним.

Если циклы переходят более чем через один уровень, это свидетельствует о том, что что-то сделано неправильно. Как видно из рис. 6.8, это может означать, что некоторые ссылки проставлены неверно. В приведенном примере такой является

ссылка от D к A. Если возникает подобная ситуация, это должно быть устранено немедленно.

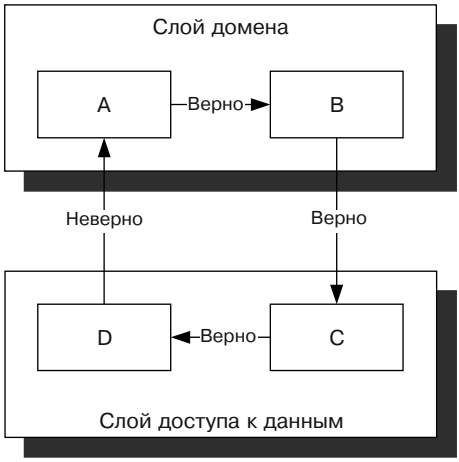


Рис. 6.8. Если цикл выходит за границы одного или более уровней, это означает, что как минимум одна ссылка является недопустимой с точки зрения архитектуры приложения

Происходящее не так очевидно, если мы имеем цикл в пределах одного слоя. Такая ситуация может возникнуть даже в результате корректных умозаключений, которые были реализованы не лучшим образом.

Цикл должен быть обязательно разорван тем или иным способом. Пока цикл существует, приложение не может быть запущено.

Любой цикл является ошибкой в проектировании, поэтому первым делом необходимо перепроектировать соответствующую часть приложения, чтобы избавиться от цикла. Таблица 6.1 показывает некоторые основные действия, которые мы должны выполнить.

Таблица 6.1. Некоторые стратегии перепроектирования, устраняющие циклические зависимости

Стратегия	Описание
События	Часто цикл может быть разорван путем перепроектирования одной из абстракций таким образом, чтобы она вызывала события вместо явной передачи зависимости информации о том, что что-то произошло. События особенно хороши, если одна из сторон только вызывает ничего не возвращающие (void) методы своей зависимости. События в .NET — это реализации шаблона проектирования «Наблюдатель» (Observer), и вы можете иногда явно реализовывать его. Это особенно верно, если вы решите использовать «События домена» (Domain Events) для устранения цикла. При этом возникает возможность применения системы однонаправленных асинхронных сообщений
Внедрение свойства	Если ни один другой метод не привел к успеху, мы можем разорвать цикл, перейдя в одном из классов от внедрения конструктора к внедрению свойства. Это может оказаться последней попыткой, так как относится только к симптомам

Я не собираюсь подробно описывать первый вариант, так как он основательно разобран в имеющейся литературе.

СОВЕТ

Пытайтесь устранить циклы используя события. Если это не получится, примените паттерн «Наблюдатель». И только если вам так и не удастся устранить цикл, попробуйте сделать это с помощью внедрения свойства.

Не сомневайтесь: циклические зависимости — это ошибка проектирования. Главным приоритетом при их выявлении является анализ кода с целью определения причин возникновения зацикливаний. Когда мы поймем, почему это произошло, мы сможем изменить код проекта.

Но в некоторых случаях изменение кода оказывается невозможным. Даже если удастся выявить главную причину возникновения зацикливания, API, в котором это происходит, может оказаться недоступным для изменений.

Прерываение цикла с помощью внедрения свойства

В некоторых случаях исправить ошибку проектирования не представляется возможным, но нам все же необходимо прервать цикл. В таком случае это можно сделать, используя внедрение свойства.

ПРЕДУПРЕЖДЕНИЕ

Внедрение свойства следует применять для устранения циклов только в последнюю очередь. Она лишь снимает симптомы, но не лечит саму болезнь.

Чтобы разорвать цикл, нам нужно проанализировать его, чтобы понять, где можно делать разрыв. Поскольку использование внедрения свойств предполагает наличие опциональной, а не обязательной зависимости, оказывается важным тщательно проанализировать все зависимости, чтобы определить, в каком месте разрыв окажется наименее болезненным.

На рис. 6.9 `B` требует экземпляр `IC` (интерфейс, который реализует `C`). Мы можем разорвать цикл, изменив зависимость `B` с внедрения конструктора на внедрение свойства. Это означает, что мы можем сначала создать `B` и внедрить его в `A`, а затем назначить `C` для `B`:

```
var b = new B();
var a = new A(b);
b.C = new C(new D(a));
```

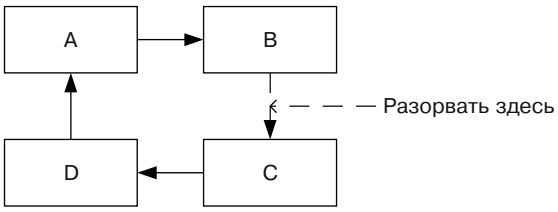


Рис. 6.9. Если имеется цикл, мы должны сначала принять решение о месте его разрыва. В данном случае мы решаем прервать цикл между B и C

Такое применение внедрения свойства повышает сложность B, так как он теперь должен быть работоспособен в случае, когда зависимость еще недоступна.

СОВЕТ

Классы никогда не должны работать, используя зависимости в своих конструкторах, поскольку внедренная зависимость может быть еще не полностью инициализирована.

Если мы не хотим менять исходные классы таким образом, мы можем использовать «Виртуальный прокси» (Virtual Proxy), что позволяет не изменять B:

```
var lb = new LazyB();
var a = new A(lb);
lb.B = new B(new C(new D(a)));
```

LazyB реализует IB точно так же, как это было сделано в B. Однако он получает зависимость IB через внедрение свойства, а не через внедрение конструктора, что позволяет разорвать цикл без изменения исходных классов.

Хотя классы с воображаемыми именами A–D показывают структуру решения, нужен более реальный пример.

6.3.2. Пример: компоновка окна

Наиболее частым случаем, когда невозможно устранить имеющиеся циклические зависимости, является использование внешних API. Одним из примеров этого служит WPF.

В WPF можно использовать шаблон MVVM[1], что позволяет реализовать разделение паттерна. Так мы можем обеспечить разделение ответственности путем разбиения кода на представления (Views) и связанные с ними модели (Models). Модели назначаются соответствующим представлениям через свойство DataContext. Это, по существу, является реализацией внедрения свойства.

СОВЕТ

Дополнительную информацию о построении WPF-приложений с использованием MVVM можно найти в разделе 7.4.

DataContext служит как зависимость для Window, но модели играют важную роль при выборе активируемого представления и времени его активации. Одно из действий, которое должна уметь выполнять модель, — это вызов диалогового окна. Одним из способов реализации этого является внедрение в модель абстракции типа показанной ниже:

```
public interface IWindow
{
    void Close();

    IWindow CreateChild(object viewModel);
```

[1] *Josh Smith* Patterns: WPF Apps With The Model-View-ViewModel Design Pattern. — MSDN Magazine, February 2009: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

```
    void Show();

    bool? ShowDialog();
}
```

Получив внедренный IWindow, любая модель может создать новые окна (Window) и показать их в модальных или немодальных диалоговых окнах. Однако для реализации интерфейса нам нужна ссылка на существующий в действительности объект Window, чтобы правильно установить значение свойства Owner. Листинг 6.2 показывает реализацию 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 представляет собой Window-объект WPF, который можно использовать для показа нового окна. Важно установить владельца окна Window до показа самого окна, или в противном случае возникнут серьезные ошибки при установке фокуса, или же модальное окно окажется скрытым за другими окнами. Для предотвращения таких ошибок нужно установить свойство Owner в значение текущего окна Window. Поле wpfWindow — это другой экземпляр System.Windows.Window.

Еще нужно назначить viewModel свойству DataContext нового окна, прежде чем обертывать его в новую реализацию IWindow и возвращать результат.

Проблема заключается в том, что при такой реализации у нас будут ViewModels, которые требуют IWindow, а реализация IWindow требует WPF Window, и, наконец, WPF-Window через свойство DataContext требуют ViewModel для своей работы. Такой цикл продемонстрирован на рис. 6.10. В MVVM Window зависит от ViewModel, который, в свою очередь, зависит от экземпляра IWindow. Правильной реализацией IWindow является WindowAdapter, который зависит от WPF-Window, так как он должен иметь возможность устанавливать владельца каждого Window, чтобы избежать ошибок установки фокуса.

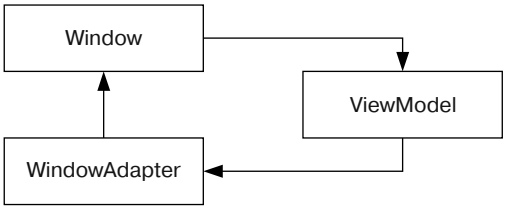


Рис. 6.10. Цикл в фреймворке WPF MVVM

В данном случае не существует обоснованного способа перепроектирования, который мог бы быть использован для устранения циклической зависимости. Отношение между Window и ViewModel является фиксированным, так как System.Windows.Window представляет собой API стороннего разработчика (определен в BCL). Подобно этому WindowAdapter зависит от Window, что позволяет избежать ошибок фокусирования, и это отношение также задано извне.

Единственное отношение, которое может быть изменено, — это отношение между ViewModel и ее IWindow. Технически можно перепроектировать такое отношение, добавив использование событий, но это даст нам более нелогичный API. Чтобы показать диалоговое окно, вам потребуется сгенерировать событие и надеяться, что один из подписчиков откроет модальное окно. Более того, вам придется возвращать результат из диалогового окна по ссылке через начальные аргументы события. Генерация события может оказаться блокирующим вызовом. Это может быть реализовано технически, но выглядит странно, поэтому следует усвоить, что такие ситуации лучше обходить.

Итак, похоже, что мы не можем перепроектировать код, содержащий цикл. Так как же нам избавиться от этого цикла?

Ликвидация цикла

Мы должны отыскать отношение, в котором мы могли бы разорвать цикл, чтобы ввести внедрение метода. В нашем случае это несложно, так как отношение между WPF Window и ViewModel уже использует внедрение свойства. Это то место, в котором цикл может быть разорван.

Простейшим решением будет подключить какой-нибудь еще элемент и установить свойство DataContext для MainWindow последним, перед тем как показать его. Это возможно, но данный способ не очень хорош при работе с контейнерами внедрения, так как потребует от нас явного назначения зависимости после того, как компоновка будет выполнена.

В качестве альтернативы можно инкапсулировать такое отложенное присваивание в адаптер отложенной загрузки (Lazy-Loading Adapter). Это позволит нам связать каждое свойство с контейнером внедрения.

ПРИМЕЧАНИЕ

Следующий пример касается того же проекта, описанного также в подразделе 7.4.2. Полный код вы можете найти в скачанном коде для книги.

Рассмотрим вопрос инкапсуляции процесса создания реализации IWindow, которая корректно загружает MainWindowViewModel и связывает ее с экземпляром WPF MainWindow. Для помощи в этом можно использовать абстрактную фабрику:

```
public interface IMainWindowViewModelFactory
{
    MainWindowViewModel Create(IWindow window);
}
```

Класс MainWindowViewModel содержит более одной зависимости, но все они, кроме IWindow, могут быть удовлетворены немедленно, поэтому их не нужно передавать в виде параметров в метод Create. Вместо этого вы можете внедрить их в конкретную реализацию IMainWindowViewModelFactory.

IMainWindowViewModelFactory может использоваться как зависимость в реализации IWindow, который наследуется из WindowAdapter. Этот адаптер был кратко рассмотрен в листинге 6.2. Это позволяет отложить инициализацию реализации IWindow до момента вызова первого метода. Вот как будет переопределен метод CreateChild из листинга 6.2:

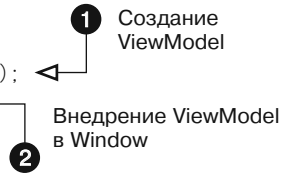
```
public override IWindow CreateChild(object viewModel)
{
    this.EnsureInitialized();
    return base.CreateChild(viewModel);
}
```

Прежде чем выполнить какое-либо реальное действие, следует позаботиться о том, чтобы все зависимости были полностью инициализированы. Когда это будет выполнено, вы можете безопасно вызвать базовую реализацию.

В листинге 6.3 показано, как реализуется метод EnsureInitialized с помощью внедренной фабрики IMainWindowViewModelFactory.

Листинг 6.3. Отложенная инициализация зависимостей

```
private void EnsureInitialized()
{
    if (this.initialized)
    {
        return;
    }

    var vm = this.vmFactory.Create(this);
    this.WpfWindow.DataContext = vm;
    this.DeclareKeyBindings(vm);

    this.initialized = true;
}
```



Когда инициализируется MainWindowAdapter, прежде всего вызывается внедренная абстрактная фабрика для получения требуемой модели ViewModel (1). Это является возможным в данной точке, так как экземпляр MainWindowAdapter уже создан; и, так как он реализует IWindow, вы можете передать этот экземпляр в метод Create.

Когда в наличии имеется ViewModel, ее можно присвоить свойству DataContext инкапсулированного окна WPF Window (2). После завершения настройки Window становится полностью инициализированным и готовым к использованию.

В корне компоновки все фрагменты, входящие в приложение, могут быть связаны воедино:

```
IMainWindowViewModelFactory vmFactory =
    new MainWindowViewModelFactory(agent);

Window mainWindow = new MainWindow();
IWindow w =
    new MainWindowAdapter(mainWindow, vmFactory);
```

Переменная `mainWindow` стала свойством в листинге 6.3, и `vmFactory` соответствует полю с таким же именем. Когда вызывается метод `Show` или `ShowDialog` в результирующем `IWindow`, вызывается метод `EnsureInitialize` и все зависимости удовлетворяются.

Представленная отложенная инициализация, реализованная с помощью абстрактной фабрики, является дополнительным преимуществом, но здесь ее следует рассматривать в первую очередь как средство избавления от цикла. В данном случае нам немного повезло, так как WPF-`Window` уже использует внедрение свойства через свойство `DataContext`.

Всегда помните, что наилучшим способом избавиться от цикла является перепроектирование API таким образом, чтобы цикл исчез. Но в небольшом количестве случаев, где такое является невозможным или трудно достижимым, цикл должен ликвидироваться путем применения внедрения свойства как минимум в одном месте.

Такой подход позволяет скомпоновать весь граф объекта, кроме зависимости, ассоциированной со свойством. Когда граф объекта полностью скомпонован, нужный экземпляр может быть внедрен через свойство. Чтобы еще более улучшить код, логика получения значения свойством может быть вынесена в дополнительный класс и можно использовать абстрактную фабрику для передачи значения свойству в последний возможный момент.

Внедрение свойства указывает на то, что зависимость является необязательной, так что не следует вносить изменения без тщательной подготовки. Внедрение конструктора является предпочтительным в большинстве случаев, но оно может оказаться неудобным в некоторых случаях. Рассмотрим почему.

6.4. Использование сверхвнедрения конструктора

Если у вас нет каких-либо специальных требований, внедрение конструктора должно быть вашим основным шаблоном реализации внедрения. Но некоторые разработчики испытывают трудности, когда количество зависимостей возрастает. Они не любят конструкторы со слишком большим количеством параметров.

В этом разделе мы рассмотрим очевидную проблему увеличения количества параметров конструктора и поймем, почему это скорее хорошо, чем плохо. Как вы увидите, это не означает, что мы должны предпочитать длинные списки параметров в конструкторах, так что мы дополнительно рассмотрим, что можно сделать со слишком большим количеством аргументов конструктора. В конце данного раздела приведен пример.

6.4.1. Распознавание и разрешение сверхвнедрения конструктора

Когда список параметров конструктора чрезмерно разрастается, мы говорим о сверхвнедрении конструктора (Constructor Over-Injection[1]) и считаем это плохим кодом.

[1] *Jeffrey Palermo* Constructor over-injection smell — follow up, 2010: http://jeffreypalermo.com/blog/constructor-over-injection-smell-ndash-follow-up.

Это плохой код как таковой, он не порождается внедрениями зависимостей, а лишь усугубляется ими. Мы сразу можем отнестись к внедрению конструктора негативно, так как оно может привести к сверхвнедрению. Но мы должны ценить такую ситуацию за то, что она позволяет диагностировать наличие одной общей проблемы проектирования.

В данном разделе мы прежде всего оценим достоинства сверхвнедрения, а затем рассмотрим, как на нее реагировать.

Сверхвнедрение конструктора — как сигнал

Хотя внедрение конструктора легко разрабатывать и применять, оно становится неудобным, когда конструкторы приобретают подобный вид:

```
public MyClass(IUnitOfWorkFactory uowFactory,
    CurrencyProvider currencyProvider,
    IFooPolicy fooPolicy,
    IBarService barService,
    ICoffeeMaker coffeeMaker,
    IKitchenSink kitchenSink)
```

Я не стану упрекать тех, кто не любит конструкторы такого рода, но не ищите проблем в самом внедрении конструктора. Можно согласиться с тем, что конструктор с шестью параметрами — это плохой код, но он свидетельствует скорее о нарушении принципа единичной ответственности, чем о наличии проблем, порожденных внедрением конструктора.

СОВЕТ

Внедрение конструктора облегчает выявление мест, в которых нарушается принцип единичной ответственности.

Вместо того чтобы сосредотачиваться на затруднениях, связанных со сверхвнедрением конструктора, мы должны воспользоваться им как удобной возможностью, предлагаемой внедрением конструктора. Это сигнал, предупреждающий нас о случаях, когда класс начинает отвечать за слишком большое количество функций.

Я установил для себя порог, равный четырем аргументам в конструкторе. Когда я добавляю третий аргумент, то уже начинаю задумываться о правильности своего проекта, но для некоторых классов я допускаю наличие четырех аргументов. Ваш порог может быть другим, но если вы превысили его, то настало время подумать о рефакторинге.

Как мы изменим класс, разросшийся слишком сильно, зависит от конкретных соображений: уже используемой модели, домена, бизнес-логики и т. д. Хорошим решением, как правило, оказывается разбиение класса, ставшего некорректным (`God Class`), на более мелкие, которые решают более узкие задачи классы. Это делается на основе хорошо известных шаблонов проектирования.

Однако остаются случаи, когда требования бизнес-логики обязывают нас выполнять множество действий одновременно. Обычно такое случается на границе приложения.

Представьте себе операцию крупно-модульного веб-сервиса, вызывающую большое количество бизнес-событий. Один из способов моделирования таких

операций — это скрытие миллиардов зависимостей за фасадными сервисами (Facade Services).

Рефакторинг в фасадные сервисы

Существует много способов проектирования и реализации взаимодействующих компонентов таким образом, что они не нарушали принципа единичной ответственности. В главе 9 мы обсудим, каким образом паттерн проектирования «Декоратор» может быть использован для создания стека сквозных функциональностей приложения без инжектирования этих функций в потребителей в качестве сервисов. Такой подход позволяет избавиться от большого количества аргументов конструктора.

Тем не менее в некоторых сценариях требуется единственная точка входа для управления многими зависимостями.

Пример — это операция веб-сервиса, вызывающая сложное взаимодействие многих различных сервисов. Точка входа назначенной пакетной фоновой операции может быть связана с аналогичной проблемой.

Рисунок 6.11 показывает, как можно провести рефакторинг ключевых отношений, преобразовав их в фасадные сервисы. На верхней диаграмме потребитель имеет пять зависимостей, что является явным признаком нарушения принципа единичной ответственности. В то же время, если задачей потребителя является управление этими пятью зависимостями, мы не можем удалить ни одну из них. Вместо этого можно ввести фасадные сервисы, которые будут управлять частями этого отношения. На нижней схеме потребитель имеет только две зависимости, в то время как фасады имеют две и три зависимости.

Рефакторинг в фасадные сервисы — это не просто трюк, помогающий сократить количество зависимостей. Главным шагом здесь является определение естественных кластеров взаимодействий. На рис. 6.11 видно, что зависимости `A`–`C` образуют естественный кластер взаимодействия, и то же самое можно сказать относительно зависимостей `D` и `E`.

Положительным дополнительным эффектом является то, что определение таких естественных кластеров позволяет увидеть ранее скрытые отношения и доменные концепции. По ходу дела мы превращаем неявные концепции в явные. Каждый фасад становится сервисом, который заключает в себе взаимодействие как услугу более высокого уровня, и потребителю остается только управлять этими высокоуровневыми сервисами.

ПРИМЕЧАНИЕ

Фасадные сервисы являются абстрактными фасадами — отсюда и название.

Фасадные сервисы связаны с объектами параметров (Parameter Objects), но вместо комбинирования и выдачи компонентов фасадный сервис выдает только инкапсулированное поведение, в то же время скрывая свои составные части.

Очевидно, что мы можем повторять такой рефакторинг, если наше приложение является слишком сложным и потребитель управляет значительным количеством зависимостей для фасадных сервисов. Создание фасадов для фасадов позволяет хорошо решить данную проблему.

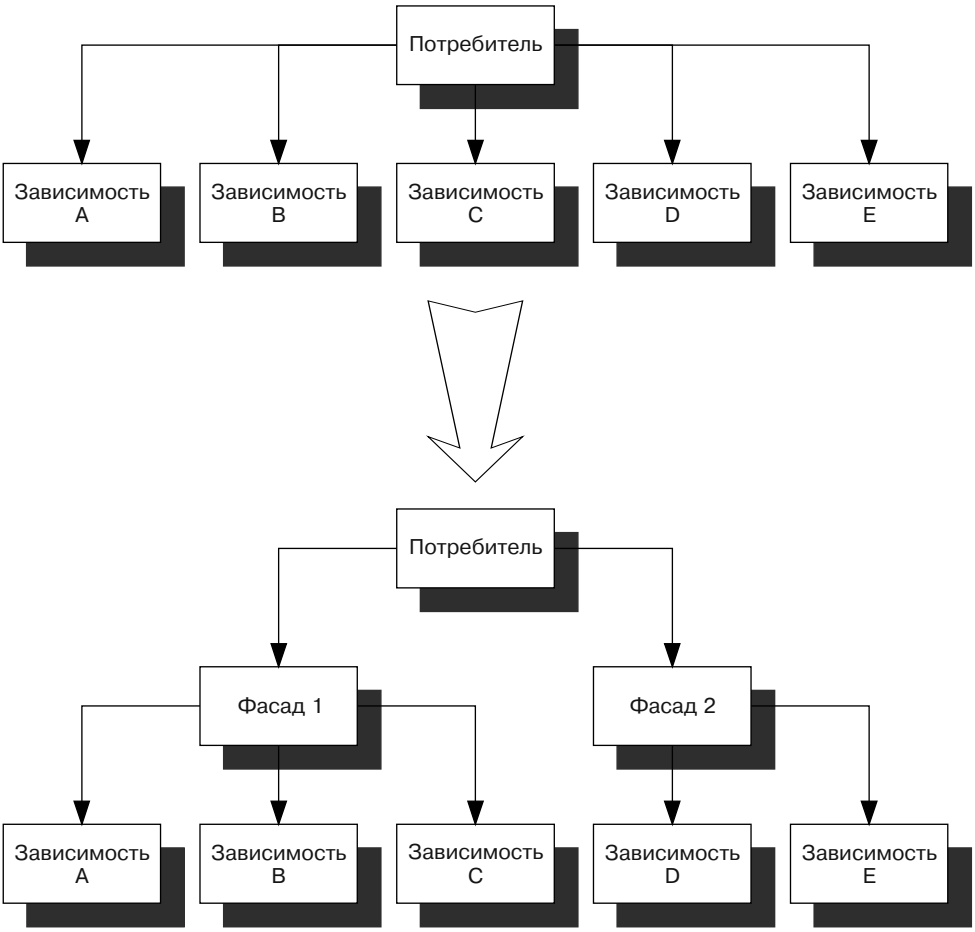


Рис. 6.11. Рефакторинг ключевых отношений в фасадные сервисы

На границах приложения (например, обеспечивающих работу с пользовательским интерфейсом или веб-сервисом) мы можем работать с набором достаточно крупно-модульных абстракций. Если мы проанализируем реализации зависимостей, то увидим, что за крупно-модульными сервисами скрываются более мелкие сервисы, которые, в свою очередь, представляют собой комбинации совсем уже мелких сервисов. Такое решение позволяет легко представить себе структуру всего уровня, в то же время гарантируя, что приложение соответствует принципу единичной ответственности.

Рассмотрим пример.

6.4.2. Пример: рефакторинг получения заказов

Демоприложение, которое мы периодически рассматриваем, должно иметь возможность получать заказы. Такая функция зачастую реализуется в виде

отдельного приложения или подсистемы, поскольку в данной точке семантика транзакции изменяется.

Во время поиска товаров и помещения их в покупательскую корзину можно в динамике пересчитывать цены, курсы конверсии, скидки; но после того как покупатель сделал заказ, все эти переменные должны быть сохранены и зафиксированы теми значениями, которые они имели в момент подтверждения заказа пользователем. Таблица 6.2 содержит описание процесса оформления заказа.

Таблица 6.2. Когда подсистема обработки заказов получает новый заказ, она должна выполнить определенный набор действий

Действие	Требуемые зависимости
Сохранить заказ	OrderRepository
Отправить подтверждение получения заказа на электронную почту покупателя	IMessageService
Известить учетную систему о сумме заказа	IBillingSystem
Выбрать наиболее подходящие хранилища для формирования и доставки заказа на основе списка товаров в нем и удаленности хранилища от адреса доставки	ILocationService, IInventoryManagement
Передать в выбранные хранилища заявки на формирование и доставку всего заказа или отдельных его частей	IInventoryManagement

Пять различных зависимостей нужны уже только для обработки получения заказа. Вообразите, сколько других зависимостей потребуется для реализации всех связанных с обработкой заказа операций!

Рассмотрим сначала, как будет выглядеть система, если использующий `OrderService` класс будет напрямую импортировать все эти зависимости. Затем вы увидите, как можно провести рефакторинг этой же функциональности с использованием фасадных сервисов.

Слишком много мелких зависимостей

Если вы реализуете `OrderService` таким образом, что он будет напрямую потреблять все пять зависимостей, структура подсистемы будет похожа на ту, что представлена на рис. 6.12.

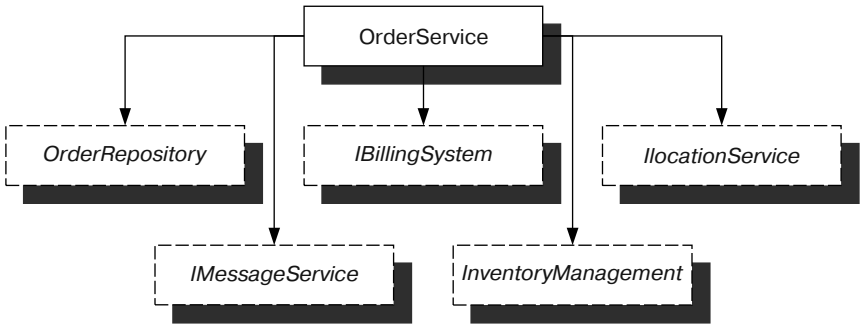


Рис. 6.12. OrderService имеет пять различных зависимостей, что является признаком нарушения принципа единичной ответственности

Если в классе `OrderService` используется внедрение конструктора (как и должно быть), вы получите конструктор с пятью параметрами. Это слишком много и свидетельствует о том, что `OrderService` функционально перегружен. С другой стороны, все эти зависимости необходимы, так как класс `OrderService` должен выполнять все требуемые функции, когда получает новый заказ.

Эта проблема может быть устранена путем перепроектирования `OrderService`.

Рефакторинг с внедрением фасадных сервисов

Первое, что нужно сделать, — выделить естественные кластеры взаимодействия для определения потенциальных фасадных сервисов. Сразу же обратите внимание на взаимодействие `ILocationService` и `IInventoryManagement`, поскольку вы используете их для нахождения ближайших хранилищ, позволяющих сформировать заказ. Это потенциально может быть сложный алгоритм; при этом после того как вы выберете хранилища, нужно будет известить их о полученном заказе.

Если вы еще раз проанализируете описанную функциональность, то заметите, что `ILocationService` является деталью реализации функции извещения подходящего хранилища о поступлении заказа. Полное взаимодействие может быть скрыто за интерфейсом `IOrderFulfillment`, как показано на рис. 6.13. Интересно, что обслуживание заказов выглядит немного похожим на концепцию домена; изменения заключаются только в том, что вы выявили неявную концепцию домена и сделали ее явной.

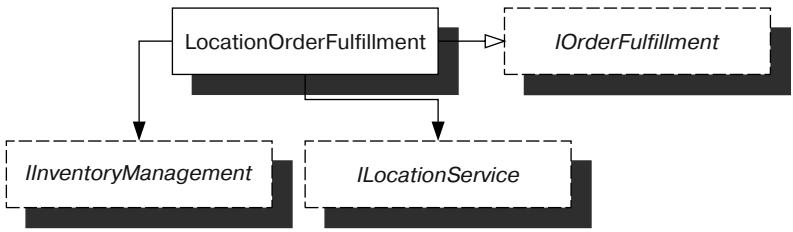


Рис. 6.13. Взаимодействие между IInventoryManagement и ILocationService реализовано в классе LocationOrderFulfillment, который реализует интерфейс IOrderFulfillment. Потребители интерфейса IOrderFulfillment не знают, что реализация имеет две зависимости

Используемая по умолчанию реализация `IOrderFulfillment` задействует две оригинальные зависимости, так что она имеет конструктор с двумя параметрами, что нас вполне устраивает. Кроме того, алгоритм нахождения наилучшего хранилища для данного заказа был инкапсулирован в повторно используемый компонент.

Проведенный рефакторинг объединил две зависимости в одну, но все равно в классе `OrderService` их осталось четыре. Нужно продолжить поиск возможностей агрегирования зависимостей в фасады.

Теперь можно заметить, что все требования включают извещение других систем о поступлении заказа. Это означает, что можно выделить общую абстракцию, которая будет моделировать нотификации (извещения) — возможно, это будет что-то вроде показанного ниже фрагмента кода:

```
public interface INotificationService
{
    void OrderAdded(Order order);
}
```

СОВЕТ

Паттерн проектирования «Событие домена» (Domain Event) — это другая хорошая альтернатива данному сценарию.

Любое извещение во внешнюю систему может быть реализовано с использованием данного интерфейса. Вы можете даже обернуть `OrderRepository` в `INotificationService`, но похоже, что класс `OrderService` потребует доступа к другим методам `OrderRepository` для реализации оставшейся функциональности. Рисунок 6.14 показывает, как реализуются другие извещения на основе `INotificationService`.

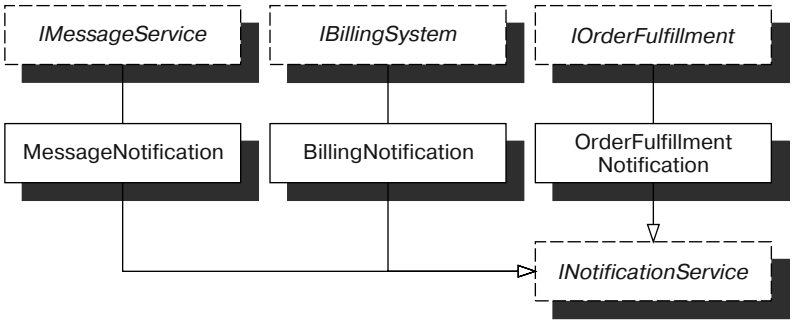


Рис. 6.14. Каждое извещение внешней системы может быть скрыто за INotificationService — даже новый интерфейс IOrderFulfillment, который был добавлен только что

Вы можете усомниться в полезности произведенных манипуляций, так как вы просто обернули каждую зависимость в новый интерфейс. Количество зависимостей при этом не уменьшилось, так выиграли ли вы что-нибудь?

Да. Поскольку все извещения реализуют один и тот же интерфейс, вы можете обернуть их в «Составной» (Composite). Получается другая реализация `Inotificationservice`, которая скрывает набор экземпляров `INotificationService` и вызывает метод `OrderAdded` на каждом из них.

С точки зрения концепции это тоже целесообразно, поскольку из высокоуровневого представления не удастся управлять деталями того, как `OrderService` извещает другие системы. Однако вы можете быть уверены, что такое извещение происходит. Рисунок 6.15 показывает окончательные зависимости для `OrderService`. `OrderRepository` оставлена отдельной зависимостью, поскольку приложению требуются ее дополнительные методы для реализации прочей функциональности `OrderService`. Все извещения скрыты интерфейсом `INotificationService`. Во время исполнения вы используете `CompositeNotificationService`, содержащий оставшиеся три извещения.

Произведенные изменения уменьшают количество зависимостей `OrderService` до двух, что уже гораздо лучше. Функциональность не изменилась, что позволяет говорить о правильно выполненном рефакторинге. С другой стороны, концепту-

альный уровень OrderService изменился. Его задача теперь заключается в приеме заказа, его сохранении и извещении других систем. Детали того, какие системы извещаются и как это реализовано, перемещены на более низкий и более детальный уровень.

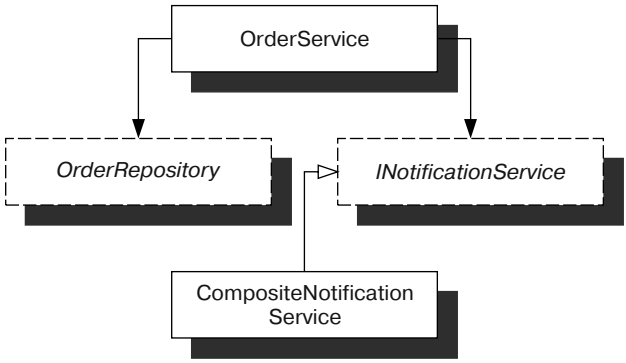


Рис. 6.15. Окончательный вариант OrderService с проведенным рефакторингом зависимостей

Даже если вы последовательно использовали повсюду внедрение конструктора, еще не встречалась ситуация, когда единственный конструктор класса требовал бы более двух параметров (CompositeNotificationService имеет единственный агрумент IEnumerable<INotificationService>).

Сверхвнедрение конструктора — это не проблема, возникающая из-за внедрения зависимостей вообще или внедрения конструктора в частности. Скорее это сигнал о том, что анализируемый класс отвечает за слишком большой объем задач. Плохой код получается из-за самого класса, а не из-за внедрения конструктора; и как всегда мы должны рассматривать данную ситуацию с точки зрения возможности улучшить код.

Существует множество способов провести рефакторинг на основе паттернов, и один из них заключается в том, чтобы добавить фасадные сервисы, моделирующие концепции на верхнем уровне абстракции. Так устраняются нарушения принципа единичной ответственности, и часто в ходе рефакторинга удается выявить скрытые до сих пор доменные концепции.

Это лишь один из многих примеров того, как внедрение зависимостей помогает разрабатывать более качественный код. Поскольку слабое связывание имеет очень большое значение, мы хотели бы быть уверены, что слабо связанный код останется таковым и после рефакторинга. В следующем разделе обсуждаются способы добиться этого.

6.5. Мониторинг связанности

Слабое связывание имеет большое значение, но на удивление легко можно написать сильно связанный код. Все, что требуется для появления жестких связей, — это новичок-разработчик и немного невнимательности. В Visual Studio очень просто

добавляются новые ссылки на существующий проект, но это как раз то, чего мы хотели бы избежать. Должен соблюдаться порядок, гарантирующий, что каждый модуль сфокусирован на решение одной конкретной задачи.

В данном разделе мы рассмотрим некоторые приемы и инструменты, которые могут оказаться полезными, если мы хотим обеспечить слабую связанность кода. Возможно, мы захотим защитить наш код от своих же ошибок, или начинающим разработчикам в команде требуется немного помощи.

Ничто не может препятствовать человеческому взаимодействию, когда речь заходит о передаче знания. Парное программирование — идеальный выход, но также не помешает заменить ручной анализ автоматизированными инструментами. В следующих подразделах мы рассмотрим, за счет чего может оказаться полезным автоматизированное тестирование, на примере инструмента, называемого NDepend.

6.5.1. Модульное тестирование связанности

Если у нас имеется набор модульных тестов, которые мы выполняем с определенной периодичностью, мы можем быстро добавить несколько таких тестов, которые проверяют зависимости и сообщают об ошибке, если какая-либо зависимость оказывается небезопасной. Используя систему типов .NET, можно легко написать модульный тест, который будет циклически просматривать все ссылки на сборки и сообщать об ошибке, если будет обнаружен тип, которого не должно там быть[1].

Наше демоприложение уже содержит некоторое количество модульных тестов, так что несложно добавить еще несколько. Листинг 6.4 демонстрирует модульный тест, защищающий модуль презентационной логики от прямого обращения к использующему SQL Server модулю доступа к данным.

Листинг 6.4. Обеспечение слабого связывания с помощью модульного теста

```
[Fact]
public void SutShouldNotReferenceSqlDataAccess()
{
    // Вспомогательная настройка
    Type sutRepresentative = typeof(HomeController);
    var unwanted = "Ploeh.Samples.Commerce.Data.Sql";
    // Испытание системы
    var references =
        sutRepresentative.Assembly
        .GetReferencedAssemblies();
    // Проверка результатов
    Assert.False(
        references.Any(a => a.Name == unwanted),
        string.Format(
             "{0} should not be referenced by SUT",
             unwanted));
    // Разрыв
}
```

[1] Эта идея была представлена в работе *Glenn Block* PrismShouldNotReferenceUnity, 2008: http://blogs.msdn.com/b/gblock/archive/2008/05/05/prismshouldnotreferenceunity.aspx.

Этот тест проверяет наличие зависимостей в модуле презентационной логики. Для получения списка ссылок вы должны запросить сборку. Вы можете получить сборку из любого типа, содержащегося в этой сборке, поэтому можно использовать имя любого типа. Часто бывает целесообразно выбирать тип, который, как вы предполагаете, будет использоваться в течение длительного времени. Ведь в противном случае придется переписывать тест, если выбранный тип будет удален. В данном тесте применен тип `HomeController`, потому что веб-сайт будет всегда гарантированно иметь домашнюю страницу.

Вам также понадобится указать сборку, на которую не должно быть ссылок. Вы можете использовать тот же подход и указать тип из сборки, но это будет означать, что вам нужна ссылка на эту сборку из модульного теста. Это не так плохо, как ссылка на нежелательную сборку из рабочего кода, но все же при этом создается искусственная связь между двумя библиотеками — вы можете сказать, что возникает порочный круг. Хотя желательно обеспечить безопасность типов, слабое связывание имеет гораздо более высокий приоритет в данном случае, поэтому вместо указанного подхода следует идентифицировать нежелательную сборку с помощью строки. Обязательно прочитайте приведенное далее обсуждение альтернативных вариантов.

Получение сборок, на которые имеются ссылки, через указанный тип не сложнее вызова отдельного метода. Теперь можно использовать простой запрос LINQ для проверки, что ссылки на нежелательные сборки отсутствуют. В коде `assertion` (утверждение, составная часть модульного теста. — *Примеч. пер.*) вы можете также выдать сообщение об ошибке, если таковая произойдет.

ПРИМЕЧАНИЕ

Код assertion в приведенном примере использует простой LINQ-запрос, но вы можете заменить его на цикл foreach, если программируете на .NET 3.0 или более ранней версии.

СОВЕТ

Можно изменить логику на зеркальную и написать тест таким образом, что только определенные ссылки из заданного списка будут разрешены, а все остальные будут рассматриваться как запрещенные.

ТЕСТИРОВАНИЕ СВЯЗАННОСТИ МЕТОДОМ «КРАСНЫЙ/ЗЕЛЕНЫЙ/РЕФАКТОРИНГ»

Если вы применяете разработку через тестирование, то вы используете цикл разработки, называемый «Красный/зеленый/рефакторинг» (Red/Green/Refactor), в котором вы сначала пишете тест, дающий ошибку, затем делаете его работоспособным, и, наконец, изменяете код таким образом, чтобы упростить его поддержку.

Оказывается, что написать ошибочный тест, предотвращающий тесное связывание, — это задача более сложная, чем вы могли вообразить. Даже если целевой проект, разрабатываемый в Visual Studio, имеет ссылку на нежелательную зависимость, компилятор включит данную ссылку в работу лишь в случае, если эта ссылка используется.

Таким образом, чтобы такой тест давал ошибку, необходимо, во-первых, добавить ссылку, которую мы не хотим иметь в окончательном варианте, а затем написать строку кода-заглушки, использующего тип из нежелательной зависимости. После того как мы увидим, что такой тест дает ошибку, мы можем вернуться назад, чтобы сделать тест рабочим. Все это, как правило, не слишком большие беды, если тестируемая библиотека *уже* нарушает ограничения связанности.

В предыдущем примере модульный тест был добавлен в набор тестов для модуля презентационной логики. Рисунок 6.16 показывает ссылки в действии. Библиотека `PresentationLogicUnitTest` — это набор тестов для библиотеки `PresentationLogic`. Она должна ссылаться на целевую библиотеку и на совместно используемую абстракцию, определенную в модели домена. Поскольку модель домена не тестируется посредством `PresentationLogicUnitTest`, модуль `DomainModel` показан со светло-серой тенью.

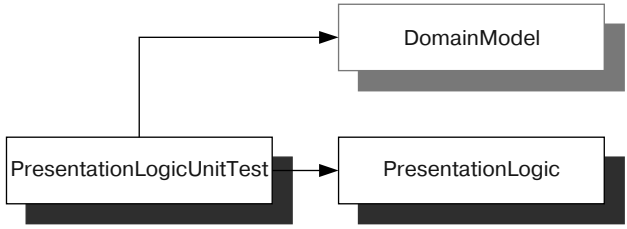


Рис. 6.16. Ссылки в действии

Код из листинга 6.4 позволяет идентифицировать нежелательную сборку с помощью простой строки, но более безопасно определять ее с помощью содержащегося в сборке типа. Однако для теста потребуется добавить ссылку на использующий SQL Server модуль доступа к данным, как показано на рис. 6.17. Когда вы обеспечиваете безопасность типов добавлением типа-представителя из библиотеки `SqlDataAccess` в `PresentationLogicUnitTest`, создается новая зависимость для набора тестов, причиной появления которой будет наше желание убедиться, что никаких лишних ссылок не имеется в библиотеке `PresentationLogic`. Забавно, не правда ли?

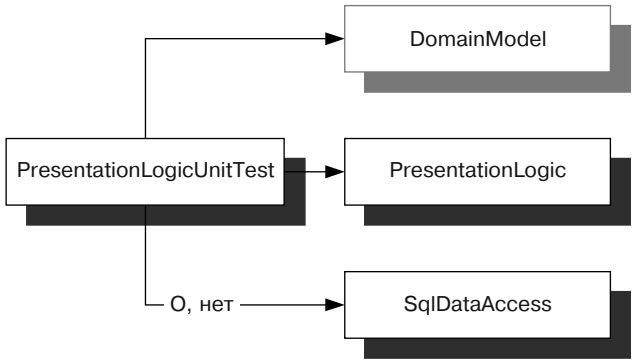


Рис. 6.17. Добавление ссылки на использующий SQL Server модуль доступа к данным

Может показаться, что добавление лишней ссылки в проекте модульного тестирования не является большой бедой, но ее наличие имеет больше недостатков, чем кажется на первый взгляд.

НЕПРЯМЫЕ ЗАВИСИМОСТИ

Подробный анализ причин — почему проект модульного тестирования должен ссылаться только на проект, который он тестирует, находится вне темы этой книги, но общей проблемой является то, что создается непрямая ссылка между `PresentationModel` и `SqlDataAccess`. Хотя оба эти проекта могут существовать и компилироваться друг без друга, проект модульного тестирования требует наличия их обоих.

Такая непрямая зависимость может быть разорвана только удалением модульного теста, приведшего к ее возникновению. Но ведь модульные тесты создаются для того, чтобы выполняться, поэтому их удаление является крайне нежелательным.

Если нужно сохранить такие, предотвращающие сильное связывание, модульные тесты в существующем проекте модульного тестирования, то цена добавления жестких ссылок на все нежелательные сборки становится слишком большой. Наилучшим вариантом в таком случае будет идентификация всех нежелательных зависимостей с помощью строк, как показано в листинге 6.4.

Недостатком данного подхода является то, что если изменится имя запрещенной сборки, то тест станет бесполезным — и даже вредным, поскольку создается ощущение защищенности кода, хотя на самом деле ее не существует.

Это не является главной проблемой, если у нас есть основания полагать, что названия сборок являются стабильными. Когда же это не выполняется, следует подобрать другую стратегию.

6.5.2. Интеграционное тестирование связывания

Имеются убедительные причины, по которым проекты модульного тестирования должны ссылаться только на тестируемые проекты. Чтобы обеспечить надежность в случае изменения имен сборок, нам время от времени могут потребоваться типизированные ссылки на нежелательные зависимости. Такое требование может показаться противоречивым, но мы можем устранить такую путаницу, добавив новый проект интеграционного тестирования.

Вы можете добавить новый тестовый проект в решение `Commerce` и проставить все необходимые ссылки. Рисунок 6.18 отображает это решение; и хотя оно немного напоминает ситуацию, показанную на рис. 6.17, отличие состоит в том, что для интеграционного теста все ссылки допустимы и корректны.

ИНТЕГРАЦИОННЫЕ ТЕСТЫ

Интеграционный тест — это еще одна разновидность автоматизированного тестирования на уровне API. Различие между модульным и интеграционным тестом заключается в том, что модульный тест обрабатывает изолированный модуль. Интеграционный же тест фокусируется на проверке того, насколько корректно отдельные модули (часто из разных библиотек) интегрируются друг с другом.

По определению проект интеграционного тестирования может ссылаться на все зависимости, требуемые для его выполнения, поэтому он хорошо подходит для включения тестов, так или иначе ограничивающих архитектуру.

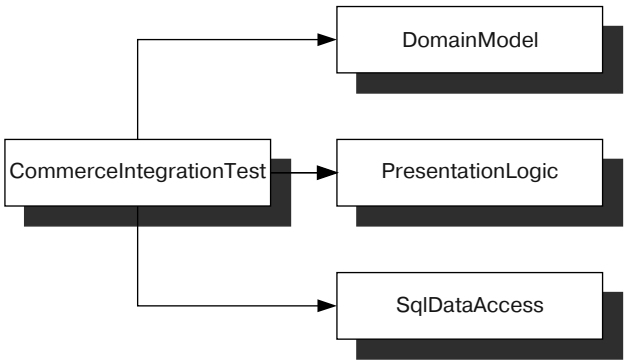


Рис. 6.18. Проект CommerceIntegrationTest включает автоматизированные тесты, гарантирующие, что отношения между модулями корректны. В отличие от модульных тестов, набор интеграционных тестов может содержать столько ссылок, сколько требуется для проведения тестирования

Набор интеграционных тестов тесно связан с конкретным набором модулей, поэтому его вряд ли возможно переиспользовать (для других модулей). В него должны входить только такие тесты, которые могут быть абсолютно определены как интеграционные. К этой категории также могут быть отнесены тесты, защищающие от нежелательного связывания. Листинг 6.5 показывает типизированный эквивалент теста, представленного в листинге 6.4. Он построен по тому же принципу и отличается возможностью идентификации нежелательной зависимости.

Листинг 6.5. Внедрение слабого связывания с помощью интеграционного теста

```
[Fact]
public void PresentationModuleShouldNotReferenceSqlDataAccess()
{
    // Вспомогательная настройка
    Type presentationRepresentative =
        typeof(HomeController);
    Type sqlRepresentative =
        typeof(SqlProductRepository);
    // Испытание системы
    var references =
        presentationRepresentative.Assembly
        .GetReferencedAssemblies();
    // Проверка результатов
    AssemblyName sqlAssemblyName =
        sqlRepresentative.Assembly.GetName();
    AssemblyName presentationAssemblyName =
        presentationRepresentative.Assembly.GetName();
    Assert.False(references.Any(a =>
        AssemblyName.ReferenceMatchesDefinition(
            sqlAssemblyName, a)),
        string.Format(
            "{0} should not be referenced by {1}",
```

1 Получить сборки, на которые есть ссылки

2 Получить имена сборок

3 Поиск нежелательной зависимости

```
            sqlAssemblyName,
            presentationAssemblyName));
    // Разрыв
}
```

Теперь, когда у вас имеются ссылки на все необходимые зависимости, можно подобрать такие типы из каждого модуля, которые можно было бы использовать для представления их сборок. В противоположность предыдущему примеру вы можете идентифицировать обе с применением типизации.

Так же как и ранее, вы получаете список всех сборок, на которые ссылается библиотека `PresentationLogic` ❶. Используя имя `AssemblyName` каждой сборки ❷, вы затем удостоверяетесь, что в число ссылок не входит ссылка на построенную на основе SQL Server сборку ❸. Встроенный статический метод `ReferenceMatchesDefinition` сравнивает `AssemblyNames`.

Возможно, вы заметили, что тесты в листингах 6.4 и 6.5 подобны друг другу. Вы можете написать новые тесты типа того, что представлен в листинге 6.5, путем смены двух типов-представителей с сохранением остальной логики в неизменном виде.

Следующим логическим шагом будет выделение общей части теста в параметризованный тест (Parameterized Test). Это позволит вам сформировать простой список почти что декларативных тестов, определяющих, что разрешено и что запрещено в данной конкретной подборке модулей.

Модульные и интеграционные тесты предоставляют великолепные возможности, если вы уже применяете автоматизированное тестирование на уровне API. Если нет, вы должны начать делать это уже с сегодняшнего дня, хотя существуют и другие альтернативы.

6.5.3. Использование NDepend для отслеживания связей

Если по какой-то невероятной причине вы не будете использовать модульное тестирование, вы можете воспользоваться инструментом, называемым NDepend (http://ndepend.com), который поможет обнаружить нежелательные связи, созданные вами или участниками вашей команды.

NDepend — это коммерческий программный продукт, анализирующий проекты или решения и генерирующий массу отчетов и статистических данных, касающихся анализируемого кода. Например, он может генерировать графы зависимостей, не похожие на те, которые вы встречали в этой книге. Если проанализировать первоначальное решение Мэри, описанное в главе 2, мы обнаружим граф, представленный на рис. 6.19. По умолчанию NDepend включает все зависимости, в том числе модули из BCL. Размер прямоугольников отражает количество строк кода в каждом модуле, а толщина стрелок соответствует количеству участников данной ссылки.

Данная схема выглядит сложной; для ее упрощения можно скрыть модули, относящиеся к BCL, и переформировать представление. На графе на рис. 6.20 были вручную удалены модули, относящиеся к BCL, а прямоугольники и стрелки сделаны одинаковых размеров.

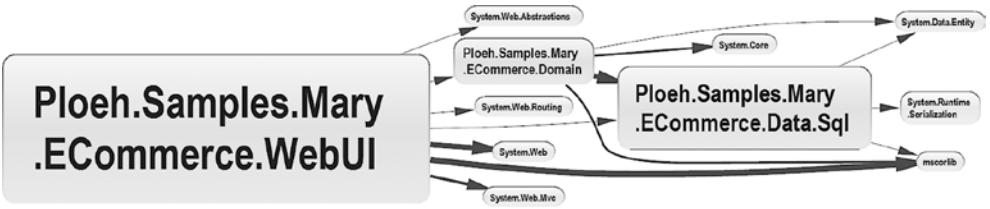


Рис. 6.19. Граф зависимостей, сгенерированный NDepend для разработанного Мэри коммерческого решения

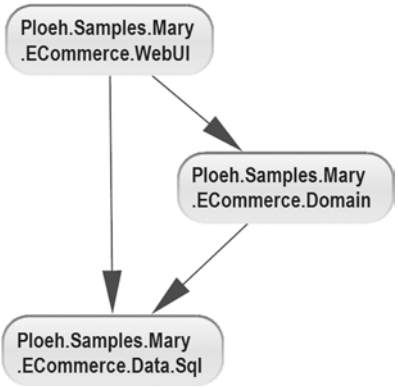


Рис. 6.20. Модифицированный граф, сформированный NDepend для разработанной Мэри программы

Не кажется ли вам знакомым рис. 6.20? Если вы обладаете хорошей эйдетической (образной) памятью, вы можете вспомнить рис. 2.10; в противном случае вы можете вернуться к нему. Обратите внимание, как разделяются одни и те же структуры и иллюстрируются одни и те же отношения.

Функционал NDepend далеко не ограничивается рисованием красивых схем. Одна из его наиболее мощных возможностей — наличие специального языка, называемого Code Query Language (CQL), позволяющего анализировать код на предмет получения большого комплекса показателей. Синтаксис этого языка напоминает SQL.

Если бы у Мэри имелся написанный на CQL проверочный код, который бы она выполнила прежде, чем разрабатывать свое решение, то она была бы предупреждена о проблемах до того, как случится что-нибудь неприятное. Ниже приведен запрос, который мог бы избавить ее от множества неприятностей:

```
WARN IF Count > 0 IN SELECT ASSEMBLIES WHERE
IsDirectlyUsing "ASSEMBLY:Ploeh.Samples.Mary.ECommerce.Data.Sql" AND
NameIs "Ploeh.Samples.Mary.ECommerce.Domain"
```

Будучи выполненным, этот CQL-запрос выдает предупреждение, если модуль `Domain` напрямую ссылается на использующий SQL Server модуль `DataAccess`. В решении Мэри этот запрос наверняка выдал бы предупреждение.

Для решения можно написать столько CQL-запросов, сколько мы захотим, и затем либо запускать их с помощью визуального редактора, либо автоматизировать процесс, использовав инструмент командной строки. В обоих случаях генерируются XML-файлы, содержащие результаты анализа, что позволяет разработать свои собственные средства автоматизации, если мы хотим включить такой этап в автоматизированный процесс построения.

ПРИМЕЧАНИЕ

Я лишь вскользь продемонстрировал возможности NDepend. Он может еще очень многое, но я хотел сфокусироваться на его возможностях, касающихся связывания.

NDepend и автоматизированные тесты — это два способа автоматического мониторинга кода с целью убедиться, что недопустимые зависимости случайно не проникли в него. Мы можем использовать один из этих способов или оба их одновременно как часть автоматизированного теста верификации сборки (Build Verification Test, BVT) или мер непрерывной интеграции (Continuous Integration, CI).

В больших фрагментах кода, поддерживаемых большими командами, это может защитить нас от значительной беды. Хотя мы не можем уследить за всем происходящим и произвести ручную проверку всех исправлений в коде, автоматизированные инструменты могут предупредить нас, когда случаются нежелательные явления.

ВНИМАНИЕ

Некоторые инструменты могут давать «ложно-положительные результаты», поэтому не принимайте слепо на веру все их сообщения о возникших проблемах. Всегда используйте свой опыт и знания для анализа полученных предупреждений. Отклоняйте их, если у вас нет полной уверенности.

Рассматривайте каждый инцидент с должным вниманием, лично участвуйте в его разрешении, если он представляет реальную проблему.

Обязательно применяйте автоматизированные инструменты для мониторинга связывания в кодах большого объема. Это поможет избежать случайного появления сильного связывания, пока вы решаете другие проблемы, описанные в этой главе.

6.6. Резюме

Внедрение зависимостей не представляет особых сложностей, если понимать несколько основных принципов, но по мере обучения, у вас непременно появятся проблемы, из-за которых вы какое-то время будете чувствовать себя в тупике. В этой главе мы попытались разрешить некоторые наиболее часто встречающиеся проблемы.

Один из наиболее разносторонних и полезных паттернов проектирования, связанных с внедрениеми зависимостей, называется «Абстрактная фабрика». Мы можем использовать ее для преобразования примитивных типов значений времени исполнения, таких как строки или числа, введенные пользователями, в экземпляры сложных абстракций. Абстрактные фабрики могут применяться

в комбинации с интерфейсом `IDisposable` для имитации краткосрочных зависимостей, таких как соединения с внешними ресурсами.

СОВЕТ

Преобразуйте значения, используемые во время исполнения, в зависимости. Это делается с помощью абстрактных фабрик.

Имитируйте соединения с помощью абстрактных фабрик, создавая, таким образом, одноразовые зависимости.

Иногда возникает проблема циклических зависимостей. Они обычно возникают при использовании слишком «требовательных» API. Чем сильнее API в ходе разработки был *завязан* на парадигме запросов, тем более вероятным становится появление циклов. Циклов можно избежать, если следовать принципу Голливуда (говори, не спрашивай). Не возвращающие значений методы (методы с `void`-сигнатурами) могут быть перепроектированы как события, которые часто могут использоваться для ликвидации циклов. Если перепроектирование оказывается невозможным, цикл можно разорвать путем замены внедрения конструктора на внедрение свойства. Однако такое изменение должно быть тщательно продумано, поскольку оно меняет семантику на стороне потребителя.

СОВЕТ

Избавляйтесь от циклов с помощью внедрения свойств.

Внедрение конструктора должно оставаться приоритетным вариантом внедрения зависимостей. Его дополнительным достоинством является то, что оно позволяет со всей очевидностью выявить места нарушения принципа единичной ответственности. Если класс имеет слишком много зависимостей, это служит явным сигналом необходимости перепроектирования. Возможно, его следует разбить на несколько более мелких классов, но в некоторых случаях функциональность должна быть сконцентрирована в одном классе.

СОВЕТ

Разрешайте сверхвнедрение конструктора путем рефакторинга в фасадные сервисы.

В таких случаях можно повысить уровень абстракции, добавив слой фасадных сервисов между потребителем и исходными зависимостями. Проведение такого рефакторинга часто дает положительный дополнительный эффект, заключающийся в том, что некоторые из введенных фасадных сервисов выявляют скрытые ранее неявные доменные концепции. Выявление скрытых концепций и преобразование их в явные оптимизирует доменную модель.

При осуществлении такого тонкого рефакторинга нельзя забывать о целостной картине. Автоматизированные тесты или инструменты могут помочь отслеживать появление сильного связывания в каких-либо фрагментах кода.

Если разрабатывается большое количество модульных тестов (и особенно, если ведется разработка через тестирование), сильное связывание будет сразу же проявляться в форме сложного и хрупкого кода тестов. Не исключено, что станет вообще невозможно проводить модульное тестирование больших частей приложения.

СОВЕТ

Используйте автоматизированные тесты для стимулирования слабого связывания.

Если не применяется модульное тестирование, случаи появления сильного связывания могут быть упущены, но известны многие их симптомы: по мере развития программного кода его становится все труднее и труднее сопровождать. Прекрасный и ясный дизайн, разработанный вначале, постепенно превращается в так называемый спагетти-код. Добавление новой функции затрагивает код во многих, казалось бы, не связанных модулях приложения.

В этой главе описаны решения проблем, обычно встречающихся при внедрении зависимостей. Вместе с двумя предшествующими главами данная глава образует каталог шаблонов, антишаблонов и методов рефакторинга. Этот каталог формирует часть 2 книги. В части 3 мы рассмотрим три измерения внедрения зависимостей: композицию объектов, управление жизненным циклом и перехват.

ЧАСТЬ 3

Самостоятельное создание внедрения зависимостей

В главе 1 я сделал краткий обзор трех измерений инъекции зависимостей: *композиции объектов*, *управления жизненным циклом* и *перехвата*. В этой части книги я расширю это описание в трех отдельных главах. Многие контейнеры внедрения зависимостей обладают возможностями, напрямую связанными с этими тремя измерениями. Одни из них функционально связаны со всеми тремя измерениями, другие же поддерживают лишь некоторые такие возможности.

Однако, поскольку контейнер внедрения зависимостей не является средством, обязательным к использованию, я чувствую, что важнее будет разъяснить основополагающие принципы и приемы, которые обычно применяются в контейнерах для реализации этих возможностей. Часть 3 показывает, как вы можете сделать это самостоятельно, без использования контейнера. Потенциально вы можете использовать эту информацию для разработки собственного контейнера внедрений (но, пожалуйста, не делайте этого — миру не нужен еще один контейнер) или применять внедрения зависимостей вообще без контейнера — я называю этот вариант *внедрением зависимостей для бедных*. Главная цель этой части книги заключается в описании основных механизмов композиции объектов, управления жизненным циклом и перехвата без использования конкретного контейнера внедрения зависимостей. Мне кажется, что если бы я применял какой-либо контейнер, мне не удалось бы отделить основные принципы от деталей конкретного API.

Глава 7 показывает, как осуществляется композиция объектов в разных фреймворках, таких как ASP.NET MVC, WPF, WCF и т. д. Не все фреймворки одинаково хорошо поддерживают внедрение зависимостей, и даже те, которые его поддерживают, делают это по-разному. Для каждого фреймворка может оказаться непросто идентифицировать шов, который обеспечивает внедрение зависимостей в этом фреймворке. Но как только шов будет найден, вы получите решение для всех приложений, использующих этот фреймворк. В главе 7 я проделаю эту работу для большинства известных фреймворков приложений .NET. Данную главу можно рассматривать как каталог используемых фреймворков со швами.

Даже если выбранный вами фреймворк не описывается здесь, я попытаюсь объяснить практически все возможные виды ограничений, связанных со фреймворками. Например, с точки зрения внедрения зависимостей PowerShell является наиболее ограничивающим типом фреймворка, поэтому я использую его в качестве примера. Вы должны уметь переносить предложенное решение на другие подобные фреймворки, даже если они не полностью представлены.

Хотя компоновка объектов не представляет особой сложности в случае использования «внедрения для бедных», вы начнете понимать преимущества реального контейнера внедрений после знакомства с управлением жизненным циклом. Существует возможность полностью управлять жизненным циклом различных объектов в графе, но при этом требуется разработка большего объема дополнительного кода, чем для простой компоновки объектов. Причем этот код не увеличивает потребительскую ценность приложения.

Помимо объяснений основ управления жизненным циклом, глава 8 содержит еще и каталог распространенных жизненных стилей. Этот каталог служит в качестве лексикона для обсуждения жизненных циклов в части 4, поэтому даже если вы

не захотите реализовывать какой-либо из них вручную, знание и понимание их в любом случае окажется очень полезным.

В главе 9 мы рассмотрим часто возникающую проблему реализации сквозных аспектов приложения с использованием компонентного подхода. Переходя от простого применения паттерна проектирования «Декоратор» к перехватам во время исполнения, мы найдем способы компоновки слабо связанных приложений в форме модулей. Я считаю эту главу самой важной в книге — в этом месте многие читатели предварительных версий книги заявляли, что они начали осознавать истинные возможности чрезвычайно мощного способа моделирования программного обеспечения.

Хотя я использую «внедрение для бедных» для демонстрационных примеров, я не рекомендую применять этот подход в профессиональной деятельности. Имеется много хороших свободных контейнеров внедрений для платформы .NET. Это объясняет, почему часть 4 выделена для детального представления API избранных контейнеров.

7 Компоновка объектов

Меню:

- консольные приложения;
- ASP.NET MVC;
- Windows Communication Foundation;
- Windows Presentation Foundation;
- ASP.NET (Web Forms);
- PowerShell.

Приготовление деликатесной еды для гурманов — это нелегкое дело, особенно если вы собираетесь сами ее откушать. Вы не можете одновременно питаться и готовить пищу, к тому же многие блюда подаются к столу свежеприготовленными.

Профессиональные повара знают, как решать подобные проблемы. Среди многих профессиональных приемов они применяют главный принцип Mise en Place (режиссуры места), что может быть приблизительно переформулировано как «все на месте»: все, что может быть приготовлено заранее, готовится заранее. Овощи очищаются и шинкуются, мясо нарезается, тарелки расставляются и т. д.

Компоненты еды готовятся, как только появляется возможность. Если частью десерта является мороженое, оно может быть приготовлено за день до банкета. Если первое блюдо включает устрицы, они могут быть очищены за несколько часов до подачи на стол. Даже такой тонкий компонент, как беарнский соус, может быть приготовлен за час до еды. Когда гости соберутся за столом, потребуются только завершающие штрихи: подогреть соус, пока дожаривается мясо, и тому подобное. Во многих случаях такая окончательная *компоновка* стола потребует не более 5 или 10 минут. Рисунок 7.1 иллюстрирует данный процесс.

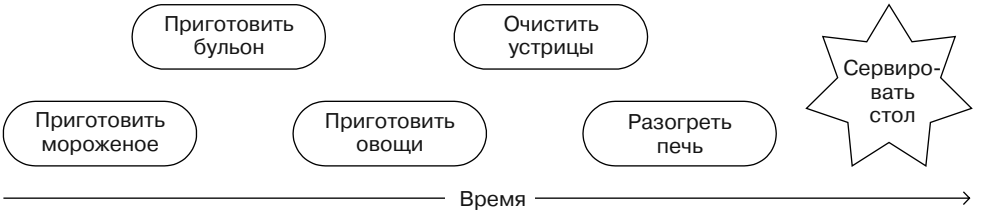


Рис. 7.1. Принцип Mise en Place означает приготовление всех блюд заранее, чтобы окончательная сервировка стола осуществлялась максимально легко и быстро

Принцип Mise En Place похож на разработку слабо связанного приложения с использованием внедрения зависимостей. Мы можем написать все требуемые компоненты заранее и скомпоновать их только в самый последний момент.

ПРИМЕЧАНИЕ

В подразделе 3.3.1 я сравнивал корень компоновки с концепцией последнего ответственного момента (Last Responsible Moment) из гибкой разработки[1]. Сравнение корня компоновки с Mise en Place — это хорошая аналогия, хотя корень определяет совершенно иной феномен: сборку.

Как всегда со всеми аналогиями, мы можем принять их только до определенной степени. Различие заключается в том, что в поварском деле приготовление еды и компоновка разделены во времени, тогда как при разработке приложений такое разделение осуществляется по модулям и слоям. Рисунок 7.2 показывает, как происходит компоновка в корне компоновки (часто в слое пользовательского интерфейса).

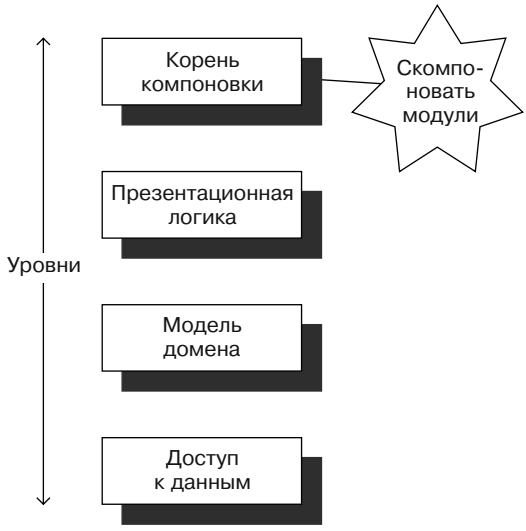


Рис. 7.2. Корень компоновки собирает все независимые модули приложения. В отличие от Mise en Place, это не делается как можно более поздно, а осуществляется в месте, где требуется интеграция различных модулей

Во время исполнения все начинается с компоновки объекта. После того как граф объекта будет сформирован, компоновка объекта завершается и все составляющие компоненты занимают свои места.

Хотя компоновка объектов является краеугольным камнем внедрения зависимостей, понять ее очень просто. Вы уже знаете, как можно добиться этого, поскольку компоновка объектов осуществляется каждый раз, когда создаются объекты, содержащие другие объекты. В разделе 3.3 мы рассматривали основы того, где и каким образом должна осуществляться компоновка приложения. Как следствие, я не

[1] Смотрите, например, *Mary Poppendieck, Tom Poppendieck* Implementing Lean Software Development: From Concept to Cash. — New York: Addison-Wesley, 2007.

собираюсь тратить следующие десятки страниц на объяснение того, как следует компоновать объекты.

Вместо этого я хочу помочь вам преодолеть некоторые трудности, которые могут возникнуть при компоновке объектов. Эти трудности возникают не из-за самой внедрения зависимостей, а скорее из-за фреймворков приложений, в которых вы компонуете свои объекты. Эти проблемы, как правило, специфичны для каждого фреймворка, так что придется анализировать каждую ситуацию. Как подсказывает мой опыт, эти сложности являются наиболее серьезными препятствиями на пути успешного внедрения зависимостей, поэтому я сфокусируюсь на них. Такой подход делает данную главу менее теоретической и более практической, чем предыдущие главы.

СОВЕТ

Если вы хотите познакомиться только со способом реализации внедрения зависимостей в том фреймворке, который вы выбрали для себя, можете пропустить остальной материал этого раздела. Каждый раздел может изучаться независимо от других.

Не составляет труда скомпоновать полную иерархию зависимостей для приложения, если мы имеем полный контроль над жизненным циклом приложения (что имеет место в приложениях, запускаемых из командной строки).

Тем не менее большинство фреймворков (ASP.NET, WCF и др.) в .NET включают инверсию управления, которая иногда может усложнить процесс внедрения зависимостей. Понимание швов для каждого фреймворка — это ключ к применению внедрения зависимостей для этого конкретного фреймворка. Как изображено на рис. 7.3, в данной главе мы покажем, как можно реализовать корень компоновки для некоторых распространенных фреймворков из библиотеки `Base Class Library` (BCL). Каждый раздел написан так, что его можно читать независимо от других.

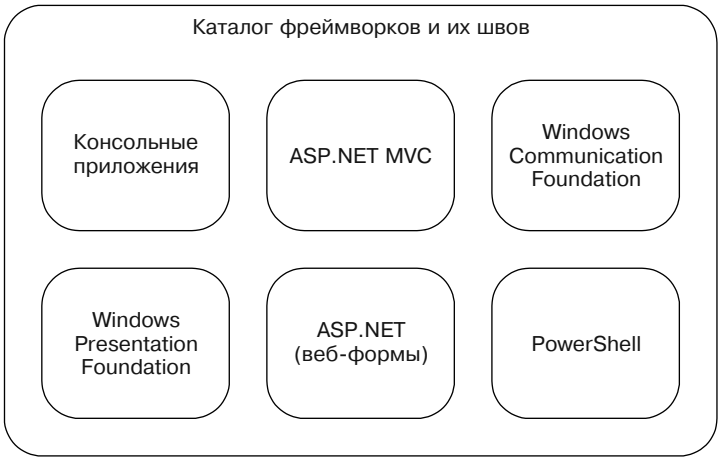


Рис. 7.3. Структура данной главы имеет форму каталога различных фреймворков из BCL, а также швов, которые они могут содержать для реализации внедрения зависимостей

ПРИМЕЧАНИЕ

По причине ограниченного размера книги я не стану рассматривать приложения, разработанные по технологии Windows Forms. Но с точки зрения реализации корня компоновки эти приложения подобны приложениям WPF.

В каждом разделе описывается один из рассматриваемых фреймворков, и они могут быть прочитаны более или менее независимо. Каждый раздел будет начинаться с общего введения по применению внедрения зависимостей в этом конкретном фреймворке, за которым следует развернутый пример, построенный применительно к примеру — приложению электронной коммерции, который рассматривается на протяжении всего издания.

Мы начнем с простейшего фреймворка, в который будет внедряться внедрение зависимостей и затем постепенно рассмотрим более сложные фреймворки. Когда мы дойдем до ASP.NET, придется пересечь пропасть, за которой можно будет осуществить внедрение зависимостей. Это возможно сделать, только пойдя на компромисс и пожертвовав как минимум несколькими принципами. Очевидно, что фреймворки, такие как ASP.NET и PowerShell, являются совершенно чужеродными средами для внедрения зависимостей. Поэтому мы сможем сделать лишь максимум возможного. Но пока мы не достигли этой точки, компромиссы будут нам не нужны.

Консольное приложение, по-видимому, является типом, в котором реализация внедрения зависимостей осуществляется самым простым образом.

7.1. Компоновка в консольных приложениях

Консольное приложение позволяет выполнить компоновку самым простым способом. В отличие от большинства других фреймворков приложений из BCL, консольное приложение не содержит инверсии управления. Во входной (стартовой) точке приложения (обычно это метод `Main`) на нас не влияет ничто. В этой точке нет ни специальных событий, на которые нужно было бы подписываться, ни интерфейсов, которые требуется реализовывать, ни сервисов, которые мы могли бы использовать.

Метод `Main` является вполне подходящим местом для реализации корня компоновки. Первое, что необходимо сделать в методе `Main`, — скомпоновать модули приложения и разрешить им начать работать. Здесь нет ничего особенного, но давайте рассмотрим пример.

7.1.1. Пример: обновление валют

В главе 4 мы рассматривали, как реализовать функцию конверсии валют в нашем демоприложении. В подразделе 4.3.4 был добавлен класс `Currency`, который предоставляет курсы конверсии одной валюты в другие. Поскольку `Currency` является абстрактным классом, мы можем создать разные варианты реализации, но в примере мы используем базу данных. В коде примера в главе 4 показывалось, как получить и реализовать конверсию валют, но мы не рассматривали способ обновления курсов обмена в базе данных.

Для продолжения примера рассмотрим, как можно написать простое консольное приложение, которое позволит администратору или суперпользователю обновлять курсы конверсии без прямого взаимодействия с базой данных.

Программа UpdateCurrency

Поскольку назначением этой программы является обновление курсов конверсии в базе данных, ее назвали `UpdateCurrency.exe`. Она будет получать три аргумента из командной строки:

- код целевой валюты (валюты назначения);
- код исходной валюты;
- курс конверсии.

Может показаться странным то, что мы собираемся указывать валюту назначения перед исходной валютой, но этот способ является самым понятным для большинства людей. Он показывает, сколько исходной валюты понадобится для покупки одной единицы валюты назначения; например, конверсионный курс для перевода `USD` в `EUR` объясняется так: «1 EUR стоит 1,44 USD»[1].

В командной строке это будет выглядеть следующим образом:

```
PS Ploeh:\> .\UpdateCurrency.exe EUR USD "1,44"
Updated: 1 EUR in USD = 1,44.
```

Выполнение данной программы изменит базу данных и выведет измененные значения обратно на консоль.

Корень компоновки

Программа `UpdateCurrency` использует точку входа, задаваемую по умолчанию для консольной программы: метод `Main` класса `Program`. Этот метод является корнем сборки для приложения, как показано в следующем листинге.

Листинг 7.1. Корень сборки консольного приложения

```
public static void Main(string[] args)
{
     var container = new CurrencyContainer();
     container.ResolveCurrencyParser()
          .Parse(args)
          .Execute();
}
```

Единственной задачей метода `Main` является сборка всех входящих в приложение модулей, после чего заботу о функциональности приложения принимает на себя сформированный граф объектов. В рассматриваемом примере компоновка модулей инкапсулируется в разработанном специально для этого контейнере. Я назвал его «контейнер», поскольку он выполняет точно такую же задачу, что и контейнер внедрения зависимостей, хотя он представляет собой специально разработанный код с жестко прописанными зависимостями. Скоро мы рассмотрим его реализацию.

---

[1] На 10 января 2010 года.

Теперь контейнер может отвечать за создание CurrencyParser, который выполняет синтаксический анализ входных аргументов, а затем выполняет соответствующие команды.

ПРИМЕЧАНИЕ

Корень компоновки должен выполнять только две задачи: настраивать контейнер и принимать решение о том, какой тип реализует запрашиваемую через команду функциональность. Как только это будет сделано, он должен завершить свою работу, оставив заботу об остальном экземпляру выбранного типа.

СОВЕТ

В своей профессиональной деятельности используйте стандартные контейнеры внедрения зависимостей, а не контейнеры собственной разработки.

В этом примере использован специально разработанный для приложения контейнер, но совершенно несложно заменить его на один из стандартных контейнеров внедрения, например на один из рассмотренных в части 4.

Контейнер

Класс CurrencyContainer — это специальной контейнер, созданный для того, чтобы четко продемонстрировать связывание воедино всех зависимостей для программы UpdateCurrency. Листинг 7.2 содержит его реализацию.

Листинг 7.2. Заказной CurrencyContainer

```
public class CurrencyContainer
{
    public CurrencyParser ResolveCurrencyParser()
    {
        string connectionString =
            ConfigurationManager.ConnectionStrings
            ["CommerceObjectContext"].ConnectionString;

        CurrencyProvider provider =
            new SqlCurrencyProvider(connectionString);
        return new CurrencyParser(provider);
    }
}
```

Получить строку соединения из конфигурации

В данном примере граф зависимостей совсем маленький. Класс CurrencyParser требует экземпляр абстрактного класса CurrencyProvider, и в CurrencyContainer вы решаете, что реализация должна быть типа SqlCurrencyProvider, обеспечивающего необходимое взаимодействие с базой данных.

В классе CurrencyParser применен внедрение конструктора, поэтому в него передается экземпляр SqlCurrencyProvider, который был создан перед возвращением его из метода.

Сигнатура конструктора CurrencyParser выглядит следующим образом:

```
public CurrencyParser(CurrencyProvider currencyProvider)
```

Вспомните, что CurrencyProvider — это абстрактный класс, реализованный как SqlCurrencyProvider. Хотя CurrencyContainer содержит жестко закодированное соотнесение между CurrencyProvider и SqlCurrencyProvider, остальной код является слабо связанным, поскольку в нем используется только абстракция.

Этот пример может показаться простым, но он компонует типы из трех разных уровней приложения. Кратко рассмотрим, как эти уровни взаимодействуют в данном примере.

Уровни приложения

Корень компоновки — это то место, где компоненты из разных уровней соединяются вместе. Код исполняемой программы состоит только из точки входа и корня компоновки. Реализация функциональности делегируется на более низкие уровни, что показано на рис. 7.4. Анализатор CurrencyParser парсирует аргументы командной строки и возвращает соответствующую ICommand. Если аргументы были верными, он возвращает CurrencyUpdateCommand, который использует экземпляр Currency для изменения курса конверсии. Вертикальная линия справа показывает соответствующий уровень приложения. Каждый уровень реализуется в отдельной сборке.

Схема на рис. 7.4 может показаться запутанной, но она представляет почти весь код приложения. Основная часть логики приложения заключается в синтаксическом анализе входных аргументов и выборе требуемой команды на основании входных данных. Все это осуществляется на уровне сервисов приложения (Application Services), который взаимодействует с моделью домена (Domain Model) через абстрактные классы CurrencyProvider и Currency.

CurrencyProvider внедряется контейнером в CurrencyParser и затем используется как «Абстрактная фабрика» для создания экземпляра Currency, который, в свою очередь, используется CurrencyUpdateCommand.

Уровень доступа к данным (Data Access) предоставляет основанные на технологии SQL Server реализации классов домена (Domain Classes). Хотя ни один из других классов приложения не взаимодействует напрямую с этими классами, CurrencyContainer соотносит абстракции с конкретными классами.

Применение внедрения зависимостей в консольном приложении не составляет труда, поскольку такие приложения не используют инверсии управления. Фреймворк .NET просто раскручивает процесс и передает управление методу Main.

В большинстве других фреймворков из BCL в той или иной мере используется инверсия управления, что оборачивается для нас необходимостью определять правильные точки расширения для подключения необходимого графа объектов. Одним из таких фреймворков является ASP.NET MVC.

7.2. Компоновка приложений ASP.NET MVC

Фреймворк ASP.NET MVC создавался для того, чтобы обеспечить максимальную адаптируемость к работе с внедрениями зависимостей, и этого удалось достичь. Он не заставляет внедрять внедрения, но легко позволяет сделать это, причем, не придавая особого значения виду реализуемого внедрения. Мы можем использовать «внедрение зависимостей для бедных» или же любой предпочтительный для нас контейнер.

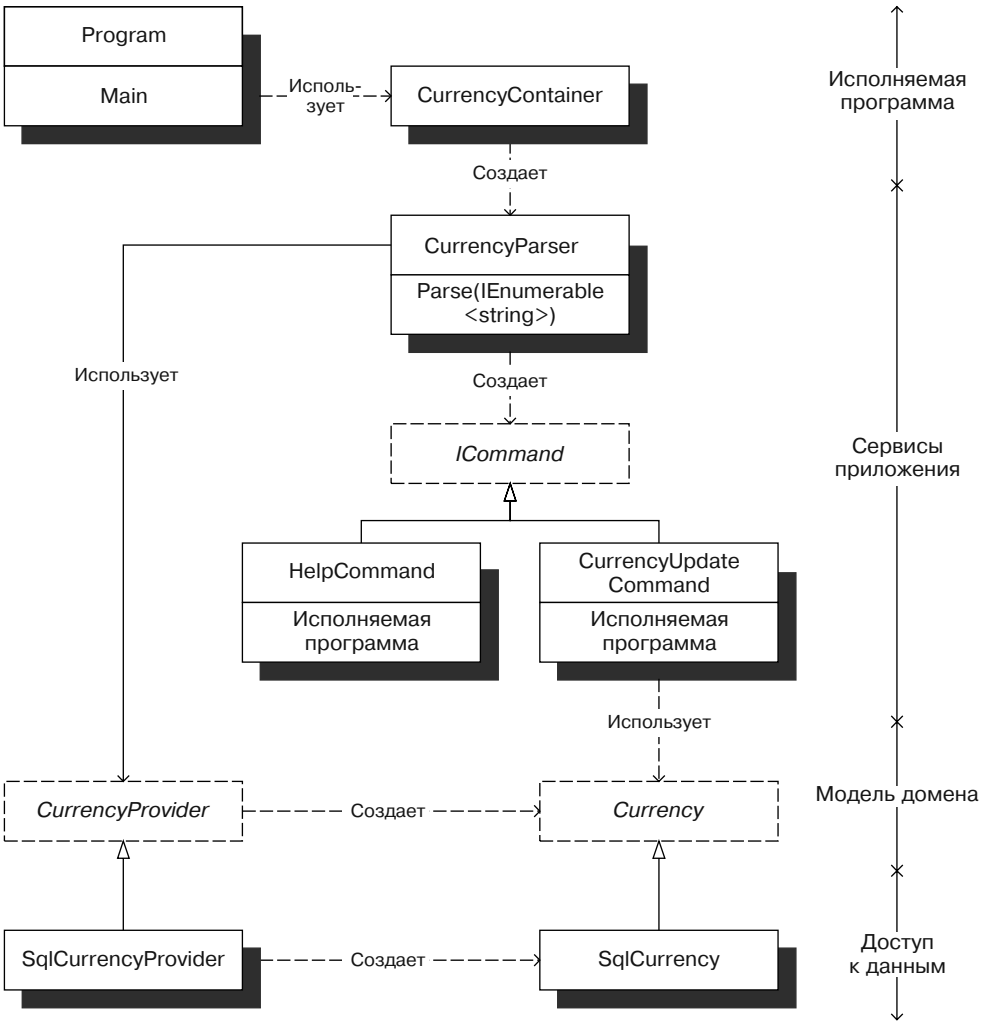


Рис. 7.4. Компоновка компонентов в приложении UpdateCurrency

7.2.1. Расширяемость ASP.NET MVC

Во всех случаях использования внедрения зависимостей ключевым фактором успешности его применения является выбор верного места расширения. В ASP.NET MVC таким местом является интерфейс, называемый `IControllerFactory`. Рисунок 7.5 показывает, как он работает в фреймворке. Когда среда времени исполнения ASP.NET MVC получает запрос, она запрашивает свою фабрику контроллеров (Controller Factory) о создании контроллера (Controller) для запрошенного URL. Фабрика контроллеров определяет необходимый тип создаваемого контроллера, который будет использоваться для обслуживания данного запроса, затем создает и возвращает новый экземпляр такого типа. После этого ASP.NET MVC вызывает требуемый

для выполнения команды метод экземпляра `Controller`. Когда данный процесс завершается, ASP.NET MVC предоставляет фабрике контроллеров возможность освободить созданные ею ресурсы, вызвав метод `ReleaseController`.

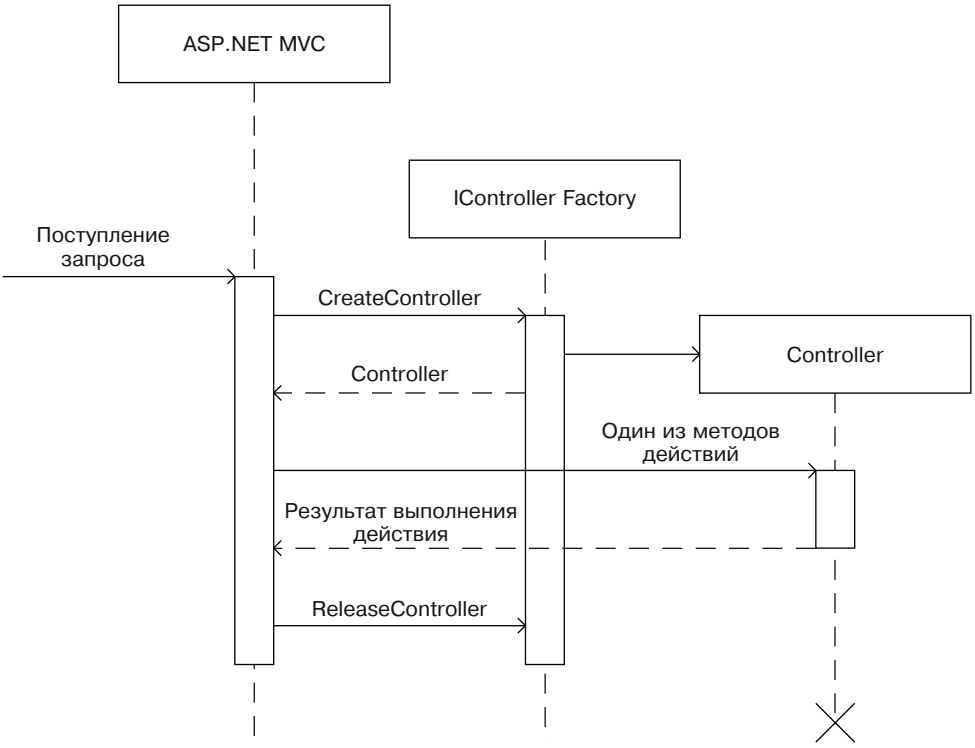


Рис. 7.5. Интерфейс IControllerFactory в фреймворке

Контроллеры — это главные сущности в ASP.NET MVC. Они обрабатывают запросы и определяют способ реакции на них. Если нам требуется запрос к базе данных, проверка и сохранение входных данных, вызов логики домена и т. д., мы инициализируем соответствующие действия из контроллера.

Контроллер должен не сам выполнять эти операции, а делегировать данную работу соответствующим зависимостям. Здесь в дело вступает внедрение зависимостей. Нам нужна возможность передачи зависимостей в соответствующий класс `Controller`, в идеале через внедрение конструктора. Это делается с помощью специального `IControllerFactory`.

IDEPENDENCYRESOLVER

Когда в 2011 году была выпущена третья версия фреймворка ASP.NET MVC, одна из ее новых возможностей была представлена как поддержка внедрения зависимостей. Оказалось, что эта поддержка организована с помощью нового интерфейса, названного IDependencyResolver. Но и сам этот интерфейс, и способ, которым предлагается использовать его в ASP.NET MVC, представляются спорными.

На концептуальном уровне предлагается задействовать IDependencyResolver как локатор сервисов, и именно таким способом фреймворк применяет его.

Если говорить более подробно, этот интерфейс обладает ограниченной полезностью, поскольку у него отсутствует метод `Release`. Другими словами, с помощью этого интерфейса мы не можем правильно управлять жизненным циклом графа объектов. В некоторых контейнерах внедрения зависимости это, как правило, приводит к утечке ресурсов[1].

Я считаю, что более безопасно и правильно будет игнорировать такую реализацию `IDependencyResolver`. Ирония ситуации заключается в том, что правильная реализация внедрения зависимостей вполне возможна в ASP.NET MVC, начиная с уже первой его версии с помощью интерфейса `IControllerFactory`.

Создание заказной фабрики контроллеров

В состав готового фреймворка ASP.NET MVC входит `DefaultControllerFactory` — фабрика, которая требует наличия конструктора по умолчанию в классах `Controller`. Это целесообразное поведение по умолчанию, которое не заставляет нас вводить внедрение зависимостей, если мы не хотим этого. Однако конструкторы по умолчанию и внедрение конструкторов являются взаимно исключающими феноменами, поэтому требуется изменить такое поведение, реализовав специальную фабрику контроллеров.

Это не слишком сложная задача. От вас требуется реализовать интерфейс `IControllerFactory`:

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
        string controllerName);

    SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName);

    void ReleaseController(IController controller);
}
```

Метод `CreateController` обеспечивает `RequestContext`, который содержит информацию о контексте `HttpContext`, тогда как `controllerName` указывает, какой именно контроллер должен быть использован.

Вы можете проигнорировать `RequestContext` и использовать только `controllerName` для определения того, какой контроллер требуется вернуть. Неважно, что вы делаете, это тот метод, где у вас имеется возможность связать необходимые зависимости и передать их в `Controller`, прежде чем возвращать созданный экземпляр. Пример рассматривается в подразделе 7.2.2.

---

[1] *Mike Hadlow* The MVC 3.0 IDependencyResolver interface is broken. Don't use it with Windsor, 2011: http://mikehadlow.blogspot.com/2011/02/mvc-30-idependencyresolver-interface-is.html.

Если были созданы какие-либо ресурсы, требующие освобождения после окончания работы с ними, вы можете сделать это в методе `ReleaseController`.

СОВЕТ

Класс DefaultControllerFactory реализует IControllerFactory и имеет несколько виртуальных методов. Вместо того чтобы реализовывать фабрику IControllerFactory с нуля, часто проще выполнить реализацию путем создания наследника DefaultControllerFactory.

Хотя реализация специальной фабрики контроллеров довольно важна, ее будет невозможно использовать, пока мы не сообщим фреймворку ASP.NET MVC о ее существовании.

Регистрация специальной фабрики контроллеров

Специальные фабрики контроллеров регистрируются во время запуска приложения — как правило, в `Global.asax`. Регистрация осуществляется путем вызова `ControllerBuilder.Current.SetControllerFactory`. Ниже представлен фрагмент кода из нашего демоприложения:

```
var controllerFactory = new CommerceControllerFactory();

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

В этом примере создается и устанавливается новый экземпляр заказной `CommerceControllerFactory`. Фреймворк ASP.NET MVC будет использовать экземпляр `controllerFactory` в качестве фабрики контроллеров в приложении.

Если этот код вам что-то напоминает, то это вызвано тем, что вы встречали нечто подобное в разделе 3.3. Тогда я обещал показать реализацию заказной фабрики контроллеров в главе 7. Вот глава 7.

7.2.2. Пример: реализация CommerceControllerFactory

Нашему демоприложению требуется специальная фабрика контроллеров для связывания контроллеров с требуемыми зависимостями. Хотя полный граф зависимостей для всех контроллеров является намного более сложным, с точки зрения самих контроллеров множество всех конкретных зависимостей сводится к представленным на рис. 7.6 трем элементам. Конкретные реализации каждой из этих зависимостей имеют свои зависимости, но это не показано на схеме. `BasketController` и `HomeController` разделяют зависимость от `CurrencyProvider`. `AccountController` наследуется в неизменном виде из шаблона по умолчанию фреймворка ASP.NET MVC; поскольку используется гибридное внедрение, он не имеет неразрешенных зависимостей.

Хотя `IControllerFactory` может быть реализован с нуля, проще определить его как наследника `DefaultControllerFactory` и переопределить в нем метод `GetControllerInstance`. Тогда `DefaultControllerFactory` будет отвечать за соотнесение имени контроллера с его типом, и все, что должны будете сделать вы, — вернуть экземпляры запрошенных типов.

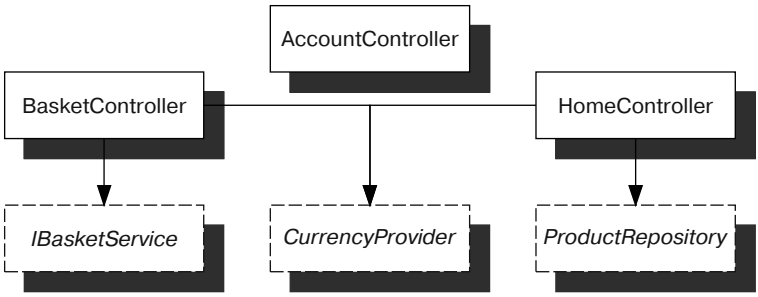


Рис. 7.6. Граф зависимостей для трех контроллеров в демоприложении

Листинг 7.3. Создание контроллеров

```
protected override IController GetControllerInstance(
    RequestContext requestContext, Type controllerType)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    var productRepository =
        new SqlProductRepository(connectionString);
    var basketRepository =
        new SqlBasketRepository(connectionString);
    var discountRepository =
        new SqlDiscountRepository(connectionString);

    var discountPolicy =
        new RepositoryBasketDiscountPolicy(
            discountRepository);

    var basketService =
        new BasketService(basketRepository,
            discountPolicy);
    var currencyProvider = new CachingCurrencyProvider(
        new SqlCurrencyProvider(connectionString),
        TimeSpan.FromHours(1));

    if (controllerType == typeof(BasketController))
    {
        return new BasketController(
            basketService, currencyProvider);
    }
    if (controllerType == typeof(HomeController))
    {
        return new HomeController(
            productRepository, currencyProvider);
    }
```

1 Переопределение

2 Создать зависимости

3 Вернуть связанные контроллеры

```
    return base.GetControllerInstance(
        requestContext, controllerType);          ❹ Использовать базовую форму
}                                                   для остальных контроллеров
```

Этот метод переопределяет `DefaultControllerFactory.GetControllerInstance` ❶, чтобы создать экземпляры требуемых типов контроллеров. Если требуемый тип является либо `BasketController`, либо `HomeController`, осуществляется явное связывание с требуемыми зависимостями ❷ и затем возврат сгенерированных экземпляров ❸. Оба эти типа используют внедрение конструктора, поэтому зависимости передаются через конструкторы этих классов.

Для упрощения кода я выполнил связывание зависимостей ❷ до проверки `controllerType`. Очевидно, это означает, что некоторые из созданных зависимостей не будут использоваться, так что рассматриваемая реализация не является слишком уж эффективной. Код из листинга 7.3 может быть преобразован в более полезную (но и значительно более сложную) форму.

Для типов, которые не обрабатываются явно, используется базовое (по умолчанию) поведение ❹, что предполагает создание запрашиваемого контроллера с помощью конструктор по умолчанию. Заметьте, что `AccountController` вообще явно не обрабатывается, так что ему задается базовое поведение. `AccountController` образуется из шаблона проекта ASP.NET MVC и использует гибридное внедрение, которое определяется конструктором по умолчанию.

ПРИМЕЧАНИЕ

Я отношу гибридное внедрение к антипаттернам, но тем не менее использую его с AccountController, потому что ранее я показывал в своих примерах много правильных образцов внедрения зависимостей. Я поступил так потому, что это демонстрационный код, но я никогда не сделаю так при разработке боевого кода.

Когда экземпляр `CommerceControllerFactory` регистрируется в `Global.asax`, он будет корректно создавать все требуемые контроллеры с необходимыми зависимостями.

СОВЕТ

Возьмите за правило не разрабатывать своих фабрик контроллеров. Вместо этого используйте ту стандартную фабрику контроллеров, которая работает совместно с выбранным вами контейнером внедрения. Для начала можете познакомиться с проектом MVC Contrib[1] или используйте какую-нибудь из переиспользуемых реализаций. Некоторые контейнеры внедрения зависимостей также имеют официальную интеграцию с ASP.NET MVC.

Достоинством ASP.NET MVC является то, что этот фреймворк разрабатывался с прицелом на внедрение зависимостей, так что нам нужно только определить и использовать одну точку расширения, чтобы реализовать внедрение зависимостей в приложении. В других фреймворках реализация внедрения зависимостей оказывается гораздо более сложной задачей. Windows Communication Foundation (WCF), хотя и является расширяемым фреймворком, служит примером этого.

[1] www.codeplex.com/MVCContrib/.

7.3. Компоновка WCF-приложений

WCF является одной из наиболее активно расширяемых частей BCL. Хотя начать использовать сервисы WCF очень просто, огромное количество точек расширяемости может усложнить поиск одной, необходимой в конкретном случае. Это относится и к внедрению зависимостей.

ПРИМЕЧАНИЕ

Аббревиатура WCF иногда шутливо расшифровывается как Windows Complication (усложнение, сложность, запутанность) Foundation. В этой шутке заключается изрядная доля правды.

Можно легко убедиться в том, что WCF не поддерживает внедрение конструктора. Если вы создадите сервис WCF с внедрением конструктора, но без конструктора по умолчанию, то во время исполнения хост WCF-сервиса сгенерирует `ServiceActivationException` с сообщением, подобным следующему:

The service type provided could not be loaded as a service because it does not have a default (parameter-less) constructor. To fix the problem, add a default constructor to the type, or pass an instance of the type to the host.

Данный тип сервиса не может быть загружен как сервис, так как он не содержит конструктор по умолчанию (не имеющий параметров). Для решения этой проблемы добавьте конструктор по умолчанию или передайте хосту экземпляр требуемого типа.

Это сообщение явно говорит о том, что конструктор по умолчанию является обязательным. Единственный возможный способ обойти это ограничение — передать уже созданный экземпляр WCF-хосту, но это решение вызывает определенные проблемы.

- Как можно сделать это, если сервис расположен на Internet Information Services (IIS)?
- Чтобы иметь возможность выполнить это, сервис должен быть запущен в `SingleInstanceContextMode`, работа в котором является нежелательной по целому ряду других причин.

Хорошая новость заключается в том, что данное сообщение об исключительной ситуации не совсем соответствует действительности. Имеются и другие способы обеспечить внедрение конструктора в WCF.

7.3.1. Расширяемость WCF

У WCF имеется множество точек расширения, но когда речь заходит о внедрении зависимостей, все, что нам нужно для работы, — это интерфейс `IInstanceProvider` и логика работы контрактов (Contract Behaviors). Логика работы контракта — это шов в WCF, который позволяет изменять поведение данного контракта (то есть сервиса).

`IinstanceProvider` — это интерфейс, который определяет, как создаются и уничтожаются экземпляры сервисов. Вот полное определение интерфейса:

```
public interface IInstanceProvider
{
    object GetInstance(InstanceContext instanceContext);
    object GetInstance(InstanceContext instanceContext, Message message);
    void ReleaseInstance(InstanceContext instanceContext, object instance);
}
```

Два перегруженных варианта `GetInstance` отвечают за создание подходящего экземпляра сервиса, а `ReleaseInstance` определяет процесс его ликвидации при необходимости.

Базовая (по умолчанию) реализация требует наличия конструктора по умолчанию в типе сервиса, но мы можем заменить данную реализацию на вариант, использующий внедрение конструктора. Рисунок 7.7 показывает весь процесс при приеме сервисом сообщения. Когда поступает сообщение (запрос) на сервисную операцию, WCF определяет, какой из типов CLR реализует данную услугу (сервис). WCF делает запрос к фабрике `ServiceHostFactory` на создание подходящего `ServiceHost`, где может располагаться запрошенный сервис. Созданный `ServiceHost` выполняет свою часть работы путем применения логики работы и созданием требуемого экземпляра.

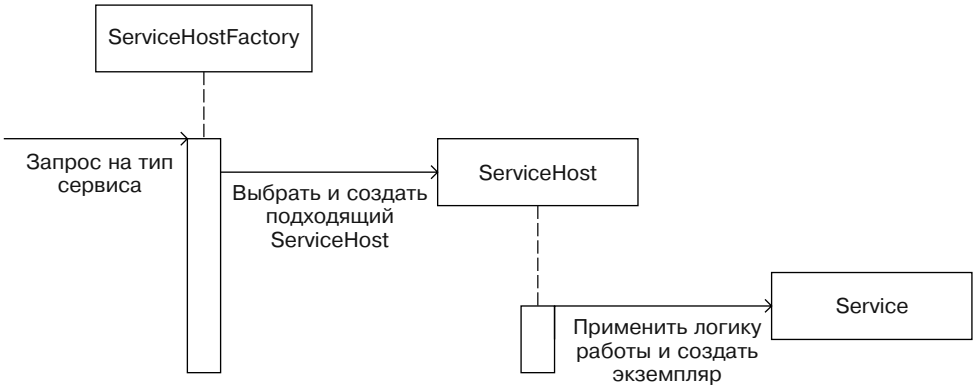


Рис. 7.7. Процесс приема сервисом сообщения

Когда мы располагаем WCF-сервис на IIS, фабрика `ServiceHostFactory` является обязательной, хотя реализация по умолчанию будет использована, если мы не укажем явно альтернативу. Если мы располагаем сервис вручную, `ServiceHostFactory` также может оказаться полезной, но не является обязательной, поскольку мы можем создать подходящий `ServiceHost` непосредственно в коде.

Когда `ServiceHost` применяет логику работы, он выбирает ее как минимум из трех различных мест, прежде чем провести агрегирование:

- атрибуты;
- файл конфигурации `.config`;
- находящиеся в памяти объекты.

Хотя можно определить логику работы через атрибуты, это не лучшая стратегия при работе с внедрением зависимостей, поскольку в таком случае мы компилируем в код специфическую стратегию создания со специфическими же зависимостями. В итоге мы получили тот же результат, как если бы жестко закодировали зависимости прямо в сервисе, только мы добились этого гораздо более сложным способом.

Файл конфигурации может показаться подходящим вариантом с точки зрения гибкости, но это не так, потому что он не позволяет нам оперативно (в произвольный момент) переконфигурировать зависимости, если понадобится сделать это.

Находящиеся в памяти объекты обеспечивают наилучшую степень гибкости, поскольку мы можем выбрать между созданием зависимостей непосредственно в коде или на основе параметров конфигурации. Если мы используем контейнер внедрения зависимостей, мы получаем обе эти возможности. Это значит, что необходимо создать специальную фабрику `ServiceHostFactory`, которая создает экземпляры заказного же `ServiceHost`, который может связать нужный сервис со всеми его зависимостями.

Мы можем создать набор классов общего назначения, реализующих эти требования на основе выбранного контейнера внедрения, или использовать одну из уже реализованных повторно и основанных на контейнере фабрик `ServiceHostFactory`. Можно также создать специализированную фабрику `ServiceHostFactory` для конкретного сервиса. Поскольку именно этот вариант обеспечивает наилучшую иллюстрацию процесса, в следующем примере задействована именно специализированная фабрика.

7.3.2. Пример: подключение сервиса управления продуктами

В качестве примера предположим, что мы получили запрос на расширение нашего демоприложения включением в него разработанного по технологии WCF-сервиса, предоставляющего операции, которые позволяют другим приложениям управлять данными о продуктах. Это дает вам возможность реализовать насыщенного клиента (это будет сделано в следующем разделе) или пакетное задание для управления данными о продуктах.

Введение в сервис управления продуктами

Чтобы не усложнить пример, предположим, что вы хотите получить набор из простых операций create — создать, read — читать, update — изменить и delete — удалить (CRUD). Рисунок 7.8 демонстрирует схему сервиса и связанных с ним контрактов данных (Data Contracts). `IProductManagementService` представляет собой сервис WCF, который определяет простые CRUD-операции, выполняемые над продуктами. Для реализации этих операций он использует связанные `ProductContract` и `MoneyContract`. Хотя это и не показано на схеме, все эти типы имеют обычные атрибуты WCF: `ServiceContract`, `OperationContract`, `DataContract` и `DataMember`.

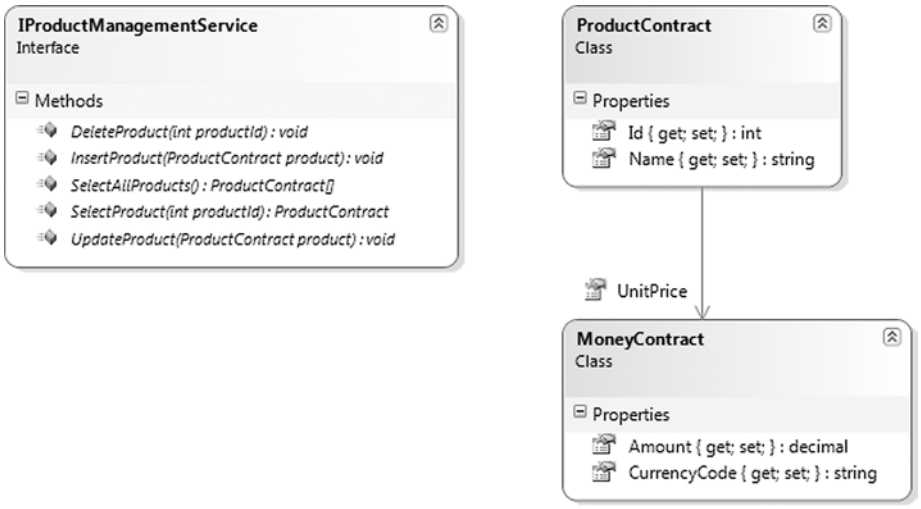


Рис. 7.8. Схема сервиса и связанные с ним контракты данных

Поскольку модель домена уже существует, желательно реализовать данный сервис путем ее расширения, а упомянутые операции — через WCF-контракт. Конкретные детали сейчас неважны; достаточно сказать, что вы расширяете абстрактный класс `ProductRepository`, который рассматривался в предыдущих главах.

СОВЕТ

Хотя я не хочу демонстрировать здесь полный код приложения, вы можете найти его и познакомиться с деталями в коде, предлагаемом для загрузки с этой книгой.

Доменная модель представляет продукт как сущность (Entity) `Product`, и контракт сервиса представляет свои операции в терминах объекта передачи данных (Data Transfer Object, DTO) `ProductContract`. Чтобы соотнести два эти различных типа, добавляется еще интерфейс, называемый `IContractMapper`.

В итоге вы получаете реализацию сервиса с двумя зависимостями; и, поскольку обе они являются обязательными, можно использовать внедрение конструктора.

Ниже приведена сигнатура конструктора сервиса:

```
public ProductManagementService(ProductRepository repository,
    IContractMapper mapper)
```

Итак, нам удалось удачно «не заметить» слона в комнате: как будет использоваться WCF для правильного подключения экземпляра `ProductManagementService`?

Подключение ProductManagementService в WCF

Как показано на рис. 7.7, корень компоновки в WCF представляет собой трио из `ServiceHostFactory`, `ServiceHost` и `IInstanceProvider`. Чтобы подключить сервис с помощью внедрения конструктора, необходимо иметь заказные реализации всех трех.

СОВЕТ

Вы можете написать полностью повторно используемые реализации, которые будут обертывать (Wrap) ваш любимый контейнер внедрения этими тремя типами и использовать их для реализации IInstanceProvider. Многие разработчики уже делали это, так что вполне вероятно, что вы найдете готовый к использованию набор для выбранного вами контейнера внедрения зависимостей.

ПРИМЕЧАНИЕ

В данном примере реализуется жестко подключенный контейнер, использующий «внедрения зависимостей для бедных». Я инкапсулировал жестко кодированные зависимости в класс специального контейнера, чтобы подсказать вам, как можно создать многократно используемое решение на основе конкретного контейнера внедрения зависимостей.

СУЩНОСТЬ ПРОТИВ DTO

Выше были введены новые термины, поэтому давайте вкратце разберем, что означают термины сущность (Entity) и DTO.

Сущность — это термин из методологии, называемой предметно-ориентированное проектирование (Domain-Driven Design), который определяет объект домена (Domain Object), имеющий идентификатор (`identity`). Такой идентификатор существует в течение длительного периода времени и не связан с конкретным экземпляром объекта. Подобное определение может показаться абстрактным и слишком теоретическим, но оно означает, что сущность представляет объект, который существует даже не будучи размещенным в битах компьютерной памяти. Любой экземпляр объекта .NET имеет адрес в компьютерной памяти (`identity`), но сущность имеет `identity`, который существует без привязки к жизненному циклу процесса. Базы данных и первичные ключи часто используются для идентификации сущностей и обеспечения гарантии того, что мы можем организовать их постоянное хранение и перечитывать их даже в случае перезагрузки компьютера.

Объект домена с именем `Product` является сущностью, поскольку концепция продукта обладает гораздо более длительным временем жизненного цикла, чем отдельный процесс, и мы используем идентификатор продукта (`ID`) для его обозначения в `ProductRepository`.

Объект передачи данных (Data Transfer Object, DTO), с другой стороны, создается только для того, чтобы быть переданным из одного уровня приложения в другой. Тогда как сущность может инкапсулировать большой объем логики работы, DTO представляет собой только структуру данных, без кода логики.

Когда требуется обеспечить доступность модели домена внешним системам, это зачастую делается с помощью сервисов и DTO, поскольку мы никогда не можем быть уверенными в том, что другие (внешние) системы могут напрямую работать с системой нашего типа (они могут даже не использовать .NET). В таких случаях нам всегда потребуется обеспечивать соотнесение между сущностями и DTO.

Начнем с рассмотрения специальной фабрии `ServiceHostFactory`, которая является хорошей точкой входа в WCF-сервис. В листинге 7.4 показана его реализация.

Листинг 7.4. Специальная фабрика ServiceHostFactory

```
public class CommerceServiceHostFactory : ServiceHostFactory
{
    private readonly ICommerceServiceContainer container;

    public CommerceServiceHostFactory()
```

```
    {
        this.container =
            new CommerceServiceContainer();
    }

    protected override ServiceHost CreateServiceHost(
        Type serviceType, Uri[] baseAddresses)
    {
          if (serviceType == typeof(ProductManagementService))
        {
            return new CommerceServiceHost(
                this.container,
                serviceType, baseAddresses);
        }
        return base.CreateServiceHost(serviceType, baseAddresses);
    }
}
```

❶ Создать экземпляр контейнера

❷ Создать заказной ServiceHost

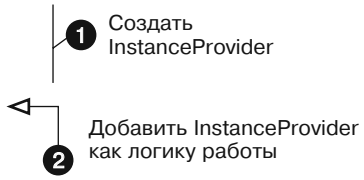
Специальная фабрика `CommerceServiceHostFactory` наследуется из `ServiceHostFactory` с единственной целью — обеспечить подключение экземпляров `ProductManagementService`. Она использует заказной `CommerceServiceContainer` для выполнения реальной работы, для чего создает экземпляр контейнера в своем конструкторе ❶. Вы можете легко расширить этот пример, перейдя к применению полноценного контейнера внедрения зависимостей путем создания и конфигурирования его экземпляра вместо `CommerceServiceContainer`.

Когда обрабатывается запрос на создание `ServiceHost`, возвращается новый `CommerceServiceHost` со сконфигурированным контейнером ❷, если запрошенный тип сервиса оказывается подходящим. `CommerceServiceHost` отвечает за назначение соответствующих логик работ всем типам сервисов в хостах. В данном случае вы хотите добавить только одну логику работы, которая назначает сервисам требуемый `IinstanceProvider`. Все эти действия могут быть выполнены в конструкторе, как показано ниже, а базовый класс выполнит всю остальную работу (листинг 7.5).

Листинг 7.5. Заказной ServiceHost

```
public class CommerceServiceHost : ServiceHost
{
    public CommerceServiceHost(ICommerceServiceContainer container,
        Type serviceType, params Uri[] baseAddresses)
        : base(serviceType, baseAddresses)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }

        var contracts = this.ImplementedContracts.Values;
```

```
public class CommerceServiceHost : ServiceHost
{
    public CommerceServiceHost(ICommerceServiceContainer container,
        Type serviceType, params Uri[] baseAddresses)
        : base(serviceType, baseAddresses)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }

        var contracts = this.ImplementedContracts.Values;
        foreach (var c in contracts)
        {
            var instanceProvider =
                new CommerceInstanceProvider(
                    container);
            c.Behaviors.Add(instanceProvider);
        }
    }
}
```



Класс `CommerceServiceHost` наследуется из конкретного класса `ServiceHost`, выполняющего основную работу. В большинстве случаев вам потребуется размещать на хосте только один тип сервиса (в данном случае `ProductManagementService`), но у вас остается возможность размещать и несколько сервисов. Это означает, что вы должны добавить `IInstanceProvider` во все эти сервисы. Свойство `ImplementedContracts` является словарем, поэтому вы должны перебрать в цикле все его значения, чтобы обработать каждое из них.

Для каждого типа сервиса вы инициализируете контейнером новый экземпляр заказного класса `CommerceInstanceProvider` ❶. Поскольку он дублирует логику работы, он может быть добавлен в число логик работы сервиса ❷.

Последним участником WCF-трио является `CommerceInstanceProvider`, дублирующий как `IInstanceProvider`, так и `IContractBehavior`. Это простая реализация, но поскольку она реализует два разных интерфейса со сложными сигнатурами, на может показаться запутанной, если вы только бегло ее просмотрели. Но я рассмотрю этот код немного подробнее; рис. 7.9 позволяет получить общее представление.

Листинг 7.6 показывает объявление класса и его конструктор. Ничто не запрещает использовать внедрение конструктора для инжектирования контейнера.

Как правило, мы применяем внедрение конструктора для того, чтобы известить контейнер внедрения о том, что классу нужны какие-то зависимости, но здесь имеет место быть обратная ситуация, потому что внедряется сам контейнер. В обычной ситуации это является образчиком плохого кода, так как обычно это свидетельствует о намерении использовать антипаттерн «Локатор сервисов», но здесь такой прием является необходимым, поскольку речь идет о реализации корня компоновки.

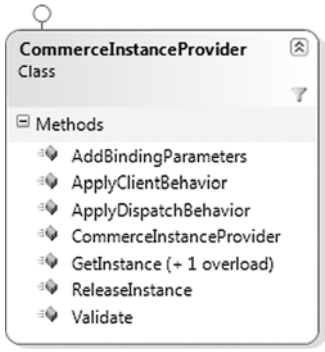


Рис. 7.9. CommerceInstanceProvider реализует и IInstanceProvider, и IContractBehavior интерфейсы, поэтому вы должны реализовать семь методов. Три из них могут быть оставлены пустыми, остальные же четыре сделаны однострочными

Листинг 7.6. Объявление и конструктор класса CommerceInstanceProvider

```
public partial class CommerceInstanceProvider :
    IInstanceProvider, IContractBehavior
{
    private readonly ICommerceServiceContainer container;

    public CommerceInstanceProvider(
        ICommerceServiceContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }

        this.container = container;
    }
}
```

(1) Реализует WCF-интерфейсы

(2) Защищенное внедрение конструктора

CommerceInstanceProvider реализует как IInstanceProvider, так и IContractBehavior (1). Контейнер передается через стандартное внедрение конструктора (2). В данном примере используется специальный CommerceServiceContainer, но замена его на стандартный контейнер является очень простой задачей.

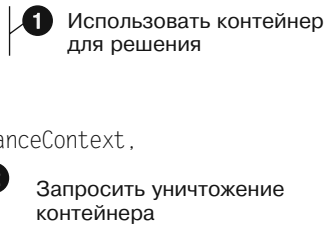
Реализация IinstanceProvider, представленная в листинге 7.7, используется средой времени исполнения WCF для создания экземпляров класса ProductManagementService.

Листинг 7.7. Реализация IInstanceProvider

```
public object GetInstance(InstanceContext instanceContext, Message message)
{
    return this.GetInstance(instanceContext);
}
```

Делегирование для перегрузки

```
public object GetInstance(InstanceContext instanceContext)
{
    return this.container
        .ResolveProductManagementService();
}

public void ReleaseInstance(InstanceContext instanceContext,
    object instance)
{
    this.container.Release(instance);
}
```



Среда времени исполнения WCF вызывает один из методов `GetInstance`, чтобы получить экземпляр сервиса требуемого типа, поэтому вы запрашиваете контейнер о подключении `ProductManagementService` (1) со всеми необходимыми зависимостями.

После того как служебная операция будет выполнена, среда времени исполнения WCF отправляет вам запрос на удаление экземпляра сервиса, а вы вновь делегируете эту работу контейнеру (2).

Оставшаяся часть кода `CommerceInstanceProvider` представляет собой реализацию `IContractBehavior`. Единственная причина, по которой вы реализуете этот интерфейс, — намерение добавить его в список логик, как показано в листинге 7.5. Для всех методов интерфейса `IContractBehavior` определен тип возврата `void`, поэтому вы можете оставить их все пустыми, так как вам не нужно их реализовывать.

Листинг 7.8 показывает реализацию единственного метода, который требует внимания.

Листинг 7.8. Базовая реализация интерфейса IContractBehavior

```
public void ApplyDispatchBehavior(
    ContractDescription contractDescription, ServiceEndpoint endpoint,
    DispatchRuntime dispatchRuntime)
{
    dispatchRuntime.InstanceProvider = this;
}
```

В этом методе вам нужно сделать всего одну совершенно простую вещь. Среда времени исполнения WCF вызывает этот метод и передает ему экземпляр `DispatchRuntime`, что позволяет сообщить, что должна использоваться эта конкретная реализация `IInstanceProvider`. Напомню, что `CommerceInstanceProvider` также реализует `IInstanceProvider`. Среда времени исполнения WCF теперь «знает», какой `IInstanceProvider` должен быть использован, и вызывается метод `GetInstance`, показанный в листинге 7.7.

Таким образом, для реализации внедрения зависимостей нужно написать достаточно много кода, и я даже не могу показать вам полную реализацию `CommerceServiceContainer`.

СОВЕТ

Помните, что можно легко написать многократно используемые версии этих трех классов, оборачивающих выбранный вами контейнер, и упаковать эту реализацию в библиотеку. Многие разработчики уже проделали такую работу, и вы, вероятно, сможете найти подходящую готовую библиотеку в Интернете.

Контейнер является последним фрагментом в мозаике внедрения зависимостей в WCF.

Реализация специализированного контейнера

`CommerceServiceContainer` — это специализированный контейнер, единственное назначение которого — подключить класс `ProductManagementService`. Напомню, что этот класс требует экземпляров `ProductRepository` и `IContractMapper` как зависимостей.

Этот контейнер совершенно не касается инфраструктуры WCF, а концентрируется на подключении графа зависимостей.

ПРИМЕЧАНИЕ

Помимо хорошего соответствия принципу единичной ответственности, такое разделение функций означает, что данный специализированный контейнер может быть с легкостью заменен на типовой контейнер внедрения зависимостей, поскольку здесь нет специфичного для WCF-кода.

Метод `ResolveProductManagementService` подключает экземпляр, используя «внедрение для бедных», как показано ниже (листинг 7.9).

Листинг 7.9. Разрешение `ProductManagementService`

```
public IProductManagementService ResolveProductManagementService()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    ProductRepository repository =
        new SqlProductRepository(connectionString);

    IContractMapper mapper = new ContractMapper();

    return new ProductManagementService(repository,
        mapper);
}
```

(1) Создать репозиторий продуктов

(2) Создать маппер для контракта

В определенном смысле, когда речь заходит о разрешении графа зависимостей, он зачастую может пригодиться вам для «прокладки обратного пути». Вы должны вернуть экземпляр `ProductManagementService` с экземплярами `ProductRepository` и `IContractMapper`. Экземпляр `IContractMapper` создать легко (2), а вот с `ProductRepository` придется повозиться.

Вы хотите использовать `SqlProductRepository` (1), но чтобы сделать это, вам понадобится строка подключения, которая может быть считана из файла `web.config`.

Если вы захотите разместить ваш сервис в вашем же приложении, вы можете сделать это, создав новый экземпляр класса CommerceServiceHostFactory и вызвав его метод CreateServiceHost с верными параметрами. Он вернет экземпляр CommerceServiceHost, который вы можете открыть, выполнит остаток работы и обеспечит хостинг для ProductManagementService.

Но если вы захотите расположить сервис на IIS, вам потребуется выполнить дополнительные шаги.

Хостинг ProductManagementService на IIS

На IIS мы не создаем вручную новые экземпляры фабрики CommerceServiceHostFactory. Вместо этого мы должны приказать IIS выполнить это действие для нас. Это может быть сделано путем передачи атрибута Factory через файл SVC:

```
<%@ ServiceHost
Factory = "Ploeh.Samples.CommerceService.CommerceServiceHostFactory,
          ➥Ploeh.Samples.CommerceService"
          Service = "Ploeh.Samples.CommerceService.ProductManagementService"
%>
```

Такой SVC-файл предписывает IIS использовать фабрику CommerceServiceHostFactory всякий раз, когда требуется создать экземпляр класса ProductManagementService. Класс ServiceHostFactory должен обязательно иметь конструктор по умолчанию, это остается справедливым и в данном примере.

Внедрение зависимостей в WCF оказывается более трудным делом, чем хотелось бы, но это как минимум возможно. Итоговый результат оказывается в конце концов вполне удовлетворительным. Можно использовать любой контейнер внедрения, и мы получим корректный корень компоновки.

Некоторые фреймворки не имеют шва, который обеспечил бы нам такие возможности. Но прежде чем мы приступим к рассмотрению такого неудобного фреймворка, изучим один гораздо более простой.

7.4. Компоновка WPF-приложений

Если вы решили, что компоновка сервиса WCF является не простым занятием (я думаю именно так), вы оцените по достоинству тот факт, что компоновка разрабатываемых по технологии Windows Presentation Foundation (WPF) приложений оказывается почто такой же простой, как компоновка консольных приложений.

Точка входа WPF приложения совершенно очевидна и несложна, и хотя технология не содержит шва, который был бы явно предназначен для внедрения зависимостей, можно легко скомпоновать приложение любым предпочтительным для вас способом.

7.4.1. WPF-компоновка

Точка входа WPF приложения определена в его классе App. Как и большинство классов в WPF, этот класс разбит на два файла: App.xaml и App.xaml.cs. Мы можем

определять, что будет происходить при запуске приложения, в обоих этих файлах, в зависимости от того, что нам нужно.

Когда вы создаете новый проект WPF в Visual Studio, в файле `App.xaml` определяется атрибут `StartupUri`, задающий, какое окно будет показано при запуске приложения — в данном случае это `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>
```

Такой декларативный подход подразумевает, что объект `Window1` создается и отображается без участия какого бы то ни было дополнительного контекста. Когда вы хотите передать зависимости в окно, лучшим может оказаться более явный подход. Можно удалить атрибут `StartupUri` и подключить окно переопределением метода `OnStartup`. Это позволит вам полностью подготовить к работе первое окно до того, как оно будет показано, но это потребует дополнительного шага: вы должны явно вызывать метод `Show` окна.

Метод `OnStartup` таким образом становится корнем компоновки окна. Для компоновки окна вы можете использовать либо внедрение конструктора, либо «внедрение для бедных». В следующем примере применяется «внедрение зависимостей для бедных» для иллюстрации, что вы необязательно должны полагаться на возможности конкретного контейнера внедрения зависимостей.

7.4.2. Пример: подключение насыщенного клиента для управления продуктами

В предыдущем примере был разработан веб-сервис, который мы могли использовать для управления каталогом продуктов нашего демоприложения. В предлагаемом примере будет создано WPF-приложение, применяющее этот веб-сервис для управления продуктами. Рисунок 7.10 представляет снимок экрана этого приложения. Главное окно приложения Product Management (Управление продуктами) представляет собой список продуктов. Имеется возможность добавлять новые продукты, редактировать находящиеся в списке или удалять их. Когда продукты добавляются или редактируются, используется модальное диалоговое окно. Все операции приложения реализуются вызовом соответствующих операций веб-сервиса управления продуктами, рассмотренного в подразделе 7.3.2.

Все приложение реализовано с помощью подхода Model View ViewModel (MVVM) и содержит три уровня, показанных на рис. 7.11. Как обычно, мы изолируем большую часть логики от других модулей — в данном случае `PresentationLogic`. `ProductManagementClient` является простым исполняемым компонентом, который немного участвует в определении пользовательского интерфейса и делегирует реализацию в другие модули. Приложение состоит из трех различных сборок. Сборка `ProductManagementClient` является исполняемым компонентом и содержит пользовательский интерфейс, реализованный на XAML без использования кода. Библиотека `PresentationLogic` содержит модели представлений (ViewModel) и поддерживающие

их классы, и, наконец, библиотека `ProductWcfAgent` содержит адаптер (Adapter) между специальной абстракцией `IProductManagementAgent` и конкретным WCF-прокси, который применяется для коммуникаций с веб-сервисом управления продуктами. Стрелки зависимостей на рисунке показывают, что `ProductManagementClient` выступает в роли корня компоновки, поскольку он соединяет остальные модули воедино.

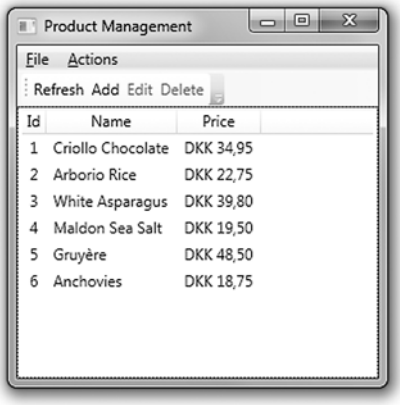


Рис. 7.10. Главное окно приложения Product Management

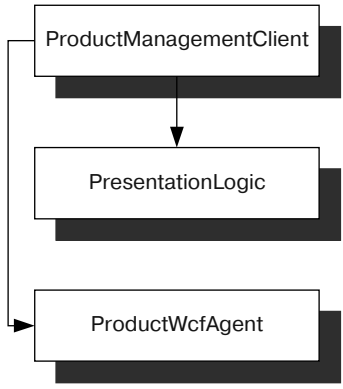


Рис. 7.11. Приложение содержит три уровня

В соответствии с концепцией MVVM, мы назначаем `ViewModel` (модель представления) свойству `DataContext` главного окна, после чего компонент связывания и шаблонизации гарантирует корректное отображение данных по мере того, как мы используем новые модели представления или изменяем данные в существующих.

MVVM

«Модель — представление — модель представления» (Model View ViewModel, MVVM)[1] — это паттерн проектирования, особенно уместный при работе с WPF. Он разделяет код пользовательского интерфейса на три отдельные функции.

Model представляет собой базовую модель для приложения. Часто, хотя и не всегда, ею оказывается модель домена (Domain Model). Зачастую она состоит из объектов POCO. В рассматриваемом примере модель домена реализована в веб-сервисе, поэтому на данном уровне доменной модели у нас нет. Однако приложение оперирует с абстракцией на верхнем уровне прокси веб-сервиса, это и будет служить нашей моделью. Заметьте, что модель обычно выражается независимым от пользовательского интерфейса способом. Не предполагается отображать ее непосредственно в пользовательском интерфейсе, поэтому она не содержит никакой специфической для WPF функциональности.

Представление (View) — это пользовательский интерфейс, с которым мы работаем. В WPF мы можем декларативно объявить представление, используя средства XAML, а затем применить связывание и шаблонизацию данных для их отображения. Возможно задействовать представления вообще без написания программного кода.

[1] Дополнительную информацию о MVVM можно найти в работе *Josh Smith* Patterns: WPF Apps With The Model-View-ViewModel Design Pattern, 2009: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

ViewModel выступает в качестве соединительного звена между представлением и моделью. Каждая ViewModel представляет собой класс, который транслирует и представляет модель специфическим для той или иной технологии способом. В WPF это означает, что можно представлять списки как ObservableCollections и т. п.

Внедрение зависимостей в главную ViewModel

Главное окно MainWindow описывается через XAML-разметку и не содержит никакого кода. Вместо этого оно применяет связывание данных их отображения и обработки команд пользователя. Чтобы обеспечить работу такого механизма, мы должны присвоить значение MainWindowViewModel свойству DataContext.

MainWindowViewModel предоставляет данные, такие как список продуктов, а также содержит команды для создания, изменения или удаления продукта. Возможность применения такой функциональности зависит от сервиса, который предоставляет доступ к каталогу продуктов: абстракция IProductManagementAgent.

Кроме IProductManagementAgent, MainWindowViewModel для работы также требуется сервис, который модель могла бы использовать для управления своей оконной средой, например, для показа модальных диалоговых окон. Предназначенная для этой цели зависимость называется IWindow.

MainWindowViewModel использует внедрение конструктора в конструкторе, имеющем такую сигнатуру:

```
public MainWindowViewModel(IProductManagementAgent agent, IWindow window)
```

Чтобы связать приложение, мы должны создать MainWindowViewModel и назначить эту модель свойству DataContext экземпляра MainWindow.

Связывание MainWindow и MainWindowViewModel

В данном примере есть дополнительный нюанс, заключающийся в следующем: чтобы корректно реализовать IWindow, требуется иметь ссылку на реальное окно WPF (MainWindow); но ViewModel требует IWindow, а свойство DataContext в MainWindow должно иметь значение ViewModel. Другими словами, мы получили циклическую зависимость.

В главе 6 мы сталкивались с циклическими зависимостями и рассматривали на примерах способы их устранения, поэтому я не буду повторяться здесь. Важно указать, что вы вводите фабрику MainWindowViewModelFactory, которая отвечает за создание экземпляров MainWindowViewModel.

Эта фабрика используется из реализации IWindow, называемой MainWindowAdapter, для создания MainWindowViewModel и присваивания ее свойству DataContext главного окна MainWindow:

```
var vm = this.vmFactory.Create(this);
this.WpfWindow.DataContext = vm;
```

Переменная экземпляра vmFactory представляет экземпляр фабрики IMainWindowViewModelFactory, и вы передаете методу Create этой фабрики экземпляр этого же класса, который реализует IWindow. Итоговый экземпляр ViewModel затем назначается свойству DataContext объекта WpfWindow, который является экземпляром MainWindow.

ПРИМЕЧАНИЕ

Я не вдаюсь в детали, поскольку мы рассматривали их в главе 6. Если вам требуется восстановить в памяти материал о циклических зависимостях, вернитесь к этой главе и перечитайте ее.

СОВЕТ

Реализуемое в WPF связывание данных требует от нас назначить зависимость (ViewModel) свойству DataContext. Это, на мой взгляд, является злоупотреблением внедрения свойств, поскольку сигнализирует о том, что зависимость является необязательной, тогда как это не так. Однако в WPF 4 появился класс, называемый XamlSchemaContext, который может использоваться как SEAM, обеспечивающий большую гибкость по сравнению с созданием экземпляров представлений с помощью разметки[1].

Рисунок 7.12 показывает полный граф зависимостей для приложения. Граф зависимостей для `MainWindowAdapter`, который выступает в качестве корневого объекта приложения. Он использует фабрику `MainWindowViewModelFactory` для создания `ViewModel` и назначает ее главному окну `MainWindow`. Для создания `MainWindowViewModel`, фабрике требуется, чтобы `WcfProductManagementAgent` был передан в `ViewModel`. Этот агент выступает в качестве адаптера (Adapter) между `IProductManagementAgent` и прокси WCF. Ему нужна `ProductChannelFactory` для создания экземпляров WCF-прокси, а также `IclientContractMapper`, который обеспечит трансляцию между `ViewModels` и контрактами данных (Data Contracts) WCF.

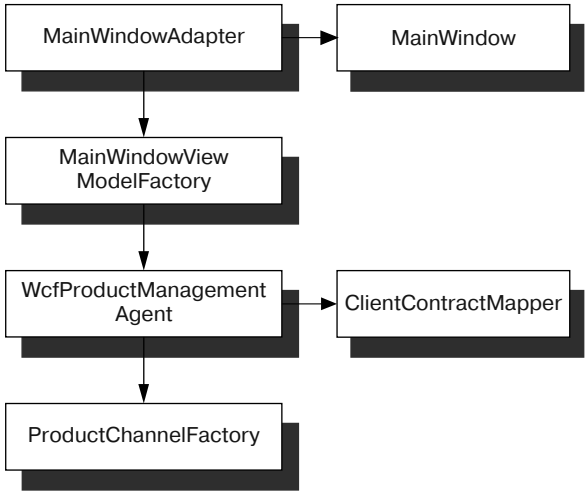


Рис. 7.12. Полный граф зависимостей для приложения

Теперь, когда все составные блоки приложения определены, можно скомпоновать его. Чтобы обеспечить равнозначность кодов, использующих «внедрение для бедных» и полновесный контейнер внедрения зависимостей, я реализовал этот код как метод `Resolve` для класса специализированного контейнера. Ниже представлена эта реализация.

[1] Дополнительную информацию смотрите в работе *Simon Ferquel* [Xaml] IoC-enabled Xaml parser, 2010: www.simonferquel.net/blog/archive/2010/02/19/xaml-ioc-enabled-xaml-parser.aspx.

Листинг 7.10. Компоновка главного окна приложения

```
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 этот код возвращает экземпляр `IWindow`, реализованный как `MainWindowAdapter`, для чего требуются WPF-`Window` и фабрика `ImainWindowViewModel Factory`. Первое окно, которое необходимо показать посетителям, должно быть `MainWindow`, поэтому именно его нужно передать в `MainWindowAdapter`.

Фабрика `MainWindowViewModelFactory` использует внедрение конструктора для передачи `IProductManagementAgent`, поэтому `WcfProductManagementAgent` должен быть скомпонован со своими двумя зависимостями.

Окончательная версия `MainWindowAdapter`, возвращаемая из метода, обернута вокруг `MainWindow`, поэтому когда вызывается метод `Show`, он делегирует свою работу методу `Show` из `MainWindow`. Это именно те задачи, которые должны решаться в корне компоновки.

Реализация корня компоновки

Теперь, когда вы знаете, как сформировать приложение, требуется только сделать это в правильном месте. Как описывалось выше, прежде всего требуется открыть `App.xaml` и удалить атрибут `StartupUri`, поскольку вы хотите явно и самостоятельно скомпоновать начальное окно.

После того как это будет сделано, вам только понадобится переопределить метод `OnStartup` в `App.xaml.cs` и вызвать контейнер (листинг 7.11).

Листинг 7.11. Реализация корня компоновки в WPF

```
protected override void OnStartup(StartupEventArgs e)
{
   base.OnStartup(e);

   var container = new ProductManagementClientContainer();
   container.ResolveWindow().Show();
}
```

В этом примере используется специализированный контейнер ProductManagementClientContainer, но можно применять и стандартный контейнер внедрения зависимостей типа Unity или StructureMap. Вы посылаете контейнеру запрос на создание экземпляра IWindow и сразу вызываете метод Show созданного экземпляра. Созданный экземпляр IWindow представляет собой MainWindowAdapter; когда вызывается его метод Show, он, в свою очередь, вызывает метод Show инкапсулированного MainWindow, который приводит к появлению специального окна на экране.

WPF определяет простое и удобное место для корня компоновки. Все, что вы должны сделать, — удалить элемент StartupUri из App.xaml, переопределить метод OnStartup в App.xaml.cs и скомпоновать приложение в этом месте.

До сих пор мы рассматривали примеры, в которых фреймворки имели такие швы, которые позволяли нам управлять жизненным циклом ключевых экземпляров (веб-страниц, экземпляров сервисов, окон и т. д.). Во многих случаях сделать это совершенно просто; но даже если возникают трудности, как в случае с WCF, все же имеется возможность реализовать настоящее внедрение зависимостей, не отступая от наших принципов.

Однако некоторые фреймворки не предоставляют нам такой возможности.

7.5. Компоновка приложений ASP.NET

Некоторые фреймворки берут на себя создание и управление жизненным циклом классов, которые мы пишем. Наиболее популярный фреймворк такого типа — это ASP.NET (Web Forms в противоположность MVC).

ПРИМЕЧАНИЕ

Некоторые другие фреймворки, также обладающие данной особенностью, — это Microsoft Management Console (MMC) управляемый SDK, и последняя новинка: PowerShell.

Наиболее общим признаком таких фреймворков является то, что для работы с ними наши классы должны иметь конструктор по умолчанию. В ASP.NET, например, любой создаваемый нами класс Page должен иметь конструктор без параметров. В таких фреймворках не может применяться внедрение конструктора, поэтому рассмотрим доступные варианты.

7.5.1. Компоновка в ASP.NET

Внедрение конструктора было бы предпочтительным вариантом, поскольку оно гарантировало бы, что наши классы Page будут всегда инициализироваться требуемыми зависимостями. Поскольку это невозможно, остается выбирать из следующих альтернатив:

- перенести и дублировать корни компоновки в каждом классе Page;
- использовать локатор сервисов для разрешения всех зависимостей в каждом классе Page.

Но помните, что «Локатор сервисов» является антипаттерном, поэтому этот вариант нежелателен. Лучшей альтернативой остается компромисс относительно расположения корня компоновки.

В идеале нам следовало бы предпочесть сценарий, показанный на рис. 7.13, в котором мы имеем лишь один корень компоновки на приложение, но это оказывается невозможным в ASP.NET, поскольку мы не сможем скомпоновать экземпляры `Page` снаружи. Другими словами, фреймворк Web Forms вынуждает нас компоновать приложение из каждой `Page`.

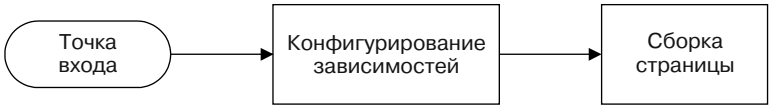


Рис. 7.13. В идеальном мире мы бы предпочли иметь возможность компоновать объекты типа Page в корне компоновки приложения

ПРИМЕЧАНИЕ

До этого момента я говорил только об объектах типа Page, но ASP.NET требует наличия конструктора по умолчанию для множества объектов, если мы хотим использовать этот фреймворк. Другим примером служит Object Data Sources. Рассуждения из этого раздела в равной степени относятся ко всем типам, которые должны иметь конструктор по умолчанию.

Чтобы разрешить эту проблему, нам придется пойти на компромисс, но я считаю, что компромисс относительно места расположения корня компоновки гораздо более безопасен, чем использование локатора сервисов.

По существу, мы превращаем каждую страницу в корень компоновки, как показано на рис. 7.14.

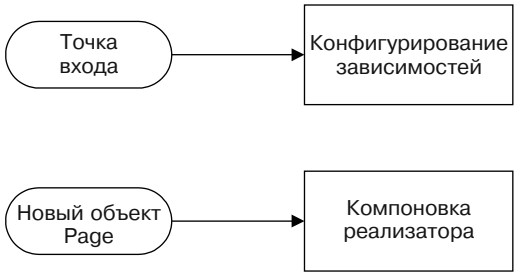


Рис. 7.14. Превращаем каждую страницу в корень компоновки

Принцип единственной ответственности напоминает нам, что каждый класс должен отвечать только за одну функцию; сейчас, поскольку мы используем класс `Page` для компоновки всех требуемых зависимостей, мы должны делегировать ответственность за реализацию нашему реализатору (Implementer). Это превращает `Page` в простой объект (Humble Object), использующий других участников, таких как обработчики событий щелчков, исключительно для запуска действий в созданном для `Page` реализаторе.

Различия между перемещением корня компоновки в каждый класс и применением локатора сервисов весьма тонки. Разница заключается в том, что при использовании локатора сервисов мы разрешаем каждую из зависимостей класса `Page` отдельно и используем их непосредственно в этом классе. Как всегда при задействовании локатора сервисов, это размывает фокус класса. Кроме того, становится заманчивым сохранить контейнер и использовать его для разрешения других зависимостей при необходимости.

Для противодействия этой тенденции важно использовать данный контейнер только для разрешения реализатора и затем забыть о нем. Это позволит нам следовать соответствующим паттернам внедрения зависимостей (таким как внедрение конструктора) в остальном коде приложения.

Хотя это только теория, но вам станет легче, когда вы услышите, что все это реализуется достаточно просто. Это лучше проиллюстрировать примером.

7.5.2 Пример: подключение CampaignPresenter

Наше известное любимое демоприложение работает со скидками цены на продукты и с предлагаемыми продуктами, но оно до сих пор не предоставляет бизнес-пользователям возможностей по управлению этими аспектами. В этом примере мы рассмотрим, как осуществляется компоновка приложения ASP.NET (показанного на рис. 7.15), которое позволит бизнес-пользователям модифицировать данные для связанных с продуктами компаний.

	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	☑	

Рис. 7.15. Приложение управления компаниями (CampaignManagement) позволяет бизнес-пользователям редактировать данные компаний (цена по прайс-листу (Featured Price) и цена со скидкой (Discount Price) для продукта)

Чтобы не усложнять ситуацию сверх меры, мы создаем приложение, состоящее из единственного элемента управления `GridView`, связанного с `ObjectDataSource`. Источник данных — это специфичный для приложения класс, делегирующий свою логику работы модели домена и через нее в конечном итоге библиотеке доступа к данным, хранящей данные в базе данных SQL Server.

Вы можете по-прежнему использовать `global.asax` для конфигурирования зависимостей, но вам придется отложить компоновку приложения до тех пор, пока `Page` и ее `ObjectDataSource` не будут созданы. Зависимости конфигурируются так, как это демонстрировалось в предыдущих примерах.

Конфигурирование зависимостей в ASP.NET

Точкой входа в ASP.NET является файл `global.asax`, и хотя вы не можете выполнить компоновку всего приложения в этом месте, вы можете создать свою Mise en Place, делая все компоненты готовыми к моменту запуска приложения:

```
protected void Application_Start(object sender, EventArgs e)
{
    this.Application["container"] =
        new CampaignContainer();
}
```

Здесь создается контейнер, который сохраняется в контексте приложения (Application Context), чтобы его можно было использовать затем в любой момент. Такое решение позволит разделять контейнер между отдельными веб-запросами, что является предпочтительным, если вам требуется сохранить некоторые зависимости на протяжении всего жизненного цикла процесса (о жизненном цикле поговорим в главе 8).

ПРИМЕЧАНИЕ

Как и в других примерах в этой главе, я использую «внедрение зависимостей для бедных», чтобы продемонстрировать применяемые основные принципы. CampaignContainer — это специальный класс, созданный исключительно для этого примера, но вы можете легко заменить его на выбранный вами контейнер внедрения зависимостей.

Многие объекты типа `Page` и объекты-источники данных могут совместно использовать один и тот же контейнер, обращаясь к контексту приложения. Но в таком подходе скрывается опасность нежелательного использования в качестве локатора сервисов, так как любой класс потенциально может получить доступ к контексту приложения.

Таким образом, важно делегировать реализацию классам, которые не могут обратиться к контексту приложения. На практике это означает, что делегирование должно осуществляться в классы, реализованные в других независимых библиотеках, которые не имеют ссылок на ASP.NET.

ПРИМЕЧАНИЕ

Можно еще попробовать соблюдать определенную дисциплину, ограничивая себя от обращений к контексту приложения, если мы не реализуем корень компоновки. Это может получиться, если все разработчики обладают опытом написания слабо связанного кода; но если имеется подозрение, что некоторые участники команды не полностью осознают возникающие проблемы, лучше защитить код путем использования независимых библиотек. Раздел 6.5 описывает, как это можно сделать.

В данном примере вы будете делегировать всю реализацию в отдельную библиотеку презентационной логики, чтобы гарантировать, что ни один класс напрямую не использует контекст приложения. Вы не разрешаете библиотеке ссылаться ни на какие сборки ASP.NET (такие как System.Web).

Рисунок 7.16 показывает фрагмент архитектуры приложения. Сразу видно, что вы используете классы в корне приложения (`Default Page` и `CampaignDataSource`) как корни компоновки, которые создают классы из уровня презентационной логики вместе с их зависимостями. Корень приложения `CampaignManagement` — это един-

ственное место в приложении, ссылающееся на ASP.NET. Класс `CampaignDataSource` имеет конструктор по умолчанию, но работает как корень компоновки и простой объект, который делегирует все вызовы методов в `CampaignPresenter`. Как обычно, стрелки обозначают ссылки, и корневое приложение ссылается на прочие модули, потому что оно связывает их воедино. Как модуль презентационной логики, так и модуль доступа к данным ссылаются на библиотеку модели домена. На схеме показаны не все участвующие классы.

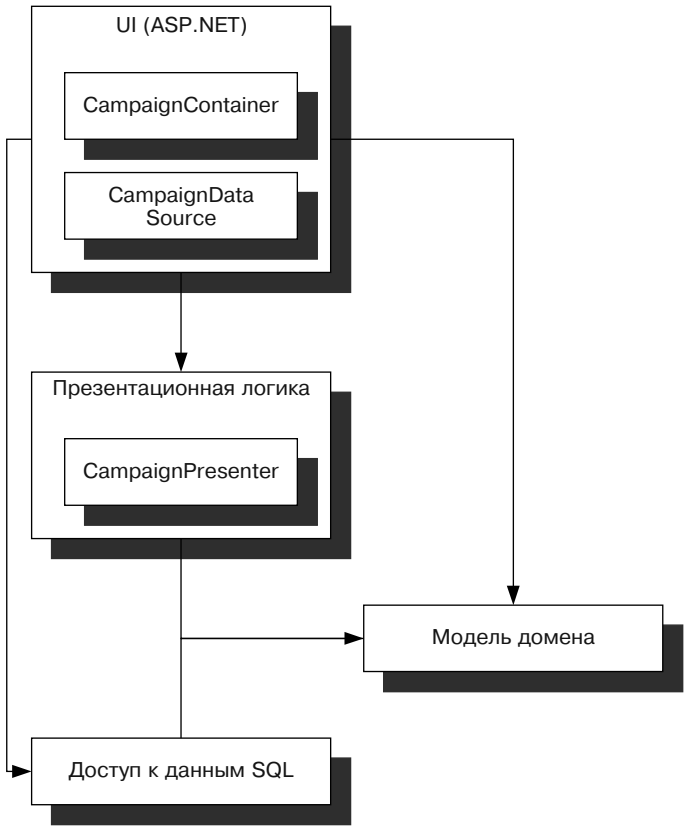


Рис. 7.16. Фрагмент архитектуры приложения

Теперь, зная граф зависимостей приложения, вы можете реализовать корень компоновки для экрана, представленного на рис. 7.15.

Компоновка ObjectDataSource

Показанная на рис. 7.15 страница Default Page состоит из элемента управления `GridView` и связанного с ним элемента управления `ObjectDataSource`. Как и в случае с классами `Page`, класс, используемый в качестве `ObjectDataSource`, должен иметь конструктор по умолчанию. Чтобы соответствовать этому требованию, вы явно создаете класс, представленный в листинге 7.12.

Листинг 7.12. Компоновка Presenter как источника данных

```
public class CampaignDataSource
{
    private readonly CampaignPresenter presenter;
    public CampaignDataSource()
    {
        var container =
            (CampaignContainer)HttpContext.Current
            .Application["container"];
        this.presenter = container.ResolvePresenter();
    }

    public IEnumerable<CampaignItemPresenter> SelectAll()
    {
        return this.presenter.SelectAll();
    }

    public void Update(CampaignItemPresenter item)
    {
        this.presenter.Update(item);
    }
}
```

Класс CampaignDataSource имеет конструктор по умолчанию, так как этого требует ASP.NET. В соответствии с принципом быстрого сбоя (Fail Fast) он пытается получить контейнер из контекста приложения и создать экземпляр CampaignPresenter ❶, который будет выступать в качестве настоящей реализации.

Все участники класса CampaignDataSource делегируют вызов созданному презентатору ❷, то есть работают как простой объект.

ПРИМЕЧАНИЕ

Для любителей паттернов проектирования класс CampaignDataSource выглядит немного похожим на «Декоратор» или «Адаптер». Он не реализует сильно типизированный интерфейс, но обертывает требуемую реализацию в класс, который соответствует требованиям, выдвинутым со стороны ASP.NET.

Вы можете поинтересоваться, зачем же нам нужен этот дополнительный уровень косвенности. Если вы используете разработку через, это должно быть очевидно: HttpContext.Current не доступен во время модульного тестирования, поэтому вы не можете провести модульное тестирование для CampaignDataSource. Это важная причина, по которой следует использовать простой объект.

Хотя эту конструкцию можно назвать, мягко говоря, неуклюжей, она позволяет следовать правильным паттернам внедрения зависимостей в классе CampaignPresenter и далее по уровням приложения.

Компоновка класса Presenter

Я не хочу заставлять вас изучать все детали класса CampaignPresenter, достаточно рассмотреть сигнатуру его конструктора, поскольку в нем используется внедрение конструктора:

```
public CampaignPresenter(CampaignRepository repository,
    IPresentationMapper mapper)
```

Зависимостями являются абстрактный класс `CampaignRepository` и интерфейс `IPresentationMapper`. Тонкости функционирования этих абстракций не так важны, как способ их компоновки. В листинге 7.13 показано, как это делает `CampaignContainer`. Вспомните, что он был сконфигурирован в `global.asax` и там же зарегистрирован в контексте приложения.

Листинг 7.13. Разрешение CampaignPresenter

```
public CampaignPresenter ResolvePresenter()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;
    CampaignRepository repository =
        new SqlCampaignRepository(connectionString);

    IPresentationMapper mapper =
        new PresentationMapper();

    return new CampaignPresenter(repository, mapper);
}
```

❶ Создать репозиторий

❷ Создать соотнесение

❸ Скомпоновать Presenter

Задача метода `ResolvePresenter` заключается в осуществлении компоновки экземпляра `CampaignPresenter`. Из его конструктора вы знаете, что ему нужен `CampaignRepository`, поэтому вы соотносите его с экземпляром `SqlCampaignRepository` ❶. Еще одной зависимостью является `IPresentationMapper`, который соотносится с конкретным классом `PresentationMapper` ❷.

Имея все требуемые зависимости, вы можете сформировать ❸ и вернуть новый экземпляр класса `CampaignPresenter`.

Применить внедрение зависимостей в ASP.NET вполне возможно, но для этого требуется много больше работы, чем нам бы хотелось. Основной недостаток использования каждого объекта `Page` и каждого объекта источника данных как комбинированного корня компоновки и пустого объекта заключается в необходимости дублировать большое количество членов класса.

Помните, как каждый член класса `CampaignDataSource` делегировал свою реализацию в совпадающий по имени метод в `CampaignPresenter`? Вам придется повторять эту идиому программирования по всему ASP.NET-приложению. Для каждого обработчика щелчка вам понадобится определить и сопровождать ассоциированный метод в классе `Presenter` и т. д.

Как обсуждалось в главе 3, я уподобил концепцию корня компоновки, почерпнутую из методологии гибкой разработки программного обеспечения (Lean Software Development), идее последнего ответственного момента (Last Responsible Moment). Когда используются фреймворки типа ASP.NET MVC или WCF, мы можем откладывать сборку приложения на всем пути по точке входа приложения, но это не получится в случае ASP.NET. Не имеет значения, насколько упорно мы пытаемся

сделать это, мы можем только отложить решения о компоновке объектов до момента, пока не появится требование конструктора по умолчанию.

Это становится «самой высокой точкой», в которой мы можем скомпоновать объекты.

Несмотря на то что мы пошли на некоторый компромисс, в целом мы следовали принципу корня компоновки. Мы собирали иерархии объектов настолько близко к архитектурной вершине, насколько это вообще возможно, и придерживались при этом настоящих паттернов внедрения зависимостей.

ASP.NET все же не так плох: мы можем совместно использовать наш экземпляр контейнера через контекст приложения. Некоторые фреймворки не позволяют нам сделать даже это.

7.6. Компоновка PowerShell cmdlets

Некоторые фреймворки вообще не предоставляют никаких швов, которые позволили бы нам управлять жизненным циклом основных элементов фреймворка. Windows PowerShell относится к их числу.

ПРИМЕЧАНИЕ

Прочитайте этот раздел, даже если вас совершенно не интересует PowerShell. Я выбрал его для рассмотрения как пример максимально проблемной области для внедрения зависимостей. Я мог бы рассмотреть еще Managed MMC SDK, но он настолько неприятен во многих других отношениях, что я все же остановился на PowerShell, посчитав его более интересным в качестве примера.

Важным элементом в PowerShell является `cmdlet`. Вы можете считать `cmdlet` продвинутой утилитой командной строки.

`Cmdlet` — это класс, который наследуется из `Cmdlet`[1], и он должен иметь конструктор по умолчанию. Как и в случае ASP.NET, это требование фактически лишает нас возможности использовать внедрение конструктора. Решение тоже похоже: мы переносим корень компоновки в конструктор каждого `cmdlet`. Единственное отличие заключается в том, что здесь нет встроенного контекста приложения, поэтому нам придется вместо этого прибегнуть к наименьшему общему знаменателю: статическому классу.

ПРИМЕЧАНИЕ

Я считаю каждое использование ключевого слова static примером дурного кода, но, в отличие от антишаблонов, дурной код всего лишь выявляет потенциальные недостатки конструкции. В некоторых специальных случаях идиома дурно пахнущего кода является оправданной, и наш случай является именно таким.

Вы можете спросить, а чем это отличается от антипаттерна «Локатор сервисов»? Точно как и для ASP.NET, главное отличие заключается не в структуре кода, а в способе использования. Вместо того чтобы пытаться использовать статический локатор сервисов как виртуальное ключевое слово `new`, мы используем его только один

[1] System.Management.Automation.Cmdlet, если уж быть совсем точным, хотя он может быть также унаследован из System.Management.Automation.PSCmdlet.

раз в cmdlet. Чтобы в дальнейшем защитить себя же от попыток злоупотребления, мы можем сделать «Компоновщик» внутренним и использовать его только для разрешения типов из других сборок, как показано на рис. 7.17. Все методы Resolve возвращают классы, определенные в других сборках. Поэтому, после того как контейнер создает реализатор (Implementer), никакие классы в созданной иерархии зависимостей больше не имеют доступа к статическому контейнеру, поскольку все эти классы находятся за пределами корневой сборки приложения, а контейнер является вложенным в нее.

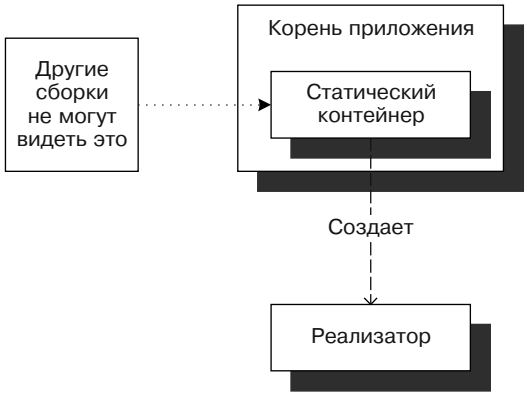


Рис. 7.17. Когда не удается избежать применения статического контейнера, мы можем сделать его внутренним и поместить в корневую сборку приложения

Результатом создания графа зависимостей будет класс, определенный в другой сборке, и этот класс не имеет доступа к статическому контейнеру, так как последний является внутренним по отношению к корневой сборке приложения. Реализатор cmdlet должен использовать корректные шаблоны внедрения зависимостей, такие как внедрение конструктора, чтобы использовать любые зависимости, и мы эффективно оградили самих себя от опасностей локатора сервисов.

Рассмотрим пример, иллюстрирующий эти принципы.

7.6.1. Пример: компоновка cmdlets для управления корзиной

Этот пример возвращает нас к нашему демоприложению. Как и все другие приложения электронной коммерции, наша программа также поддерживает функциональность корзины покупок. Часто бывает так, что пользователи набирают товары в свои корзины, а затем уходят с сайта и никогда больше не возвращаются.

Хотя в наши дни компьютерная память является дешевым ресурсом, заказчики потребовали, чтобы у них имелась возможность очищать заброшенные тележки на основе различных критериев. Прежде всего они хотят иметь возможность отбирать заброшенные тележки на основании времени их последнего изменения. Если стоимость отобранных в тележку товаров велика, она не должна удаляться (или,

возможно, ей должен быть предоставлена отсрочка), но итог должен подсчитываться в соответствии с действующими бизнес-правилами.

Похоже, что хорошим решением будет использование скриптового API, поскольку администратор должен иметь возможность определять и планировать простые скрипты очистки. PowerShell удовлетворяет этому требованию, поскольку имеет развитую функциональность для фильтрации и организации взаимодействия отдельных программных компонентов через конвейеры.

Требуемый API может быть реализован с помощью двух `cmdlets`: первого для получения всех корзин и второго для удаления корзины заданного пользователя. Листинг 7.14 содержит пример того, как это может выглядеть в интерактивной сессии.

Листинг 7.14. Удаление корзин, сформированных более месяца тому назад

```
PS C:\> Get-Basket

LastUpdated        Owner                          Total
-----------        -----                          -----
19.03.2010 20:5... pløeh                         89,4000
22.01.2010 19:5... nd h                         199,0000
21.03.2010 09:1... fnaah                        171,7500

PS C:\> $nøw = [System.DateTime]::Nøw
PS C:\> $mønth = [System.TimeSpan]::FrømDays(30)
PS C:\> $øld = $nøw - $mønth

PS C:\> Get-Basket | ? { $_.LastUpdated -lt $øld } |
Remøve-Basket
PS C:\> Get-Basket

LastUpdated        Owner                          Total
-----------        -----                          -----
19.03.2010 20:5... pløeh                         89,4000
21.03.2010 09:1... fnaah                        171,7500

PS C:\>
```

(1) Старая корзина

(2) Вычислить дату начала периода

(3) Удалить старые корзины

Прежде чем приступить к удалению корзин, вы хотите просмотреть список всех корзин, имеющихся в системе. Для этого можно использовать специальный `cmdlet` называемый `Get-Basket`. Заметьте, что у каждой корзины имеется три свойства, показывающих, когда была создана корзина, кто является ее владельцем и на какую сумму (с учетом скидок) в ней лежит товар.

Текущей датой в примере является 22 марта 2010 года.

Обратите внимание, что вторая корзина в списке (1) создана более чем 30 дней назад. Можно вычислить дату отсечки (2), начиная с текущей даты, и использовать ее для выражения фильтрации (3). Вы можете удалить все старые корзины, используя конвейер сначала для фильтрации полученного применением списка `Get-Basket`, а затем применяя конвейер еще раз для удаления отфильтрованных корзин с помощью `Remove-Basket cmdlet`. Кроме того, можно выполнить фильтрацию еще и по свойству `Total`.

В конце еще раз просматривается список оставшихся корзин для контроля удаления старых корзин.

ПРИМЕЧАНИЕ

Не беспокойтесь, если вы не понимаете всех деталей процесса фильтрации. Эта книга не о PowerShell, поэтому я не хочу углубляться в этот предмет.

Чтобы получить такой скриптовый API, вы должны реализовать два специальных `cmdlets`. Поскольку одно из требований состоит в том, что при вычислении `Total` должны приниматься во внимание все действующие бизнес-правила, вы должны скомпоновать эти `cmdlets` с вашей моделью домена.

Компоновка GetBasketCmdlet

Рассмотрим реализацию `Get-Basket cmdlet`. `Remove-Basket` реализуется подобным образом, так что я не буду показывать еще и его.

Чтобы уйти от приманки статического контейнера, вы реализуете целый мост между PowerShell `cmdlet` и моделью домена в отдельной библиотеке `BasketPowerShellLogic`. Рисунок 7.18 показывает, как приложение компонуется из разных библиотек. Библиотека `BasketPowerShell` содержит только инфраструктуру, необходимую для PowerShell, — она является простым объектом. Как только статический `BasketContainer` разрешает `BasketManager`, вся последующая реализация производится в различных сборках. Класс `BasketManager` не имеет доступа к внутреннему `BasketContainer`, но использует IBasketService из модели домена. Как обычно, стрелки обозначают ссылки. Показаны не все участвующие классы.

ПРИМЕЧАНИЕ

Если вы считаете, что рис. 7.18 немного похож на рис. 7.16, то вы начали распознавать паттерны. Вы можете припомнить IBasketService из подраздела 2.3.2.

Класс `GetBasketCmdlet` должен иметь конструктор по умолчанию, чтобы соответствовать требованиям PowerShell, поэтому его можно использовать в качестве корня компоновки и оставить его простым объектом. Листинг 7.15 показывает, каким простым он является.

Листинг 7.15. Реализация GetBasketCmdlet

```
[Cmdlet(VerbsCommon.Get, "Basket")]
public class GetBasketCmdlet : Cmdlet
{
    private readonly BasketManager basketManager;

    public GetBasketCmdlet()
    {
        this.basketManager =
            BasketContainer.ResolveManager();   // 1 Корень компоновки
    }

    protected override void ProcessRecord()
```

```
    {
        var baskets =
            this.basketManager.GetAllBaskets();
        this.WriteObject(baskets, true);
    }
}
```

❷ Делегирование реализатору

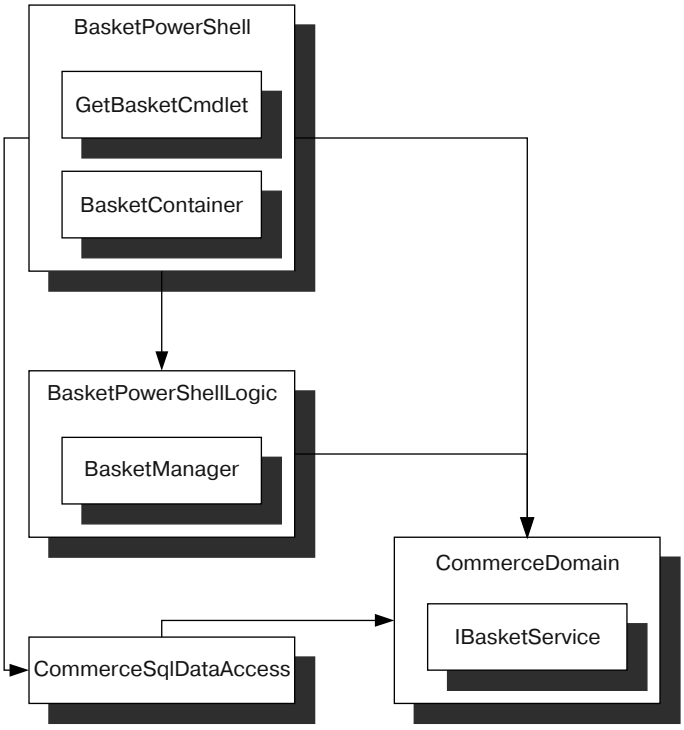


Рис. 7.18. Компоновка приложения из различных библиотек

В требуемом конструкторе используется контейнер по умолчанию для разрешения `BasketManager` ❶, который служит в качестве реализации ❷. `BasketManager` применяет внедрение конструктора, чтобы запросить экземпляр `IbasketService`. Сейчас вам должен быть понятен этот паттерн и реализация `BasketContainer`, показанная в листинге 7.16.

Листинг 7.16. Разрешение BasketManager

```
internal static BasketManager ResolveManager()
{
    BasketRepository basketRepository =
        new SqlBasketRepository(
            BasketContainer.connectionString);
    DiscountRepository discountRepository =
```

❶ Внутренний метод

```
            new SqlDiscountRepository(
                BasketContainer.connectionString);

        BasketDiscountPolicy discountPolicy =
            new RepositoryBasketDiscountPolicy(
                discountRepository);

        IBasketService basketService =
            new BasketService(basketRepository,
                discountPolicy);

        return new BasketManager(basketService);
    }
```

❷ Вернуть диспетчер корзины

Метод, как и весь класс, является внутренним ❶, что делает возможным его вызов из `GetBasketCmdlet`, как показано в листинге 7.15. Но при этом невозможно случайно использовать его из `BasketManager` или его зависимостей.

Реализация метода теперь должна быть вам понятна. Я нахожу его слишком простым, чтобы разъяснять еще раз. Класс `BasketManager` требует экземпляр `IBasketService` ❷, и вы применяете класс `BasketService` (не то что вы имеете какую-то другую реализацию на выбор).

`BasketService` требует `BasketRepository` и `BasketDiscountPolicy`. Для последнего используется `RepositoryBasketDiscountPolicy`. Этот класс требует другой абстракции репозитория, и для обоих репозиториев применяются основанные на SQL Server реализации.

Реализация `BasketManager` является базовой, поэтому я не буду показывать ее. Все, что она делает, — выражает требуемую операцию в терминах модели домена.

`Remove-Basket cmdlet` построен по точно такому же образцу: он использует статический внутренний контейнер `BasketContainer` для разрешения экземпляра `BasketManager` и затем делегирует реализацию разрешенному экземпляру. Оба `cmdlets` построены как комбинация корня компоновки и простого объекта.

Класс `BasketManager` реализован в другой сборке. Как только код выйдет из `cmdlets`, будет исключено, что какая-либо из реализаций сможет использовать статический контейнер в качестве локатора сервисов, поскольку этот контейнер является внутренним для сборки, содержащей `cmdlets`.

ПРИМЕЧАНИЕ

Очевидно, что никакой код никогда не сделает ничего случайно, но разработчик, который пишет этот код, от случайностей не застрахован. Мы изолируем статический контейнер от остального кода, чтобы защитить себя же от возможных ошибок.

Фреймворк типа PowerShell крайне не приспособлен к работе с внедрениями зависимостей. Простой прием, который заключается в том, чтобы делать каждый элемент интерфейса одновременно и корнем компоновки, и простым объектом, предоставляет вам простой способ решения этой проблемы.

7.7. Резюме

Компоновка объектов является одним из трех важных измерений внедрения зависимостей (другими такими измерениями являются управление жизненным циклом и перехват). В данной главе я показал, как выполняется компоновка приложений из слабо связанных модулей во множестве различных сред.

Некоторые фреймворки упрощают такие операции. Когда мы пишем консольные приложения и клиентов Windows (WPF или Windows Forms), мы в большей или меньшей степени можем непосредственно управлять тем, что происходит в точке входа приложения. Это предоставляет нам возможность создать ясный и легко реализуемый корень компоновки в точке входа.

Другие фреймворки, такие как ASP.NET MVC и WCF, делают нашу работу несколько более сложной, но они тем не менее предоставляют швы, которые мы можем использовать для определения того, как приложение должно быть скомпоновано. ASP.NET MVC разрабатывался с ориентацией на внедрение зависимостей, компоновка приложения сводится к реализации заказной фабрики на основе интерфейса `IControllerFactory` и регистрации ее в фреймворке. При использовании WCF имеющийся шов выглядит как чужеродный; но хотя работать с ним труднее, чем реализовать единственный интерфейс, все же возможно получить все преимущества внедрения зависимостей, которые мы только захотим.

Остальные фреймворки явно плохо приспособлены к работе с внедрениями зависимостей и требуют наличия конструкторов по умолчанию. Наиболее известный из них — это ASP.NET (Web Forms), кроме того, следует упомянуть PowerShell и Managed MMC SDK. Данные фреймворки сами управляют жизненным циклом предоставляемых нами классов, поэтому единственным возможным вариантом является оформление каждого класса как отдельного корня компоновки. Это нелегко, поэтому лично я предпочитаю использовать фреймворки, удобные для работы с внедрениями зависимостей, если только у меня имеется возможность выбора.

Без компоновки объектов не может быть внедрения зависимостей, но определенное воздействие оказывает и жизненный цикл объектов, когда мы перемещаем место создания объектов за пределы потребляющих классов. Вы можете посчитать очевидным, что внешний код, который вызывает наш код (часто это контейнер внедрения зависимостей), создает новые экземпляры зависимостей — но когда же внедренные экземпляры уничтожаются? И что произойдет, если внешний вызывающий код решит не создавать каждый раз новый экземпляр, а передаст вам уже используемый? Все это является предметом обсуждения следующей главы.

8 Время жизни объектов

Меню:

- управление временем жизни зависимостей;
- одноразовые зависимости;
- Singlton;
- Transient;
- Per Graph;
- Web Request Context;
- Pooled.

Количество прошедшего времени оказывает огромное влияние на большинство продуктов и напитков, но имеет разные последствия. Лично я нахожу вино «Грюер» 12-месячной выдержки гораздо более вкусным, чем «Грюер» 6-месячной выдержки, но спаржу я люблю свежую, а не размороженную.

Зачастую бывает просто оценить настоящий возраст какого-то элемента, но иногда это оказывается весьма сложным делом. Это особенно примечательно, когда речь идет о вине (рис. 8.1).

Рис. 8.1. Вино, сыр и спаржа: их возраст значительно влияет на их качество

Вина имеют свойство улучшаться с возрастом — до тех пор, пока они не станут слишком старыми и не потеряют свой вкус. Это зависит от массы факторов, включая происхождение и марку вина. Хотя я и интересуюсь винами, не думаю, что овладею способностью предсказывать, когда вино окажется на максимуме. В этом вопросе я полагаюсь на экспертные источники: мои домашние энциклопедии и сомелье в ресторанах. Они разбираются в винах лучше меня, поскольку это их специальность, так что, поскольку я доверяю им, я с удовольствием передаю им бразды правления.

Если вы прочитали предыдущие главы книги, вы знаете, что передача управления является одной из ключевых концепций внедрения зависимостей. Это важный аспект инверсии управления (Inversion of Control), но он влияет далеко не только на возможность разрешить кому-то еще выбирать реализацию нужной абстракции. Если мы разрешаем компоновщику (Composer) передавать зависимости, нам придется согласиться с тем, что мы не можем управлять их жизненным циклом.

Мы же не сомневаемся, что сомелье досконально знает содержимое винного погреба ресторана и может принять гораздо более обоснованное решение, чем мы. Поэтому нам нужно согласиться с тем, что компоновщик будет управлять жизненным циклом зависимостей более эффективно, чем потребитель. Компоновка и управление компонентами — это его единственная функция.

ОПРЕДЕЛЕНИЕ

Компоновщик (как я использую этот термин здесь) — является универсальным термином, обозначающим любой объект или метод, который занимается компоновкой зависимостей. Часто в этой роли выступает контейнер внедрения зависимостей, но это может быть и метод, использующий «внедрение зависимостей для бедных», например метод Main консольного приложения.

В этой главе мы будем исследовать управление жизненным циклом зависимостей. Понимание этой темы сложно переоценить, поскольку вы несомненно пожалеете, если отведаете вина не в том возрасте[1]. Вы неизбежно получите снижение производительности при неправильном конфигурировании жизненного цикла зависимостей. И даже больше, вы можете получить при управления жизненным циклом такой же результат, как от употребления испорченной пищи: нехватку ресурсов. Понимая, как правильно работать с областями жизненных циклов компонентов, вы сможете принимать обоснованные решения и верно конфигурировать ваши приложения.

ПРИМЕЧАНИЕ

В этой главе я использую термины «тип жизнедеятельности» (Lifestyle Type), «стратегия жизненного цикла» (Life Cycle Strategy), «диапазон (область) жизненного цикла» (Life Cycle Scope) и другие странные комбинации как синонимы.

Как показано на рис. 8.2, мы начнем с общего введения в саму концепцию, за которой последует обсуждение так называемых одноразовых зависимостей. Предполагается, что эта первая часть главы содержит всю базовую информацию и описание принципов, которые потребуются вам для принятия обоснованных решений относительно конфигурирования диапазонов жизненных циклов ваших приложений.

[1] Причем это как в вашем «не том возрасте», так и вина!

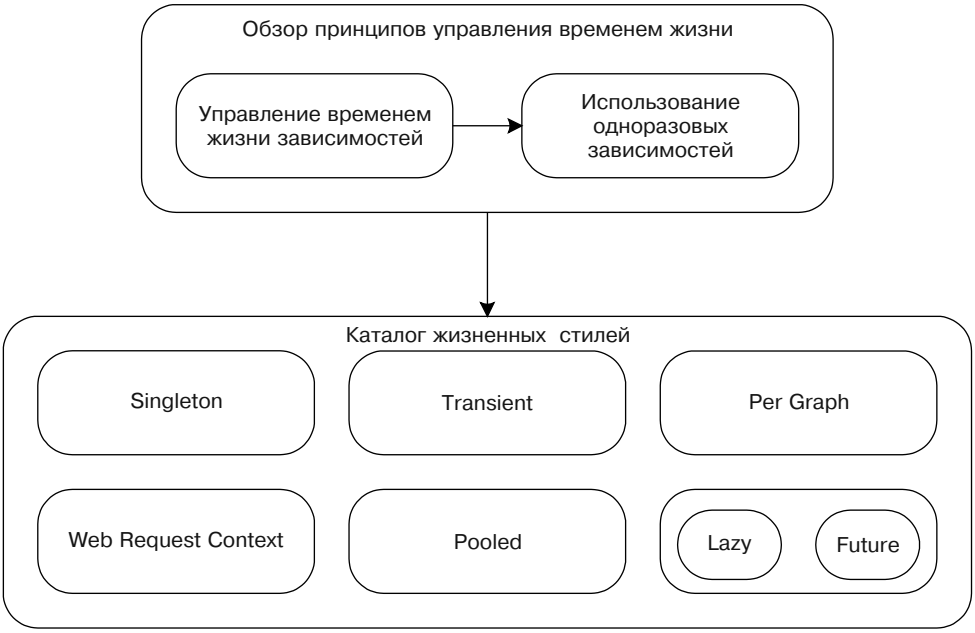


Рис. 8.2. Общая структура этой главы

После этого в оставшейся части главы будут рассмотрены различные стратегии времени жизни. Эта часть главы написана как каталог имеющихся жизненных стилей. В большинстве случаев один из этих паттернов жизненных стилей хорошо подойдет для решения конкретных проблем. Поэтому, заблаговременно с ними разобравшись, вы будете знать, как следует действовать в различных сложных ситуациях. После того как мы рассмотрим каталог, вы вполне будете представлять себе, как управлять временем жизни и работать с распространенными паттернами.

Прежде всего рассмотрим время жизни объектов и узнаем, как оно связано с внедрением зависимостей.

8.1. Управление временем жизни зависимостей

До сих пор мы в основном обсуждали, как внедрение влияет на компоновку зависимостей. В предыдущей главе этот вопрос рассматривался очень подробно, но, как я упоминал в разделе 1.4, композиция объектов является лишь одним из аспектов внедрения зависимостей. Второй важный аспект — управление жизненным циклом объектов.

ПРИМЕЧАНИЕ

В .NET жизненный цикл объектов очень прост: объект создается, используется и уничтожается механизмом сборки мусора. Наличие интерфейса IDisposable немного усложняет эту картину, но сам жизненный цикл не становится сложнее. Когда мы обсуждаем время жизни объектов, мы говорим о том, как мы управляем жизненными циклами объектов.

Когда я впервые узнал, что сфера влияния внедрения зависимостей включает управление временем жизни, у меня не получалось понять глубину связи между композицией объектов и их временем жизни. В конце концов я понял это, и оказалось, что это совсем просто. Итак, приступим!

В этом разделе я введу понятие «управление временем жизни» и покажу, как оно применяется к зависимостям. Мы начнем с рассмотрения общего случая компоновки объектов и того, как это влияет на время жизни зависимостей. Затем мы перейдем к изучению вопроса, как контейнеры внедрения зависимостей могут управлять жизненным циклом зависимостей. Хотя большинство примеров представляет собой специализированный код, предназначенный для запуска в конкретных средах исполнения, мы сделаем небольшое отступление, посвященное модели контейнера внедрения зависимостей. Так мы сможем получить некоторое представление о том, как может конфигурироваться время жизни.

Прежде всего разберемся, почему компоновка объектов влияет на управление жизненным циклом.

8.1.1. Введение в управление временем жизни

Нужно приложить некоторые усилия и признать, что правильный подход к работе требует не управлять зависимостями непосредственно, а запрашивать их через внедрение конструктора или какой-то другой паттерн внедрения. Это нужно усвоить раз и навсегда. Чтобы понять, почему это так, рассмотрим проблему поступательно. Сначала разберем, что означает для зависимостей феномен жизненного цикла стандартных объектов .NET. Вы должны уже знать это, но прочтите еще полстраницы, пока я расскажу все по порядку.

Простой жизненный цикл зависимости

Как известно, при применении внедрения зависимостей мы разрешаем третьей стороне (часто контейнеру внедрения зависимостей) поставлять нам нужные зависимости. Это означает также, что мы должны разрешить ему управлять временем жизни зависимостей. Это наиболее простая вещь, когда речь заходит о создании объектов. Ниже представлен фрагмент кода из корня компоновки нашего демоприложения (полный код приведен в листинге 7.3):

```
var discountRepository =
    new SqlDiscountRepository(connectionString);
var discountPolicy =
    new RepositoryBasketDiscountPolicy(discountRepository);
```

Я полагаю, очевидно, что класс `RepositoryBasketDiscountPolicy` не управляет временем создания `discountRepository`. В данном случае, скорее всего, на все про все уйдет не более одной миллисекунды. Но хотя бы в порядке эксперимента мы можем вставить вызов `Thread.Sleep` между этими двумя строками кода, чтобы продемонстрировать, что можно произвольно разделить их во времени. Это довольно странный поступок, но мы добивались именно этого.

Потребители не управляют созданием требуемых им зависимостей. А что насчет их *ликвидации*? Как правило, в .NET мы не управляем временем удаления объектов.

Неиспользуемые объекты утилизируются сборщиком мусора, и если только речь не идет об одноразовых объектах, мы не можем уничтожить объект вручную.

ПРИМЕЧАНИЕ

Я использую термин «одноразовый объект» как общее краткое название для экземпляров объектов, типы которых реализуют интерфейс IDisposable.

Объекты подпадают под сборку мусора, когда они становятся неиспользуемыми. И наоборот, они существуют, пока на них стоят ссылки. Хотя потребитель не может явно уничтожить объект, он может продлить его существование, сохраняя ссылку на него. Именно это мы и делаем, используя внедрение конструктора, поскольку ссылка на зависимость сохраняется в приватном (Private) поле. Но как только потребитель завершает работу, это же происходит и с зависимостью (рис. 8.3).

Рис. 8.3. Тот, кто внедряет зависимость в потребителя, принимает решение о времени ее создания, но потребитель может продлевать ее существование, сохраняя ссылку на эту зависимость. Когда потребитель завершает работу, зависимость может подпасть под сборку мусора

Даже когда потребитель завершает работу, зависимость может продолжить существование, если другие объекты сохраняют ссылку на нее. В противном случае она будет уничтожена механизмом сборки мусора. Поскольку вы — опытный разработчик для платформы .NET, это для вас не новость, но сейчас мы поговорим о более интересных вещах.

Усложнение жизненного цикла зависимостей

До сих пор наш анализ жизненного цикла зависимостей был довольно прозаическим, но мы можем немного его усложнить. Что произойдет, если одна и та же зависимость потребуется более чем одному потребителю? Можно, например, предоставить каждому потребителю по его собственному экземпляру, как сделано в листинге 8.1.

Листинг 8.1. Компоновка с несколькими экземплярами одной и той же зависимости

```
var repositoryForPolicy =
    new SqlDiscountRepository(connectionString);
var repositoryForCampaign =
    new SqlDiscountRepository(connectionString);

var discountPolicy =
```

```
    new RepositoryBasketDiscountPolicy(
        repositoryForPolicy);

var campaign =
    new DiscountCampaign(repositoryForCampaign);
```

В этом примере два потребителя требуют экземпляр `DiscountRepository`, поэтому создаются два независимых экземпляра на основе одной и той же соединительной строки. Теперь можно передать `repositoryForPolicy` в новый экземпляр `RepositoryBasketDiscountPolicy`, а `repositoryForCampaign` — в новый экземпляр `DiscountCampaign` (1).

Когда речь заходит о жизненных циклах каждого репозитория из листинга 8.1, ничего нового, по сравнению с предыдущим примером, в сущности, не появляется. Каждый из них прекращает использоваться и подвергается сборке мусора, когда соответствующие потребители завершают работу. Это может случиться в разное время, но ситуация лишь немного отличается от описанной ранее.

Другая ситуация возникает, если оба потребителя разделяют одну и ту же зависимость, как показано в следующем примере:

```
var repository =
     new SqlDiscountRepository(connectionString);

var discountPolicy =
     new RepositoryBasketDiscountPolicy(repository);

var campaign = new DiscountCampaign(repository);
```

Вместо создания двух разных экземпляров `SqlDiscountRepository` вы создаете единственный экземпляр, который внедряется в обоих потребителей. Оба потребителя сохраняют ссылки на зависимость для дальнейшей работы.

ПРИМЕЧАНИЕ

Потребители совершенно не подозревают о том, что зависимость используется совместно. Поскольку каждый из них способен работать с любой переданной версией зависимости, не требуется никаких изменений в исходном коде для внедрения этого изменения в конфигурацию зависимости. Это происходит благодаря принципу подстановки Лисков.

ПРИНЦИП ПОДСТАНОВКИ ЛИСКОВ

В своей первоначальной редакции принцип подстановки Лисков является академической и абстрактной концепцией. Но в объектно-ориентированном проектировании мы можем перефразировать его следующим образом: методы, использующие абстракции, должны обладать способностью применять любой класс-наследник этой абстракции без знания этого класса. Другими словами, мы должны обладать способностью заменять абстракцию любой ее реализацией без изменения корректности системы.

Ситуация с жизненным циклом зависимости репозитория явно изменилась по сравнению с предыдущим примером. Оба потребителя должны завершить свою работу, чтобы репозиторий подпадал под сборку мусора. Это может быть сделано в разное время. Становится невозможным предсказать, когда зависимость достиг-

нет конца своего жизненного цикла, и эта особенность только усугубляется, когда количество потребителей возрастает.

Когда количество потребителей достаточно велико, можно предположить, что в любой момент будет существовать как минимум один действующий потребитель, что не позволит уничтожить зависимость. Может показаться, что это является проблемой, но такое случается редко: вместо огромного количества одинаковых экземпляров мы имеем всего один, что позволяет экономить память. Это настолько удобно, что мы формализовали данное качество в паттерне жизненного стиля Singleton (Одиночка). Не путайте его с шаблоном проектирования «Одиночка», хотя между ними и имеется некоторое сходство. Мы рассмотрим этот паттерн жизненного стиля подробнее в разделе 8.3.1.

Главное, что следует учитывать: компоновщик имеет более высокую степень влияния на время жизни, чем любой отдельный потребитель. Компоновщик принимает решение о времени создания экземпляров, и в зависимости от реализованного в нем правила совместного использования экземпляров между потребителями определяет, перестает ли зависимость использоваться, когда завершает работу единственный потребитель или все потребители должны закончить работу, прежде чем зависимость может быть удалена.

Ситуация напоминает посещение ресторана, в котором работает хороший сомелье. Сомелье значительную часть времени тратит на работу с винным погребом, закупая новые вина, дегустируя их для контроля качества и общаясь с поварами, чтобы определить наилучшее сочетание вин с подаваемыми блюдами. Когда мы изучаем винную карту, в нее включены только те сорта, которые сомелье решил предложить в продажу. Мы можем выбрать вино по своему вкусу, но мы не можем лучше разбираться в наборе вин в ресторане и в их сочетании с блюдами, чем сомелье.

Сомелье может принять решение закупить партию вина про запас; и как вы увидите в следующем разделе, компоновщик может принять решение поддерживать экземпляры «живыми», сохраняя ссылки на них.

8.1.2. Управление временем жизни с помощью контейнера

В предыдущем разделе было показано, как можно изменять компоновку зависимостей, чтобы повлиять на их время жизни. В этом разделе мы рассмотрим, как эти изменения можно реализовать с помощью контейнера внедрения зависимостей.

ПРИМЕЧАНИЕ

В данном разделе обсуждаются общие принципы организации управления временем жизни с применением контейнеров внедрения, поэтому я не хочу вдаваться в детали, относящиеся к конкретным контейнерам. Как и во всей третьей части книги, я использую «внедрение для бедных», чтобы проиллюстрировать концепцию.

Мы начнем со знакомства с тем, как может быть организовано управление жизненным циклом зависимостей с использованием универсального контейнера, а затем перейдем к краткому примеру управления временем жизни в реальном контейнере внедрения.

Управление временем жизни с помощью специализированного контейнера

В главе 7 мы создали специализированные контейнеры для компоновки приложений. Одним из них был `CommerceServiceContainer`. Листинг 7.9 демонстрирует реализацию его метода `ResolveProductManagementService`; и, как показано на рис. 8.4, этот метод содержит весь код данного класса. Метод Release не делает совершенно ничего, и в классе отсутствуют как поля, так и свойства. К методу Release мы вернемся в разделе 8.2.2.

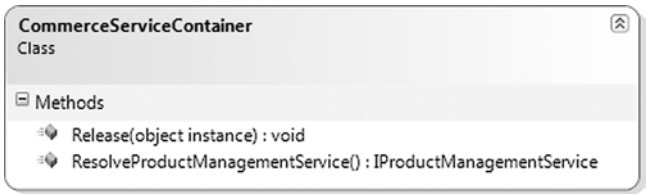


Рис. 8.4. Все реализации класса CommerceServiceContainer в настоящее время заключены в методе ResolveProductManagementService

Как вы помните из листинга 7.9, метод `Resolve` создает весь граф зависимостей динамически каждый раз, когда его вызывают. Другими словами, каждая зависимость является приватной для вызванного `IProductManagementService`, и такие зависимости совместно не используются. Когда экземпляр `IProductManagementService` перестает использоваться (что происходит всякий раз после того, как сервис ответил на запрос), все зависимости также перестают использоваться. Такая ситуация часто называется жизненным стилем Transient (Кратковременный), мы поговорим о нем подробнее в разделе 8.3.2.

Проанализируем граф объектов, созданный контейнером `CommerceServiceContainer` (рис. 8.5), чтобы понять, можем ли мы его улучшить. Каждый созданный экземпляр `ProductManagementService` содержит свои собственные `ContractMapper` и `SqlProductRepository`, который, в свою очередь, содержит собственную строку соединения. Зависимости, расположенные справа, являются неизменяемыми.

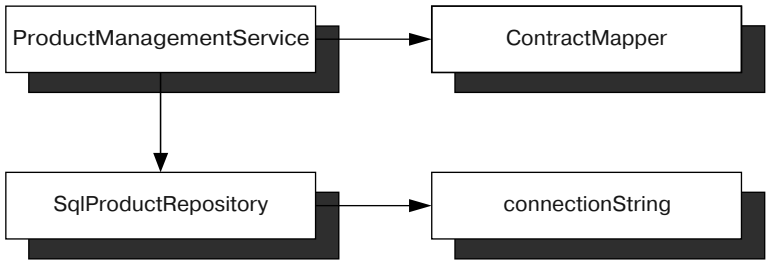


Рис. 8.5. Граф объектов, созданный CommerceServiceContainer

Класс `ContractMapper` является не сохраняющим состояния (Stateless) сервисом, поэтому нет причин создавать новый экземпляр всякий раз, когда требуется обслу-

жить запрос. Строка соединения тоже вряд ли будет меняться, так что мы можем решить использовать ее во всех запросах.

С другой стороны, класс `SqlProductRepository` полагается на контекст объекта (Object Context) фреймворка `Entity`, и представляется более правильным использовать для каждого запроса его новый экземпляр.

С такой конфигурацией получается лучшая реализация `CommerceServiceContainer`, которая будет использовать одни и те же экземпляры `ContractMapper` и строки соединения, но создавать новые экземпляры `SqlProductRepository`. То есть мы должны сконфигурировать `ContractMapper` и строку соединения как жизненный стиль Singleton, а `SqlProductRepository` как Transient. В листинге 8.2 показана реализация этого изменения.

Листинг 8.2. Управление жизненным стилем в контейнере

```
public partial class LifetimeManagingCommerceServiceContainer :
    ICommerceServiceContainer
{
    private readonly string connectionString;
    private readonly IContractMapper mapper;

    public LifetimeManagingCommerceServiceContainer()
    {
        this.connectionString =
            ConfigurationManager.ConnectionStrings
            ["CommerceObjectContext"].ConnectionString;

        this.mapper = new ContractMapper();
    }

    public IProductManagementService
        ResolveProductManagementService()
    {
        ProductRepository repository =
            new SqlProductRepository(
                this.connectionString);
        return new ProductManagementService(
            repository, this.mapper);
    }
}
```

1 Создать Singleton-зависимости

2 Создать Transient-зависимость

Поскольку мы хотим повторно использовать одну и ту же строку соединения и один и тот же `ContractMapper` для всех запросов, мы сохраняем ссылки на них в приватных полях и инициализируем их в конструкторе 1. Ключевое слово `readonly` дополнительно гарантирует, что, получив однажды значения, эти Singleton-экземпляры станут неизменяемыми и их будет невозможно заменить. Но кроме как для обеспечения этой дополнительной защиты `readonly` не требуется больше ни для чего, в том числе не требуется для реализации жизненного стиля Singleton.

Каждый раз, когда у контейнера запрашивается новый экземпляр, он создает новый Transient-экземпляр `SqlProductRepository`, используя строку соединения типа Singleton ❷. В заключение Transient-репозиторий применяется вместе с Singleton-сопоставителем для компоновки и последующего возврата экземпляра `ProductManagementService`.

ПРИМЕЧАНИЕ

Код листинга 8.2 функционально эквивалентен коду листинга 7.9 — он только значительно более эффективен.

Храня ссылки на созданные им зависимости, контейнер может сохранять зависимости живыми так долго, как это понадобится. В предыдущем примере обе Singleton-зависимости создаются сразу при инициализации контейнера, но он вполне может использовать отложенную (Lazy) инициализацию.

Этот пример должен сформировать у вас представление о том, как контейнер внедрения управляет жизненными циклами. Поскольку контейнер внедрения представляет собой многократно используемую библиотеку, мы не можем модифицировать его исходный код всякий раз, когда захотим переконфигурировать жизненный стиль зависимостей. Далее будет показано, как конфигурируются жизненные стили для демонстрационного контейнера.

Управление жизненным стилем с помощью Autofac

Иногда в этой книге я делаю переход от чистого «внедрения для бедных» к примеру того, как можно достичь результата, используя контейнер-образец. Каждый контейнер имеет свой специальный API для реализации многих различных возможностей; но хотя их детали отличаются друг от друга, принципы остаются одинаковыми. Это справедливо и для управления временем жизни.

ПРИМЕЧАНИЕ

Даже термин «управление временем жизни» (Lifetime Management) используется не везде. Например, в Autofac[1] это называется областью видимости экземпляра (Instance Scope).

В этом разделе мы кратко рассмотрим технику конфигурирования времени жизни с помощью `Autofac`.

ПРИМЕЧАНИЕ

У меня нет особых причин предпочесть для этого примера Autofac перед прочими контейнерами. Я мог бы использовать любой другой.

Листинг 8.3 показывает, как сконфигурировать `Autofac` для использования с Transient-зависимостями, пример эквивалентен коду из листинга 7.9.

Листинг 8.3. Конфигурирование Autofac с Transient-зависимостями

```
var builder = new ContainerBuilder();
builder.RegisterType<ContractMapper>()
    .As<IContractMapper>();
builder.Register((c, p) =>
```

[1] http://code.google.com/p/autofac/.

```
    new SqlProductRepository(
        ConfigurationManager
        .ConnectionStrings["CommerceObjectContext"]
        .ConnectionString))
    .As<ProductRepository>();
builder.RegisterType<ProductManagementService>()
    .As<IProductManagementService>();
var container = builder.Build();
```

Одна особенность Autofac заключается в том, что вы конфигурируете не сам контейнер, а ContainerBuilder. Этот построитель и используется для создания контейнера после того, как конфигурирование будет выполнено.

Простейшая форма регистрации — когда требуется только определить соответствие между абстракцией и конкретным типом, например соответствие между IContractMapper и ContractMapper. Обратите внимание, что конкретный тип указывается перед абстракцией, то есть порядок является обратным по отношению к используемому в большинстве других контейнеров.

Хотя Autofac поддерживает автоматическое подключение, как и другие контейнеры, внедрение примитивных типов, таких как строки, всегда является особым случаем, поскольку такие типы могут потенциально оказаться набором различных строк. В нашем случае имеется только одна строка соединения, но ее по-прежнему нужно передать в регистрируемый SqlProductRepository. Вы можете сделать это, используя лямбда-выражение[1], которое будет выполняться, когда поступит запрос на создание экземпляра типа ProductRepository.

Наличие лямбда-выражений — это своеобразный «конек» Autofac. Большинство контейнеров внедрения не позволяют работать с лямбда-выражениями, и Autofac был практически первым, предоставившим такую возможность. Вы можете использовать лямбда-выражение, чтобы определить, как будет создаваться класс SqlProductRepository, и вы получите параметр конструктора connectionString из конфигурации приложения.

Преимущество использования лямбда-выражений заключается в том, что они типизированы, поэтому во время компиляции происходит проверка конструкции SqlProductRepository. Недостатком является то, что вы не получаете автоматического подключения, поэтому, если только вам не требуется явно указывать параметр конструктора, предпочтительным будет простое соотнесение с помощью метода RegisterType. Это тот способ, которым соотносятся IProductManagementService и ProductManagementService, поэтому здесь могут использоваться преимущества от автоматического подключения.

Теперь экземпляр контейнера может быть использован для создания новых экземпляров IProductManagementService наподобие следующего:

```
var service = container.Resolve<IProductManagementService>();
```

Но подождите, а где же здесь управление временем жизни? В большинстве контейнеров внедрения определен жизненный стиль по умолчанию. Для Autofac такой

[1] Технически это не лямбда-выражение, а скорее блок кода. Большинство .NET-разработчиков знают такие кодовые конструкции, как лямбда-выражения, так что я предпочел более известный термин более правильному.

стиль называется Per Dependency (На каждую зависимость), что соответствует стилю Transient (Кратковременный). Поскольку это является умолчанием, вам не нужно указывать стиль, но если вам очень хочется, то это можно сделать следующим образом:

```
builder.RegisterType<ContractMapper>()
    .As<IContractMapper>()
    .InstancePerDependency();
```

Заметьте, что используется гибкий интерфейс регистрации для указания области видимости экземпляра (это используемый в Autofac термин для обозначения жизненного стиля) с методом InstancePerDependency.

Есть еще область видимости экземпляра, называемая Single (Одиночная) и соответствующая жизненному стилю Singleton. Зная это, вы можете написать для Autofac эквивалент листинга 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 жизненный стиль Singleton, поэтому вы определяете его, вызывая метод SingleInstance. Когда дело доходит до SqlProductRepository, процесс немного усложняется, поскольку экземпляр SqlProductRepository должен быть Transient (кратковременным), но внедренная строка состояния должна быть Singleton. Вы можете осуществить это, выделяя connectionString из конфигурации приложения (не показано, но похоже на приведенные ранее примеры), и использовать эту внешнюю переменную внутри замыкания для указания конструктора. Поскольку connectionString представляет собой внешнюю переменную, она остается неизменной при множестве вызовов конструктора. Заметьте, что вы неявно сделали и SqlProductRepository, и ProductManagementService кратковременными (Transient), так как не указывали для них жизненный стиль.

Хотя этот пример показывает лишь то, как можно определять жизненные стили на примере Autofac, прочие контейнеры внедрения имеют похожие API, предназначенные для тех же целей.

Наличие точной настройки жизненного стиля для каждой зависимости важно не только из соображений производительности, но и с функциональной точки зрения. Например, паттерн проектирования (Mediator) полагается на «разделяемого директора» (Shared Director), через которого взаимодействуют различные компоненты. Этот подход работает, только если «Посредник» совместно используется между участвующими элементами.

Итак, инверсия управления подразумевает, что потребители не могут управлять временем жизни применяемых ими зависимостей, так как они не управляют созданием объектов. Поскольку .NET использует сборку мусора, они еще и не могут явно удалить объекты.

Остается без ответа вопрос: как обстоит дело с одноразовыми зависимостями? Обратим наше внимание на эту деликатную тему.

8.2. Использование одноразовых зависимостей

Хотя .NET является управляемой платформой со сборкой мусора, она может работать с неуправляемым кодом. Когда такое случается, код .NET взаимодействует с неуправляемой памятью, которая не обрабатывается сборщиком мусора. Чтобы предотвратить утечки памяти, нужно иметь механизм, который мог бы детерминированно освобождать неуправляемую память. Это является основным назначением интерфейса `IDisposable`[1].

Похоже, некоторые реализации зависимостей будут содержать неуправляемые ресурсы. Например, соединения ADO.NET являются одноразовыми, так как они стремятся использовать неуправляемую память. Поэтому реализации, основанные на базах данных, например соответствующие репозитории, будут одноразовыми как таковые.

Как нужно моделировать одноразовые зависимости? Могут ли абстракции быть одноразовыми? Рассмотрим следующий код:

```
public interface IMyDependency : IDisposable { }
```

Технически такое возможно, но нежелательно, поскольку является дурно пахнущим дизайном, содержащим протекающую абстракцию:

> *...интерфейс [...] не может быть одноразовым. Не существует способа, которым кто-либо, определяющий интерфейс, мог бы предусмотреть все его возможные реализации — всегда может быть создана одноразовая реализация практически любого интерфейса.*
>
> *Николас Блумхард, на дискуссионном форуме Common Context Adapters*[2]

Если вы очень хотите добавить `IDisposable` в свой интерфейс, это может объясняться тем, что у вас в уме уже имеется схема его конкретной реализации. Но вы не должны позволять такому знанию влиять на дизайн интерфейса. Если вы сделаете это, другим классам будет сложнее реализовать его, кроме того, в абстракцию будет добавлена неопределенность. Кто будет отвечать за уничтожение одноразовой зависимости? Должен ли это быть потребитель?

8.2.1. Использование одноразовых зависимостей

Предположим, мы имеем одноразовую абстракцию типа абстрактного класса `OrderRepository`:

```
public abstract class OrderRepository : IDisposable
```

---

[1] По моему мнению, основополагающей статьей об IDisposable является следующая работа: *Shawn Farkas* CLR Inside Out: Digging into IDisposable. — MSDN Magazine, July 2007: http://msdn.microsoft.com/en-us/magazine/cc163392.aspx.

[2] http://cca.codeplex.com/discussions/82987?ProjectName=cca.

Как такую зависимость должен использовать класс `OrderService`? Большинство руководств по проектированию (включая FxCop и встроенный в Visual Studio инструмент Code Analysis) четко указывают, что если класс определяет одноразовый ресурс как своего члена, он сам должен реализовать `IDisposable` и уничтожать этот ресурс примерно таким способом:

```
protected virtual void Dispose(bool disposing)
{
   if (disposing)
   {
      this.repository.Dispose();
   }
}
```

Но эта идея не выдерживает критики, поскольку участник репозитория был сначала внедрен, и он может быть разделен между другими потребителями, что показано на рис. 8.6. Один и тот же экземпляр `SqlOrderRepository` внедрен как в `OrderService`, так и в `SupplierReorderPolicy`. Эти два экземпляра совместно используют одну зависимость. Если `OrderService` уничтожает свой внедренный `OrderRepository`, он тем самым уничтожает и зависимость, используемую `SupplierReorderPolicy`. В результате будет сгенерирована исключительная ситуация (Exception) при попытке `SupplierReorderPolicy` обратиться к зависимости.

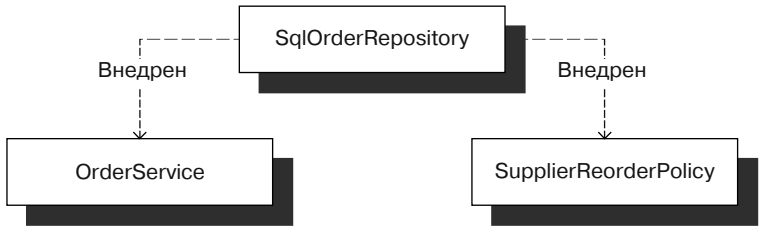


Рис. 8.6. Один и тот же экземпляр SqlOrderRepository внедрен как в OrderService, так и в SupplierReorderPolicy

Более целесообразно будет не уничтожать внедренный репозиторий, но это будет означать, что мы совершенно игнорируем тот факт, что абстракция является одноразовой. Другими словами, объявление абстракции наследницей `IDisposable` не дает никаких преимуществ.

Существуют сценарии, в которых требуется сигнализировать о начале и окончании краткосрочной области видимости, и `IDisposable` иногда используется в этой цели. Прежде чем мы разберемся, как компоновщик будет управлять временем жизни одноразовой зависимости, мы должны решить, что делать с такими короткоживущими элементами.

Создание эфемерных (короткоживущих) элементов

Многие API в .NET BCL используют `IDisposable` для сигнализации о том, что особая область видимости закончилась. Одним из известных примеров такого их применения служат WCF-прокси.

WCF-ПРОКСИ И IDISPOSABLE

Все генерируемые автоматически WCF-прокси реализуют `IDisposable`, и важно не забывать выполнять метод `Dispose` (или `Close`) в прокси настолько скоро, насколько это возможно. Многие подключения автоматически создают соединение на сервисе при подтверждении первого запроса, и это соединение задерживается на сервисе, пока не будет явно закрыто или не истечет время задержки.

Если мы забудем отключиться от нашего WCF-прокси по завершении работы, количество соединений будет увеличиваться, пока не будет достигнут предел числа разрешенных параллельных подключений от одного источника. Когда достигается этот предел, генерируется исключительная ситуация. Кроме того, слишком большое количество соединений чрезмерно перегружает сервис, поэтому важно освобождать WCF-прокси при первой же возможности.

Чтобы обеспечить полную техническую корректность, мы не должны вызывать метод `Dispose` для WCF-прокси. Использование метода `Close` дает тот же результат.

Важно помнить, что применение `IDisposable` для таких целей не свидетельствует о наличии протекающей абстракции, поскольку эти типы не всегда являются абстракциями. С другой стороны, некоторые из них абстракциями являются; и как работать с ними в таком случае?

К счастью, после того как объект удален, переиспользовать его уже невозможно. Это означает, что если мы вызовем этот же API снова, нам придется создать новый экземпляр. Например, это хорошо соответствует тому, как мы используем WCF посредник или команды ADO .NET: мы создаем посредник, вызываем его операции и сразу же уничтожаем его после завершения работы. Как мы согласуем этот сценарий с внедрениями зависимостей, если мы считаем одноразовые абстракции протекающими?

Как всегда, нам может помочь скрытие путаных деталей за интерфейсом. Если вернуться к WPF-приложению из раздела 7.4, то там мы спрятали WCF-прокси за интерфейсом `IProductManagementAgent`.

ПРИМЕЧАНИЕ

Наиболее примечательным моментом в листинге 7.10 является интерфейс IProductManagementAgent, но мы не рассматриваем его в подробностях. По существу, такой агент выполняет ту же функцию, что и репозиторий, но много лет назад я взял за привычку называть компоненты доступа к данным «агентами», а не «репозиториями».

Вот как удаляется продукт с точки зрения `MainViewModel`:

```
this.agent.DeleteProduct(productId);
```

Вы вызываете внедренный агент для удаления продукта. `MainViewModel` может безопасно хранить ссылку на агент, потому что интерфейс `IProductManagementAgent` не наследуется из `IDisposable`.

Иная картина получается, если рассмотреть WCF-реализацию этого же интерфейса. Вот реализация метода `DeleteProduct`:

```
public void DeleteProduct(int productId)
{
   using (var channel = this.factory.CreateChannel())
   {
```

```
            channel.DeleteProduct(productId);
        }
    }
```

Класс `WcfProductManagementAgent` не имеет изменяемого состояния, но содержит внедренную абстрактную фабрику, которую можно использовать для создания канала (Channel). Канал — это всего лишь синонимичное название для WCF-посредника и это автоматически генерируемый клиентский интерфейс, который вы получаете, когда создается ссылка на сервис с помощью Visual Studio или `svcutil.exe`. Поскольку этот интерфейс наследуется из `IDisposable`, вы можете обернуть его в инструкцию `using`.

Вы используете канал для удаления продукта. Когда вы покидаете область `using`, канал удаляется.

Но подождите! Разве я не утверждал, что одноразовая абстракция является протекающей? Да, утверждал, но я стараюсь найти золотую середину между прагматизмом и принципиальностью. В данном случае `WcfProductManagementAgent`, абстрактная фабрика `IProductChannelFactory` и `IProductManagementServiceChannel` определены в одной и той же библиотеке, ориентированной на WCF, которая представлена на рис. 8.7. `WcfProductManagementAgent` использует фабрику `IProductChannelFactory` для создания экземпляров канала `IProductManagementServiceChannel`, который является одноразовым. Хотя его и можно считать протекающей абстракцией, он не наносит особого вреда, так как все его потребители и реализации находятся в одной и той же сборке.

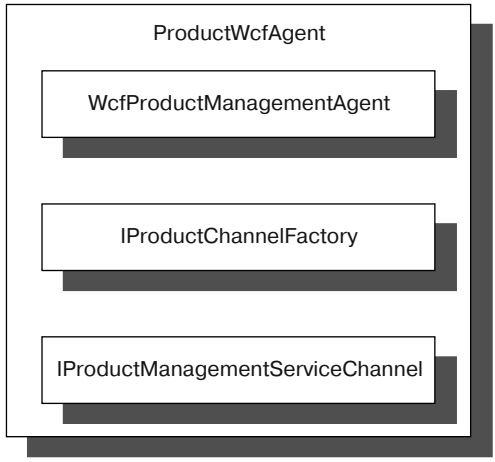


Рис. 8.7. Помимо других типов, библиотека ProductWcfAgent содержит реализацию IProductManagementAgent и его вспомогательных типов

Каждый раз, когда вы вызываете метод класса `WcfProductManagementAgent`, он открывает новый канал и ликвидирует его после использования. Время его жизни очень невелико, по этой причине я назвал такую одноразовую абстракцию «эфемерным ресурсом».

Заметьте, что эфемерный ресурс никогда не внедряется в потребитель. Вместо этого внедряется абстрактная фабрика, которая и используется для управления временем жизни эфемерного ресурса.

Итак, одноразовые абстракции протекают. Иногда приходится допускать такие утечки, чтобы избежать ошибок (в частности, потери WCF-соединений). Но в таком случае мы можем сделать все возможное, чтобы локализовать утечку и чтобы она не распространилась по всему приложению.

Теперь мы рассмотрели способы использования одноразовых зависимостей. Перейдем к тому, как мы можем применять их и управлять ими для потребителей.

8.2.2. Управление одноразовыми зависимостями

Поскольку я так последовательно настаиваю на том, что одноразовые абстракции протекают, я, очевидно, подразумеваю, что абстракции не должны быть одноразовыми. С другой стороны, иногда реализации являются одноразовыми, и если мы неправильно их ликвидируем, то будем иметь утечки ресурсов в своих приложениях. Кто-то *должен* избавиться от них.

СОВЕТ

Всячески старайтесь реализовывать сервисы так, чтобы они не удерживали ссылки на одноразовые сущности (Disposables), но надежнее будет создавать и ликвидировать их по запросу. Это значительно упрощает управление памятью, поскольку сервис может быть уничтожен посредством сборки мусора, как и другие объекты.

Как всегда, ответственность возлагается на компоновщик (такой как контейнер внедрения зависимостей). Он лучше, чем кто бы то ни было, знает, когда он создает одноразовый экземпляр, поэтому он знает и то, что этот экземпляр должен быть уничтожен. Компоновщик может легко сохранить ссылку на одноразовый экземпляр и вызвать его метод `Dispose` в нужное время.

Проблема состоит в определении момента времени, когда экземпляр должен быть уничтожен. Как мы можем узнать, что все потребители закончили работать с ним?

Мы не можем этого знать, если только кто-то не сообщит нам, что это случилось. Часто наш код существует в определенном контексте, у которого четко определено время жизни. Плюс имеются события, сообщающие нам, что та или иная область видимости завершилась. Таблица 8.1 показывает области видимости для технологий, которые рассматривались в главе 7.

Таблица 8.1. Точки входа и выхода для различных фреймворков .NET

Технология	Точка входа	Точка выхода
Консольные приложения	Main*	Main*
ASP.NET MVC	IControllerFactory. CreateController	IControllerFactory.ReleaseController
WCF	IInstanceProvider. GetInstance	IInstanceProvider.ReleaseInstance
WPF	Application.OnStartup	Application.OnExit
ASP.NET	Конструкторы**, Page_Load	IDisposable.Dispose**, Page_Unload
PowerShell	Конструкторы**	IDisposable.Dispose**

* Метод `Main` является одновременно и точкой входа, и точкой выхода, поскольку приложение запускается, когда начинает выполняться метод `Main`, и завершается, когда заканчивается работа метода. Разрешайте зависимости в начале метода `Main` и уничтожайте их в его конце.

** Мы можем разрешать зависимости в конструкторах, и фреймворки ASP.NET и PowerShell позволяют вызывать метод `Dispose`, если мы реализуем `IDisposable`.

Мы можем использовать различные точки выхода, чтобы сообщить компоновщику, что он должен уничтожить все зависимости для данного объекта. После этого компоновщик должен сохранить ссылки на нужные зависимости и решить, какие из них должны быть уничтожены.

Уничтожение зависимостей

Высвобождение (Release) графа объектов — это не то же самое, что уничтожение (Dispose). Это сигнал компоновщику, что корень графа выходит из области видимости, поэтому если сам корень реализует `IDisposable`, он должен быть уничтожен. Но зависимости корня графа могут совместно использоваться другими графами, поэтому компоновщик может принять решение не уничтожать такие зависимости, поскольку он знает, что они нужны другим объектам. Рисунок 8.8 иллюстрирует последовательность событий. Когда компоновщик получает запрос на создание объекта (Resolve), он собирает все ссылки этого объекта. В нашем случае создаваемый объект имеет три зависимости, и две из них являются одноразовыми. Одна из этих одноразовых зависимостей совместно используется с другими потребителями, поэтому для нее не создается новый экземпляр, а используется существующий. Для других зависимостей экземпляры создаются на месте. Когда поступает запрос на высвобождение объекта (Release), компоновщик уничтожает приватную одноразовую зависимость и позволяет неодноразовой зависимости и самому объекту выйти из области видимости и завершить работу, тем самым позволяя сборщику мусора уничтожить их. В созданный объект внедряется только совместно используемая зависимость, и именно потому, что она используется совместно, она не уничтожается вместе с объектом.

Чтобы выполнять высвобождение, компоновщик должен отслеживать все одноразовые зависимости, обслуживаемые им, и знать, какими потребителями они используются. Соответственно, он может уничтожить их, когда последний потребитель их высвобождает.

СОВЕТ

Если вы когда-либо использовали счетчики ссылок (или сталкивались с ошибками из-за плохой реализации), вы знаете, насколько сложно отслеживать закладки на все зависимости и на их потребителей. Это та область, в которой не обойтись без контейнера внедрения, который выполняет для вас всю эту работу. Применяйте стандартный контейнер внедрения, а не разрабатывайте свой код управления временем жизни. Нет никаких сомнений, что реализованная в контейнерах поддержка управления временем жизни протестирована и реализована более тщательно, чем все, что вы можете осуществить за сколь-нибудь обозримый период времени.

Вернемся к примеру WCF-сервиса из подраздела 8.1.2. Как оказалось, в листинге 8.2 имеется ошибка, поскольку, как показывает рис. 8.9, класс `SqlProductRepository` реализует `IDisposable`. Кроме того, он наследуется из абстрактного класса `ProductRepository`, который не реализует `IDisposable`.

Код в листинге 8.2 создает новые экземпляры `SqlProductRepository`, но никогда не высвобождает их. Это приведет к утечкам ресурсов, поэтому устраним эту ошибку в новой версии специализированного контейнера.

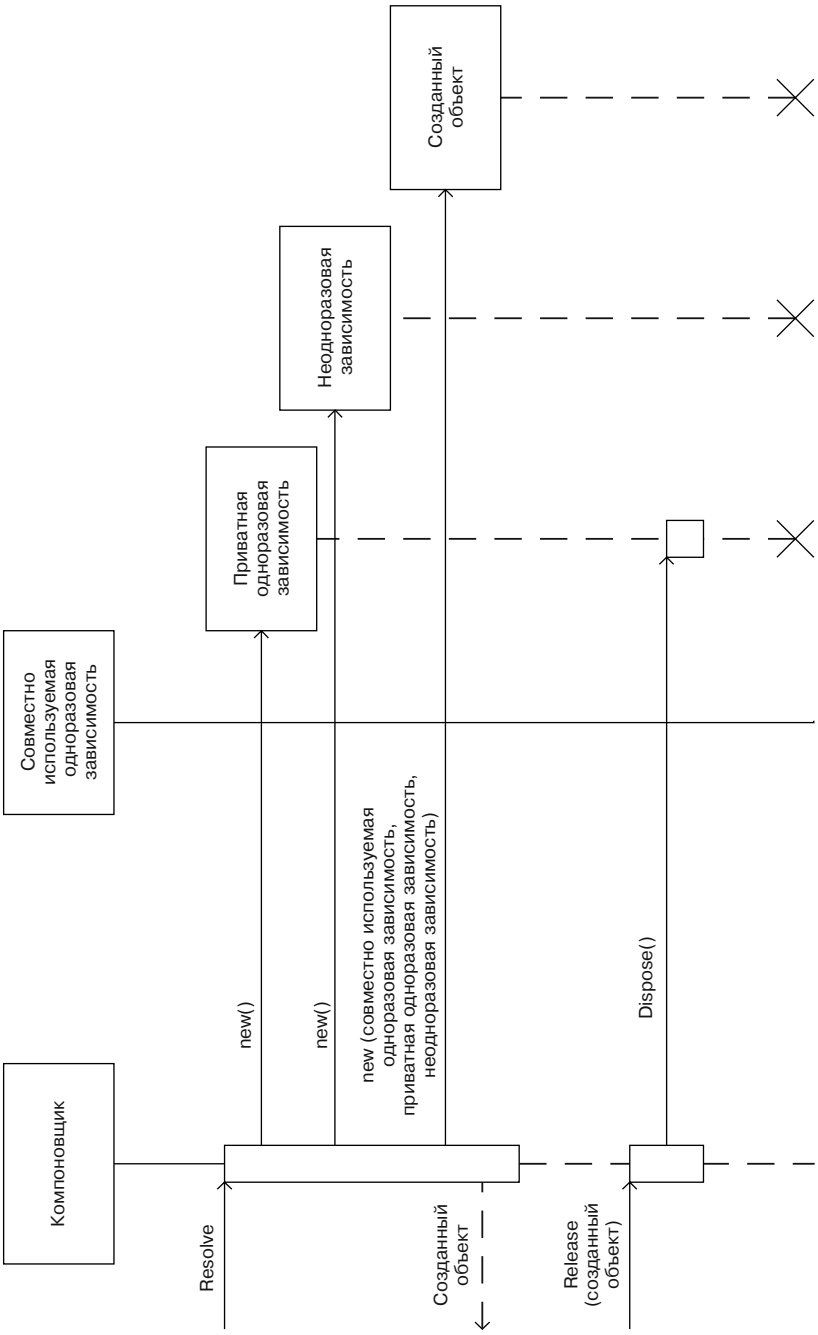


Рис. 8.8. Последовательность событий

Рис. 8.9. SqlProductRepository реализует IDisposable, поскольку он инкапсулирует одноразовый ресурс

Прежде всего запомните, что контейнер должен уметь обслуживать множество параллельных запросов, поэтому он должен ассоциировать каждый экземпляр `SqlProductRepository` с сервисом, создаваемым на основе `IProductManagementService`. Контейнер использует экземпляр `Dictionary<IProductManagementService, SqlProductRepository>` с именем `repositories` для хранения этих ассоциаций. Листинг 8.4 показывает, как контейнер разрешает запросы на создание экземпляров `IProductManagementService`.

Листинг 8.4. Ассоциирование одноразовых зависимостей с создаваемым корнем графа

```
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 создания всех зависимостей. Это похоже на реализацию в листинге 8.2. Но прежде чем возвращать созданный сервис, контейнер должен запомнить ассоциацию между сервисом и репозиторием.

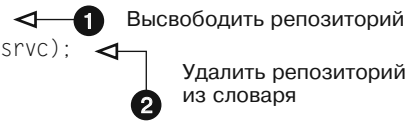
В WCF-приложении имеется лишь один экземпляр контейнера, и поскольку вполне вероятно, что он будет получать параллельные запросы, вы должны блокировать словарь, прежде чем вы добавите в него репозиторий. Добавление элементов в словарь является потокобезопасной операцией, поэтому вы должны выполнять блокировку, чтобы гарантировать сохранность всех репозиториев для последующего использования, даже в параллельных запросах.

Если вернуться к листингу 7.7, видно, что реализация `IInstanceProvider` уже вызывает метод `Release` контейнера. Вы пока еще не реализовывали этот метод, полагая, что эту работу выполнит сборщик мусора; но при наличии одноразовых зависимостей важно учитывать такую удобную возможность для выполнения уборки. Вот возможная реализация.

Листинг 8.5. Высвобождение одноразовых зависимостей

```
public void Release(object instance)
{
    var srvc = instance as IProductManagementService;
```

```
    if (srvc == null)
    {
        return;
    }

    lock (this.syncRoot)
    {
        SqlProductRepository repository;
        if (this.repositories.TryGetValue(srvc, out repository))
        {
            repository.Dispose();
            this.repositories.Remove(srvc);
        }
    }
}
```



Поскольку метод `Release` работает с объектами любых типов, то прежде всего нужно задействовать граничный оператор, чтобы в метод попали только экземпляры `IProductManagementService`.

Параллельные потоки могут вызывать метод `Release` одновременно, поэтому нужно сериализовать доступ к словарю репозиториев. Так мы гарантируем, что параллельные потоки не повредят содержимое словаря. Если репозитории не будут удаляться из библиотеки, могут возникнуть утечки памяти.

Переменная `srvc` выступает в роли ключа для словаря, поэтому ее можно использовать для поиска одноразовых зависимостей. Когда такая зависимость будет найдена, вы можете высвободить ее ❶ и удалить ее из словаря ❷, чтобы гарантировать, что контейнер не сохраняет ее по ошибке.

Примеры, продемонстрированные в листингах 8.4 и 8.5, ориентированы на работу с единственной одноразовой зависимостью: `SqlProductRepository`. Несложно расширить этот код, чтобы можно было обрабатывать любой вид одноразовых зависимостей. Но после того, как это будет сделано, код усложнится. Представьте себе работу со множеством одноразовых зависимостей одного и того же объекта или вложенные одноразовые зависимости, когда некоторые из них являются Singleton (одиночками), а другие — Transient (кратковременными). А ведь мы даже еще не начинали обсуждать более сложные жизненные стили!

СОВЕТ

Сделайте себе одолжение и используйте контейнер внедрения вместо того, чтобы решать все эти проблемы в своем коде. Я показываю специальный код только для того, чтобы разъяснить основные принципы управления временем жизни.

Контейнеры внедрения зависимостей могут работать со сложными комбинациями жизненных стилей, и они обеспечивают возможности (такие как метод `Release`) для явного высвобождения компонентов, когда мы заканчиваем работать с компонентами. Не забывайте использовать эти методы, чтобы избежать утечек памяти, особенно когда одна или более из сконфигурированных зависимостей являются одноразовыми.

Мы в общих чертах обсудили управление временем жизни. Как потребитель мы не можем управлять временем жизни внедренных зависимостей; ответственность

за это возлагается на компоновщик, который может решить, будет ли один экземпляр совместно применяться несколькими потребителями либо мы будем передавать каждому потребителю его собственный приватный экземпляр. Жизненные стили Singleton и Transient являются единственными и самыми известными участниками большого количества жизненных стилей. В оставшейся части главы мы познакомимся с каталогом стратегий жизненного цикла.

8.3. Каталог жизненных стилей

После того как в предыдущем разделе мы изучили основные принципы управления временем жизни, в оставшейся части главы мы рассмотрим распространенные шаблоны жизненных стилей.

ПРИМЕЧАНИЕ

В этом разделе я буду использовать сопоставимые примеры. Но чтобы сфокусировать внимание на существенных вещах, я буду строить поверхностные иерархии и кое-где буду игнорировать существование одноразовых зависимостей, чтобы избежать излишней сложности.

Поскольку вы уже познакомились с Singleton и Transient, мы начнем с них, а затем приступим к другим типам. По мере освоения жизненных стилей мы будем переходить от распространенных стилей к экзотическим, что отражено в табл. 8.2.

Таблица 8.2. Рассматриваемые в этом разделе жизненные стили

Название	**Описание**
Singleton	Единственный экземпляр, всегда разделяется (используется повторно)
Transient	Всегда создаются новые экземпляры
Per graph	В каждом графе используется один экземпляр
Web request context	На каждый веб-запрос подается не более одного экземпляра каждого типа
Pooled	Экземпляры берутся из пула предварительно созданных объектов
Lazy	Энергозатратная зависимость создается и обслуживается в отложенный момент времени
Future	Зависимость становится доступной в будущем

Хотя вы будете редко использовать такие жизненные стили, как Pooled, будет полезно познакомиться с ними, и приведенный список дает вам хорошее представление о диапазоне доступных жизненных стилей. По сравнению с развитыми стилями, Singleton может показаться простеньким, но он является полезной и широко используемой стратегией жизненного цикла.

8.3.1 Singleton

Жизненный стиль Singleton время от времени использовался неявно в этой книге. Его название в одно и то же время является и совершенно ясным, и вводящим в заблуждение. Это связано с тем, что логика его работы похожа на логику шаблона проектирования «Одиночка», но их структура различается.

ПРЕДУПРЕЖДЕНИЕ

Не путайте жизненный стиль Singleton и паттерн проектирования «Одиночка».

В области действия отдельного компоновщика компонент с жизненным стилем Singleton ведет себя подобно паттерну проектирования Singleton, но структурно ситуация отличается. Всякий раз, когда потребитель запрашивает компонент, подается один и тот же экземпляр.

Но на этом сходство заканчивается. Потребитель не может получить через статический член доступ к зависимости, находящейся в области видимости Singleton, и если мы запросим экземпляры у двух разных компоновщиков, мы получим два разных экземпляра.

СОВЕТ

Используйте жизненный стиль Singleton везде и всегда, где только возможно.

Поскольку используется только один экземпляр, жизненный стиль Singleton обычно потребляет минимальный объем памяти. Единственный случай, когда это не выполняется, — при редком использовании экземпляра. Но данный сценарий потребляет неоправданно большое количество памяти. В таких случаях лучшим вариантом конфигурации будет жизненный стиль Lazy (Отложенный) с поддержкой Transient (но я потратил много времени, пытаясь подобрать пример обоснованного использования такой конфигурации).

Когда применяется

Используйте стиль Singleton где только возможно. Главная причина отказа от его применения — потоковая небезопасность компонента. Поскольку Singleton-экземпляр может совместно использоваться между большим количеством потребителей, он должен быть способен обрабатывать параллельный доступ.

Все не сохраняющие информацию о состоянии (Stateless) сервисы по определению являются потокобезопасными, как, например, неизменяемые типы и классы, специально разработанные как потокобезопасные. В этом случае нет оснований не конфигурировать их как Singleton.

Не считая аргументов в пользу эффективности, необходимо сказать, что некоторые зависимости могут использоваться только несколькими элементами совместно. Например, это относится к реализациям паттерна проектирования «Прерыватель цепи» (Circuit Breaker), а также к кэшам памяти. В этих случаях важно, что реализации являются потокобезопасными.

Подробнее рассмотрим расположенный в памяти репозиторий.

Пример: применение потокобезопасного находящегося в памяти репозитория

Еще раз рассмотрим реализацию `ICommerceServiceContainer`, как мы уже делали в подразделах 7.3.2, 8.1.2 и 8.2.2. Допустим, мы решили использовать вместо `ProductRepository`, разработанного на базе SQL Server, потокобезопасную, находящуюся в памяти реализацию. Чтобы применение расположенного в памяти хранилища данных имело хоть какой-то смысл, оно должно совместно использо-

ваться всеми запросами, поэтому оно должно быть потокобезопасным, как показано на рис. 8.10.

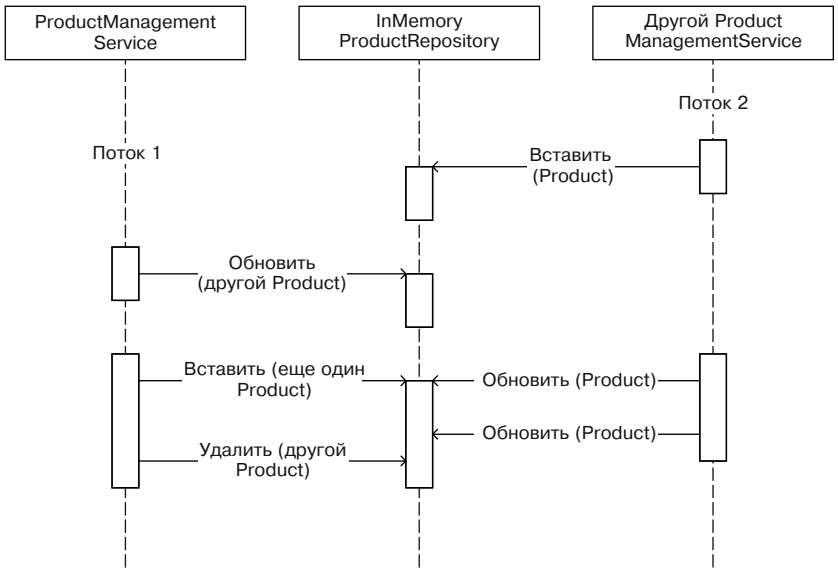


Рис. 8.10. Когда многие экземпляры ProductManagementService выполняющиеся в разных потоках, совместно используют определенный ресурс, такой как расположенный в памяти ProductRepository, мы должны гарантировать, что этот ресурс является потокобезопасным

Вместо того чтобы явно реализовывать такой репозиторий, как Singleton, мы можем использовать конкретный класс и правильно определить его область действия. Единственное требование — он должен быть потокобезопасным.

Листинг 8.6 показывает, как контейнер может возвращать новые экземпляры каждый раз, когда у него запрашивается `IProductManagementService`. При этом `ProductRepository` совместно используется всеми экземплярами.

Листинг 8.6. Управление Singleton

```
public class SingletonContainer : ICommerceServiceContainer
{
    private readonly ProductRepository repository;
    private readonly IContractMapper mapper;

    public SingletonContainer()
    {
        this.repository =
            new InMemoryProductRepository();
        this.mapper = new ContractMapper();
    }

    public IProductManagementService
        ResolveProductManagementService()
```

(1) Singleton-экземпляры

Создание Singletons

```
        ResolveProductManagementService()
    {
        return new ProductManagementService(
            this.repository, this.mapper);
    }

    public void Release(object instance) { }
}
```

❷ Создать сервис

Ничего не делать

Стиль Singleton реализуется очень просто: на время жизни контейнера сохраняется ссылка ❶ на каждую зависимость. Обратите внимание, что применение ключевого слова `readonly` гарантирует, что эти ссылки нельзя будет случайно изменить. Данной дополнительной меры не требуется с точки зрения реализации стиля Singleton, но так повышается степень безопасности, а дописать нужно всего одно слово.

Всякий раз, когда у контейнера запрашивается экземпляр `IProductManagementService`, он создает Transient-экземпляр сервиса с инжектированными в него Singleton-зависимостями ❷. В данном примере такими зависимостями будут `repository` и `mapper`, обе они являются Singleton. Если потребуется, можно применять любые другие жизненные стили.

Жизненный стиль Singleton — один из самых простых для реализации. Все, что для этого требуется, — хранить ссылку на объект и предоставлять один и тот же объект по каждому запросу. Экземпляр этот существует, пока существует экземпляр компоновщика. Когда компоновщик завершает свою работу, он должен уничтожить этот объект, если его тип является одноразовым.

Еще один простой в реализации жизненный стиль называется Transient.

8.3.2. Transient

Жизненный стиль Transient (Кратковременный) предполагает возврат нового экземпляра при каждом запросе. Если только возвращаемый экземпляр не реализует `IDisposable`, то не требуется ничего запоминать. И наоборот, если экземпляр реализует `IDisposable`, компоновщик должен сохранять ссылку на него и явно уничтожать при поступлении запроса на очистку соответствующего графа объекта.

Неважно, что в настольных и подобных им приложениях мы стремимся создавать иерархию объектов лишь однажды: при запуске приложения. Даже для Transient-компонентов это означает, что будет создано всего несколько экземпляров и они могут существовать в течение длительного времени. В вырожденном случае, когда для зависимости имеется только один потребитель, окончательный результат создания графа чистых Transient-компонентов будет эквивалентен графу чистых Singleton-компонентов или смеси этих двух жизненных стилей. Это происходит потому, что граф создается лишь однажды, поэтому различия в поведении никогда не проявятся.

Когда применяется

Жизненный стиль Transient является самым безопасным, но и наименее эффективных из всех стилей, потому что он приводит к созданию множества экземпляров,

впоследствии требующих сборки мусора, хотя, вероятно, хватило бы и одного. Но если вы сомневаетесь относительно потоковой безопасности самих компонентов, стиль Transient обеспечит вам требуемую безопасность на уровне приложения, поскольку каждый потребитель получает свой собственный экземпляр зависимости.

Во многих случаях можно безопасно сменить стиль Transient на другой, связанный с контекстом, такой как Web Request Context. В таком стиле доступ к зависимости еще гарантированно и сериализируется, но это зависит от среды времени исполнения (применение Web Request Contexts не имеет смысла в настольных приложениях).

Пример: создание множества репозиториев

Ранее в этой главе вы видели различные примеры применения жизненного стиля Transient. В листинге 8.2 репозиторий `repository` создается и внедряется на месте в методе создания, и контейнер не сохраняет ссылку на него. В листингах 8.4 и 8.5 мы рассматривали, как используется одноразовый Transient-компонент.

Можно заметить, что в приведенных здесь примерах `mapper` (сопоставитель) остался Singleton, как и раньше. Этот сервис не хранит состояние (`stateless`), поэтому нет необходимости создавать новый экземпляр для каждого созданного `ProductManagementService`. Примечательно, что можно смешивать зависимости, имеющие разные жизненные стили.

При использовании Transient, когда много компонентов требует одну и ту же зависимость, каждый из них получает независимый экземпляр. Листинг 8.7 демонстрирует разрешение метода в контроллере ASP.NET MVC.

Листинг 8.7. Разрешение Transient DiscountRepositorys

```
public IController ResolveHomeController()
{
    var connStr = ConfigurationManager
        .ConnectionStrings["CommerceObjectContext"]
        .ConnectionString;

    var discountCampaign =
        new DiscountCampaign(
            new SqlDiscountRepository(connStr));
    var discountPolicy =
        new RepositoryBasketDiscountPolicy(
            new SqlDiscountRepository(connStr));

    return new HomeController(
        discountCampaign, discountPolicy);
}
```

❶ Новый экземпляр SqlDiscountRepository

❷ Другой экземпляр SqlDiscountRepository

Оба класса `DiscountCampaign` и `RepositoryBasketDiscountPolicy` требуют зависимости `DiscountRepository`. Если `DiscountRepository` является Transient, каждый потребитель получает свой собственный приватный экземпляр, так что `DiscountCampaign` получает один экземпляр ❶, а `RepositoryBasketDiscountPolicy` — другой ❷.

Жизненный стиль Transient определяет, что каждый потребитель получает приватный экземпляр зависимости, даже если несколько потребителей в одном и том же графе объектов получают одну и ту же зависимость (как показано в листинге 8.7). Если несколько потребителей совместно используют одну и ту же зависимость, такой подход оказывается неэффективным; но если реализация не является потокобезопасной, более эффективный стиль Singleton применять нельзя. В таких случаях самым подходящим вариантом может оказаться стиль Per Graph.

8.3.3. Per Graph

Singleton является наиболее эффективным жизненным стилем, а Transient — наиболее безопасным, а можно ли создать стиль, который объединял бы достоинства их обоих? Хотя этого нельзя сделать в полной мере, в ряде случаев имеет смысл разделять один экземпляр в пределах одного создаваемого графа. Мы можем считать такую схему разновидностью одиночки (Singleton) с локальной областью видимости. Можно получить экземпляр, совместно используемый в пределах одного графа, и не разделять его с другими графами.

Каждый раз, когда мы создаем граф объекта, мы создаем только один экземпляр каждой зависимости. Если эту зависимость использует несколько потребителей, они разделяют один и тот же экземпляр; но когда мы создаем новый граф объекта, мы создаем новый экземпляр.

Когда применяется

Жизненный стиль Per Graph (На граф) применяется в основном там, где также может использоваться и Transient. Обычно мы предполагаем, что поток, создающий граф объекта, является единственным потребителем этого графа. Даже когда зависимость в запросе не является потокобезопасной, можно использовать жизненный стиль Per Graph, поскольку совместно применяемый экземпляр разделяется только потребителями, запущенными в том же потоке.

В тех немногих случаях, когда один или более потребителей запускают новые потоки и используют зависимость из этих потоков, Transient остается самым безопасным стилем, но это должно быть редким явлением. Могут встречаться также случаи, когда зависимость представляет собой изменяемый ресурс и каждому потребителю должно быть установлено свое приватное значение. В таком случае корректным жизненным стилем будет Transient, который гарантирует, что экземпляры никогда не будут использоваться совместно.

По сравнению с Transient, стиль Per Graph не добавляет никаких дополнительных издержек и часто может служить заменой для Transient. Но хотя издержки и отсутствуют, он не гарантирует и каких-либо дополнительных преимуществ. Мы получаем некоторый выигрыш в эффективности, только если один граф объекта содержит много потребителей одной и той же зависимости. В этом случае мы можем совместно использовать экземпляр между этими потребителями; но если разделяемые зависимости отсутствуют, то нечего будет разделять и никаких преимуществ не будет.

ПРИМЕЧАНИЕ

Per Graph оказывается лучше Transient в большинстве случаев, но не многие контейнеры включают его реализацию.

Когда реализация является потокобезопасной, жизненный стиль Singleton остается более эффективным вариантом.

Пример: совместное использование репозитория в графе

В листинге 8.7 было продемонстрировано, как каждый потребитель получает собственный приватный экземпляр `SqlDiscountRepository`. Этот класс не является потокобезопасным, поэтому его нельзя конфигурировать как Singleton. Но вы не планируете предоставлять доступ к отдельным экземплярам `HomeController` из множества потоков, поэтому экземпляр `SqlDiscountRepository` может быть безопасно разделен между обоими потребителями. Листинг 8.8 показывает, как создавать единственный экземпляр стиля Per Graph в методе `ResolveHomeController`.

Листинг 8.8. Разрешение единственного репозитория на граф

```
public IController ResolveHomeController()
{
    var connStr = ConfigurationManager
        .ConnectionStrings["CommerceObjectContext"]
        .ConnectionString;
    var repository =
        new SqlDiscountRepository(connStr);

    var discountCampaign =
        new DiscountCampaign(repository);
    var discountPolicy =
        new RepositoryBasketDiscountPolicy(repository);
    return new HomeController(discountCampaign, discountPolicy);
}
```

❶ Разделяемый экземпляр SqlDiscountRepository

❷ Внедрение разделяемого экземпляра

Вместо того чтобы создавать отдельные экземпляры для каждого потребителя, вы создаете один экземпляр, который совместно используете между всеми потребителями ❶. Этот единственный экземпляр внедряется как в `DiscountCampaign`, так и в `RepositoryBasketDiscountPolicy` ❷. Заметьте, что, в отличие от Singleton, где совместно используемый экземпляр — приватным членом контейнера, экземпляр `repository` является локальным в методе `ResolveHomeController`; при следующем вызове метода будет создан новый экземпляр, который и будет разделяться между двумя потребителями.

Жизненный стиль Per Graph является хорошей альтернативой для Transient в случаях, когда Singleton не используется лишь потому, что реализация не является потокобезопасной. Хотя Per Graph предоставляет вполне приемлемое решение по совместному применению зависимостей в четко определенной области видимости, имеются и другие, более специализированные альтернативы.

8.3.4. Web Request Context (Контекст веб-запроса)

Как пользователи веб-приложения, мы ожидаем отклика так скоро, как только это возможно, даже если другие пользователи работают с приложением в это же время. Мы не хотим, чтобы наш запрос был поставлен в очередь вместе с запросами всех остальных пользователей. Время ожидания может быть слишком большим, если в очереди перед нами окажется много других запросов

Чтобы решить эту проблему, веб-приложения обрабатывают запросы параллельно. Инфраструктура .NET защищает нас, обеспечивая выполнение каждого запроса в собственном контексте и со своими экземплярами контроллеров (Controller, если вы используете ASP.NET MVC) или страниц (Page, если применяете ASP.NET Web Forms).

При параллельной обработке зависимости, которые не являются потокобезопасными, не могут использоваться как Singleton. С другой стороны, применение их как Transient может оказаться неэффективным или даже совершенно проблематичным, если нам требуется совместно использовать зависимость между разными потребителями в одном и том же запросе.

Хотя среда ASP.NET не гарантирует, что один запрос выполняется строго в одном потоке, она гарантирует, что код выполняется сериализированным образом. Из этого следует, что если вы можете совместно использовать зависимость только в пределах одного запроса, то потоковая безопасность не станет проблемой.

Рисунок 8.11 показывает, как работает жизненный стиль Web Request Context Зависимости ведут себя как Singleton в пределах одного запроса, но не разделяются между запросами. Каждый запрос содержит свой собственный набор ассоциированных с ним зависимостей. Экземпляр `DiscountRepository` разделяется между `BasketDiscountPolicy` и `DiscountCampaign`, но только в запросе `Request 1`. Запрос `Request 2` использует такую же конфигурацию, но экземпляры ограничены уже его пределами.

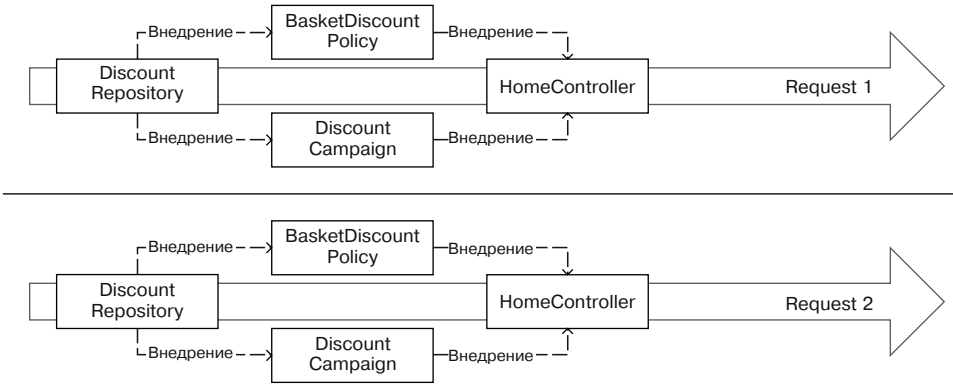


Рис. 8.11. Жизненный стиль Web Request Context означает, что мы создаем не более одного экземпляра зависимости на каждый запрос

Все одноразовые зависимости должны быть уничтожены при завершении запроса.

Когда применяется

Жизненный стиль Web Request Context очевидно целесообразен только в веб-приложениях. Но даже в веб-приложениях он может использоваться только в запросах. Хотя запросы имеют огромное значение в веб-приложениях, стоит отметить, что если мы запустим фоновый поток для асинхронной обработки, этот жизненный стиль будет неприменим в нем, поскольку фоновый поток не синхронизирован с веб-запросом.

Стиль Web Request Context более предпочтителен, чем Transient, но стиль Singleton все равно оказывается более эффективным. Используйте Web Request Context только в ситуациях, когда Singleton не работает.

ПРИМЕЧАНИЕ

Если вы стараетесь создавать только один граф объектов на веб-запрос, то стили Web Request Context и Per Graph оказываются функционально эквивалентными.

СОВЕТ

Если вам понадобится скомпоновать в веб-запрос экземпляр ObjectContext из фреймворка Entity, стиль Web Request Context подойдет лучше всего. Экземпляры ObjectContext не являются потокобезопасными, но на каждый веб-запрос должен существовать только один ObjectContext.

Не все контейнеры внедрения зависимостей поддерживают этот жизненный стиль, поэтому, очевидно, мы можем использовать его, только если он доступен.

СОВЕТ

Некоторые контейнеры внедрения позволяют создавать собственные расширения для жизненных стилей, что может оказаться вариантом, если ваш контейнер не поддерживает стиль Web Request Context. Но это может оказаться сложной задачей.

Как и в случае с другими жизненными стилями, их можно смешивать, поэтому, например, одни зависимости могут быть сконфигурированы как Singleton, а другие сделаны совместно используемыми в пределах веб-запроса.

Пример: компоновка HomeController с совместно используемым в запросе репозиторием

В этом примере вы увидите, как собирается экземпляр ASP.NET MVC `HomeController` с двумя зависимостями, каждая из которых требует `DiscountRepository`. Эта ситуация представлена на рис. 8.11: `HomeController` требует `BasketDiscountPolicy` и `DiscountCampaign`, и каждая из них требует `DiscountRepository`.

ПРИМЕЧАНИЕ

Код примера в этом разделе более сложен, чем в случае одноразового решения. Я надеюсь, что вам никогда не придется писать специальный код для реализации жизненного цикла Web Request Context, подобного представленному здесь, но хочу показать вам, как он работает. Старайтесь использовать контейнеры зависимостей, поддерживающие этот жизненный стиль.

Вы желаете использовать разделяемый `SqlDiscountRepository`, но поскольку этот класс не является потокобезопасным, вы не можете совместно применять его, как Singleton. Вместо этого вы разделяете его в каждом веб-запросе.

Специализированный контейнер компонует экземпляры HomeController, как показано в листинге 8.9.

Листинг 8.9. Компоновка HomeController

```
public IController ResolveHomeController()
{
    var discountPolicy =
        new RepositoryBasketDiscountPolicy(
            this.ResolveDiscountRepository());

    var campaign = new DiscountCampaign(
        this.ResolveDiscountRepository());

    return new HomeController(
        campaign, discountPolicy);
}
```

1 Делегировать создание репозитория

Вернуть сформированный HomeController

Сейчас вам должен быть понятен механизм работы этого метода. Единственное, что требует комментария, — этап делегирования создания DiscountRepository 1 отдельному методу. Этот метод обеспечивает создание не более одного экземпляра на веб-запрос.

Когда поступает запрос на создание DiscountRepository, контейнер должен проверить, существует ли уже экземпляр, ассоциированный с данным веб-запросом. Если такой экземпляр существует, он и будет возвращен; в противном случае создается новый экземпляр, который ассоциируется с веб-запросом, прежде чем будет возвращен. Как показано в листинге 8.10, в ASP.NET (как в MVC, так и в Web Forms) вы можете использовать текущий HttpContext, чтобы поддерживать такую ассоциацию.

Листинг 8.10. Разрешение Web Request Context зависимости

```
protected virtual DiscountRepository ResolveDiscountRepository()
{
    var repository = HttpContext.Current
        .Items["DiscountRepository"]
        as DiscountRepository;
    if (repository == null)
    {
        var connStr = ConfigurationManager
            .ConnectionStrings["CommerceObjectContext"]
            .ConnectionString;
        repository = new SqlDiscountRepository(connStr);
        HttpContext.Current
            .Items["DiscountRepository"] = repository;
    }

    return repository;
}
```

1 Поиск репозитория в контексте запроса

2 Сохранение репозитория в контексте запроса

Основным моментом в жизненном стиле Web Request Context является повторное использование экземпляров, уже ассоциированных с текущим запросом.

Поэтому первое, что нужно сделать, — проверить, существует ли уже требуемый экземпляр (1). Если да, то нужно вернуть его. Если же экземпляр не найден, он должен быть создан и ассоциирован с текущим веб-запросом (2) до возврата.

При первом вызове метода `ResolveDiscountRepository` экземпляр `repository` создается и ассоциируется с запросом, так что каждый последующий вызов использует этот же экземпляр.

Когда обработка запроса заканчивается, одноразовые зависимости могут остаться в нем, что приведет к утечкам памяти, поэтому должно быть обеспечено освобождение всех зависимостей при окончании обработки запроса. Один из способов сделать это — зарегистрировать специальный `IHttpModule`, который подписывается на событие `EndRequest` для корректного высвобождения всех одноразовых зависимостей. В листинге 8.11 приведен пример реализации.

Листинг 8.11. Высвобождение одноразовых зависимостей с областью видимости Web Request Context

```
public class DiscountRepositoryLifestyleModule : IHttpModule
{
    public void Init(HttpApplication context)
    {
        context.EndRequest += this.OnEndRequest;
    }

    public void Dispose() { }

    private void OnEndRequest(object sender, EventArgs e)
    {
        var repository = HttpContext.Current
            .Items["DiscountRepository"];
        if (repository == null)
        {
            return;
        }

        var disposable = repository as IDisposable;
        if (disposable != null)
        {
            disposable.Dispose();
        }

        HttpContext.Current
            .Items.Remove("DiscountRepository");
    }
}
```

(1) Поиск репозитория в контексте запроса

(2) Уничтожение репозитория

(3) Удаление ассоциации репозитория из контекста запроса

Когда завершается обработка веб-запроса, вы пытаетесь найти репозиторий в контексте запроса (1). Если репозиторий находится, вы можете уничтожить его (2). Независимо от того, является репозиторий одноразовым или нет, вы должны не забыть удалить его ассоциацию (3) из контекста запроса.

Жизненный стиль Web Request Context ассоциирует зависимость с текущим запросом, сохраняя и затем извлекая ассоциацию в `HttpContext.Current`. В приведенном примере показано специализированное решение, но оно может быть обобщено таким образом, что произвольное количество зависимостей множества различных типов может быть связано с контекстом запроса. Это сфера настоящего контейнера внедрения зависимостей.

Разновидность: Session Request Context (Контекст сессии запроса)

В редко используемой разновидности жизненного стиля Web Request Context время жизни зависимости ассоциируется не с отдельным веб-запросом, а с целой сессией. Этот жизненный стиль является весьма экзотическим, и следует быть чрезвычайно осторожным при попытке реализовать его.

Технически реализация этой разновидности может показаться похожей на Web Request Context, наиболее существенное различие заключается в том, что HTTP-запрос имеет четко определенное время жизни, а сессия — нет. Сессия редко заканчивается явно, скорее она устаревает по истечении времени задержки при отсутствии активности со стороны посетителя. Это означает, что все зависимости, зарегистрированные таким образом, остаются «висящими», хотя они и не используются. Все это время они занимают память, что может сильно сказаться на возможностях приложения.

ПРЕДУПРЕЖДЕНИЕ

Используйте стиль Session Request Context, только если он действительно вам нужен. Он может сильно ограничить возможности приложения[1].

СОВЕТ

Если вам нужно связать некоторые зависимости с сессией, лучше всего сконфигурировать ее со стилем Web Request Context и использовать фабрику, которая свяжет каждый экземпляр с ключом соответствующей сессии. Такой подход позволит вам более явно управлять жизненным стилем зависимости, в то же время сохраняя ее связь с сессией.

Еще одна проблема заключается в том, что состояние сессии может сохраняться в хранилище, находящемся вне пределов процесса, например на выделенном сервере сессий или в расположенном на SQL Server хранилище состояний сессий. В таких конфигурациях все данные сессии должны позволять сериализацию, и это относится и к зависимостям. Чтобы разрешить сериализацию, достаточно использовать атрибут `[Serializable]` при объявлении типа, но это нельзя забывать делать.

Резюмируя, я нахожу стиль Session Request Context непривлекательным, и я даже не хочу приводить примеров его использования.

Разновидность: Thread Context (Контекст потока)

Еще одна более используемая разновидность позволяет ассоциировать зависимость с отдельным потоком. Основной принцип тот же: зависимость управляется

[1] Для лучшего понимания, почему объекты с областью видимости в пределах сессии всегда создают проблемы, смотрите работу *Michael T. Nygard* Release It! Design and Deploy Production-Ready Software. — Raleigh, NC: Pragmatic Bookshelf, 2007. — 175.

как Singleton в каждом потоке, в каждом потоке существует свой независимый экземпляр.

Этот подход наиболее употребителен в сценариях, где мы запускаем несколько одинаковых рабочих потоков и используем точку начала каждого потока как корень композиции. Эта ситуация проиллюстрирована на рис. 8.12.

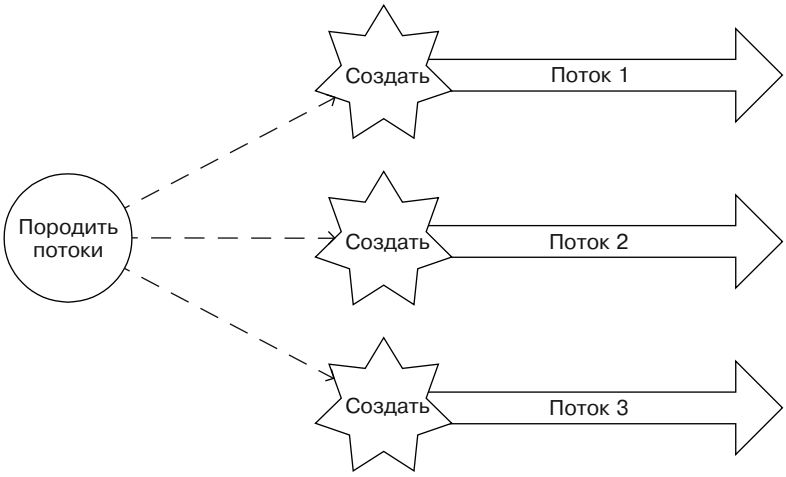


Рис. 8.12. Когда приложение немедленно запускает какое-то количество параллельных задач и создает зависимости в каждом потоке, можно применять стиль THREAD CONTEXT, чтобы гарантировать, что любая зависимость, которая не является потокобезопасной, может совместно использоваться любым количеством потребителей в одном и том же потоке. Каждый поток имеет свои собственные экземпляры

Чтобы реализовать стиль Thread Context, при поступлении запроса следует произвести поиск экземпляра запрошенной зависимости в локальной памяти потока (Thread Local Storage, TLS)[1]. Если экземпляр найден, используется он; в противном случае мы создаем его и сохраняем в TLS.

Несмотря на то что Session Request Context может быть довольно небезопасным, а Thread Context несколько экзотичен, жизненный стиль Web Request Context можно все же считать полезным. Он позволяет разделять зависимости в пределах веб-запроса, не учитывая того, являются ли они потоконезависимыми. Он является золотой серединой между Singleton и Transient.

Стиль Thread Context является более эффективной альтернативой стилю Transient, но может использоваться только в веб-приложениях. Если нам нужно управлять энергозатратными зависимостями в других типах приложений, то следует применять другие методы оптимизации.

8.3.5. Pooled (Пулированный, реализованный как пул)

Иногда создание компонентов оказывается довольно затратной операцией. Распространенное решение в таком случае — создание пула заранее подготовленных

[1] Мы уже использовали TLS в подразделе 4.4.1.

компонентов, легко доступных для последующего использования. Хорошо известный пример — соединения с базами данных, для которых практически всегда создаются пулы. Мы автоматически применяем пулы соединений к базам данных и можем задействовать этот же подход, если у нас имеются специальные компоненты различных типов, создание которых оказывается затратным (необязательно по цене — по времени, затраченным ресурсам и т. п.).

Хотя общая концепция пулированных объектов должна быть вам знакома, в табл. 8.3 перечислены некоторые разновидности их реализации.

Таблица 8.3. Параметры реализации пулов объектов

Параметр	Описание
Подготовка пула	Как осуществляется подготовка пула? Создаются ли все объекты в пуле при его создании или это осуществляется постепенно, по мере поступления запросов? Начальное заполнение всего пула требует, чтобы мы знали, как минимум, начальный размер пула. Это может быть затратной операцией, поскольку назначение пула — сделать затратные объекты легкодоступными. Преимущество такого решения заключается в том, что объекты, создание которых длится достаточно долго, становятся удобны для быстрого доступа. Поскольку загрузку пула можно осуществлять из фонового потока, то появляется возможность приступать к использованию объектов до окончания полной загрузки. Альтернативный подход — создание пустого пула и загрузка его по мере поступления запросов. Это приводит к снижению скорости доступа при начальных загрузках, но поможет поддерживать размер пула соответствующим нагрузке на него (количеству используемых объектов)
Минимальный размер	Проблема подготовки пула может быть решена установкой конфигурируемого начального размера. Если минимальный размер задается в отличное от нуля значение, то пул должен сначала загрузить такое количество объектов, прежде чем он начнет обслуживать их. С другой стороны, при установке минимального размера в ноль, пул может начать обслуживать объекты немедленно, продолжая в то же время загрузку объектов
Максимальный размер	Каков максимальный размер пула?
Поведение на границе	Что произойдет, если мы достигнем максимального размера пула? Разрешено ли пулу расти? Если да, то мы рискуем выйти за пределы памяти. Если нет, то как будут обрабатываться дополнительные запросы на объекты? Один из вариантов поведения — блокировать вызов, пока объект не станет доступным. Но если мы поступим так, мы, как минимум, должны предоставить вызывающей стороне возможность задать задержку. Другой вариант — немедленно генерировать исключительную ситуацию
Очистка пула	Должны ли мы держать пул заполненным до окончания работы приложения или мы должны очищать его, если выяснится, что он имеет избыточную емкость?

В таблице перечислены все важные проблемы, касающиеся пулов объектов. Но, как и в случае жизненного стиля Web Request Context, следует воздерживаться от собственных реализаций пулов объектов и использовать пулы, предоставляемые типовыми контейнерами внедрения зависимостей.

ПРИМЕЧАНИЕ

Не все контейнеры предоставляют жизненный стиль Pooled, поэтому, естественно, применять его можно, только если он поддерживается контейнером.

При использовании стиля Pooled в типовом контейнере не все возможности, перечисленные в табл. 8.3, могут быть доступны. Мы можем рассчитывать только на то, что реализовано.

Когда применяется

Жизненный цикл Pooled может применяться, когда имеются специфические компоненты, которые часто используются, но являются затратными на этапе создания. Даже если компонент требует много ресурсов при создании, следует все же выбирать жизненный стиль Singleton, когда только это возможно. Так мы сможем обходиться одним экземпляром и расплачиваться за его создание лишь однажды.

Отсюда следует, что Pooled применим только в тех случаях, когда компонент в запросе не должен использоваться совместно, что часто означает, что такой вариант не является потокобезопасным. Если речь идет о веб-приложениях, то хорошей альтернативой может быть стиль Web Request Context; в большинстве случаев применение стиля Pooled предполагается не в веб-приложениях.

Обязательным требованием является то, что запрашиваемый компонент должен быть переиспользуемым. Если он имеет естественный жизненный цикл, исключающий повторное использование элементов, пул применять нельзя. Примером является интерфейс `ICommunicationObject` из фреймворка WCF, который имеет четко определенный жизненный цикл. Когда `ICommunicationObject` является либо `Closed` (закрытый) либо `Faulted` (неисправный), он по определению никогда не выйдет из этого состояния. Такой тип объектов не подходит для использования в пуле. Мы должны иметь возможность вернуть объект обратно в пул в первоначальном состоянии.

Пример: повторное использование дорогостоящих репозиториев

Я однажды участвовал в проекте, в котором требовалось взаимодействовать с мейнфреймом из кода .NET. Консультанты, работавшие в проекте в начале, создали неуправляемую COM-библиотеку, которая могла взаимодействовать с некоторой точкой подключения на мейнфрейме, и мы решили обернуть эту библиотеку в управляемую сборку.

COM-библиотека взаимодействовала с мейнфреймом по патентованному протоколу через сетевые сокеты. Чтобы использовать ее, нужно было открыть соединение и выполнить рукопожатие (квитирование). Когда соединение открывалось, мы могли передавать сообщения с приемлемой скоростью, но на открытие соединения уходило ощутимое количество времени.

Посмотрим, как можно создать пул экземпляров `ProductRepository`, которые могут взаимодействовать по такому протоколу. В том проекте, про который я рассказывал, мы дали COM-библиотеке название `Xfer` (очень общая), поэтому давайте создадим пул экземпляров `XferProductRepository`.

ПРИМЕЧАНИЕ

Как и в случае с Web Request Context, я не думаю, что вы станете писать специальные диспетчеры для управления жизненным стилем пулов объектов. Хотя следует использовать подходящий контейнер внедрения зависимостей для управления пулами, я хочу показать вам упрощенный пример, чтобы пояснить принцип их работы.

ВНИМАНИЕ

Следующий далее пример не является потокобезопасным. Я не рассматривал вопросы синхронизации кода, чтобы чрезмерно не усложнять пример, и я оставляю разработку потокобезопасной версии в качестве упражнения для читателей (мне всегда хотелось написать такой пример).

Данный пример — это еще один вариант `ICommerceServiceContainer`, различные версии реализации которого уже встречались в этой главе. Листинг 8.12 демонстрирует основу контейнера.

Листинг 8.12. Базовый код для контейнера пулов

```
public partial class PooledContainer : ICommerceServiceContainer
{
   private readonly IContractMapper mapper;
   private readonly List<XferProductRepository> free;
   private readonly List<XferProductRepository> used;
   public PooledContainer()
   {
      this.mapper = new ContractMapper();
      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;
      }
   }
}
```

Хотя планируется поддерживать пул экземпляров типа `XferProductRepository`, `ContractMapper` все же конфигурируется как Singleton, поскольку этот сервис не сохраняет состояния.

Для реализации пула используются две коллекции: одна содержит свободные репозитории, вторая — репозитории, которые применяются в настоящее время. Когда компоненты создаются и освобождаются, репозитории перемещаются между этими двумя коллекциями.

Свойство `MaxSize` позволяет задавать максимальный размер пула, и свойство `HasExcessCapacity`, по существу, является инкапсулированным вычислением, который применяется для определения — имеется ли еще свободная память в стеке.

В такой версии пула вы будете заполнять пул постепенно по мере поступления запросов, пока не будет достигнут максимум. Как показано в листинге 8.13, при поступлении нового запроса после того, как пул будет заполнен, будет генерироваться исключительная ситуация.

Листинг 8.13. Разрешение репозиториев из пула

```
public IProductManagementService ResolveProductManagementService()
{
    XferProductRepository repository = null;
    if (this.free.Count > 0)
    {
        repository = this.free[0];
        this.used.Add(repository);
        this.free.Remove(repository);
    }
    if (repository != null)
    {
        return this.ResolveWith(repository);
    }

    if (!this.HasExcessCapacity)
    {
        throw new InvalidOperationException(
            "The pool is full.");
    }

    repository = new XferProductRepository();
    this.used.Add(repository);

    return this.ResolveWith(repository);
}

private IProductManagementService ResolveWith(
    ProductRepository repository)
{
    return new ProductManagementService(repository,
        this.mapper);
}
```

❶ Выбрать репозиторий из пула, если есть свободный

❷ Вернуть из пула

❸ Добавить новый репозиторий

Разрешение экземпляра `IProductManagementService` начинается с проверки, имеется ли свободный репозитарий в пуле. Если имеется, то один экземпляр выбирается из коллекции свободных репозиториев и помещается в список используемых репозиториев ❶. Если поиск репозитория завершился успешно, то сервис можно вернуть незамедлительно ❷.

Если свободного репозитория в пуле нет, это может произойти по двум причинам: пул заполнен и все репозитории используются или пул еще не заполнен. Если свободное место в пуле есть и вы не попадаете в блок защиты, контролирующий наполненность пула, вы создаете новый экземпляр репозитория и добавляете его в коллекцию применямых репозиториев ❸ перед тем, как вернуть скомпонованный сервис.

Метод `ResolveProductManagementService` только перемещает репозитории из коллекции свободных в коллекцию используемых репозиториев, поэтому важно ос-

вободить сервисы после использования. Листинг 8.14 показывает, как это может быть выполнено.

Листинг 8.14. Возвращение репозиториев в пул

```
public void Release(object instance)
{
    var service = instance as ProductManagementService;
    if (service == null)
    {
        return;
    }
    var repository = service.Repository
        as XferProductRepository;
    if (repository == null)
    {
        return;
    }
    this.used.Remove(repository);
    this.free.Add(repository);
}
```

Блок защиты

❶ Вернуть репозиторий в пул

Вернуть репозиторий в пул легко: нужно просто переместить его ❶ из коллекции используемых в коллекцию свободных репозиториев.

Заметьте, что, хотя этот пример и выглядит сложным, я еще не решал несколько проблем.

- Пример по определению не является потокобезопасным. Рабочая реализация должна позволять нескольким потокам создавать и высвобождать экземпляры параллельно.
- Поскольку класс `XferProductRepository` инкапсулирует неуправляемый код, он реализует `IDisposable`. Пока вы сохраняете используемые повторно экземпляры, вы не должны освобождать их, но это необходимо сделать, когда контейнер завершает свою работу. Таким образом, контейнер сам по себе должен реализовывать `IDisposable` и уничтожать все репозитории в своем методе `Dispose`.

Поддержка пула объектов — это хорошо известный паттерн проектирования, но он часто инкапсулируется в существующих API. Например, ADO .NET использует пулы соединений, но нам не требуется явно работать с ними. Применение стиля Pooled целесообразно только в том случае, если нам явно требуется оптимизировать доступ к энергозатратным ресурсам.

Жизненный стиль Pooled помогает разрешить ситуацию, в которой нам нужно оптимизировать использование дорогих ресурсов. Это последний из распространенных типов жизненных стилей зависимостей.

8.3.6. Прочие жизненные стили

Ранее в этой главе рассматривались самые распространенные разновидности жизненных стилей, но у вас могут появиться более экзотические потребности, которые

удовлетворяются не полностью. Когда я попадаю в такую ситуацию, я, конечно же, горжусь, что мне попался редкий и драгоценный случай, требующий от меня применения какого-нибудь экзотического инструмента из моего программистского арсенала.

Но очень быстро я осознаю, что такой подход совершенно неверен, и если я просто немного изменю выбранный дизайн, все прекрасно впишется в стандартные шаблоны. Это, конечно, не самый приятный компромисс, но в результате получается более качественный и удобный в поддержке код. Вывод из сказанного таков, что если у вас возникает необходимость в применении редкого или вообще заказного жизненного стиля, вам следует прежде всего серьезно переосмыслить свой дизайн.

Как уже было сказано, некоторые контейнеры внедрения зависимостей включают возможность разработки дополнительных жизненных стилей. Бегло рассмотрим два технически реализуемых, но экзотических жизненных стиля.

В обоих случаях я даю лишь краткий набросок того, как жизненный стиль будет работать. Я не выделяю для этого полномасштабных разделов, поскольку мне трудно подобрать целесообразный сценарий, в котором они могли бы быть использованы.

Lazy (Отложенный, задержанный)

Жизненный стиль Lazy, или Delayed — это виртуальный посредник для более затратной зависимости. Идея заключается в том, что если у нас имеется затратная зависимость, которую мы не хотим использовать часто, мы можем отложить ее создание вплоть до момента, когда она окажется совершенно необходимой. Рисунок 8.13 показывает, как в потребителя может быть внедрен упрощенный дублер вместо реальной, более затратной реализации. Потребителю требуется зависимость типа `IService`, но если он использует эту зависимость лишь в течение короткого периода времени, он может довольно долго существовать, прежде чем ему потребуются услуги от `IService`. Когда он, наконец, вызывает метод `IService.SelectItem()`, `LazyService` использует свою внедренную фабрику `IServiceFactory` для создания экземпляра другого `IService`. Вплоть до этого момента экземпляр `ExpensiveService` не создается. После того как `ExpensiveService` будет создан, все последующие вызовы будут делегироваться ему.

Данный жизненный стиль следует использовать, только если потребитель использует энергозатратную зависимость в течение краткого периода своего жизненного цикла или если достаточно вероятно, что пройдет значительный период времени, прежде чем зависимость будет применена. Если зависимость используется немедленно после ее получения или же задействуется часто, «Отложенный декоратор» (Lazy Decorator) ничего нам не дает, но тратит дополнительные ресурсы.

Если наблюдается последняя ситуация, то затратная зависимость должна регистрироваться как Singleton, так что расплачиваться за ее создание нам придется только один раз. Если таковое невозможно из соображений потоковой безопасности, зачастую бывает лучше решить проблему, используя пулы компонентов. Даже если у нас имеется только один экземпляр, такой единичный пул в сочетании с задержкой фактически обеспечит нам сериализированный доступ к зависимости.

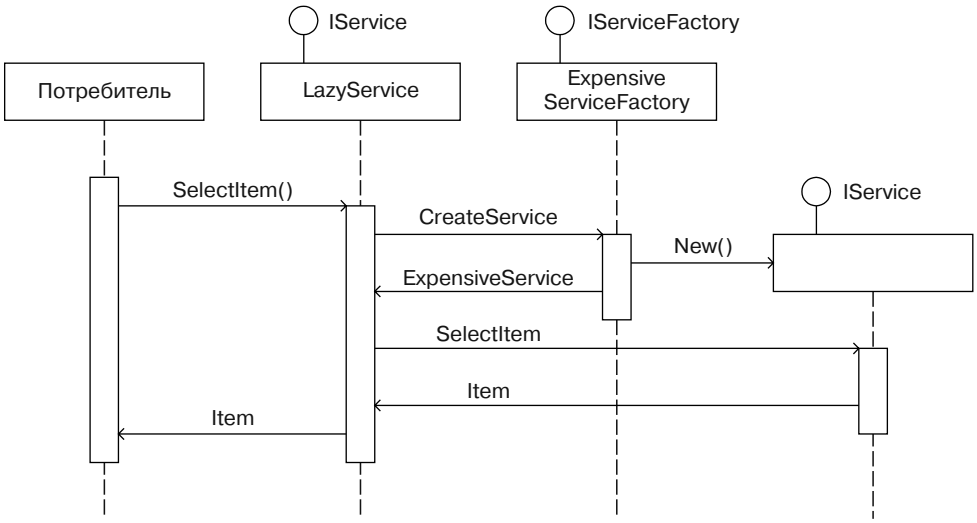


Рис. 8.13. Внедрение упрощенного дублера в потребителя

Жизненный стиль Lazy любопытен технически, но практически не слишком полезен; если это вам интересно, я отсылаю вас к соответствующей литературе[1].

Future (Будущее)

Жизненный стиль Future экзотичен в еще большей степени. Основная его идея заключается в том, что мы можем захотеть использовать зависимость, которая недоступна в данный момент, но которую мы сможем применять, когда она станет доступной.

Наилучший способ реализовать этот жизненный стиль похож на реализацию стиля Lazy: мы можем использовать паттерн «Декоратор», который делегирует некоторую начальную реализацию, пока не станет доступна требуемая зависимость. Рисунок 8.14 иллюстрирует принцип взаимодействия между компонентами. Начальная реализация применяется как дублер, пока `Future Decorator` ожидает требуемую зависимость. Часто такая начальная реализация является реализацией шаблона проектирования «Нулевой объект» Null Object. Потребителю требуется экземпляр, но служба `DesiredService` может быть еще недоступной. В таком случае можно инкапсулировать `NullService` как дублер, который будет использоваться, пока мы будем «ожидать Годо» (от названия пьесы «В ожидании Годо». — *Примеч. пер.*). `FutureService` — это машина состояний, которая периодически проверяет, стал ли `DesiredService` доступным. До тех пор пока он недоступен, у `FutureService Decorator` нет другого выбора, кроме как вернуть реализацию, сформированную сервисом `NullService`. Когда, в конце концов, `DesiredService` становится доступным, все последующие запросы направляются ему.

---

[1] *Mark Seemann* Rebuttal: Constructor over-injection anti-pattern, 2010: http://blog.ploeh.dk/2010/01/20/RebuttalConstructorOverinjectionAntipattern.aspx.

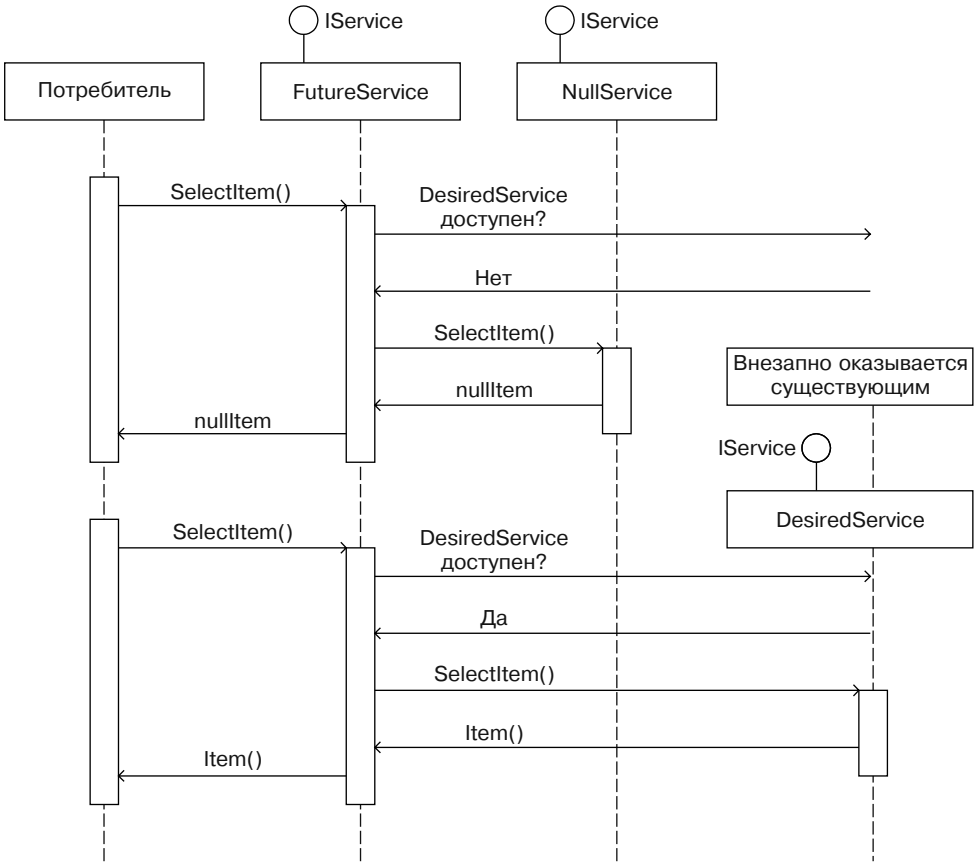


Рис. 8.14. Взаимодействие между компонентами

Я должен признать, что у меня возникли огромные трудности с подбором осмысленного примера, в котором зависимость бы становилась доступной только после того, как мы сформировали весь граф объектов. Ситуация выглядит похожей на ту, когда мы ждем ответа от внешнего ресурса, такого как база данных или веб-сервис. Но помните, что даже если реальный ресурс оказывается недоступным, программная зависимость может существовать. Например, веб-сервис может быть выключен, но WCF-посредник, используемый для коммуникаций с ним, все еще будет доступен.

Для обработки ситуаций, когда внешние ресурсы оказываются недоступными, лучше подходит шаблон «Прерыватель цепи» (Circuit Breaker), который мы будем рассматривать в следующей главе. До тех пор пока кто-нибудь не представит мне имеющий смысл сценарий, я буду полагать стратегию жизненного цикла Future просто техническим курьезом.

Итак, мы рассмотрели некоторые доступные жизненные стили зависимостей, начиная от широко распространенных и заканчивая совершенно экзотическими.

8.4. Резюме

Когда мы применяем инверсию управления к зависимостям, мы инвертируем управление не только над выбором типов, но и над управлением временем жизни. Когда потребитель более не создает собственные экземпляры зависимостей, он не может решить, когда зависимость должна быть создана или будет ли она разделяться с другими потребителями.

Компоновщики решают, позволить ли нескольким потребителям совместно использовать единственный экземпляр или предоставлять каждому потребителю собственный. Могут применяться и более сложные стратегии.

Хотя компоновщики в очень высокой степени контролируют процесс создания объектов, модель управляемой памяти .NET обусловливает то, что они очень мало влияют на время уничтожения объектов. Зависимости могут выйти за границу области видимости, после чего они будут утилизированы сборщиком мусора. Но остается необходимость явно управлять компонентами, реализующими интерфейс `IDisposable`, потому что мы должны гарантировать, что все неуправляемые ресурсы также очищаются, иначе в нашем приложении скоро возникнут утечки памяти.

Так же как мы вызываем метод `Resolve` (он может называться и иначе), мы всегда должны помнить о вызове метода `Release` при выходе графа объектов за границу области видимости (Out of Scope). Это дает компоновщику возможность высвободить любые одноразовые компоненты, которые становятся неиспользуемыми.

Каждый граф зависимости может содержать смесь нескольких различных жизненных стилей, и нам нужно сохранить контроль над тем, являются ли компоненты одноразовыми. Добавьте в эту смесь потоковую безопасность, и станет сложно контролировать все эти вещи. Это область, в которой блистают полномасштабные контейнеры внедрения зависимостей, и это одна из множества причин, по которым мы должны использовать контейнер внедрения, а не «внедрение зависимостей для бедных».

Каждый из известных контейнеров внедрения имеет свой набор жизненных стилей. Некоторые контейнеры поддерживают небольшой набор стилей, другие же позволяют работать с большинством или со всеми из них, но многие еще имеют точки расширений, позволяющие разработчикам реализовывать собственные жизненные стили.

Самым безопасным стилем является Transient, поскольку экземпляры никогда не используются одновременно несколькими элементами. От также наименее эффективный, поскольку предполагается, что много экземпляров одного и того же типа будет находиться в памяти.

Наиболее эффективный жизненный стиль — Singleton, поскольку лишь один экземпляр находится в памяти (на контейнер). Но требуется, чтобы компонент был потокобезопасным, поэтому данный жизненный стиль можно применять не всегда.

Стили Web Request Context и Pooled являются хорошими альтернативами стилям Singleton и Transient, но только в более ограниченных сценариях.

Существуют и более экзотические жизненные стили. Стиль Future на первый взгляд может показаться хорошим механизмом для работы с недоступными ресурсами, но, как вы увидите в следующей главе, эта проблема гораздо лучше решается с помощью перехватов (Interception).

9 Перехват

Меню:

- сквозные аспекты приложения;
- аспектно-ориентированное программирование;
- динамический перехват.

Одна из самых интересных вещей в поварском деле — это способы, которыми мы можем комбинировать различные ингредиенты. Некоторые из них могут быть не очень вкусны сами по себе, но образуют некое целое, которое представляет собой нечто большее, чем простая сумма слагаемых. Часто мы начинаем с простого ингредиента, составляющего основу блюда, и затем дорабатываем и украшаем его, пока у нас не получится восхитительное яство.

Представим себе телячьи отбивные. С голоду их можно есть и сырыми, но, как правило, вы предпочтете их пожарить. Однако если вы просто бросите их на горячую сковороду, результат будет, мягко говоря, скромный. Скорее всего, они просто подгорят.

К счастью, имеется множество способов приобрести необходимый опыт.

- Если жарить на масле, то отбивные не подгорят, их вкус будет мягким.
- Добавление соли улучшит вкус мяса.
- Добавление других специй, например перца, сделает вкус более многогранным.
- Панировка отбивной в смеси соли со специями не только улучшит вкус, но и выведет начальные ингредиенты на новый уровень. В этой точке мы приближаемся к получению Отбивной с большой буквы.

Если сделать на отбивной продольный надрез и добавить туда ветчину, сыр и чеснок перед панировкой, наша отбивная станет шедевром. Теперь у нас будет продукт, достойный Cordon Bleu[1], то есть кулинарно безупречный.

Разница между подгоревшей отбивной и Cordon Bleu огромна, но базовый ингредиент остался тем же самым. Различие является следствием добавления разнообразных улучшений. Имея телячью отбивную, мы можем приукрасить ее, не меняя основного компонента, чтобы получить совершенно разные блюда.

---

[1] Cordon Bleu — самая известная в мире кулинарная школа, имеет 40 филиалов в 20 странах. Центральное отделение находится в Париже. — *Примеч. пер.*

Имея слабое связывание, мы можем выполнить такой же подвиг при разработке программного обеспечения. Когда мы программируем, используя интерфейсы, мы можем преобразовать или улучшить базовую реализацию, декорируя ее в другие реализации этого интерфейса. Вы уже встречали фрагменты такого подхода в подразделе 8.3.6, когда мы использовали его для модификации времени жизни затратной зависимости, декорируя ее в посредник.

Данный подход может быть обобщен, что предоставит нам возможность перехватить вызов, идущий от клиента к сервису. Об этом мы и поговорим в данной главе. Как и при жарке телячьей отбивной, мы начнем с основного ингредиента и добавим дополнительные, чтобы сделать его лучше, при этом не внося изменений в нашу исходную «телятину». Перехват — это одна из самых мощных возможностей, которые мы получаем от слабого связывания. Он позволяет нам легко применять принцип единственной ответственности и разделение ответственности.

В предыдущих главах мы постарались, чтобы сделать наш код по-настоящему слабо связанным. В этой главе мы начнем пожинать плоды этих усилий.

На рис. 9.1 представлена структура этой главы. После того как вы прочтете ее, вы сможете использовать перехват для разработки слабо связанного кода в соответствии с принципами объектно-ориентированного проектирования. В частности, вы получите возможность успешно соблюдать разделение ответственности и использовать сквозные аспекты приложения, обеспечивая хорошее качество своего кода.

Рис. 9.1. Структура данной главы

Понять, как работает перехват, нетрудно, и мы начнем с простого примера, который позволит определить предмет разговора. Чтобы полностью оценить потенциал перехватов, нам понадобится познакомиться с некоторыми связанными с ними концепциями, такими как аспектно-ориентированное программирование и SOLID, и продемонстрировать их взаимосвязь на примерах. В заключение рассмотрим, как контейнеры внедрения зависимостей могут использоваться для обобщения перехватов и облегчения работы с ними.

Концепции, на которых основаны перехваты, являются хорошо известными паттернами проектирования и принципами объектно-ориентированного программирования, и эта глава содержит много примеров. Структура главы совершенно линейна, начинается она с простого вводного примера, затем рассматриваются более сложные концепции и примеры. Заключительную и наиболее мощную концепцию можно быстро объяснить на теоретическом уровне, но, поскольку вам нужно только щелкнуть кнопкой мышью, чтобы перейти к основательному примеру, кульминацией главы будет многостраничный пример, демонстрирующий, как это работает.

Прежде чем мы приступим к рассмотрению деталей, нам придется познакомиться с основами.

9.1. Вводная информация о перехватах

Концепция перехвата проста: мы хотим иметь возможность перехватить вызов, идущий от потребителя к сервису, и выполнить некоторый код до или после того, как будет вызван требуемый сервис. На рис. 9.2 нормальный вызов сервиса со стороны потребителя прерывается неким промежуточным участником, который может выполнить свой собственный код до или после того, как вызов будет передан реальному сервису.

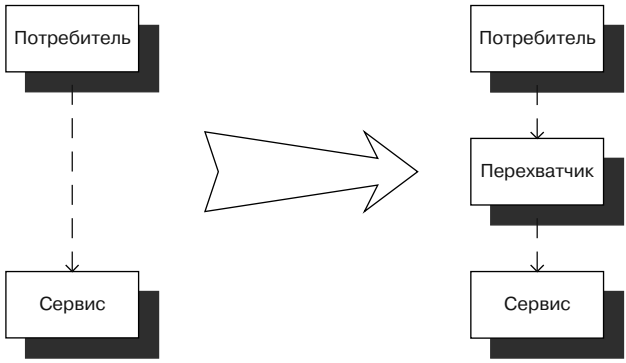


Рис. 9.2. Суть перехвата

В этом разделе мы познакомимся с перехватами и выясним, что, по существу, перехват — это вариант применения шаблона проектирования «Декоратор». Мы рассмотрим данный шаблон в процессе обсуждения, поэтому если вы не знакомы с ним, у вас появится понимание того, как он работает. Сначала изучим простой пример, демонстрирующий этот шаблон, а затем обсудим, как перехват связан с шаблоном «Декоратор».

9.1.1. Пример: реализация аудита

Мы реализуем функцию контроля для `ProductRepository`. Контроль (Auditing) — это распространенный пример сквозного аспекта приложения: он может быть обязателен, но основная функциональность по редактированию и чтению продуктов не должна им затрагиваться. Поскольку принцип единственной ответственности запрещает нам реализовывать контроль непосредственно в `ProductRepository`, наилучшим способом для реализации данной функции будет использование паттерна «Декоратор».

Реализация AuditingProductRepository

Мы можем внедрить перехват в приложение, добавив новый класс `AuditingProductRepository`, декорирующий другой `ProductRepository` и реализующий к тому же функцию контроля. Рисунок 9.3 показывает, как используемые в примере типы связаны друг с другом. `AuditingProductRepository` наследуется из абстрактного класса `ProductRepository` и в то же время используется как обертка для экземпляра любой

другой реализации ProductRepository. Он оставляет основную работу декорированному им ProductRepository, но выполняет контроль в нужных местах. Догадываетесь, где здесь панировка?

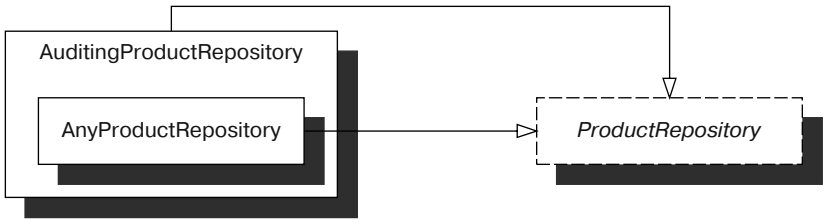


Рис. 9.3. Связь типов друг с другом

Кроме декорированного ProductRepository, классу AuditingProductRepository нужен также свой сервис, который выполнял бы контроль. В листинге 9.1 эту роль выполняет интерфейс IAuditor.

Листинг 9.1. Объявление AuditingProductRepository

```
public partial class AuditingProductRepository :
    ProductRepository
{
    private readonly ProductRepository
        innerRepository;
    private readonly IAuditor auditor;

    public AuditingProductRepository(
        ProductRepository repository,
        IAuditor auditor)
    {
        if (repository == null)
        {
            throw new ArgumentNullException("repository");
        }
        if (auditor == null)
        {
            throw new ArgumentNullException("auditor");
        }

        this.innerRepository = repository;
        this.auditor = auditor;
    }
}
```


AuditingProductRepository наследуется из той же самой абстракции, которую он декорирует ❶. Он использует стандартную внедрение конструктора, чтобы запросить экземпляр типа ProductRepository, который будет декорирован и которому будет затем делегирована базовая реализация. В дополнение к декорированному

репозиторию он требует еще IAuditor ❷, который применяется для контроля операций, реализованных репозиторием.

Листинг 9.2 демонстрирует пример реализации двух методов класса 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 все операции требуют контроля. Как правило, нужен контроль операций Create, Update и Delete, тогда как операции Read (считывание) игнорируются. Так, поскольку метод SelectProduct представляет именно Read-операцию, вызов делегируется декорированному репозиторию, и результат немедленно возвращается.

С другой стороны, метод UpdateProduct должен проходить контроль. Реализация также делегируется декорированному репозиторию, но после возврата результата внедренный IAuditor используется для аудита операции.

Decorator, как и AuditingProductRepository, подобен панировке для телячьих отбивных: он украшает основной ингредиент, не меняя его. Панировка сама по себе не является простой оболочкой, она состоит из собственных ингредиентов. Настоящая панировка состоит из сухарей и специй; так и AuditingProductRepository содержит IAuditor.

Обратите внимание, что внедренный IAuditor сам по себе есть абстракция, то есть его реализацию можно менять независимо от AuditingProductRepository. Все, чем занят класс AuditingProductRepository, — это координация действий декорированного ProductRepository и IAuditor.

Конечно, вы можете создать ту реализацию IAuditor, какая вам будет нужна, но самой распространенной является реализация, основанная на технологии SQL Server. Рассмотрим, как можно подключить все необходимые зависимости, чтобы получить требуемый результат.

Компоновка AuditingProductRepository

Многие приложения используют класс ProductRepository для получения информации о продукте, поскольку веб-сервис CommerceService WCF из подраздела 7.3.2 предоставляет CRUD-операции для продуктов. Вот здесь мы и займемся этими вопросами.

В главе 8 рассматривалось несколько примеров компоновки экземпляров ProductManagementService. Листинги 8.4 и 8.5 содержали наиболее корректную реализацию, но в листинге 9.3 мы игнорируем то обстоятельство, что SqlProductRepository является одноразовым, и сосредоточимся на компоновке декораторов (Decorators).

Листинг 9.3. Компоновка декоратора

```
public IProductManagementService ResolveProductManagementService()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings
        ["CommerceObjectContext"].ConnectionString;

    ProductRepository sqlRepository =
        new SqlProductRepository(connectionString);

    IAuditor sqlAuditor =
        new SqlAuditor(connectionString);

    ProductRepository auditingRepository =
        new AuditingProductRepository(
            sqlRepository, sqlAuditor);

    IContractMapper mapper = new ContractMapper();

    return new ProductManagementService(
        auditingRepository, mapper);
}
```

❶ Внутренний ProductRepository

❷ Декоратор

❸ Внедрить декоратор

Как показано в листинге 7.9, поскольку вы хотите использовать `ProductRepository`, построенный на базе SQL Server, вы создаете новый экземпляр `SqlProductRepository` ❶. Но вместо того, чтобы внедрять его прямо в экземпляр `ProductManagementService`, вы декорируете его в `AuditingProductRepository`.

Вы внедряете реализации как `SqlProductRepository` ❶, так и `IAuditor`, основанный на SQL Server, в экземпляр `AuditingProductRepository` ❷. Заметьте, что и `sqlRepository`, и `auditingRepository` объявлены как экземпляры, имеющие тип `ProductRepository`.

Вы можете теперь внедрить `auditingRepository` в новый экземпляр `ProductManagementService` ❸ и вернуть этот экземпляр. `ProductManagementServices` видит только `auditingRepository` и не знает ничего о `sqlRepository`.

ВНИМАНИЕ

Листинг 9.3 содержит упрощенный пример, в котором игнорируются связанные с временем жизни проблемы. Поскольку SqlProductRepository и SqlAuditor являются одноразовыми типами, использование этого кода приведет к утечкам ресурсов. Более корректной реализацией может быть объединение листинга 9.3 с листингами 8.4 и 8.5 — но я уверен, что вы понимаете, насколько более сложным будет получившийся код.

СОВЕТ

Используйте контейнер внедрения вместо того, чтобы вручную управлять реализацией композиции объектов, временем жизни и перехватами.

Заметьте, что вы добавили функционал в `ProductRepository`, не меняя исходного кода существующих классов. Нам не понадобилось менять `SqlProductRepository`, чтобы добавить контроль. Это желательное качество, соответствующее принципу открытости/закрытости.

НЕОБХОДИМАЯ КУЛИНАРНАЯ АНАЛОГИЯ

Мне кажется, что это похоже на обваливание телячьих отбивных в панировке. Хотя мы изменяем отбивную, мы сохраняем ее размер вместо того, чтобы измельчить ее и превратить в тушеное мясо.

Теперь, когда мы изучили пример перехвата конкретного `SqlProductRepository` декорирующим его классом `AuditingProductRepository`, сделаем шаг назад и рассмотрим паттерны и связанные с ними принципы.

9.1.2. Паттерны и принципы перехвата

Как и многие другие паттерны, используемые при реализации внедрения зависимостей, «Декоратор» — это уже не новый и хорошо описанный паттерн проектирования, созданный на несколько лет раньше внедрения зависимостей. Он является настолько важной частью технологии перехвата, что, независимо от степени вашего знакомства с ним, не помешает напомнить основные его принципы.

Вы можете обратить внимание на сравнительно частое использование терминов «принцип единственной ответственности» и «принцип открытости/закрытости». Это элементы набора принципов SOLID[1].

Эти паттерны и принципы считаются ценным руководством по разработке ясного кода. Главная цель этого раздела — связать это существующее руководство с внедрениеми зависимостей, чтобы продемонстрировать, что внедрение зависимостей — это только средство для достижения цели. Внедрение зависимостей используется как инструмент, позволяющий получить легко поддерживаемый код.

Все потребители при вызове зависимостей должны соблюдать принцип подстановки Лисков. Это позволяет заменять одну реализацию другой, имеющей такой же тип абстракции. Поскольку декорирующий класс (декоратор) реализует ту же абстракцию, что и класс, который он декорирует, мы можем заменить исходный класс на декоратор.

Именно это было сделано в листинге 9.3, когда оригинальный `SqlProductRepository` был заменен на `AuditingProductRepository`. Вы можете сделать это, не меняя код потребителя — `ProductManagementService`, — потому что он придерживается принципа подстановки Лисков: ему требуется экземпляр `ProductRepository` и подойдет любая реализация.

Способность расширения поведения класса без модификации его кода известна как принцип открытости/закрытости, и это второй из пяти принципов, в совокупности известных как SOLID.

SOLID

Все хотят разрабатывать *надежное* программное обеспечение, правда? Программное обеспечение, которое проходило бы проверку временем и сохраняло бы ценность для пользователей, — то, ради чего стоит работать. Итак, аббревиатуру SOLID[2] можно считать акронимом термина «разработка качественного программного обеспечения».

---

[1] http://ru.wikipedia.org/wiki/SOLID_(объектно-ориентированное_программирование). — *Примеч. ред.*

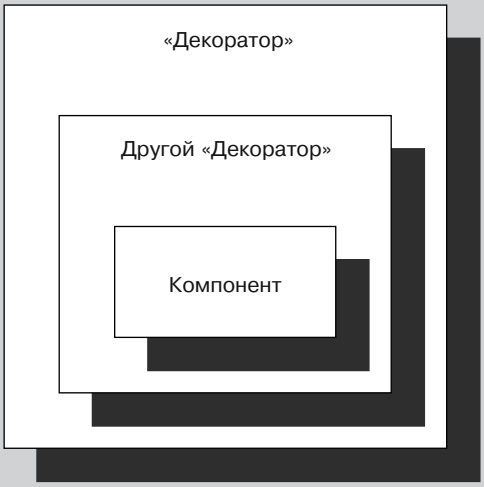
[2] *Robert C Martin* The Principles of OOD: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

«ДЕКОРАТОР» (DECORATOR)

Паттерн «Декоратор» впервые был описан в книге «Design Patterns»[1]. Его назначение — «динамическое придание объекту дополнительной функциональности. Декораторы являются гибкой альтернативой наследования для расширения функциональности».

«Декоратор» работает, обертывая (декорируя) одну реализацию абстракции в другую реализацию. Декорирующий (обертывающий) класс делегирует выполнение операций внутренней (декорированной) реализации, выполняя при этом свои дополнительные операции до или после вызова декорированного объекта.

«Декоратор» может декорировать другой «Декоратор», который, в свою очередь, декорирует еще один «Декоратор» и т. д. На следующем рисунке показано, как «Декораторы» могут обертывать друг друга. Ядром такой конструкции должна быть не декорирующая больше ни один компонент реализация, которая выполняет основную работу.



«Декоратор» внешнего уровня декорирует другой «Декоратор», который, в свою очередь, декорирует конечный компонент. Когда вызывается член (Member) «Декоратора» самого верхнего уровня, он делегирует вызов своему вложенному компоненту. Поскольку этот вложенный компонент, в свою очередь, также является «Декоратором», он делегирует вызов содержащемуся в нем компоненту. При каждом вызове «Декоратор» имеет возможность использовать полученное входное значение или же значение, возвращенное вложенным компонентом, для выполнения дополнительных действий.

Когда «Декоратор» получает вызов одного из членов абстракции, которую он реализует, он может просто делегировать вызов без выполнения каких-либо действий:

```
public string Greet(string name)
{
   return this.innerComponent.Greet(name);
}
```

[1] *Erich Gamma* Design Patterns: Elements of Reusable Object-Oriented Software. — New York: Addison-Wesley, 1994.

Или он может модифицировать входное значение прежде, чем делегировать вызов:

```
public string Greet(string name)
{
   var reversed = this.Reverse(name);
   return this.innerComponent.Greet(reversed);
}
```

Наконец, он может похожим образом модифицировать возвращаемое значение перед тем, как вернуть его:

```
public string Greet(string name)
{
    var returnValue = this.innerComponent.Greet(name);
    return this.Reverse(returnValue);
}
```

Можно обернуть код из последнего примера вокруг кода из первого, чтобы получить комбинацию, модифицирующую как вход, так и выход.

«Декоратор» может и не вызывать вложенную реализацию:

```
public string Greet(string name)
{
    if (name == null)
    {
        return "Hello world!";
    }
    return this.innerComponent.Greet(name);
}
```

В данном примере контрольный оператор реализует поведение, заданное по умолчанию для ситуации, когда на вход передано значение `null`; в этом случае вложенный компонент не вызывается вообще.

«Декоратор» от любого класса, содержащего зависимости, отличается тем, что он реализует тот же тип абстракции, что и декорируемый объект. Это позволяет компоновщику заменять оригинальный компонент на «Декоратор» без изменения кода потребителя. Декорируемый объект часто внедряется в «Декоратор», будучи объявленным как абстрактный тип. В таком случае «Декоратор» должен придерживаться принципа подстановки Лисков и одинаково рассматривать все декорированные объекты.

Мы уже видели «Декораторы» в действии в нескольких местах в книге, например, в подразделах 9.1.1 и 4.4.4.

За акронимом SOLID скрываются пять принципов объектно-ориентированного проектирования, полезных при разработке хорошо поддерживаемого кода. Эти принципы перечислены в табл. 9.1.

ПРИМЕЧАНИЕ

Ни один из принципов, объединенных в аббревиатуре SOLID, не является догмой. Это руководства, помогающими разрабатывать ясный код. Я считаю их целями, помогающими определить, в каком направлении я должен развивать свои API. Я испытываю удовлетворение, если у меня получается добиться успеха, но так бывает не всегда.

Таблица 9.1. Пять принципов SOLID

Принцип	Описание	Отношение к внедрения зависимостей
Принцип единственной ответственности (Single Responsibility Principle, SRP)	Класс должен иметь только одну ответственность. Он должен выполнять только одну вещь, но делать это хорошо. Противоположностью этого принципа является антишаблон, известный как «Класс-Бог» (God Class), когда один класс может делать все, даже варить кофе	Следовать этому принципу может оказаться сложно, но одно из многих достоинств внедрения конструктора заключается в том, что оно делает очевидным каждое нарушение этого принципа. В примере контролирующего сервиса в подразделе 9.1.1 у вас имелась возможность придерживаться SRP путем разделения ответственности между разными типами: SqlProductRepository выполняет только хранение и предоставление данных о продуктах, тогда как SqlAuditor концентрируется на операциях контроля базы данных. Единственная функция класса AuditingProductRepository заключается в координации действий ProductRepository и IAuditor
Принцип открытости/закрытости (Open/Closed Principle, OCP)	Класс должен быть открыт для расширения, но закрыт для модификации. Это значит, что должна иметься возможность добавления функциональности в существующий класс без модификации его кода. Этого не всегда легко достичь, но следование SRP как минимум упрощает процесс, поскольку чем проще код, тем легче определить потенциальные швы	Существует много способов сделать класс расширяемым, включая виртуальные методы, внедрение стратегий[1] и применение «Декораторов», но независимо от деталей внедрение зависимостей предоставляет такие возможности, позволяя нам компоновать объекты
Принцип подстановки Лисков (Liskov Substitution Principle, LSP)	Клиент должен рассматривать все реализации абстракции как эквивалентные. Мы должны иметь возможность заменять одну реализацию на другую, не разрушая потребителя	LSP является основой внедрения зависимостей. Когда код потребителей написан без следования этому принципу, вы вообще не можете заменять зависимости, кроме того, теряются некоторые (если не все) достоинства внедрения зависимостей
Принцип разделения интерфейсов (Interface Segregation Principle, ISP)	Интерфейсы должны проектироваться как мелкомодульные. Мы не хотим сваливать в кучу слишком много функций в одном интерфейсе, поскольку он становится слишком громоздким для реализации. Я нахожу, что ISP является концептуальной основой SRP. ISP требует,	На первый взгляд может показаться, что ISP лишь отдаленно касается внедрения зависимостей. Но этот принцип важен, так как интерфейс, который моделирует все, включая кухонную раковину, подталкивает вас в направлении конкретизированной реализации. Часто это

Продолжение ⇩

[1] Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1994), 315.

Таблица 9.1 *(продолжение)*

Принцип	Описание	Отношение к внедрения зависимостей
	чтобы интерфейсы моделировали только одну концепцию, тогда как SRP гласит, что реализации должны отвечать только за одно дело	является проявлением протекающей абстракции и усложняет замену зависимостей, поскольку некоторые из членов интерфейса могут оказаться бессмысленными в контексте, отличном от того, для которого была инициализирована разработка интерфейса[1]
Принцип инверсии зависимости (Dependency Inversion Principle, DIP)	Фактически синоним для требования «Программировать в соответствии с интерфейсом, а не с конкретной реализацией»	DIP является руководящим принципом для применения внедрения зависимостей

«Декоратор» (и паттерны проектирования вообще) и руководства, такие как принципы SOLID, существуют уже многие годы, и, как правило, они чрезвычайно полезны. В данном разделе я объясняю, как они связаны с внедрениеми зависимостей.

Принципы SOLID используются во всех главах книги, и я упоминаю некоторые из них в разных местах. Но когда мы начинаем говорить о перехвате и его связи с «Декораторами», оказывается, что связь с SOLID проявляется особенно отчетливо. Одни связи являются более слабыми, другие более сильными, но добавление функциональности (такой, как контроль) с помощью «Декоратора» — наглядное применение принципа открытости/закрытости и сопутствующего ему принципа единственной ответственности, поскольку первый позволяет нам создавать реализации, имеющие четко определенные области видимости.

В этом разделе мы рассмотрим паттерны и принципы, чтобы понять связь внедрения зависимостей с другими существующими руководствами. Познакомившись с ними, мы можем вернуться к цели главы, которая заключается в разработке ясного и хорошо поддерживаемого кода в условиях нечетких или меняющихся требований, а также в реализации сквозных аспектов приложения.

9.2. Реализация сквозных аспектов приложения

Большинство приложений должны работать с аспектами, которые не относятся непосредственно к каким бы то ни было функциям приложения, а скорее касаются более широких областей. Эти проблемы касаются многих практически не связанных областей кода, возможно, даже находящихся в разных модулях или даже на различных уровнях. Поскольку они распространяются на большие области кодовой базы, я называю их сквозными аспектами приложения (Cross-Cutting Concerns).

[1] *Mark Seemann* Interfaces are not abstractions, 2010: http://blog.ploeh.dk/2010/12/02/InterfacesAreNotAbstractions.aspx.

В табл. 9.2 представлены некоторые примеры. Эта таблица не является всеобъемлющим списком; скорее, это просто иллюстративный пример.

Таблица 9.2. Распространенные примеры сквозных аспектов приложений

Аспект	Описание
Контроль (Auditing)	Каждая изменяющая данные операция должна регистрироваться в системе контроля, указывая дату и время, идентификационные данные пользователя, который произвел изменения, и информацию о самом изменении. Вы видели пример в подразделе 9.1.1
Логирование (Logging)	Логирование, несколько отличное от контроля, фокусируется на фиксации событий, которые влияют на состояние приложения. Это могут быть события, связанные функционированием информационных технологий, но могут быть и события, которые связаны с бизнес-логикой
Контроль производительности (Performance monitoring)	Несколько отличается от логирования, поскольку здесь в основном регистрируются параметры производительности, а не какие-либо события. Если у вас имеются специфические соглашения об уровне обслуживания (Service Level Agreements, SLA), которые могут быть отслежены с помощью стандартной инфраструктуры, вы должны реализовать свой мониторинг производительности. Для этого хорошо подходит система Custom Windows Performance Counters, но вам все равно придется добавить код, собирающий данные
Безопасность (Security)	Некоторые операции могут быть разрешены только определенным пользователям, и вы должны обеспечить это
Кэширование (Caching)	Иногда производительность может быть повышена путем реализации кэшей, но не существует причин, по которым определенные компоненты доступа к данным обязательно должны их использовать. Вам может потребоваться возможность разрешать или запрещать кэширование данных для разных реализаций доступа к данным. Мы уже видели фрагменты реализации кэширования данных с помощью «Декораторов» в подразделе 4.4.4
Обработка ошибок (Error handling)	Нам может потребоваться обрабатывать определенные исключительные ситуации и либо логировать их, либо показывать сообщение пользователю. Можно использовать обрабатывающий ошибки «Декоратор», чтобы работать с ошибками единообразным способом
Отказоустойчивость (Fault tolerance)	Несомненно, что не управляемые процессом ресурсы будут время от времени недоступны. Можно реализовать отказоустойчивые паттерны, такие как «Прерыватель цепи» (Circuit Breaker), используя «Декоратор»

Когда мы рисуем схемы архитектуры многоуровневых приложений, сквозные аспекты приложения часто представляются в виде вертикальных блоков, размещенных рядом со слоями, как показано на рис. 9.4.

В данном разделе мы рассмотрим некоторые примеры, иллюстрирующие способы использования перехвата в виде декораторов, чтобы реализовать сквозные аспекты приложения. Мы возьмем несколько аспектов из табл. 9.2, чтобы почувствовать, как они реализуются в соответствии с принципами SOLID, но используем лишь небольшое подмножество. Как и в случае со многими другими концепциями, перехват несложно усвоить как абстракцию, но «дьявол скрывается в деталях». Требуются усилия, чтобы правильно освоить эту технологию, и я лучше дам вам слишком много примеров, чем слишком мало. Когда мы изучим их, вы будете иметь ясное представление о том, что такое перехват и как его следует применять.

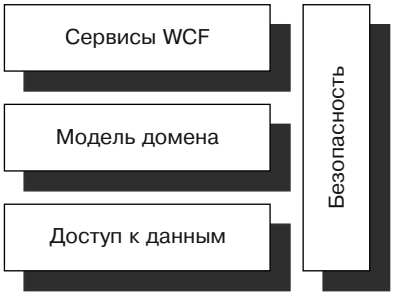


Рис. 9.4. Сквозные аспекты приложений на схемах архитектуры приложений часто представляются в виде вертикальных блоков, распространяющихся вдоль всех слоев. В приведенном примере сквозным аспектом является безопасность (Security)

Поскольку мы уже рассматривали вводный пример в подразделе 9.1.1, мы возьмемся теперь за более сложный пример, чтобы проиллюстрировать, как перехват может применяться со сложной логикой. Затем опишем пример, в котором используется декларативный подход.

9.2.1. Перехват с «Прерывателем цепи» (Circuit Breaker)

Любое приложение, взаимодействующее с ресурсом, не управляемым из него, будет время от времени сталкиваться с недоступностью этого ресурса. Сетевые соединения отключаются, базы данных становятся недоступными, а веб-сервисы подвергаются DDOS-атакам «Отказ от обслуживания» (Distributed Denial of Service). В таких случаях вызывающее приложение должно иметь возможность восстанавливаться и решать проблему соответствующим образом.

Большинство API, включенных в платформу .NET, имеют установленные по умолчанию задержки, которые гарантируют, что обращение к не управляемому приложением ресурсу не блокирует основной процесс. Но остается вопрос — как нужно относиться к следующему запросу к сбойному ресурсу при возникновении исключительной ситуации по задержке? Нужно ли пытаться обратиться к нему снова? Поскольку задержка обычно означает, что сервис на другой стороне либо выключен, либо перегружен запросами, формирование нового запроса, блокирующего работу клиента, может оказаться не самой хорошей идеей. Правильнее было бы предположить худшее и немедленно сгенерировать исключительную ситуацию. Это можно успешно реализовать с помощью шаблона «Прерыватель цепи» (Circuit Breaker).

«Прерыватель цепи» — это устойчивый паттерн, повышающий работоспособность приложения путем немедленного перехода к обработке ошибки вместо того, чтобы повторно пытаться работать с отказавшими ресурсами. Это хороший пример не связанных с функциональностью приложения требований и настоящий сквозной аспект приложения, поскольку он практически ничего не должен делать с функциями вызываемого не управляемого приложением процесса.

Паттерн «Прерыватель цепи» сам по себе немного сложен, и его реализация может оказаться запутанной, но достаточно разобраться с ним лишь однажды.

Можно даже реализовать его в виде многократно используемой библиотеки. Если у нас имеется многоразовый «Прерыватель цепи», мы можем легко применить его ко множеству компонент, используя шаблон «Декоратор».

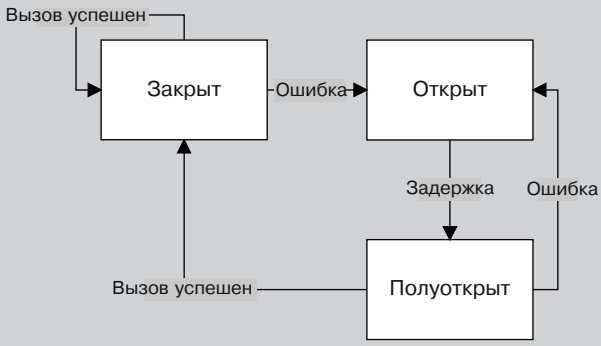
«ПРЕРЫВАТЕЛЬ ЦЕПИ»

Паттерн проектирования «Прерыватель цепи» назван так по аналогии с электрическим предохранителем. Он предназначен для прерывания соединения, если возникает сбой, чтобы предотвратить распространение этого сбоя.

В программных приложениях при возникновении задержек или подобных ошибок связи плохая ситуация зачастую может сильно ухудшиться, если вы будете продолжать пытаться работать с отказавшей системой. Если удаленная система захлебывается, многочисленные повторные обращения к ней могут вывести ее из строя. Пауза же, возможно, предоставит ей возможность восстановиться. На уровне вызовов потоки, блокированные в ожидании окончания задержки, могут совершенно вывести из строя приложения-потребители. Лучше определить, что соединения разорваны и немедленно перейти к обработке ошибки.

Внутренняя структура «Прерыватель цепи» решает эти проблемы путем отключения переключателя при возникновении ошибок. Она обычно использует задержку, которая позволяет повторно запросить соединение по истечении некоторого периода времени. Таким образом, соединение может автоматически восстанавливаться, когда восстанавливается удаленная система.

Следующий рисунок иллюстрирует упрощенное представление смены состояний в «Прерывателе цепи».



«Прерыватель цепи» начинает работать, находясь в состоянии `Закрыт` (Closed), сигнализируя, что цикл закрыт, и сообщения могут быть переданы. Когда возникает ошибка, прерыватель срабатывает, и состояние меняется на `Открыт` (Open). В этом состоянии прерыватель не разрешает делать вызовы к удаленной системе; вместо этого он немедленно генерирует исключительную ситуацию. По истечении времени задержки наступает состояние `Полуоткрыт` (Half-Open), в котором разрешается выдача единичного вызова к удаленной системе. Если он оказывается успешным, состояние возвращается обратно в `Закрыт` (Closed), но если происходит сбой, прерыватель возвращается в `Открыт` (Open), запуская новую задержку.

Вы можете сделать «Прерыватель цепи» более сложным, чем показано здесь. Прежде всего вы, возможно, не захотите, чтобы прерыватель срабатывал каждый раз, когда возникает случайная кратковременная ошибка, а вступал в дело только при превышении некоторого порога. Кроме того, прерыватель должен срабатывать при возникновении ошибок только некоторых типов. Задержки и исключительные ситуации, определенные для работы со связью, конечно, не помешают, но `NullReferenceException` скорее указывает на ошибку в программировании, а не на *случайную* ошибку.

Рассмотрим пример, показывающий, как паттерн «Декоратор» может добавить функционал паттерна «Прерыватель цепи» к имеющемуся нерабочему компоненту. В данном примере мы сфокусируемся на применении переиспользуемого «Прерывателя цепи», но не на том, как он реализован.

Пример: реализация «Прерывателя цепи»

В подразделе 7.4.2 мы создали WPF-приложение, взаимодействующее с WCF-сервисом с помощью интерфейса `IProductManagementAgent`. Хотя мы и рассматривали его еще раз в подразделе 8.2.1, мы не изучали этот интерфейс детально.

В предыдущих примерах мы использовали `WcfProductManagementAgent`, реализующий этот интерфейс путем вызова операций сервиса WCF. Поскольку эта реализация не содержит кода для явной обработки ошибок, любая ошибка связи будет передаваться в вызывающий код.

Именно здесь удобно использовать «Прерыватель цепи». Вы хотите обрабатывать ошибки сразу же, как только появляются исключительные ситуации. При таком подходе вам не понадобится блокировать вызывающий поток и перегружать сервис запросами. Как показано на рис. 9.5, сначала вы объявляете декоратор для `IProductManagementAgent`, и при этом необходимые зависимости запрашиваются через внедрение конструктора. Обратите внимание: `CircuitBreakerProductManagementAgent` реализует интерфейс и при этом содержит экземпляр, внедренный через конструктор. Еще одной зависимостью является `ICircuitBreaker`, который может использоваться для реализации паттерна «Прерыватель цепи».

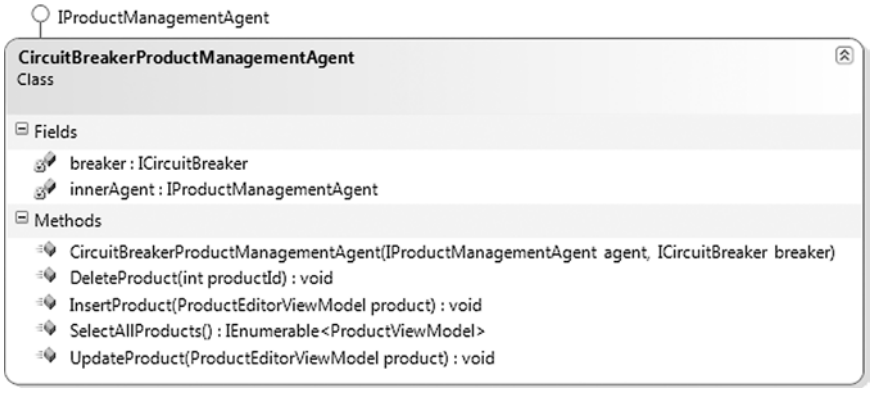


Рис. 9.5. CircuitBreakerProductManagementAgent является декоратором для IProductManagementAgent

Теперь каждый вызов декорированного `IProductManagementAgent` может быть декорирован примерно так, как это сделано в примере из листинга 9.4.

Листинг 9.4. Декорирование с использованием "Прерывателя цепи"

```
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 следует сделать до вызова декорированного агента, — проконтролировать состояние «Прерыватель цепи». Метод `Guard` позволит вам начать работу, если состояние будет `Closed` (Закрыт) или `Half-Open` (Полуоткрыт), но если состояние будет `Open` (Открыт), то будет генерироваться исключительное состояние. Таким образом, ошибка возникнет сразу же, как только у вас появятся подозрения, что вызов не выполняется.

Если вызову удалось пройти метод `Guard`, вы можете попытаться вызвать декорированный агент. Обратите внимание, что вызов обернут в блок `try`: если вызов выдаст ошибку, вы включаете прерыватель. Данный пример намеренно упрощен, но в реальной реализации на прерывателе нужно ловить ошибки (`catch` и `trip`) из списка типов исключений. Поскольку `NullReferenceExceptions` и подобные ему типы исключительных ситуаций редко обозначают случайные ошибки, в таких случаях нет причин запускать прерыватель.

Ошибка прерывателя, когда он находится как в состоянии `Closed`, так и в состоянии `Half-Open`, переводит его в состояние `Open`. Из состояния `Open` выдается новый запрос, и задержка определяет, когда прерыватель вернется в состояние `Half-Open`.

Если вызов завершился успешно, вы сами сигнализируете прерывателю `Circuit` об этом, вызывая метод `Succeed`. Если вы уже находитесь в состоянии `Closed`, вы остаетесь в нем же. Если вы находитесь в состоянии `Half-Open`, вы будете переведены в состояние `Closed`. Невозможно просигнализировать об успехе, когда «Прерыватель цепи» находится в состоянии `Open`, поскольку метод `Guard` не позволит вам оказаться в состоянии успеха.

Все другие методы `IProductManagementAgent` выглядят подобным образом, с единственными отличиями в том, какой метод вызывается на вложенном компоненте `innerAgent` и добавлением еще одной строки в случае, если метод должен возвращать значение. Эти отличия видны внутри блока `try` метода `SelectAllProducts`:

```
var products = this.innerAgent.SelectAllProducts();
this.breaker.Succeed();
return products;
```

Поскольку вы должны сигнализировать об успехе в «Прерывателе цепи», вы сохраняете возвращаемое значение декорированного агента перед тем, как вернуть его; но это единственное отличие между методами, которые возвращают значения, и методами, которые не делают этого.

В нашем примере реализация `ICircuitBreaker` остается незавершенной, но в реальном приложении она представляет собой полностью переиспользуемый набор

классов, реализующих шаблон проектирования «Состояние» (State). Рисунок 9.6 демонстрирует эти классы.

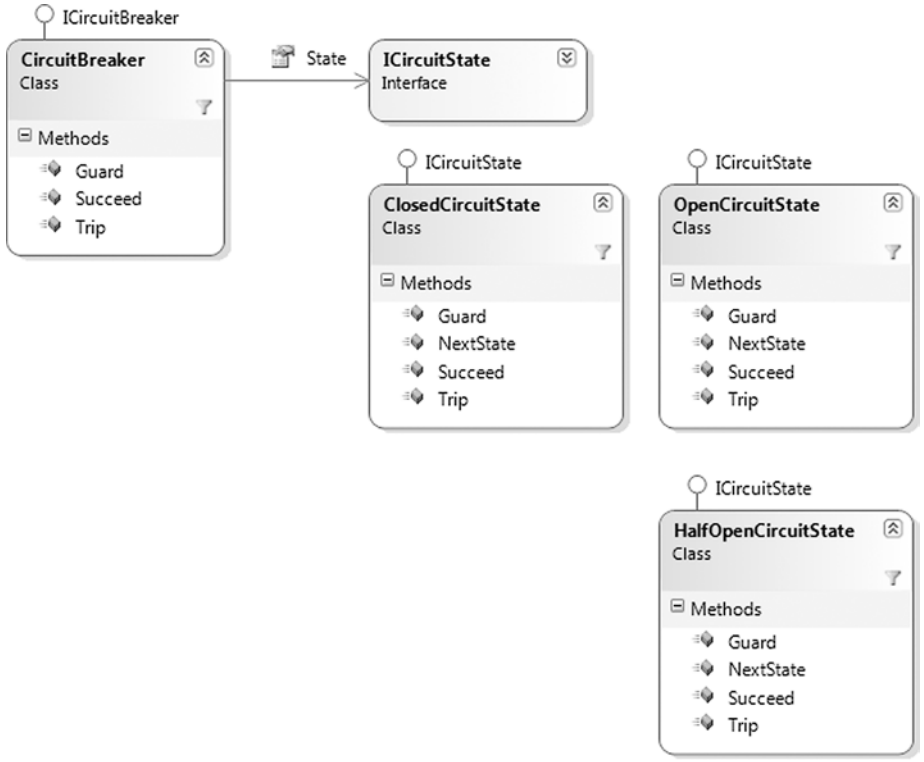


Рис. 9.6. Класс CircuitBreaker реализует интерфейс ICircuitBreaker, используя шаблон State. Все эти методы реализуются делегированием полиморфного участника типа State, который изменяется, когда состояния переходят из одного в другое

Хотя в этой книге мы не углубляемся более подробно в реализацию «Прерывателя цепи», важно знать, что вы можете выполнить перехват с любым сложным кодом.

СОВЕТ

Если у вас есть вопросы по поводу реализации класса CircuitBreaker, можете увидеть ее в коде, предлагаемом вместе с книгой.

Чтобы скомпоновать `ProductManagementAgent`, в который добавлена функциональность «Прерывателя цепи», его можно обернуть вокруг другой реализации:

```
var timeout = TimeSpan.FromMinutes(1);
ICircuitBreaker breaker = new CircuitBreaker(timeout);
IProductManagementAgent circuitBreakerAgent =
    new CircuitBreakerProductManagementAgent(wcfAgent, breaker);
```

В листинге 7.10 мы компонуем WPF-приложение с несколькими зависимостями, включая экземпляр `WcfProductManagementAgent`. Мы можем задекорировать пе-

ременную wcfAgent, внедряя ее в экземпляр CircuitBreakerProductManagementAgent, который реализует этот же интерфейс. В данном конкретном примере мы создаем новый экземпляр класса CircuitBreaker каждый раз, когда вы разрешаете зависимости, и это соответствует жизненному стилю Transient.

В WPF-приложении, в котором мы разрешаем зависимости только однажды, использование «Прерыватель цепи» со стилем Transient не представляет проблем, но вообще Transient не является оптимальным жизненным стилем для такой функциональности. На другой стороне будет находиться только один сервис. Если этот сервис станет недоступным, «Прерыватель цепи» должен пресечь все попытки подключиться к нему. Если используется несколько экземпляров CircuitBreakerProductManagementAgent, это касается их всех.

БОЛЕЕ КОМПАКТНЫЙ ICIRCUITBREAKER

Как показано здесь, интерфейс ICircuitBreaker содержит три члена: Guard, Succeed и Trip. Альтернативное определение интерфейса может использовать передачу продолжений (Continuation Passing)[1] для уменьшения количества методов:

```
public interface ICircuitBreaker
{
   void Execute(Action action);

   T Execute<T>(Func<T> action);
}
```

Это позволит нам более лаконично использовать ICircuitBreaker в каждом методе, вот так:

```
public void InsertProduct(ProductEditorViewModel product)
{
   this.breaker.Execute(() =>
       this.innerAgent.InsertProduct(product));
}
```

Я использовал более наглядную и старомодную версию ICircuitBreaker, потому что я хочу сфокусироваться на перехвате. Хотя лично мне нравится передача продолжений, я считаю, что лямбда-код и дженерики являются сами по себе продвинутыми темами, и думаю, что в данном контексте они скорее отвлекут читателя, чем помогут ему.

То, что мы в конечном итоге предпочли одно определение интерфейса другому, не меняет выводов этой главы.

Представляется очевидным выбор варианта настройки CircuitBreaker с жизненным стилем Singleton, но это означает, что он должен быть потокобезопасным. CircuitBreaker по определению хранит информацию о состояниях; поэтому потоковая безопасность должна быть реализована явно. Это делает ее еще более сложной.

---

[1] Хорошее введение в стиль продолжений дается в работе *Jeremy Miller* Patterns in practice: Functional Programming for Everyday .NET Development. — MSDN Magazine, October 2009. Доступно онлайн на сайте http://msdn.microsoft.com/en-us/magazine/ee309512.aspx.

Несмотря на упомянутую сложность, можно легко перехватить экземпляр `IProductManagementAgent` с помощью «Прерывателя цепи». Хотя первый пример перехвата в подразделе 9.1.1 был совершенно прост, пример с «Прерывателем цепи» показывает, что можно перехватить класс со сквозным аспектом приложения, реализация которого является намного более сложной, чем оригинальная реализация.

Паттерн «Прерыватель цепи» гарантирует, что приложение быстро дает ошибку вместо того, чтобы связывать драгоценные ресурсы; но в идеале приложение вообще не должно терпеть сбоев. Чтобы разрешить эту проблему, вы можете реализовать обработку некоторых видов ошибок путем использования перехвата.

9.2.2. Обработка исключительных состояний

Применение зависимостей приводит к периодическому возникновению исключительных ситуаций. Даже написанный наилучшим образом код будет (и должен) порождать исключительные ситуации, если возникают непредвиденные обстоятельства, которые он не может обработать. Клиенты, работающие с ресурсами, не управляемыми приложением, особенно часто попадают в такие ситуации. Класс, подобный классу `WcfProductManagementAgent` из примера WPF-приложения, является примером такого случая. При недоступности веб-сервиса агент должен начать генерировать исключительные ситуации.

«Прерыватель цепи» не изменяет этот фундаментальный признак. Хотя он и перехватывает WCF-клиент, он все еще генерирует исключительные ситуации.

Вместо того чтобы аварийно завершать приложение, вы можете выдать сообщение о том, что операция завершилась неудачей и что можно попробовать выполнить ее снова позднее.

Вы можете использовать перехват, чтобы добавить обработку ошибок на манер SOLID. Вы не хотите нагружать зависимости обработкой ошибок. Поскольку зависимость должна рассматриваться как переиспользуемый компонент, который может быть задействован во множестве различных сценариев, вряд ли окажется возможным добавить в нее такую стратегию обработки ошибок, которая могла бы удовлетворить всем возможным сценариям. Кроме того, реализация обработки ошибок в самой зависимости будет нарушать принцип единственной ответственности.

Используя перехват для обработки исключительных ситуаций, вы следуете принципу открытости/закрытости. Это позволяет вам реализовать наилучшую стратегию обработки ошибок для любой ситуации. Рассмотрим пример.

Пример: обработка исключительных ситуаций

В предыдущем примере вы декорировали `WcfProductManagementAgent` в «Прерыватель цепи», чтобы использовать его в клиентском приложении управления продуктами, первоначально рассмотренном в подразделе 7.4.2. «Прерыватель цепи» обрабатывает ошибочные ситуации единственным образом — завершая работу клиента; но он еще и генерирует исключительные ситуации. Если они остаются необработанными, то происходит крах приложения. Поэтому требуется реализовать «Декоратор», который будет обрабатывать хотя бы некоторые виды ошибок. Когда генери-

руется исключительная ситуация, код должен выдавать сообщение, как показано на рис. 9.7. Обратите внимание, что в данном случае сообщение об ошибке происходит из «Прерывателя цепи», а не из базового сбоя соединения.

Рис. 9.7. Приложение для управления продуктами обрабатывает связанные с соединениями исключения, показывая сообщение пользователю

Реализовать такое поведение легко. Так же, как и в подразделе 9.2.1, вы добавляете новый класс `ErrorHandlingProductManagementAgent`, который декорирует интерфейс `IProductManagementAgent`. Листинг 9.5 показывает пример одного из методов такого интерфейса, остальные методы похожи на этот.

Листинг 9.5. Обработка исключительных ситуаций

```
public void InsertProduct(ProductEditorViewModel product)
{
    try
    {
        this.innerAgent.InsertProduct(product);
    }
    catch (CommunicationException e)
    {
        this.AlertUser(e.Message);
    }
    catch (InvalidOperationException e)
    {
        this.AlertUser(e.Message);
    }
}
```

1 Делегировать операцию декорированному агенту

2 Выдать сообщение пользователю

Метод `InsertProduct` служит иллюстрацией для всей реализации класса `ErrorHandlingProductManagementAgent`. Вы пытаетесь вызвать декорированный агент 1 и предупреждаете пользователя с помощью сообщения 2, если генерируется исключительная ситуация. Обратите внимание, что обрабатывается только конкретный набор

известных исключительных ситуаций, поскольку подавление всех возможных исключений может оказаться опасным.

Предупреждение пользователя включает в себя подготовку строки сообщения и отображение ее с помощью метода `MessageBox.Show`.

Еще раз отмечу, что вы добавили функциональность в первоначальный вариант (`WcfProductManagementAgent`), реализовав «Декоратор». Вы следовали как принципу единственной ответственности, так и принципу открытости/закрытости, последовательно добавляя новые типы вместо того, чтобы модифицировать существующий код. Теперь вы получили шаблон, поддерживающий больше соглашений, чем «Декоратор».

Для данного сквозного аспекта приложения реализация, основанная на «Декораторе», используется многократно. Реализация «Прерыватель цепи» означает включение одного и того же шаблона кода во все методы интерфейса `IProductManagementAgent`. Если вы захотите добавить «Прерыватель цепи» в другую абстракцию, вы должны будете добавить такой же код в другие методы. Хотя шаблоны отличаются, то же относится и к рассмотренному только что коду обработки исключительных ситуаций.

Кратко рассмотрим реализацию функции безопасности. Это позволит ввести более общий подход к компоновке сквозных аспектов приложения, о чем мы подробнее поговорим в разделе 9.3.

9.2.3. Добавление функционала безопасности

Безопасность — это еще один распространенный сквозной аспект приложения. Мы хотим максимально обезопасить свои приложения, чтобы предотвратить неавторизованный доступ к той или иной функциональности.

ПРИМЕЧАНИЕ

Безопасность — это обширная тема, охватывающая множество областей, включая несанкционированное разглашение информации и взлом компьютерных сетей. В этом подразделе я кратко затрону только вопросы авторизации — то есть обеспечение того, что только авторизованные люди (или системы) могут выполнить некоторые действия.

Как и при работе с «Прерывателем цепи», мы хотели бы перехватить вызов метода и проверить, должен ли данный вызов быть разрешен. В противном случае вместо выполнения вызова следует сгенерировать исключительную ситуацию. Используемый принцип остается прежним: различие заключается в критерии, который мы задействуем для определения валидности (допустимости) вызова.

Распространенный подход в реализации логики авторизации заключается в применении ролевой безопасности с использованием `Thread.CurrentPrincipal`. Начать можно с использования декоратора `SecureProductRepository`. Поскольку, как вы видели в предыдущих разделах, все методы подобны друг другу по внутренней структуре, листинг 9.6 содержит лишь пример реализации 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 контрольного оператора, который явно обращается к `Thread.CurrentPrincipal` и запрашивает, имеет ли он роль `ProductManager`. В противном случае немедленно генерируется исключительная ситуация. Только если вызывающий `IPrincipal` имеет требуемую роль, он проходит через контрольный оператор и вызывает декорированный репозиторий.

ПРИМЕЧАНИЕ

Как вы помните, Thread.CurrentPrincipal является примером шаблона «Окружающий контекст».

Это настолько универсальная идиома программирования, что она инкапсулирована в классе `System.Security.Permissions.PrincipalPermission`; поэтому предыдущий пример можно переписать немного короче:

```
public override void InsertProduct(Product product)
{
    new PrincipalPermission(null, "ProductManager").Demand();

    this.innerRepository.InsertProduct(product);
}
```

Класс `PrincipalPermission` инкапсулирует запрос для проверки того, что текущий `IPrincipal` имеет заданную роль. Вызов метода `Demand` будет генерировать исключительную ситуацию, если `Thread.CurrentPrincipal` не имеет роли `ProductManager`. Этот пример функционально эквивалентен листингу 9.6.

Если вам просто нужно проверить, имеет ли текущий `IPrincipal` заданную роль, можно перейти к чисто декларативному стилю:

```
[PrincipalPermission(SecurityAction.Demand,Role = "ProductManager")]
public override void InsertProduct(Product product)
{
    this.innerRepository.InsertProduct(product);
}
```

Атрибут `PrincipalPermission` обеспечивает такую же функциональность, что и класс `PrincipalPermission`, но выглядит как атрибут. Поскольку фреймворк .NET знает, что это за атрибут, он, встречая данный атрибут, выполняет соответствующий код `PrincipalPermission`.

На данном этапе использование отдельного «Декоратора» только для поддержки атрибута начинает выглядеть несколько излишним. Почему бы не применить этот атрибут прямо в оригинальном классе?

Хотя использование декларативного стиля выглядит заманчиво, имеется несколько причин, почему нежелательно так поступать.

- Применение атрибутов исключает более сложную логику. Что если вы захотите, чтобы большинство пользователей имели возможность редактировать описания продуктов, но только имеющие роль `ProductManagers` могли бы изменять цену? Такая логика может быть реализована в программном коде, но ее нелегко задать посредством атрибутов.
- Что если вы захотите убедиться, что правила разрешения доступа работают независимо от типа выбранной реализации `ProductRepository`? Поскольку атрибуты конкретных классов не могут быть повторно использованы между реализациями, это будет нарушать принцип DRY[1].
- У вас нет возможности изменять логику безопасности независимо от `ProductRepository`.

Идея реализации сквозных аспектов приложения с применением декларативного подхода не нова. Она часто используется в аспектно-ориентированном программировании, и именно поэтому мы рассматриваем ее и возможности ее использования для реализации слабо связанного перехвата.

9.3. Объявление аспектов

В предыдущих разделах мы рассмотрели шаблоны перехвата и познакомились с тем, как они могут помочь нам реализовать сквозные аспекты приложения с помощью принципов SOLID. В разделе 9.2.3 вы видели, как можно свести реализацию проверки безопасности к чисто декларативному подходу.

Применение атрибутов для объявления аспектов — это общая методика аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP). Но несмотря на первоначальную заманчивость такого подхода, использование атрибутов сопровождается некоторыми характерными проблемами, делающими этот подход менее чем идеальным. В первой части этого раздела я сделаю обзор данной концепции и нескольких известных ее недостатков.

ПРИМЕЧАНИЕ

Я использую термин атрибут аспекта (Aspect Attribute), чтобы обозначить атрибут, который реализует или означает аспект.

Поскольку мы решили не применять атрибуты для объявления аспектов, мы потратим остаток этого раздела на рассмотрение динамических перехватов с помощью контейнеров внедрения зависимостей, что является более приемлемой альтернативой.

9.3.1. Использование атрибутов для объявления аспектов

Атрибуты имеют те же характерные особенности, что и декораторы: хотя они могут добавить или подразумевать модификацию поведения члена, они не изменяют его

[1] Don't Repeat Yourself — не повторяй себя самого.

сигнатуру. Как вы видели в подразделе 9.2.3, можно заменить явный, императивный код авторизации на определение через атрибут. Вместо того чтобы писать строки явного кода, вы можете получить тот же результат, используя атрибут `[PrincipalPermission]`.

Представляется привлекательным экстраполировать данный концепт на другие сквозные аспекты приложения. Как удобно было бы декорировать метод или класс атрибутом `[HandleError]` или даже специальным атрибутом `[CircuitBreaker]` и таким образом использовать аспект с единственной строкой декларативного кода?

Такое возможно, но здесь возникает несколько проблем, которые нужно понимать и устранять.

Первая и основная проблема проистекает из того факта, что атрибуты по своей сути пассивны. Хотя определение специального атрибута и его применение сводится просто к произведению класса из `System.Attribute` и декорированию им других классов, этим функционал специального атрибута и ограничивается.

Но подождите! Разве использование атрибута `[PrincipalPermission]` не изменяет поведения метода? Да, но этот атрибут (и некоторые другие атрибуты, доступные в Base Class Library) является особенным. Фреймворк .NET понимает и обрабатывает этот атрибут, но он не будет делать этого для любого произвольного атрибута, объявленного вами.

У вас имеется два варианта действий, если вы хотите разрешить специальным атрибутам модифицировать поведение приложения:

- модифицировать этап компиляции;
- ввести дополнительный узел для обработки во время исполнения.

Кратко проанализируем каждый из вариантов.

Модификация компиляции

Один из наиболее популярных фреймворков AOP, PostSharp[1], позволяет добавлять специальные атрибуты в ваш код. Эти атрибуты должны быть наследниками специального атрибута, определенного в PostSharp SDK и предоставляющего виртуальные методы, которые вы можете переопределить, чтобы реализовать требуемое поведение аспекта. После этого дополнительные атрибуты могут использоваться в ваших классах или в участниках классов. Рисунок 9.8 показывает, что происходит далее. PostSharp позволяет добавлять этап дополнительной компиляции после того, как будет выполнена нормальная компиляция. Поскольку дополнительные атрибуты PostSharp в вашем коде воспринимаются нормальным компилятором (например, `csc.exe`) точно так же, как и любой другой атрибут, на выходе получается обычная сборка с пассивными атрибутами. PostSharp включает этап посткомпиляции, на котором он берет уже скомпилированную сборку и включает код ваших специальных атрибутов в снабженный атрибутами код приложения. Результатом является новая сборка .NET с внедренными аспектами.

PostSharp использует посткомпиляцию для включения пассивных атрибутов в активный код. Процессор PostSharp находит атрибуты, наследуемые из атрибутов PostSharp, и соединяет код этих атрибутов с кодом, содержащим эти атрибуты.

---

[1] www.sharpcrafters.com/postsharp.

Результатом будет новая сборка с кодом аспекта, соединенного с оригинальным кодом.

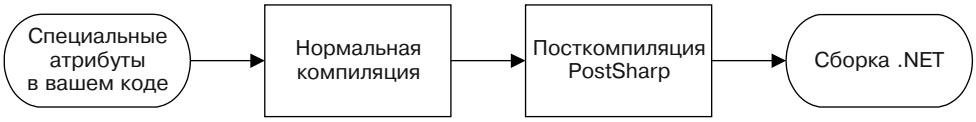


Рис. 9.8. PostSharp позволяет добавлять этап дополнительной компиляции

Эта сборка является совершенно нормальной и запускается точно так же, как и все другие коды .NET. Во время работы ей не требуется дополнительных компонентов, добавленных в среду времени исполнения.

К достоинствам такого подхода относится то, что не требуются никакие дополнительные усилия по проектированию с вашей стороны. Внедрение зависимостей не является необходимым, хотя его использование и не исключается. Я уверен, что у данного подхода можно обнаружить и другие достоинства.

Один из недостатков этого подхода заключается в том, что код, который исполняется, отличается от кода, который вы пишете. Если вы захотите отлаживать этот код, вам потребуется выполнять специальные шаги, и хотя поставщики с готовностью предложат вам средства, позволяющие сделать это, вы попадаете в зону действия антипаттерна «Блокировка поставщиком» (Vendor Lock-In).

Но самый значительный недостаток заключается собственно в использовании атрибутов. Этот же недостаток имеется и у варианта с применением пользовательского хоста для активации атрибутов. Рассмотрим этот вариант, прежде чем вернемся к анализу недостатков использования атрибутов.

Использование специального хоста

В другом варианте активации атрибутов требуется, чтобы весь код был активирован или инициализирован специальным хостом или фабрикой. Такая фабрика должна иметь возможность проверять все атрибуты класса, который она инициализирует, и действовать соответственно.

Подобный прием используется несколькими технологиями .NET, работающими с атрибутами. Вот несколько примеров.

- WCF включает множество атрибутов, таких как `[ServiceContract]`, `[OperationContract]` и т. д. Эти атрибуты действуют только в случае, если вы размещаете сервис на экземпляре `ServiceHost` (это одна из услуг, предоставляемых IIS).
- ASP.NET MVC позволяет указать через атрибут `[AcceptVerbs]`, какие глаголы HTTP вы будете принимать, а также указать через атрибут `[HandleError]`, какие исключительные ситуации вы будете обрабатывать, а также задать еще некоторые данные. Это возможно, поскольку ASP.NET MVC сам по себе является специальным пользовательским хостом и управляет временем жизни своих контроллеров.
- Все фреймворки модульного тестирования .NET, которые я знаю, используют атрибуты для идентификации вариантов тестирования. Фреймворк модульно-

го тестирования инициализирует тестовые классы и интерпретирует атрибуты, чтобы определить, какие тесты следует выполнять.

Компоновка объектов с помощью контейнера внедрения зависимостей похожа на эти примеры. Поскольку контейнер внедрения зависимостей инициализирует экземпляры требуемых классов, он имеет возможность проверить каждый класс на предмет поиска специальных атрибутов.

Неудивительно, что многие контейнеры внедрения зависимостей поставляются с возможностями, которые позволяют вам делать именно это. Если вы уже решили использовать контейнер, должны ли вы проходить весь этот путь и определять и использовать специальные атрибуты?

Я вижу только одно преимущество, которое мы получаем от использования динамического перехвата: атрибут очень легко обнаружить. Даже несмотря на то, что он имеет весьма высокий уровень косвенности, вы все же получаете ценное указание на то, что что-то еще происходит, помимо работы кода метода, который вы рассматриваете.

Но в реализации сквозных аспектов приложения с помощью атрибутов имеются и недостатки. Они являются общими при использовании как посткомпиляции, так и специальных хостов.

Недостатки определения аспектов через атрибуты

Каким бы привлекательным ни выглядело определение аспектов через атрибуты, у этого подхода имеются определенные недостатки.

Атрибуты компилируются вместе с кодом, который они декорируют. Это означает, что вы не можете легко изменить поведение всего приложения. Возьмем в качестве примера обработку ошибок. В подразделе 9.2.2 вы видели, как можно использовать паттерн проектирования «Декоратор» для реализации обработки ошибок в любом `IProductManagementAgent`. Интересно то, что `WcfProductManagementAgent` ничего не знает о `ErrorHandlingProductManagementAgent`. Как показано на рис. 9.9, они даже реализованы в двух различных библиотеках. Поскольку сборка `ProductManagementClient` содержит корень компоновки, она зависит и от `ProductWcfAgent`, и от `PresentationLogic`.

Базовая реализация, представляемая `WcfProductManagementAgent`, не выполняет явной обработки ошибок, потому что правильная обработка ошибок зависит от контекста. Например, в приложениях с графическим интерфейсом, типа WPF-приложения, которое мы будем использовать в качестве примера, может быть целесообразен вывод сообщения в диалоговом окне. Но в консольном приложении предпочтительнее выдавать сообщения в выходной поток ошибок. Наконец, автоматизированный сервис может ставить сообщение об ошибке в очередь для возврата и переключаться на выполнение другой задачи.

Чтобы оставаться открытой и гибкой, библиотека `ProductWcfAgent` не должна включать обработку ошибок. Но если вы используете атрибут аспекта в `WcfProductManagementAgent` (или, тем более, в `IProductManagementAgent`), этот аспект окажется сильно связан с реализацией (или даже с абстракцией). Если вы сделаете так, вы получите обработку ошибок в `WcfProductManagementAgent`, привязанную к определенному контексту, и потеряете способность изменять аспект независимо от реализации.

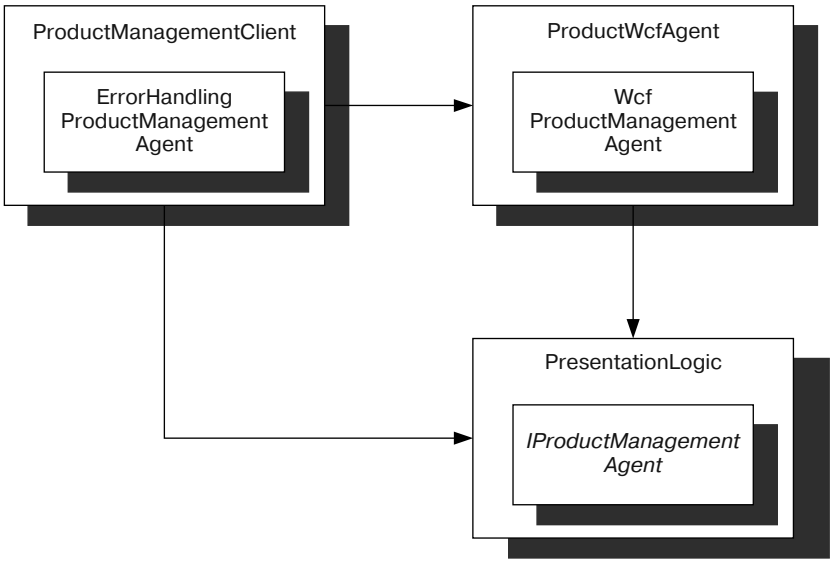


Рис. 9.9. Как ErrorHandlingProductManagementAgent, так и WcfProductManagementAgent реализуют IProductManagementAgent, но находятся они в двух разных библиотеках

Вторая проблема, связанная с определением аспектов в атрибутах, заключается в том, что ваши возможности очень ограниченны, когда дело доходит до применения атрибутов. Применение атрибутов может осуществляться только на следующих уровнях:

- параметры, включая возвращаемые значения;
- члены, такие как методы, свойства и поля;
- типы, такие как классы и интерфейсы;
- библиотеки.

Хотя здесь получается широкий набор вариантов, вы не можете легко выразить условные конфигурации типа «Я хочу применить аспект `Circuit Breaker` ко всем типам, имя которых начинается с `Wcf`». Вместо этого вам придется применить гипотетический атрибут `[CircuitBreaker]` ко всем соответствующим классам, нарушая принцип DRY.

Последний недостаток атрибутов аспектов заключается в том, что атрибуты должны иметь простой конструктор. Если вы хотите использовать зависимости в аспекте, вы можете сделать это только с помощью окружающего контекста. Вы уже видели пример такого решения в подразделе 9.2.3, где `Thread.CurrentPrincipal` является окружающим контекстом. Но этот паттерн подходит нам редко, так как он усложняет управление временем жизни. Например, совместное использование одного экземпляра `ICircuitBreaker` между многочисленными WCF-клиентами внезапно значительно усложняется.

Несмотря на все эти недостатки, привлекательность атрибутов аспектов заключается в том, что код аспекта должен быть реализован только в одном месте. В следующем подразделе будет показано, как можно использовать возможности пере-

хвата, которыми располагают контейнеры внедрения, для достижения этой цели без сильного связывания атрибутов аспектов.

9.3.2. Применение динамического перехвата

Вы уже видели, как декораторы могут применяться для реализации сквозных аспектов приложения. Использованная при этом техника написания кода удовлетворяла принципам SOLID, но нарушала принцип DRY. Из примеров данной главы это может показаться неочевидным, но реализация аспекта путем ручного создания декорирующих классов приводит к появлению большого количества повторяющегося кода.

Повторяемость кода декораторов

Примеры в подразделах 9.1.1 и 9.2.1 демонстрируют только методы-представители, так как остальные методы реализуются таким же образом, и я не хочу выкладывать многостраничную распечатку с практически идентичным кодом, так как это уведет нас далеко от предмета разговора. Листинг 9.7 демонстрирует, насколько похожи методы из `CircuitBreakerProductmanagementAgent`. В этом листинге показаны только два метода интерфейса `IProductManagementAgent`, но вы можете экстраполировать и представить себе, как выглядит остальная часть реализации.

Листинг 9.7. Нарушение принципа DRY

```
public void DeleteProduct(int productId)
{
    this.breaker.Guard();
    try
    {
        this.innerAgent.DeleteProduct(productId);
        this.breaker.Succeed();
    }
    catch (Exception e)
    {
        this.breaker.Trip(e);
        throw;
    }
}

public void InsertProduct(ProductEditorViewModel product)
{
    this.breaker.Guard();
    try
    {
        this.innerAgent.InsertProduct(product);
        this.breaker.Succeed();
    }
    catch (Exception e)
    {
        this.breaker.Trip(e);
```

```
            throw;
        }
    }
```

Поскольку вы уже видели метод `InsertProduct` в листинге 9.4, код этого примера предназначен только для иллюстрации повторяемости декораторов, используемых как аспекты. Единственное различие (1) между методами `DeleteProduct` и `InsertProduct` заключается в том, что каждый из них вызывает свой собственный метод декорированного агента.

Даже несмотря на то, что мы успешно делегировали реализацию «Прерывателя цепи» в отдельный класс, используя интерфейс `ICircuitBreaker`, данный код нарушает принцип DRY. Он кажется сравнительно постоянным, но тем не менее без него не обойтись. Каждый раз, когда вы захотите добавить новый член в декорируемый тип или когда захотите применить «Прерыватель цепи» к новой абстракции, вы должны написать такой же фрагмент кода.

Один из способов, который вы можете захотеть применить для устранения данной проблемы, это использование генераторов кода типа Text Template Transformation Toolkit (T4) из Visual Studio, но многие контейнеры предоставляют лучшее решение на основе использования динамических перехватов.

Автоматизация декораторов

Код в каждом методе в листинге 9.7 выглядит похожим на шаблон. Трудность при реализации «Декоратора» как аспекта заключается в проектировании такого шаблона, но после этого работа становится чисто механическим процессом с такой последовательностью действий.

1. Создать новый класс.
2. Наследовать из родительского интерфейса.
3. Реализовать каждого участника интерфейса, применяя разработанный шаблон.

Этот процесс настолько механистичен, что можно использовать специальный инструмент, чтобы автоматизировать его. Такой инструмент будет использовать рефлексию (Reflection) или другой подобный ей API для определения всех членов, которые должны быть реализованы, и затем применения шаблона ко всем участникам. Рисунок 9.10 демонстрирует, как эта процедура может быть применена с использованием образца (Template) T4[1]. T4 делает возможной автоматическую генерацию кода декораторов из образцов. Начальная точка — это прототип образца, который понимает базовую концепцию декоратора. Прототип образца содержит код генерации кода, который будет генерировать основу декорируемого класса, но он не определяет никакого кода аспекта. Из прототипа образца разрабатывается образец аспекта, который описывает, как специфический аспект (такой как «Прерыватель цепи») будет применяться, когда будет декорироваться любой интерфейс. Результатом этого является специализированный образец (`SomeAspect.tt`) для данного конкретного аспекта, кото-

[1] Прочитать подробнее о декораторах и образцах T4 можно в *Oleg Sych* How to use T4 to generate Decorator classes, 2007: www.olegsych.com/2007/12/how-to-use-t4-to-generate-decorator-classes/.

рый может использоваться для генерации декораторов из требуемых интерфейсов. Результатом является файл обычного кода (`SomeAspectDecorator.cs`), который компилируется обычным образом вместе с другими файлами кода.

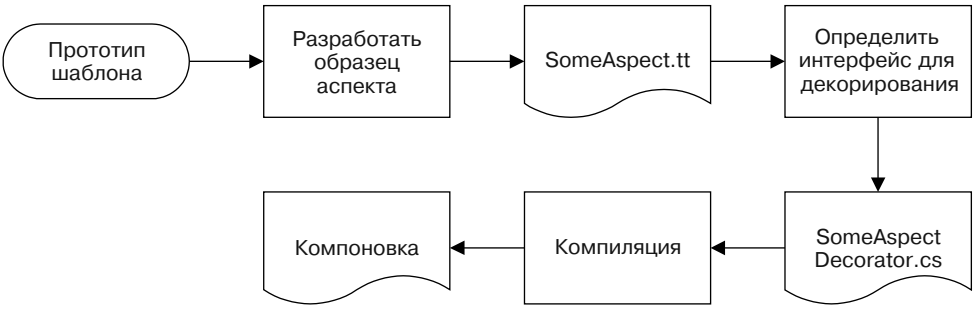


Рис. 9.10. Использованием образца T4

Хотя генераторы кода позволяют вам устранить симптомы повторяющегося кодирования, они все же оставляют огромное количество повторяющегося кода в конечном приложении. Если вы полагаете, что этот код обязателен[1], учтите, что увеличение объема кода влечет за собой возрастание издержек независимо, генерировался код вручную или автоматически.

Если даже вы не согласны с последним аргументом, вы остаетесь пока со статическим набором автоматически сгенерированных декораторов. Если вам понадобится новый декоратор для данной комбинации аспекта и абстракции, вы должны будете вручную добавить его класс. Этот класс будет генерироваться автоматически, но вы должны помнить о необходимости написать и подключить его. Более традиционный подход в данном случае неприменим.

Некоторые контейнеры предлагают нечто лучшее, чем автоматически генерируемый код: автоматически генерируемые классы. Может показаться, что различие невелико, но рассмотрим этот вариант.

Динамический перехват

Среди множества мощных возможностей фреймворка .NET имеется возможность динамически создавать типы. Помимо автоматической генерации кода во время разработки, можно писать код, который создает полнофункциональный класс во время исполнения. У такого класса нет файла исходного кода, он компилируется непосредственно из некоторой абстрактной модели.

Тем же способом, которым вы можете автоматизировать генерацию файлов исходного кода для декораторов, можно автоматизировать генерацию декораторов, которые будут создаваться во время выполнения приложения. Как показывает рис. 9.11, именно это нам позволяет получить динамический перехват. Перехватчик (Interceptor) — это фрагмент кода, реализующий аспект и интегрированный с контейнером.

---

[1] *Tim Ottinger* Code is a Liability, 2007: http://blog.objectmentor.com/articles/2007/04/16/code-is-a-liability.

Регистрация перехватчика в контейнере позволяет контейнеру динамически создавать и запускать декораторы, содержащие поведение аспекта. Эти классы существуют только во время исполнения.

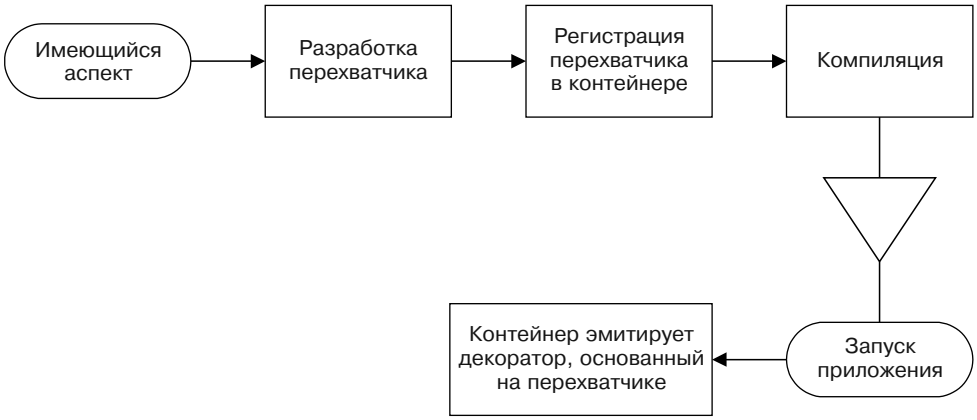


Рис. 9.11. Некоторые контейнеры позволяют определить аспекты как перехватчики

ПРИМЕЧАНИЕ

Не все контейнеры поддерживают перехваты времени исполнения; если вам нужна такая возможность, выберите подходящий контейнер.

Чтобы использовать динамические перехваты, вам все равно придется написать код, который реализует аспект. Этот код может совпадать с тем, который использовался в аспекте «Прерыватель цепи», что показано в листинге 9.7. После того как код аспекта будет реализован, придется известить контейнер о существовании нового аспекта и о том, при каких условиях он должен применяться.

Во время исполнения контейнер будет автоматически создавать и запускать новые классы в работающем `AppDomain`, основанном на зарегистрированных аспектах. Лучшее, что есть в данном подходе, — это возможность применения традиционной конфигурации для определения способа использования аспектов. Кроме того, можно применять разные соглашения в различных приложениях (например, хотя вы можете совместно использовать множество библиотек, у вас могут существовать разные стратегии обработки ошибок для WPF- и PowerShell-приложений).

ПРИМЕЧАНИЕ

В аспектно-ориентированном программировании соглашение, соотносящее аспекты с классами и участниками, называется срезом (Pointcut).

Достаточно теории — давайте рассмотрим пример.

9.3.3. Пример: перехват с Windsor

Имеющие повторяющийся код «Прерыватель цепи» и обрабатывающие ошибки аспекты из подразделов 9.2.1 и 9.2.2 отлично подходят для реализации динамического перехвата. Рассмотрим в качестве примера, как можно создать соответству-

ющий принципам DRY и SOLID код с использованием возможностей перехвата, предоставляемых контейнером `Castle Windsor`[1].

ПРИМЕЧАНИЕ

Я мог бы выбрать другой контейнер (не Castle Windsor), но не любой из существующих. Некоторые контейнеры поддерживают перехваты, тогда как другие нет — в части 4 описываются возможности распространенных контейнеров.

В этом примере вы реализуете и регистрируете перехватчики как для обработки ошибок, так и для «Прерывателя цепи». Добавление аспекта в `Windsor` — это трехэтапный процесс, как показано на рис. 9.12.

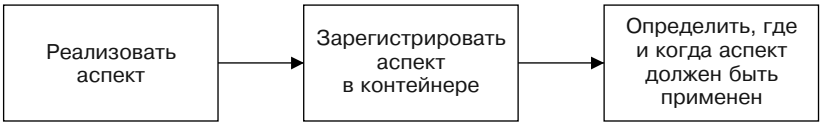


Рис. 9.12. Три шага процесса добавления аспекта в Windsor

Эти шаги должны выполняться для обоих аспектов в этом примере. Обработка ошибок более проста в реализации, так как не использует зависимости; давайте начнем с нее.

Реализация перехватчика обработки исключительных ситуаций

Реализация перехватчика для `Windsor` требует, чтобы мы создали интерфейс `IInterceptor`, имеющий только один метод. Листинг 9.8 показывает, как реализовать стратегию обработки исключительных ситуаций, подобную представленной в листинге 9.5. Но, в отличие от листинга 9.5, здесь показан весь класс.

Листинг 9.8. Реализация перехватчика обработки исключительных ситуаций

```
public class ErrorHandlingInterceptor : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        try
        {
            invocation.Proceed();
        }
        catch (CommunicationException e)
        {
            this.AlertUser(e.Message);
        }
        catch (InvalidOperationException e)
        {
            this.AlertUser(e.Message);
        }
```

1 Реализация IInterceptor
2 Реализация аспекта
3 Вызов декорированного метода

[1] www.castleproject.org/.

```
    }

    private void AlertUser(string message)
    {
        var sb = new StringBuilder();
        sb.AppendLine("An error occurred.");
        sb.AppendLine("Your work is likely lost.");
        sb.AppendLine("Please try again later.");
        sb.AppendLine();
        sb.AppendLine(message);

        MessageBox.Show(sb.ToString(), "Error",
            MessageBoxButton.OK, MessageBoxImage.Error
    }
}
```

Отображение диалогового окна

Перехватчик должен быть наследником интерфейса `IInterceptor` (1), определенного в `Windsor`. Этот интерфейс имеет только один метод, который должен быть реализован, и делается это тем же самым кодом (2), который переиспользовался в реализации `ErrorHandlingProductManagementAgent`.

Единственное отличие от листинга 9.5 состоит в том, что, вместо делегирования вызова специфическому методу, нужно найти более общее решение, так как этот код должен использоваться для вызова практически любого метода. Вы указываете контейнеру `Windsor` передать вызов в декорированный элемент, для чего применяется метод `Proceed` (3) на входном параметре `invocation`.

Интерфейс `IInvocation` передается в метод `Intercept` как параметр, представляющий вызов метода. Он может, к примеру, представлять вызов метода `InsertProduct`. Метод `Proceed` является одним из ключевых членов этого интерфейса, поскольку он позволяет нам разрешать вызову переходить к следующей реализации в стеке.

Интерфейс `IInvocation` также дает вам возможность назначить возвращаемое значение до того, как передать вызов дальше, а также предоставить доступ к детальной информации о вызове метода. Из параметра `invocation` можно получить информацию об имени и значениях параметров метода, а также большое количество другой информации о текущем вызове метода.

Реализация перехватчика — это трудная задача. Следующий шаг более прост.

Регистрация перехватчика обработки исключительных ситуаций

Перехватчик должен быть зарегистрирован в контейнере, прежде чем его можно будет использовать. На этом шаге не конфигурируются правила, определяющие, как и когда перехватчик активируется (срез), здесь перехватчик только делается доступным как компонент.

ПРИМЕЧАНИЕ

Вы можете считать этот шаг церемонией для удовлетворения контейнера Windsor. Одна из причуд Windsor заключается в том, что каждый компонент должен быть явно зарегистрирован, даже когда он является конкретным типом, имеющим конструктор по умолчанию. Не все контейнеры внедрения зависимостей работают подобным образом, но у Windsor такое поведение заложено прямо в коде.

Регистрация класса `ErrorHandlingInterceptor` выполняется просто (контейнер является экземпляром `IWindsorContainer`):

```
container.Register(Component.For<ErrorHandlingInterceptor>());
```

Эта процедура не отличается от регистрации любого другого компонента в `Windsor`, и вы можете даже прибегнуть к основанному на соглашениях подходу для регистрации всех реализаций интерфейса `IInterceptor`, находящегося в конкретной сборке. Это похоже на код примера в разделе 3.2.

Последний этап в активации перехватчика — определение правил, когда и как он применяется. Но поскольку эти правила должны затрагивать еще и перехватчик `Circuit Breaker`, мы отложим рассмотрение этого шага до тех пор, как второй перехватчик также будет готов к использованию.

Реализация перехватчика Circuit Breaker

Перехватчик `Circuit Breaker` немного более сложен, так как он требует зависимость `ICircuitBreaker`. Но, как видно из листинга 9.9, эта проблема решается использованием стандартного внедрения конструктора. Когда осуществляется компоновка этого класса, `Windsor` считает его подобным любому другому компоненту: пока он может разрешить зависимость, все хорошо.

Листинг 9.9. Реализация перехватчика Circuit Breaker

```
public class CircuitBreakerInterceptor : IInterceptor
{
    private readonly ICircuitBreaker breaker;

    public CircuitBreakerInterceptor(
        ICircuitBreaker breaker)
    {
        if (breaker == null)
        {
            throw new ArgumentNullException(
                "breaker");
        }
        this.breaker = breaker;
    }

    public void Intercept(IInvocation invocation)
    {
          this.breaker.Guard();
        try
        {
            invocation.Proceed();
            this.breaker.Succeed();
        }
        catch (Exception e)
        {
            this.breaker.Trip(e);
```

1 Внедрение конструктора

2 Применить аспект

3 Вызов декорированного метода

```
            throw;
        }
    }
}
```

Класс `CircuitBreakerInterceptor` требует зависимость `ICircuitBreaker`, внедрение зависимости в `IInterceptor` ❶ осуществляется посредством внедрения конструктора, как и в любом другом сервисе.

Как видно из листинга 9.8, интерфейс `IInterceptor` реализуется путем применения шаблона ❷, основанного на предыдущей, повторяющейся реализации из листинга 9.4. Опять же, вместо вызова конкретного метода вы вызываете метод `Proceed` ❸, чтобы указать перехватчику, что работа должна быть продолжена в следующем компоненте из стека декоратора.

Теперь вы должны представлять себе формирование шаблона. Вместо повторения кода `Circuit Breaker` в каждом методе абстракции вы можете определить его лишь однажды, в перехватчике.

Еще необходимо зарегистрировать класс `CircuitBreakerInterceptor` в контейнере; для этого, поскольку он имеет зависимость, требуется не одна, а две строки кода.

Регистрация перехватчика Circuit Breaker

Перехватчик для обработки исключительных ситуаций требует только одной строки кода для регистрации, но поскольку `CircuitBreakerInterceptor` еще зависит от `ICircuitBreaker`, потребуется зарегистрировать еще и эту зависимость:

```
container.Register(Component
    .For<ICircuitBreaker>()
    .ImplementedBy<CircuitBreaker>()
    .DependsOn(new
    {
        timeout = TimeSpan.FromMinutes(1)
    }));
container.Register(Component.For<CircuitBreakerInterceptor>());
```

Вы соотносите интерфейс `ICircuitBreaker` с конкретным классом `CircuitBreaker`, который сам по себе требует параметра задержки в конструкторе.

Теперь, когда оба перехватчика готовы, остается только определить правила их активации.

Активация перехватчиков

Итак, перехватчики реализованы и зарегистрированы в контейнере `Windsor`, но нужно еще определить, когда они будут активироваться. Если не сделать этого, перехватчики останутся просто пассивными регистрациями в контейнере, которые даже не будут вызваны.

Этот этап может рассматриваться как эквивалент применения атрибутов аспекта. Если мы применим гипотетический атрибут `[CircuitBreaker]` к методу, мы присоединим аспект `Circuit Breaker` к этому методу. Определение и применение специальных атрибутов является одним из способов, которым мы можем активировать перехватчики `Windsor`, но у нас имеются и другие, более оптимальные способы сделать это.

Самым гибким способом является реализация и регистрация интерфейса IModelInterceptorsSelector. Так мы сможем написать код, который решит, какой перехватчик следует применять к каждому типу или члену. Поскольку мы можем написать код любой сложности, мы можем применять наши аспекты значительно более традиционным способом.

В листинге 9.10 вы используем простую реализацию такого среза.

Листинг 9.10. Реализация среза

```
public class ProductManagementClientInterceptorSelector :
    IModelInterceptorsSelector
{
    public bool HasInterceptors(ComponentModel model)
    {
        return typeof(IProductManagementAgent)
            .IsAssignableFrom(model.Service);
    }

    public InterceptorReference[]
        SelectInterceptors(ComponentModel model,
            InterceptorReference[] interceptors)
    {
        return new[]
        {
            InterceptorReference
                .ForType<ErrorHandlingInterceptor>(),
            InterceptorReference
                .ForType<CircuitBreakerInterceptor>()
        };
    }
}
```

❶ Применить перехватчики к IProductManagementAgent

❷ Вернуть перехватчики

Интерфейс IModelInterceptorsSelector основан на шаблоне Tester-Doer. Windsor будет прежде всего вызывать метод HasInterceptors для получения информации о том, имеются ли у инициализируемого компонента какие-нибудь перехватчики. Если они имеются, вы отвечаете утвердительно, если компонент реализует интерфейс IProductManagementAgent ❶, но в этом месте можно написать и значительно более сложный код, если вы хотите реализовать более эвристический алгоритм.

Когда метод HasInterceptors возвращает значение true (истинно), будет вызван метод SelectInterceptors. Этот метод возвращает ссылки на уже зарегистрированные перехватчики ❷. Обратите внимание, что возвращаются не экземпляры перехватчиков, а ссылки на тех из них, которые вы уже реализовали и зарегистрировали.

Это позволяет контейнеру Windsor автоматически подсоединить любые перехватчики, которые сами по себе могут иметь зависимости (такие как CircuitBreakerInterceptor из листинга 9.9).

И еще, только представьте! Нужно еще зарегистрировать в контейнере класс ProductManagementClientInterceptorSelector. Это делается немного иначе, но также в одной строке:

```
container.Kernel.ProxyFactory.AddInterceptorSelector(
    new ProductManagementClientInterceptorSelector());
```

Это окончательно активизирует перехватчики, так что когда ваше приложение будет работать на платформе Windsor, они автоматически будут появляться при необходимости.

Можно подумать, что такое многостраничное пошаговое руководство по перехватчикам `Windsor` выглядит достаточно сложным, но вам следует учитывать несколько моментов:

- вы реализовали два перехватчика, а не один;
- я использовал часть кода из предыдущих примеров, чтобы показать, как он вписывается сюда. Независимо от того, решили ли вы написать декораторы вручную, использовать ли фреймворк AOP или использовать динамический перехват, вам в любом случае будет нужно писать код, реализующий сам аспект.

Динамический перехват дает вам массу преимуществ. Он позволяет разрешать сквозные аспекты приложения, следуя принципам SOLID и DRY. Он дает возможность разработать по-настоящему слабо связанные аспекты, а также применить соглашения или сложную эвристику для определения, когда и где должен быть применен каждый аспект. Это максимальный уровень свободы и гибкости.

Вы можете беспокоиться о снижении производительности вследствие компиляции и создания специальных типов на лету, но, по моему опыту, контейнер `Windsor` делает такую работу лишь однажды и затем использует полученный тип для всех последующих вызовов. Я выполнил несколько неофициальных тестов и не выявил никакого заметного снижения производительности.

Еще одним поводом для беспокойства является дополнительный уровень косвенности. Можно справедливо заметить, что, применяя атрибуты аспектов, мы все же можем отслеживать основной метод, с которым работают аспекты, влияющие на поведение. В случае декораторов и, в частности, динамического перехвата такое отслеживание не получается. Начинающие разработчики могут потенциально пойти вразрез с этим наполовину магическим подходом, в результате чего они застрянут на несколько дней, пока кто-нибудь не поможет им, объяснив данную концепцию.

Это является реальной проблемой в некоторых организациях. Определитесь, как вы будете преодолевать это препятствие, если решите применять динамический перехват.

9.4. Резюме

Внедрение зависимостей отлично работает, когда оно реализуется на основе четко определенных принципов объектно-ориентированной разработки, таких как SOLID. В частности, поскольку внедрение зависимостей является слабо связанным по своей природе, становится возможным использовать паттерн «Декоратор» на основе принципов открытости/закрытости и единственной ответственности. Это ценно в различных ситуациях, поскольку позволяет поддерживать код ясным и хорошо организованным. Но особенно удачно данная черта проявляется, когда речь заходит о реализации сквозных аспектов приложений.

Сквозные аспекты приложений традиционно относятся к области аспектно-ориентированного программирования, но могут с успехом реализовываться с использованием внедрения зависимостей. Паттерн проектирования «Декоратор» является основным инструментом, позволяющим декорировать существующую функциональность в дополнительные слои поведения без изменения оригинального кода.

Однако несмотря на многие достоинства, основная проблема с реализацией декораторов заключается в том, что они являются многословными и содержат много повторов. Даже если мы будем следовать принципам SOLID, мы, в конце концов, получим нарушение принципа DRY, поскольку нам придется писать один и тот же код снова и снова для каждого участника каждого интерфейса, который мы захотим использовать в конкретном аспекте.

Атрибуты выглядят привлекательной альтернативой декораторам, поскольку позволяют добавлять аспекты намного более компактным способом. Но поскольку атрибуты компилируются вместе с кодом, который они декорируют, они являются сильно связанными и от их использования следует отказываться.

Некоторые контейнеры внедрения предоставляют более привлекательную альтернативу за счет способности динамического создания «Декораторов» во время исполнения. Такие динамические декораторы обеспечивают перехват, соответствующий принципам и SOLID, и DRY.

Следует отметить, что динамический перехват — это одна из возможностей контейнеров внедрения зависимостей, не имеющих прямого соответствия «во внедрении зависимостей для бедных». В части 3 вы видели, как реализуется компоновка объектов и управление временем жизни путем использования шаблонов, но когда дело доходит до перехвата, все, что мы получаем, это множество декораторов.

Хотя концепция декоратора сравнима с перехватом, переход от многих кодируемых вручную декораторов к единственному, разработанному в соответствии с принципом DRY перехватчику, является значительным достижением. Если первый подход приводит к обилию многократно повторяющегося вспомогательного кода, то второй позволяет реализовать сквозные аспекты приложения всего в нескольких строках кода с дополнительным бонусом в виде возможности задействовать традиционное применение аспектов.

Именно здесь, в конце части 3, вы узнали, в каких вопросах контейнеры внедрения безоговорочно оставляют «внедрение зависимостей для бедных» позади. Даже без перехвата контейнер намного лучше управляет сложными вопросами соотнесения абстракций с конкретными типами, а также управления их временем жизни; но когда рассматривается еще и перехват, победить такое сочетание становится невозможно.

На этой ноте мы счастливо оставляем «внедрение для бедных» в части 3 и переходим к рассмотрению конкретных контейнеров в части 4.

ЧАСТЬ 4

Контейнеры внедрения зависимостей

Предыдущие части этой книги были посвящены рассмотрению различных принципов и паттернов, определяющих внедрение зависимостей. Как говорилось в главе 3, контейнер внедрения зависимостей — это необязательный инструмент, который вы можете использовать для реализации большого количества общих модулей инфраструктуры, которые в противном случае пришлось бы разрабатывать с использованием «внедрения зависимостей для бедных».

На протяжении этой книги я старался минимально затрагивать тему контейнеров. Не считайте это рекомендацией в пользу «внедрения зависимостей для бедных». Я просто хотел продемонстрировать работу с чистым внедрением зависимостей, не рассматривая никаких (возможно, причудливых) API-контейнеров.

Нет особых причин тратить свое время на работу с «внедрением зависимостей для бедных», в то время когда в .NET доступно множество контейнеров. В этой части будет описано два распространенных контейнера. В каждой главе я предоставлю детальный обзор API одного из контейнеров с точки зрения паттернов и методов, рассмотренных в части 3. Мы также разберем многие другие особенности, которые могут немного озадачить новичков.

В этой части описаны контейнеры, которые называются Castle Windsor и StructureMap. Обратите внимание на то, что предложенные к рассмотрению контейнеры не являются идеальными. Множество других контейнеров также очень хороши, но этот критерий не является единственным.

Обе главы строятся по одному принципу. Вы можете ощущать дежа вю, когда будете читать одно и то же предложение несколько раз, но я считаю это преимуществом, поскольку вы сможете быстро находить похожие разделы в разных главах, если захотите сравнить, как одна особенность реализована в нескольких контейнерах.

В следующей таблице приведено сравнение некоторых контейнеров.

Контейнер ВЗ	**Преимущества**	**Недостатки**
Castle Windsor	Завершенность. Понимание паттерна «Декоратор». Типизированные фабрики. Доступна коммерческая поддержка	Местами причудливый API
StructureMap	Просто хорошо работает в большинстве случаев	Нет поддержки перехвата
Spring.NET	Поддержка паттерна перехвата. Подробная документация. Доступна коммерческая поддержка	Излишняя ориентация на XML. API не основан на соглашениях. Не поддерживаются специальные жизненные циклы. Ограниченное применение автоматического связывания
Autofac	Простой для понимания API. Доступна коммерческая поддержка	Нет поддержки перехвата. Частичная поддержка настройки специальных жизненных циклов
Unity	Поддержка перехвата. Хорошая документация. Последовательный API	Слабое управление жизненным циклом. API не основан на соглашениях

Контейнер ВЗ	Преимущества	Недостатки
MEF	Доступен в .NET 4/Silverlight 4 BCL. Коммерческая поддержка	Не является полноценным контейнером ВЗ. Нет поддержки XML. Нет поддержки конфигурирования в коде. API не основан на соглашениях. У пользователя нет возможности создать собственный жизненный цикл. Нет поддержки перехвата

Многие описанные здесь контейнеры являются проектами с открытым исходным кодом, их новые версии выходят достаточно часто. Информация, представленная в главе 4, была актуальна на момент написания книги, однако всегда старайтесь проверять более свежие источники.

Эти главы послужат вам для вдохновения. Если вы еще не выбрали предпочтительный для вас контейнер внедрения зависимостей, можете прочесть эти главы для того, чтобы сравнить их все. Вы можете также изучить лишь те несколько, которые вас заинтересовали.

10 Castle Windsor

Меню:

- знакомство с Castle Windsor;
- управление жизненным циклом;
- работа с несколькими компонентами;
- конфигурирование сложных API.

В предыдущих девяти главах мы рассмотрели паттерны и принципы, применяемые к внедрениям зависимостей вообще, но помимо нескольких примеров, мы пока не разбирались в деталях их применения в отдельных контейнерах. В этой главе вы узнаете, как все данные принципы работают в контейнере Castle Windsor; вам не нужно досконально знать материал предыдущих глав, чтобы получить максимальную выгоду от использования этого контейнера.

Castle Windsor — это второй по возрасту контейнер внедрения зависимостей для .NET. Он является частью большего проекта с открытым исходным кодом, известного как Castle Project (www.castleproject.org/), который предоставляет многоцелевые библиотеки, пригодные для многократного использования. Хотя Windsor и является частью проекта Castle Project, он может использоваться независимо от любого другого компонента этого проекта. В этой главе он будет рассматриваться как отдельный компонент.

Castle Windsor — не просто один из самых старых контейнеров внедрения зависимостей. Он также самый зрелый и, если верить нескольким совершенно ненаучным интернет-опросам, самым популярным. Хотя Castle Windsor довольно прост в освоении, он предлагает богатый и расширяемый API.

В этой главе мы подробно рассмотрим контейнер Castle Windsor. После ее прочтения вы будете знать достаточно, чтобы разу же приступить к использованию этого контейнера. Мы не будем изучать продвинутые сценарии расширения, вместо этого сфокусируемся на основных шаблонах использования. Структура этой главы показана на рис. 10.1, она является древовидной. В первом разделе вы познакомитесь с контейнером Castle Windsor и узнаете, как сконфигурировать и разрешить его компоненты. После изучения этого раздела вы можете читать остальные последовательно или независимо друг от друга. Последний раздел использует синтаксис и несколько методов, появлявшихся в том разделе, где описывалась работа с несколькими компонентами. Поэтому если вы решите пропустить предпоследний раздел, вам, возможно, потребуется вернуться назад и прочесть его.

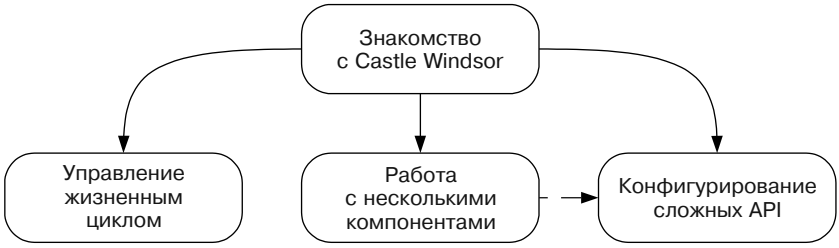


Рис. 10.1. Структура данной главы

Прочтение данной главы позволит вам начать работу с контейнером, а также справиться с наиболее распространенными проблемами, с которыми вы можете столкнуться при повседневном использовании этого контейнера. Глава не является полным обзором контейнера Castle Windsor — на это потребовалась бы целая книга.

Вы можете прочесть эту главу отдельно от прочих глав части 4 для того, чтобы научиться работать с контейнером Castle Windsor. Или же вы можете изучить эту часть до конца, чтобы сравнить контейнеры внедрения зависимостей. Цель этой главы заключается в том, чтобы показать, как Castle Windsor работает и реализует паттерны и принципы, описанные в предыдущих девяти главах.

10.1. Знакомство с Castle Windsor

В этом разделе вы узнаете, где можно достать Castle Windsor, что он собой представляет и то, как начать его использовать. Мы рассмотрим самые популярные конфигурации, а также поместим настройки конфигурации в пакеты, которые возможно использовать повторно. В табл. 10.1 предоставляется основная информация, которая, скорее всего, понадобится вам, чтобы начать работу с контейнером.

Таблица 10.1. Первый взгляд на контейнер Castle Windsor

Вопрос	**Ответ**
Где достать контейнер?	Перейдите по ссылке www.castleproject.org/castle/projects.html и нажмите подходящую ссылку в разделе Stable release (Стабильная версия)
Что нужно загрузить?	Вы можете скачать ZIP-архив со скомпилированными бинарными файлами. Вы также можете скачать текущую версию исходного кода и скомпилировать его самостоятельно. Бинарные файлы представляют собой DLL-библиотеки, которые вы можете поместить, куда пожелаете, и ссылаться на них из своего кода
Какие платформы поддерживаются?	.NET 3.5 SP1, .NET 4 Client Profile, .NET 4, Silverlight 3, Silverlight 4
Сколько это стоит?	Нисколько. Это проект с открытым исходным кодом под невирусной лицензией

Продолжение ⇩

Таблица 10.1 *(продолжение)*

Вопрос	Ответ
Где можно получить помощь?	Вы можете получить коммерческую поддержку от Castle Stronghold. Прочесть больше можно здесь: www.castlestringhold.com/services/support. Кроме того, поскольку Castle Windsor является проектом с открытым исходным кодом и процветающей экосистемой, вы, скорее всего (но необязательно), можете получить поддержку на официальном форуме, расположенном по адресу http://groups.google.com/group/castle-project-users. Портал Stack Overflow (http://stackoverflow.com/) также является местом, где можно задать вопрос
О какой версии эта глава?	2.5.2

Как показывает рис. 10.2, для работы с контейнером Castle Windsor вам нужно лишь следовать простому ритму: сконфигурировать контейнер путем добавления компонентов и последовательного их разрешения. В большинстве случаев следует создать экземпляр класса WindsorContainer и полностью сконфигурировать его перед тем, как начать разрешать компоненты из него. Компоненты разрешаются из сконфигурированного экземпляра

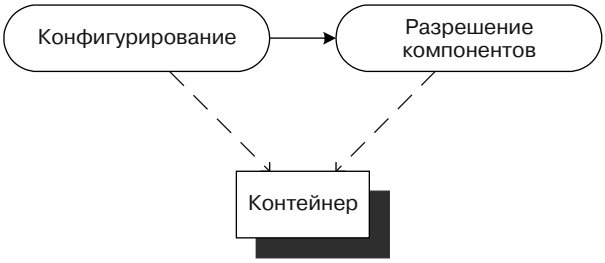


Рис. 10.2. Общий шаблон использования контейнера Castle Windsor

Как только вы закончите читать этот раздел, вы усвоите общий принцип использования контейнера Castle Windsor, а также сможете начать применять его в ситуациях, где все компоненты следуют подходящим паттернам внедрения зависимостей, например при внедрении конструктора (Constructor Injection). Мы начнем с простейшего сценария и рассмотрим, как можно разрешать объекты с помощью контейнера Windsor.

10.1.1. Разрешение объектов

Основная цель каждого контейнера внедрения зависимостей — разрешение объектов путем объединения их с зависимостями. Контейнер Castle Windsor предоставляет простой API, предназначенный для разрешения служб. Но перед тем, как разрешить какую-либо службу, она должна быть *зарегистрирована* в контейнере. Рассмотрим наиболее простой пример использования контейнера Castle Windsor:

```
var container = new WindsorContainer();
container.Register(Component.For<SauceBéarnaise>());
SauceBéarnaise sauce = container.Resolve<SauceBéarnaise>();
```

Перед тем как вы прикажете экземпляру класса `WindsorContainer` разрешить что-либо, вы обязаны зарегистрировать подходящие компоненты. В этом случае вы можете зарегистрировать компонент одного конкретного типа. Однако, как вы увидите дальше, гораздо чаще вы будете регистрировать преобразование от абстракции к конкретному типу.

Как только контейнер будет подходящим образом сконфигурирован, вы сможете разрешить тип `SauceBéarnaise`, чтобы получить экземпляр этого типа. Вам не нужно выполнять проверку на нулевое значение, поскольку контейнер `WindsorContainer` сгенерирует исключение в том случае, если не сможет выполнить автоподключение (Auto-wire), и вернет экземпляр требуемого типа.

ВНИМАНИЕ

Контейнер Windsor требует, чтобы все его компоненты были зарегистрированы, даже если они имеют конкретные типы. Это решение специфично для данного контейнера[1], но для других контейнеров внедрения зависимостей, возможно, это будет не так.

Первый пример по своему результату подобен прямому созданию экземпляра класса `SauceBéarnaise` с помощью ключевого слова `new`: пока мы не получили преимущества от использования зависимостей. Напомню, что внедрение зависимостей является лишь средством достижения цели, а цель — это слабое связывание. Чтобы достичь слабого связывания, вы должны преобразовывать абстракции к конкретным типам.

Преобразование абстракций к конкретным типам

Хотя иногда необходимо зарегистрировать конкретный класс, вы будете чаще регистрировать преобразование абстракции к конкретному типу. Это в конечном счете основная служба, выполняемая контейнерами внедрения зависимостей.

В следующем примере мы преобразовываем интерфейс `IIngridient` к конкретному классу `SauceBéarnaise`, что позволит вам успешно разрешить `IIngridient`:

```
var container = new WindsorContainer();
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
IIngredient ingredient = container.Resolve<IIngredient>();
```

Вместо регистрирования конкретного типа вы преобразовываете абстракцию к конкретному типу. Когда вы в дальнейшем обратитесь к экземпляру класса `IIngridient`, контейнер вернет экземпляр класса `SauceBéarnaise`.

Строго типизированный Fluent API, доступный в классе `Component` (`Castle.MicroKernel.Registration.Component`, а не `System.ComponentModel.Component`), помогает избегать ошибок конфигурации, поскольку метод `ImplementedBy` имеет общее ограничение.

[1] Для получения более подробного объяснения перейдите по ссылке http://docs.castleproject.org/Default.aspx?Page=FAQ&NS=Windsor&AspxAutoDetectCookieSupport=1.

Благодаря ему можно убедиться, что определенный вами тип реализует тип аргумента абстракции, определенный в методе `For`. Код предыдущего примера компилируется потому, что `SauceBéarnaise` реализует `IIngridient`.

Во многих случаях строго типизированный API — это все, что вам нужно, и поскольку он предоставляет желаемую проверку во время компиляции, вам следует использовать его при любой возможности. В то же время есть ситуации, когда вам понадобится способ, позволяющий разрешать слабо типизированные службы. Это также возможно.

Разрешение слабо типизированных служб

В некоторых случаях для разрешения типа нельзя написать универсальный код, поскольку невозможно знать наверняка, какой тип представляет собой абстракция во время разработки. Хорошим примером такой ситуации является фабрика `DefaultControllerFactory`, написанная с помощью ASP.NET MVC, ее мы рассмотрели в разделе 7.2. Интересующей нас частью этого класса является виртуальный метод `GetControllerInstance`:

```
protected internal virtual IController GetControllerInstance(
    RequestContext requestContext, Type controllerType);
```

В этом API нет строго типизированных общих конструкций. Вместо них предлагается работать с типом `Type`, а метод должен возвращать экземпляр класса `IController`. Класс `WindsorContainer` имеет также и слабо типизированную версию метода `Resolve`, позволяющего реализовать метод `GetControllerInstance`:

```
return (IController)this.container.Resolve(controllerType);
```

Обратите внимание на то, что в этом случае вы передаете методу `Resolve` аргумент `controllerType`. Поскольку слабо типизированная версия метода `Resolve` возвращает экземпляр класса `System.Object`, вы должны явно преобразовывать его к типу `IController` перед тем, как возвращать результат.

Независимо от того, какой из перегруженных методов `Resolve` вы используете, контейнер Windsor гарантирует, что он вернет экземпляр требуемого типа или сгенерирует исключение, если существуют зависимости, которые не могут быть соблюдены. Как только все требуемые зависимости будут соответствующим образом зарегистрированы, контейнер Windsor выполнит операцию автоматического подключения для требуемого типа, основываясь на его конфигурации.

В предыдущем примере `this.container` — это экземпляр класса `IWindsorContainer`. Чтобы разрешать требуемые типы, все типы и их зависимости должны быть зарегистрированы. Существует много способов сконфигурировать контейнер Windsor, следующий подраздел продемонстрирует наиболее распространенные.

10.1.2. Конфигурирование контейнера

Как мы говорили в разделе 3.2, есть несколько способов конфигурирования контейнера внедрения зависимостей, которые концептуально отличаются друг от друга. Они показаны на рис. 10.3. Конфигурация в коде (Code as configuration) — это строго типизированный явный способ. XML, с другой стороны, отличается позд-

ним связыванием, но при этом также является явным. Авторегистрация (Auto-registration) в отличие от них полагается на соглашения, которые могут быть как строго, так и слабо типизированными.

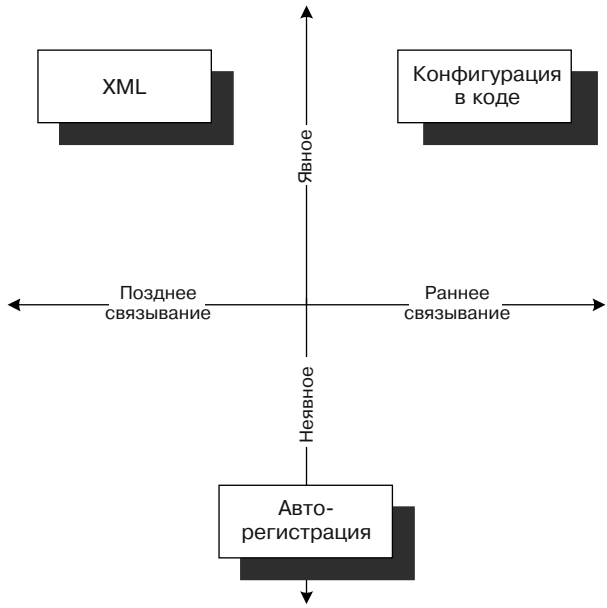


Рис. 10.3. Концептуально разные способы конфигурации

Как и прочие контейнеры внедрения зависимостей с богатой историей, Castle Windsor первоначально выполнял конфигурирование с помощью XML. Но прошло совсем немного времени, и многие команды разработчиков поняли, что определение регистрации типов с помощью XML очень ненадежно. В наше время предпочитается строго типизированное конфигурирование. Так, сконфигурировать контейнер возможно с помощью конфигурации в коде или же, что более эффективно, с использованием более согласованной авторегистрации (Auto-registration).

Контейнер Castle Windsor поддерживает все три подхода и даже позволяет объединять их в одном и том же контейнере. В этом отношении он предоставляет нам все, что только может понадобиться. В этом подразделе мы рассмотрим, как можно использовать каждый из этих трех типов конфигурирования контейнера.

Конфигурация в коде

В главе 3 вы видели примеры API конфигурации контейнера Castle Windsor в коде. Каждая регистрация инициируется с помощью метода `Register` и часто определяется с использованием Fluent API.

Контейнер `CastleWindsor` конфигурируется с помощью метода `Register`, который в качестве входного параметра принимает массив элементов типа `IRegistration`. На первый взгляд это кажется довольно абстрактным. Но вместо того, чтобы заставить нас определять, какую реализацию `IRegistration` следует использовать

в этот раз, контейнер Castle Windsor также предоставляет Fluent Registration API. Этот интерфейс позволяет создавать экземпляры класса `IRegistration` с помощью более понятного синтаксиса.

Чтобы использовать Fluent Registration API, мы назначим в качестве точки входа статический класс `Component`.

ВНИМАНИЕ

Не путайте класс Castle.MicroKernel.Registration.Component с классом System.ComponentModel.Component, находящимся в базовой библиотеке классов.

Как вы могли заметить ранее, наиболее простая из возможных регистраций — это регистрация конкретного типа:

```
container.Register(Component.For<SauceBéarnaise>());
```

Эта строка регистрирует класс `SauceBéarnaise` в контейнере, но в таком случае не выполняется никакого преобразования. Даже несмотря на то, что класс `SauceBéarnaise` реализует `IIngridient`, контейнер выбросит исключение, если вы укажете ему разрешить `IIngridient`:

```
container.Resolve<IIngredient>()
```

Чтобы осуществить этот более подходящий сценарий, вы должны преобразовать конкретный тип к абстракции:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

Обратите внимание на то, что теперь вы регистрируете интерфейс `IIngridient` вместо класса `SauceBéarnaise`. Это позволит вам разрешить экземпляр класса `IIngridient`, но, возможно, как это ни удивительно, вы потеряете возможность разрешать конкретный класс `SauceBéarnaise`. Это редко представляет проблему, когда ваш код слабо связан. Однако иногда, когда вам может понадобиться разрешать оба типа, вы можете определить эту возможность с помощью переопределения метода `For`:

```
container.Register(Component
    .For<SauceBéarnaise, IIngredient>());
```

В этой строке регистрируется компонент `SauceBéarnaise` и в то же время регистрируется интерфейс `IIngredient`. Это означает, что и `SauceBéarnaise`, и `IIngredient` зарегистрированы как разрешаемые типы. В обоих случаях реализация предоставляется классом `SauceBéarnaise`. Обратите внимание на то, что, когда используется подобное переопределение, вам не нужно неявно применять метод `ImplementedBy`.

Очевидно, вы можете зарегистрировать несколько типов с помощью успешных вызовов метода `Register`:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
container.Register(Component
```

```
    .For<ICourse>()
    .ImplementedBy<Course>());
```

Этот фрагмент кода регистрирует интерфейсы `IIngredient` и `ICourse`, а также преобразует их в конкретные типы. Регистрация одинаковых абстракций, выполненная несколько раз, может привести к интересным результатам:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

В этом примере интерфейс `IIngredient` регистрируется дважды. Если вы разрешите этот интерфейс, то получите экземпляр класса `Steak`. Первая регистрация всегда имеет приоритет, а последующие регистрации забываются. Контейнер Castle Windsor имеет сложную модель работы с множественными регистрациями, мы рассмотрим ее в разделе 10.3.

Во Fluent Registration API существуют и более продвинутые параметры, но все приложение возможно сконфигурировать и подобным образом. Однако чтобы удержаться от слишком явного обслуживания конфигурации контейнера, мы рассмотрим более традиционный подход — использование автоматической регистрации.

Автоматическая регистрация

Во многих случаях большинство регистраций будут похожими друг на друга. Такие регистрации тяжело обслуживать, а явная регистрация каждого компонента часто является контрпродуктивной.

Рассмотрим библиотеку, содержащую множество реализаций `IIngridient`. Вы можете зарегистрировать каждый класс отдельно, но в результате все равно будет множество похожих вызовов метода `Register`. Более того, всякий раз, когда вы будете добавлять реализацию `IIngredient`, вы должны будете явно зарегистрировать ее внутри контейнера, если хотите, чтобы она была доступна для дальнейшей работы. Было бы более продуктивно поместить все реализации `IIgredient` в заданную сборку, которая будет зарегистрирована.

Это становится возможным благодаря использованию статического класса `AllTypes`, роль которого похожа на роль класса `Component`. Мы можем использовать один из этих методов для проверки сборок на наличие типов, соответствующих определенным критериям. Это поможет зарегистрировать все реализации `IIngredient` за один раз:

```
container.Register(AllTypes
    .FromAssemblyContaining<Steak>()
    .BasedOn<IIngredient>());
```

Класс `AllTypes` предоставляет множество методов, позволяющих нам указать на определенную сборку, но я нахожу общий метод `FromAssemblyContaining` довольно четким: предоставьте ему определенный тип в качестве параметра, и он будет

использовать сборку, содержащую этот тип. Существуют также и другие методы, позволяющие нам предоставлять сборки иными способами.

В предыдущем примере мы, безусловно, регистрировали все реализации интерфейса `IIngredient`, но контейнер позволяет определить критерий выборки — либо сузить выбор, либо выбирать что-либо, отличное от интерфейсов и базовых классов. Рассмотрим пример регистрации, основанной на соглашении, в ходе которой добавляются все классы, чьи имена начинаются на `Sauce`, и они регистрируются к реализуемым ими интерфейсам:

```
container.Register(AllTypes
    .FromAssemblyContaining<SauceBéarnaise>()
    .Where(t => t.Name.StartsWith("Sauce"))
    .WithService.AllInterfaces());
```

Обратите внимание на то, что вы можете передать методу `Where` предикат, с помощью которого он будет фильтровать имена типов. Любой тип, чье название начинается с `Sauce`, будет выбран из сборки, содержащей класс `SauceBéarnaise`. Свойство `WithService` позволяет вам задавать правило для регистрации типа; в этом случае вы регистрируете все типы к реализуемым ими интерфейсам.

Применение подобного способа регистрации поможет вам перейти от строгой типизации к работе, в ходе которой нет необходимости следить за безопасностью типов. Примеры, подобные предыдущему, будут компилироваться, но нет гарантии, что будет зарегистрирован хотя бы один тип. Это зависит от того, удовлетворяют ли типы критериям выборки. Вы можете переименовать все классы, чьи имена начинаются с `Sauce` (Соус), во что-нибудь еще, и тогда вы останетесь совсем без соусов!

В классе `AllTypes` существует метод, принимающий в качестве входного параметра имя сборки. Он использует Fusion (движок, загружающий сборки в фреймворке .NET), чтобы найти соответствующую сборку. Объединив сборку с поздним связыванием и предикат без типа, вы углубитесь в область позднего связывания. Этот прием может быть оправдан при реализации надстроек, поскольку контейнер Castle Windsor также может просматривать все сборки в каталоге.

Еще одним способом регистрации надстроек и прочих служб с поздним связыванием является конфигурация с помощью XML.

XML-конфигурирование

Конфигурация с помощью XML — это отличная альтернатива, если вам нужно сменить конфигурацию без перекомпиляции приложения.

СОВЕТ

Используйте конфигурацию с помощью XML только для тех типов, которые вам нужно изменить без перекомпиляции приложения. Для прочих типов применяйте автоматическую регистрацию или конфигурацию в коде.

Мы можем встроить XML-конфигурацию в обычные конфигурационные файлы .NET или импортировать XML из специализированных файлов. Как и всегда при работе с контейнером Castle Windsor, ничто не произойдет, пока мы явно не укажем контейнеру выполнить это, поэтому необходимо также учитывать, хотим ли мы загружать конфигурацию из XML.

Существует несколько способов сделать это, но рекомендуется использовать метод `Install` (об установщиках мы поговорим подробнее в подразделе 10.1.3):

```
container.Install(Configuration.FromAppConfig());
```

Метод `FromAppConfig` возвращает экземпляр класса `ConfigurationInstaller`, который считывает XML-конфигурацию для контейнера Castle Windsor из конфигурационного файла и преобразовывает ее в объекты, понятные контейнеру.

Чтобы подключить конфигурационный файл контейнера Castle Windsor, сначала необходимо добавить в код раздел конфигурации:

```
<configSections>
  <section name="castle"
           type="Castle.Windsor.Configuration.AppDomain
              ➥.CastleSectionHandler, Castle.Windsor" />
</configSections>
```

Этот фрагмент кода позволяет добавить конфигурацию контейнера в конфигурационный файл. Рассмотрим простой пример, преобразующий интерфейс `IIgredient` к классу `Steak`:

```
<castle>
  <components>
    <component id="ingredient.sauceBéarnaise"
               service="IIngredient"
               type="Steak"/>
  </components>
</castle>
```

Обратите внимание на то, что вам не нужно передавать имя типа сборки ни для службы, ни для класса. До тех пор пока имя каждой загруженной сборки уникально, они будут корректно разрешаться — но даже если у вас возникнут конфликты имен, вы по-прежнему можете использовать имена типов сборок.

Очевидно, вы можете добавить столько компонентов, сколько вам нужно. Экземпляр класса `Configuration-installer` преобразует XML-конфигурацию к объектам регистрации, которые конфигурируют контейнер, и вы можете последовательно разрешить все сконфигурированные типы.

Конфигурирование с помощью XML — это хороший вариант, когда вам нужно изменить конфигурацию одного или более компонентов без перекомпиляции приложения. Однако поскольку этот способ довольно неустойчив, вы не должны злоупотреблять им и применять в качестве основного способа конфигурации автоматическую регистрацию или конфигурацию в коде.

СОВЕТ

Вы помните, что учитывается именно первая конфигурация типа? Вы можете использовать это поведение для того, чтобы переписать конфигурацию XML, записанную вручную. Чтобы сделать это, вы должны создать экземпляр класса ConfigurationInstaller перед регистрацией каких-либо компонентов.

В этом разделе мы рассмотрели основные конфигурационные API контейнера Castle Windsor. Хотя возможно написать один большой блок неструктурированного кода, лучше всего разбить конфигурацию на модули. Для этих целей можно использовать установщики контейнера Windsor.

10.1.3. Размещение конфигурации

Иногда есть необходимость разбить логику конфигурации на группы, которые можно переиспользовать. И даже когда переиспользование не является основной целью, целесообразно немного структурировать код, особенно если нужно сконфигурировать большое и сложное приложение.

С помощью контейнера Castle Windsor возможно разместить конфигурацию в установщике (Installer). Установщик представляет собой класс, который реализует интерфейс `IWindsorInstaller`:

```
public interface IWindsorInstaller
{
    void Install(IWindsorContainer container, IConfigurationStore store);
}
```

Все способы конфигурирования, которые вы изучили, также могут быть применены и внутри установщика. В листинге 10.1 показан установщик, который регистрирует все реализации класса `IIgredient`.

Листинг 10.1. Реализация загрузчика для контейнера Castle Windsor

```
public class IngredientInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container,
        IConfigurationStore store)
    {
        container.Register(AllTypes
            .FromAssemblyContaining<Steak>()
            .BasedOn<IIngredient>());
    }
}
```

Класс `IngredientInstaller` реализует интерфейс `IWindsorInstaller`, используя рассмотренный ранее API, который регистрировал все реализации `IIngredient`.

Чтобы зарегистрировать установщик, вызовите метод `Install`:

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

Хотя возможно вызвать метод `Install` несколько раз, в документации контейнера Castle Windsor рекомендуется выполнять все операции конфигурирования при одном вызове метода `Install`[1]. Метод `Install` принимает в качестве входного параметра массив экземпляров класса `IWindsorInstances`:

```
public IWindsorContainer Install(params IWindsorInstaller[] installers);
```

СОВЕТ

Установщики контейнера Castle Windsor позволяют вам размещать в пакетах и структурировать конфигурационный код вашего контейнера. Используйте его вместо строкового конфигурирования, это улучшит читаемость корня дерева зависимостей.

Помимо всех преимуществ, которые дают вашему коду установщики, разработчики контейнера Castle Windsor оптимизируют контейнер так, чтобы его API строились вокруг установщиков. Это устоявший-

[1] http://stw.castleproject.org/Windsor.Installers.ashx.

ся и рекомендованный способ конфигурирования контейнера Castle Windsor, подходящий ему в большей степени, нежели прочим контейнерам.

Вы также можете определить один или несколько установщиков с помощью XML и загружать конфигурационный файл описанным ранее способом:

```
<installers>
  <install type="IngredientInstaller" />
</installers>
```

При использовании установщиков вы сможете сконфигурировать экземпляр класса `WindsorContainer` любым способом: подойдет и конфигурирование в коде, и автоматическая регистрация, и регистрация с помощью XML. Вы также можете объединить все три подхода. Как только контейнер сконфигурируется, вы можете указать ему разрешить службы.

В этом разделе мы рассмотрели контейнер внедрения зависимостей Castle Windsor в общих чертах и продемонстрировали его основные механизмы: конфигурирование контейнера и последовательное разрешение его служб. Разрешение служб может быть с легкостью выполнено с помощью единственного вызова метода `Resolve`, поэтому вся сложность заключается лишь в конфигурации контейнера. Это может быть сделано несколькими различными способами, включая необходимый код и XML. До этого момента мы изучали лишь самые простые API, впереди нас ждут более продвинутые техники. Одна из наиболее важных тем — управление жизненным циклом компонента.

10.2. Управление жизненным циклом

В главе 8 мы обсудили управление жизненным циклом, включая наиболее распространенные стили жизненных циклов, такие как Singleton и Transient. Контейнер Castle Windsor поддерживает многие виды жизненных циклов и позволяет конфигурировать жизненный цикл всех служб. Жизненные циклы, показанные в табл. 10.2, доступны как часть API.

Таблица 10.2. Стили жизненных циклов контейнера Castle Windsor

Название	Комментарии
Singleton	Стиль жизненных циклов, используемый контейнером Castle Windsor по умолчанию
Transient	Новый экземпляр создается каждый раз, но он по-прежнему отслеживается контейнером
PerThread	Новый экземпляр создается для каждого потока
PerWebRequest	Требует регистрации в web.config (см. подраздел 10.2.2)
Pooled	Часто целесообразно конфигурировать размер пула (см. подраздел 10.2.2)
Специальный	Создайте ваш собственный стиль жизненных циклов (см. подраздел 10.2.3)

Некоторые встроенные стили жизненных циклов полностью эквивалентны общим шаблонам стилей жизненных циклов, описанным в главе 8. Это верно для стилей жизненных циклов Singleton и Transient, поэтому я не буду останавливаться на них подробно в этой главе.

ВНИМАНИЕ

Стиль жизненных циклов, используемый контейнером Castle Windsor по умолчанию, — это Singleton. Этим он отличается от множества других контейнеров. Как говорилось в главе 8, Singleton — это наиболее эффективный, хотя и не всегда безопасный стиль. Контейнер Castle Windsor ставит эффективность превыше безопасности.

В этом разделе вы увидите, как можно сконфигурировать стили жизненных циклов для компонентов и использовать некоторые особенные стили, такие как PerWebRequest и Pooled. Мы также рассмотрим реализацию пользовательского жизненного цикла для того, чтобы показать, что мы не ограничены лишь встроенными жизненными циклами. После прочтения этого раздела вы сможете использовать стили жизненных циклов контейнера Castle Windsor в вашем собственном приложении.

Начнем с рассмотрения конфигурирования стилей жизненных циклов для компонентов.

10.2.1. Конфигурирование стилей жизненных циклов

В этом разделе мы вспомним, как управлять стилем жизненных циклов компонентов с помощью контейнера Castle Windsor. Стиль жизненных циклов конфигурируется во время регистрации компонента, поэтому для достижения наших целей доступны все те же способы — код или XML. Рассмотрим каждый из них.

Конфигурирование стилей жизненных циклов в коде

Стиль жизненных циклов конфигурируется с использованием Fluent Registration API, с помощью которого регистрируются компоненты. Применять его довольно просто:

```
container.Register(Component
    .For<SauceBéarnaise>()
    .LifeStyle.Transient);
```

Обратите внимание на то, что вы определяете стиль жизненного цикла, используя свойство `Lifestyle`. В этом примере значение этого свойства задается равным `Transient`, поэтому всякий раз, когда разрешается класс `SauceBéarnaise`, будет возвращаться его новый экземпляр.

Вы по-прежнему можете определить стиль Singleton, даже несмотря на то, что он является стилем по умолчанию. Следующие два примера полностью одинаковы:

```
containér.Régistér(Componént
    .For<SaucéB arnaisé>()
    .LiféStylé.Singléton);
```

```
containér.Régistér(Componént
For<SaucéB arnaisé>());
```

Поскольку Singleton является стилем жизненных циклов, применяемым по умолчанию, вам не нужно явно определять его, но вы можете сделать это, если захотите.

Конфигурирование стилей жизненных циклов, как и конфигурирование компонентов, может быть выполнено как в коде, так и в XML.

Конфигурирование стилей жизненных циклов в XML

В подразделе 10.1.2 вы увидели, как можно сконфигурировать компоненты, используя XML. Однако мы не указывали никаких жизненных циклов. Как и в случае конфигурирования компонентов с использованием Fluent Registration API, стиль жизненного цикла Singleton применяется по умолчанию, но вы можете явно указать другой стиль, если это вам нужно:

```
<component id="ingredient.sauceBéarnaise"
           service="IIngredient"
           type="Steak"
           lifestyle="transient" />
```

При сравнении с примером в подразделе 10.1.2 можно заметить, что появился лишь атрибут `lifestyle`. Как вы можете видеть, определение стилей жизненных циклов легко выполнить как в коде, так и в XML.

Разрешение компонентов

Как мы говорили в подразделе 8.2.2, очень важно разрешать объекты, когда мы заканчиваем работу с ними. Это можно легко сделать с помощью вызова метода `Release`:

```
container.Release(ingredient);
```

Вызов этого метода разрешит экземпляр класса, переданный ему в качестве параметра (в предыдущем примере этим параметром являлась переменная `ingredient`), а также все его зависимости, чей жизненный цикл уже закончился. Так и есть, если экземпляр разрешаемого класса имеет зависимость со стилем жизненного цикла Transient, то эта зависимость будет разрешена (и, скорее всего, удалена), а зависимость Singleton останется в контейнере.

СОВЕТ

Контейнер Castle Windsor отслеживает все, даже Transient-компоненты, поэтому важно не забывать удалять все разрешенные экземпляры классов во избежание утечек памяти.

Явно удаляйте все, что было разрешено.

Не забывайте удалять и сам контейнер при завершении работы приложения. Это действие удалит все компоненты Singleton и позволит убедиться в том, что приложение не оставит после себя мусора в памяти.

Теперь обратим внимание на стили жизненных циклов, для которых требуется производить немного больше конфигурационных манипуляций, нежели запись одного утверждения.

10.2.2. Использование продвинутых стилей жизненных циклов

В этом подразделе мы рассмотрим два стиля жизненных циклов, применяемых в контейнере Castle Windsor, которые требуется конфигурировать дольше, чем написать одно простое утверждение: Pooled и PerWebRequest.

Использование стиля Pooled

В подразделе 8.3.5 мы рассмотрели общую концепцию стиля Pooled, в этом разделе вы увидите, как использовать его реализацию в контейнере Castle Windsor. У стиля Pooled в контейнере Castle Windsor имеется размер пула по умолчанию, но поскольку оптимальный размер пула зависит от самых различных обстоятельств, вам следует явно задавать этот размер. Вы можете указать контейнеру использовать пул с размерами по умолчанию. Это делается с помощью тех же команд, которые применялись бы для любого иного стиля жизненных циклов:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.Pooled);
```

Однако мы ничего не делаем с размерами пула. Хотя я не смог найти никакой *документации*, в которой был бы указан размер пула по умолчанию, в исходном коде версии контейнера 2.1.1 указано, что его стандартный размер — 5, а максимальный — 15. Мне кажется, что эти значения произвольны, что является еще одной причиной определять размер пула явно.

Для явного задания размера пула вы можете использовать метод `PooledWithSize`:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.PooledWithSize(10, 50));
```

В этом примере исходный размер пула устанавливается равным 10, а максимальный — 50. Пулы контейнера Castle Windsor имеют два конфигурационных значения — исходный размер и максимальный размер, управляющие, соответственно, исходным и максимальным размеров пула. Иногда поведение контейнера может быть удивительным в спорных ситуациях. На рис. 10.4 показано, как изменяется размер пула во время существования контейнера. Даже несмотря на то, что исходный размер пула равен 3, он остается пустым до того, как разрешится первый экземпляр класса. С этого момента создаются все три экземпляра, составляющие исходный размер, и один из них немедленно становится используемым. Как только экземпляр освобождается, он возвращается в пул. Пул увеличивается в размерах, если необходимо создать больше экземпляров, чем количество, составляющее минимальный размер пула. Обратите внимание на то, что возможно создать больше экземпляров, чем позволяет максимальный размер пула, но в этом случае создадутся избыточные экземпляры, которые не удалятся при освобождении.

Как только экземпляр разрешается из пула, он отмечается как *используемый* (in use). Поскольку он будет оставаться в таком состоянии до тех пор, пока не будет явно удален из контейнера, важно помнить, что объекты необходимо удалять из контейнера, как только вы закончили с ними работу. Это позволит контейнеру полностью удалить их:

```
container.Release(ingredient);
```

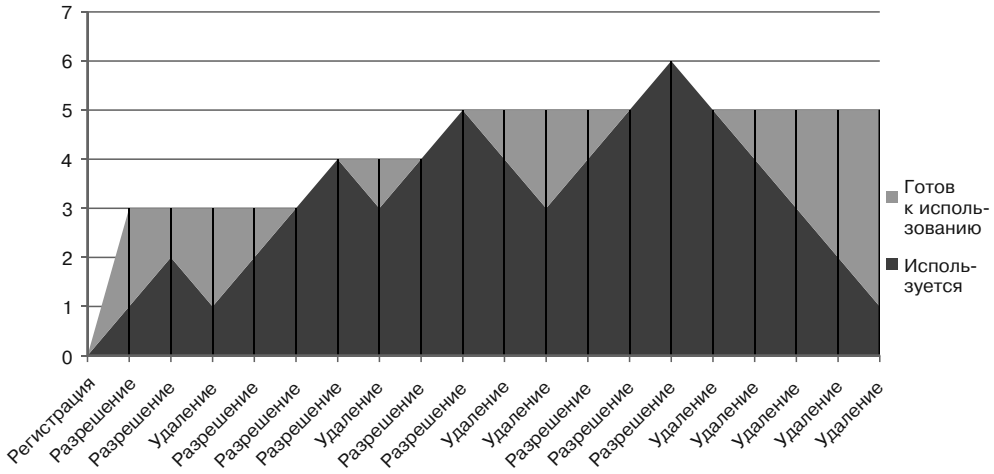


Рис. 10.4. Изменение размера пула с исходным размером, равным 3, и максимальным размером, равным 5

ВНИМАНИЕ

Поведение контейнера в моменты, когда пул заполнен, может вас удивить. Вместо генерации исключения или блокирования вызова создаются *избыточные* экземпляры. Они выбрасываются после использования вместо удаления.

Хотя этот стиль является немного более продвинутым, чем Singleton или Transient, он также прост в использовании. Единственное дополнительное усилие, которое вам нужно выполнить, — передать два дополнительных числа конфигурирования размера пула. Стиль PerWebRequest несколько отличается от остальных, и его несколько сложнее конфигурировать.

Использование стиля PerWebRequest

Как понятно из названия, стиль жизненных циклов PerWebRequest (На веб-запрос) создает новый экземпляр класса при веб-запросе. Его определение не сложнее, чем жизненного стиля Transient:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.PerWebRequest);
```

Однако если мы попробуем использовать его без дальнейшей конфигурации, то получим следующее исключение:

Looks like you forgot to register the HTTP module Castle.MicroKernel.Lifestyle.PerWebRequestLifestyleModule

Add '<add name=»PerRequestLifestyle» type=»Castle.MicroKernel.Lifestyle.PerWebRequest-LifestyleModule, 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>

(Кажется, вы забыли зарегистрировать HTTP-модуль Castle.MicroKernel.Lifestyle.PerWeb-RequestLifestyleModule

Добавьте строку '<add name=»PerRequestLifestyle» type=»Castle.MicroKernel.Lifestyle.PerWebRequest-LifestyleModule, Castle.Windsor» />' в раздел <httpModules> файла web.config. Если у вас запущен IIS7 в интегрированном режиме, вам необходимо добавить эту строку в раздел <modules> под <system.webServer>)

Это сообщение об ошибке — просто образцовое. Оно указывает вам все, что нужно сделать.

Важно отметить, что стиль PerWebRequest использует HTTP-модуль для определения того, какой веб-запрос сейчас выполняется. Одним из следствий является тот факт, что нам необходимо регистрировать HTTP-модуль так, как я описывал. Другое следствие заключается в том, что этот стиль жизненных циклов работает только с веб-запросами. Если мы попробуем использовать его в приложениях другого типа, мы получим то же исключение, что и в предыдущем примере.

Оба стиля, Pooled и PerWebResquest, требуют ненамного больше работы для своего использования, чем простое утверждение, но они так же просты и в конфигурировании, и в использовании. Встроенные стили контейнера Castle Windsor предоставляют обширный и полезный набор стилей жизненных циклов, подходящих для большинства ситуаций. Но если ни один из них не подходит для ваших нужд, вы можете написать свой стиль жизненных циклов.

10.2.3. Разработка специальных стилей жизненных циклов

В большинстве случаев встроенные стили жизненных циклов контейнера Castle Windsor должны удовлетворить любые потребности, но если вам необходимо что-то особенное, вы можете написать собственный стиль.

В этом подразделе вы узнаете, как это сделать. Сначала мы взглянем на соответствующий шов, обеспечивающий такую возможность, а затем рассмотрим пример.

Понимание API жизненных стилей

Вы можете создать собственный стиль жизненных циклов, реализовав интерфейс `ILifestyleManager`:

```
public interface ILifestyleManager : IDisposable
{
    void Init(IComponentActivator componentActivator,
        IKernel kernel, ComponentModel model);
    bool Release(object instance);
    object Resolve(CreationContext context);
}
```

Одно из довольно странных требований к реализации интерфейса `ILifestyleManager` — это наличие конструктора по умолчанию. Да-да, внедрение конструктора здесь запрещено. Вместо этого у нас есть один из относительно редких случаев

внедрения методов. Вызывается метод `Init`, предоставляя среди прочих параметров экземпляр класса `IKernel`, который мы можем использовать как локатор сервисов (Service Locator). Это точно мне не по душе, и как только мы рассмотрим код примера, вы увидите, как это усложняет реализацию по сравнению с использованием внедрения конструктора.

Другими методами интерфейса `ILifestyleManager` являются `Resolve` и `Release`, но мы будем использовать их скорее как ловушки (Hooks), а не предоставлять собственные реализации разрешения и освобождения компонентов — это задача для `IComponentActivator`, решаемая в методе `Init`. На рис. 10.5 показано, что мы должны использовать эти методы только для того, чтобы перехватывать вызовы методов `Resolve` и `Release`, благодаря чему мы сможем контролировать жизненный цикл каждого компонента. Ожидается, что реализация интерфейса `ILifestyleManager` будет применять `IComponentActivator` для создания объектов. Поскольку интерфейс `ILifestyleManager` находится в середине структуры, у него есть возможность перехватить каждый вызов и выполнить свою собственную логику. Он может свободно переиспользовать объекты вместо работы с `IComponentActivator`.

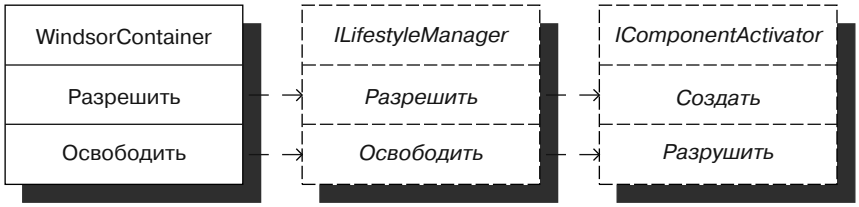


Рис. 10.5. Интерфейс ILifestyleManager работает как перехватчик, который вызывается вместо базового IComponentActivator

Контейнер Castle Windsor предоставляет реализацию по умолчанию интерфейса `ILifestyleManager`, представленную в виде класса `AbstractLifestyleManager`. Этот класс реализует интерфейс и содержит неплохую стандартную реализацию большинства методов. Этот класс вы будете использовать для реализации собственного стиля жизненных циклов.

Разработка жизненного стиля, работающего с кэшем

Поскольку контейнер Castle Windsor предлагает обширный набор стандартных стилей жизненных циклов, довольно сложно придумать хороший пример. Однако представьте, что вы хотите разработать кэширующий стиль, при котором экземпляр класса хранится некоторое время, а затем освобождается. Это хороший пример, поскольку он достаточно сложен для того, чтобы продемонстрировать различные аспекты реализации специального стиля жизненных циклов. Но этот пример не так огромен, чтобы занять больше пары страниц.

ПРИМЕЧАНИЕ

Кэширующий стиль — это отвлеченный пример. Существуют более качественные способы реализовать функциональность кэширования, поскольку нет необходимости кэшировать. Скорее понадобится работа с данными, которыми эти службы управляют.

ВНИМАНИЕ

Приведенный код игнорирует потоковую безопасность. Реальная реализация интерфейса ILifestyleManager должна быть потокобезопасной.

Самый простой способ реализации собственного стиля жизненных циклов заключается в наследовании от класса `AbstractLifestyleManager`, что показано в листинге 10.2.

Листинг 10.2. Определение пользовательского стиля жизненных циклов

```
public partial class CacheLifestyleManager :
    AbstractLifestyleManager
{
    private ILease lease;

    public ILease Lease
    {
        get
        {
            if (this.lease == null)
            {
                this.lease = this.ResolveLease();
            }
            return this.lease;
        }
    }

    private ILease ResolveLease()
    {
        var defaultLease = new SlidingLease(TimeSpan.FromMinutes(1));
        if (this.Kernel == null)
        {
            return defaultLease;
        }
        if (this.Kernel.HasComponent(typeof(ILease)))
        {
            return this.Kernel.Resolve<ILease>();
        }
        return defaultLease;
    }
}
```

1 Наследование от класса AbstractLifestyleManager

2 Ленивая загрузка

3 Попытка обнаружить ILease

Одно из преимуществ наследования от класса `AbstractLifestyleManager` (1) заключается в том, что метод `Init` уже реализован за вас. Вы можете переопределить его, но в данном примере такой необходимости нет. Все, что он делает, — сохраняет внедренные службы таким образом, что вы можете получить к ним доступ через защищенные свойства.

Для реализации функциональности, определяющей, когда следует удалять кэшированный объект, вам понадобится экземпляр класса `ILease`. Если бы у вас была

возможность использовать внедрение конструктора, вы могли бы запросить его в конструкторе с помощью трех строк кода (включая граничный оператор). В этом случае вам потребуется 12 строк кода, поскольку необходимо отследить множество потенциальных состояний экземпляра класса `CacheLifestyleManager`: был ли вызван метод `Init`? Имеет ли объект `Kernel` экземпляр класса `ILease`?

С этим можно справиться, работая с так называемым лениво загруженным свойством `Lease` ❷. В первый раз, когда вы считываете его, оно вызывает метод `ResolveLease`, чтобы определить, каким будет свойство `Lease`. Обычно используется стандартный объект `defaultLease`, однако также предпринимается попытка получить альтернативный вариант из объекта `Kernel` ❸ — если, конечно, этот объект существует. Думаю, это довольно хорошая иллюстрация недостатков внедрения методов. Обратите внимание: если какой-либо элемент считает свойство `Lease` до того, как вызовется метод `Init`, будет возвращен стандартный объект, даже если в объекте `Kernel` содержится компонент `ILease`. Однако поскольку контейнер Castle Windsor ничего не знает о свойстве `Lease`, при обычном использовании ничего не произойдет.

ПРИМЕЧАНИЕ

Интерфейс ILease, использованный в этом примере, является специальным интерфейсом, который определен для данной конкретной цели. Он отличается от интерфейса System.Runtime.Remoting.Lifetime.ILease, имеющего хоть и похожий, но более сложный API.

По сравнению со всеми препятствиями, которые приходится преодолеть при внедрении зависимости в специальный жизненный стиль, реализация метода `Resolve` гораздо проще, что вы и можете увидеть в листинге 10.3.

Листинг 10.3. Реализация метода Resolve

```
private object obj;

public override object Resolve(CreationContext context)
{
    if (this.Lease.IsExpired)
    {
        base.Release(this.obj);
        this.obj = null;
    }
    if (this.obj == null)
    {
        this.Lease.Renew();
        this.obj = base.Resolve(context);
    }
    return this.obj;
}
```

❶ Удаление объекта

❷ Обновление объекта Lease

Всякий раз, когда экземпляру класса `CacheLifestyleManager` требуется разрешить компонент, он начинает свою работу с проверки, не истек ли срок действия текущего объекта `Lease`. Если это случилось, он освобождает кэшированный объект и чистит всю память, занимаемую им ❶. Метод `Release` явно вызывается в базовом классе, а затем управление передается классу `IComponentActivator`, что показано

на рис. 10.5. Это важно делать, поскольку у менее высокоуровневой реализации появляется шанс удалить объект, если она реализует IDisposable.

Следующее, что необходимо сделать, — проверить, не является ли кэшированный объект нулевым указателем. Такое может произойти, если объект только что был освобожден или же это первый вызов метода Resolve. В обоих случаях вы обновляете объект ❷ и указываете базовой реализации разрешить компонент. Здесь также базовым классом вызывается соответствующий метод интерфейса IComponentActivator.

В этой реализации стиля жизненных циклов вы переопределяете метод Release так, чтобы он не выполнял никаких действий:

```
public override bool Release(object instance)
{
    return false;
}
```

Это может показаться странным, хотя это вполне нормальное явление. Вы должны понимать, что метод Release является подключаемым и частью жизненного стиля Seam контейнера Castle Windsor. Вас информируют о том, что компонент может быть освобожден, но это не означает, что вы должны делать это. Например, жизненный стиль Singleton по определению никогда не освобождает экземпляр класса, поэтому его реализация метода Release такая же, как и в предыдущем примере.

В случае класса CacheLifestyleManager вы время от времени освобождаете кэшированный экземпляр. Как показано в листинге 10.3, вы делаете это в методе Resolve, что является наиболее логичным.

Класс CacheLifestyleManager сохраняет в кэше разрешенный экземпляр до тех пор, пора не пройдет срок его хранения, а затем разрешает новый объект и обновляет объект Lease. Существует несколько способов реализации логики класса Lease, но мы рассмотрим только один.

Реализация класса Lease

Вам понадобится как минимум одна реализация ILease для того, чтобы работать с классом CacheLifestyleManager. Действие объекта класса SlidingLease прекращается спустя определенный промежуток времени, но вы можете создать и другие его реализации, прекращающие свое действие в определенное время дня или после разрешения каждого компонента определенное количество раз.

ПРИМЕЧАНИЕ

Интерфейс ILease и класс SlidingLease сами по себе не будут работать с контейнером Castle Windsor, я хочу показать их реализацию лишь для того, чтобы завершить разработку стиля жизненных циклов. Вы можете пропустить этот пункт, если вам неинтересна разработка класса SlidingLease, и прочесть о том, как зарегистрировать специальный стиль жизненных циклов.

В листинге 10.4 показана реализация интерфейса SlidingLease.

Листинг 10.4. Реализация интерфейса ILease

```
public class SlidingLease : ILease
{
    private readonly TimeSpan timeout;
```

```
    private DateTime renewed;

    public SlidingLease(TimeSpan timeout)
    {
        this.timeout = timeout;
        this.renewed = DateTime.Now;
    }

    public TimeSpan Timeout
    {
        get { return this.timeout; }
    }
    public bool IsExpired
    {
        get { return DateTime.Now >
            this.renewed + this.timeout; }
    }

    public void Renew()
    {
        this.renewed = DateTime.Now;
    }
}
```

❶ Действие объекта прекращается, если время вышло

❷ Обновляем объект

Класс `SlidingLease` реализует интерфейс `ILease`, отслеживая момент, когда обновляется объект `Lease`. Каждый раз, когда вы спрашиваете у него, вышло ли время действия объекта, он сравнивает текущее время с суммой времени обновления и задержки ❶. Когда объект `Lease` обновляется, время обновления устанавливается равным текущему времени ❷. Я мог бы использовать контекст `TimeProvider`, описанный в подразделе 4.4.4, вместо метода `DateTime.Now`, но я предпочел максимально упростить свой пример.

Теперь, когда вы знаете, как реализовать собственный стиль жизненных циклов и любую пользовательскую зависимость, которая может его иметь, вам остается лишь научиться их использовать.

Конфигурирование компонентов с использованием собственного стиля жизненных циклов

Использовать класс `CacheLifestyleManager` для конфигурации компонентов так же легко, как и прочие стили жизненных циклов:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .LifeStyle.Custom<CacheLifestyleManager>());
```

Вы применяете перегруженный метод `Custom`, чтобы определить, какой `ILifestyleManager` задействовать в данный момент, хотя также существует перегруженный метод, принимающий экземпляр типа `Type`.

Если вы не зарегистрируете интерфейс `ILease`, класс `CacheLifestyleManager` будет использовать объект Lease, заданный по умолчанию (`SlidingLease`), с тайм-аутом, равным одной минуте. Существует способ зарегистрировать пользовательскую реализацию интерфейса `ILease`:

```
container.Register(Component
    .For<ILease>()
    .Instance(new SlidingLease(TimeSpan.FromHours(1))));
```

В этом фрагменте регистрируется экземпляр класса `SlidingLease` с задержкой, равной одному часу. Необходимо регистрировать реализацию интерфейса `ILease` перед использованием собственного стиля, в противном случае будет применен объект Lease по умолчанию.

Разработка собственного стиля жизненных циклов для контейнера Castle Windsor совсем не сложна. В большинстве случаев класс `AbstractLifestyleManager` является хорошей исходной точкой, и нам нужно лишь переопределить методы, которые будут необходимы. Часто таким методом является `Resolve`, для остальных методов можно использовать и стандартную реализацию. Создавать собственные стили жизненных циклов вам придется нечасто, поскольку стандартный набор стилей контейнера Castle Windsor довольно обширен.

Наша экскурсия по управлению жизненными циклами контейнера Castle Windsor закончена. Компоненты могут быть сконфигурированы с использованием нескольких стилей жизненных циклов, это верно и для множественных реализаций одной абстракции. Далее нам нужно рассмотреть работу с несколькими компонентами, поэтому обратимся к этому вопросу.

10.3. Работа с несколькими компонентами

Контейнеры внедрения зависимостей отлично проявляют себя в ситуациях с четкими условиями и, наоборот, пробуксовывают, если условия нечеткие. При использовании внедрения конструкторов исходный конструктор предпочитается перегруженным, поскольку всегда очевидно, какой конструктор использовать, если нет другого выбора. Это происходит и в случаях, когда абстракции преобразовываются к конкретным типам. Если мы попытаемся преобразовать несколько конкретных типов к одной абстракции, мы создадим неопределенность.

Несмотря на неприятный побочный эффект в виде таких неопределенностей, нам зачастую необходимо работать с несколькими реализациями одного интерфейса. Это могут быть следующие ситуации:

- различные конкретные типы должны использоваться различными потребителями;
- зависимости являются последовательностями;
- используются декораторы.

В этом разделе мы рассмотрим все перечисленные случаи и увидим, как контейнер Castle Windsor справляется с каждым из них. Когда мы закончим, вы сможете регистрировать и разрешить компоненты даже в тех случаях, когда используется несколько реализаций одной абстракции.

Сначала взглянем на то, как мы можем предоставить более детализированный контроль, чем тот, который обеспечивает автоматическое связывание.

10.3.1. Выбираем одного кандидата из нескольких

Автоматическое подключение — очень удобный и мощный инструмент, но он предоставляет сравнительно немного возможностей управления. До тех пор пока абстракции явно преобразовываются к конкретным типам, никаких проблем не возникает, но как только мы предоставим несколько реализаций одного интерфейса, начнутся сложности с неопределенностью.

Вспомним, как контейнер Castle Windsor справляется со множественными регистрациями одной абстракции.

Регистрация нескольких реализаций одной службы

Как вы видели в подразделе 10.1.2, вы можете зарегистрировать несколько компонентов одной службы:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>());
```

В этом примере классы `Steak` и `SauceBéarnaise` регистрируются к службе `IIngredient`. Первая регистрация имеет приоритет, поэтому если вы разрешите `IIngredient` с помощью утверждения `container.Resolve<IIngredient>()`, то получите экземпляр класса `Steak`. Однако вызов `container.ResolveAll<IIngredient>()` возвратит массив экземпляров `IIngredient`, содержащих как `Steak`, так и `SauceBéarnaise`. Так и есть, последующие регистрации не забываются, но добраться до них труднее.

СОВЕТ

Первая регистрация заданного типа имеет приоритет. Она определяет стандартную регистрацию для этого типа.

ВНИМАНИЕ

Если существуют регистрации заданного типа, которые не могут быть разрешены из-за отсутствующих зависимостей, метод ResolveAll просто игнорирует их и возвращает только те, которые могут быть разрешены. Из-за того, что не генерируется никакого исключения, это иногда может привести к некоторым ошибкам, которые трудно распознать[1].

Листинг 10.5 демонстрирует способ, который позволяет реализовать подсказки. В дальнейшем такие подсказки могут быть использованы для выбора одного кандидата из нескольких.

[1] Пока я заканчивал книгу, были внесены изменения в эту часть кода контейнера Castle Windsor, хотя новая версия, содержащая их, еще не вышла. Однако сейчас, когда книга уже готова, поведение текущей версии контейнера Castle Windsor, скорее всего, будет отличаться от описанного здесь.

Листинг 10.5. Именуем компоненты

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Steak>()
    .Named("meat"));
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<SauceBéarnaise>()
    .Named("sauce"));
```

Вы можете дать каждой регистрации уникальное имя, которое впоследствии может быть использовано для отделения одного компонента от других.

Основываясь на именах, заданных в листинге 10.5, вы можете разрешить как `Steak`, так и `SauceBéarnaise` следующим образом:

```
var meat = container.Resolve<IIngredient>("meat");
var sauce = container.Resolve<IIngredient>("sauce");
```

Обратите внимание: вы передаете те имена, которые были использованы при регистрации компонентов.

Учитывая, что вы всегда должны разрешить службы в одном корне компоновки, вы не должны ожидать появления неопределенности на этом уровне.

СОВЕТ

Если вы вызываете метод Resolve с определенным ключом, подумайте, можете ли вы решить задачу иначе, чтобы повысить определенность.

Однако вы можете использовать названные компоненты, чтобы выбрать один вариант из множества доступных при конфигурировании зависимостей для заданной службы.

Регистрация названных зависимостей

Не менее полезной, чем автоматическое подключение, может быть переопределение нормального поведения для получения полного контроля над зависимостями. Кроме того, это может быть полезным, чтобы справиться с неопределенным API. В качестве примера рассмотрим такой конструктор:

```
public ThreeCourseMeal(ICourse entrée,
    ICourse mainCourse, ICourse dessert)
```

В этом случае у вас есть три зависимости с одинаковым типом, каждая из которых представляет собой концепцию, *отличную от других*. В большинстве случаев вам, вероятно, потребуется преобразовать каждую зависимость в отдельный тип. Листинг 10.6 показывает, как можно было бы зарегистрировать преобразования типа `ICourse`.

Листинг 10.6. Регистрация именованных преобразований

```
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"));
```

В листинге 10.6 вы регистрируете три именованных компонента, преобразовывая тип `Rilettes` к регистрации, названной `entrée` (это слово означает *закуска* или *аперитив*), `CordonBleu` — к регистрации, названной `mainCourse`, и `MousseAuChocolat` к регистрации, названной `dessert`.

Основываясь на этой конфигурации, вы можете зарегистрировать класс `ThreeCourseMeal` так, как показано в листинге 10.7.

Листинг 10.7. Переопределение автоматического подключения

```
container.Register(Component
    .For<IMeal>()
    .ImplementedBy<ThreeCourseMeal>()
    .ServiceOverrides(new
        {
            entrée = "entrée",
            mainCourse = "mainCourse",
            dessert = "dessert"
        }));
```

Вы можете предоставить явные переопределения для тех параметров (или свойств), которые хотите явно адресовать. В случае класса `ThreeCourseMeal` вам необходимо адресовать все три параметра конструктора. Однако в других случаях вам может понадобиться переопределить один из нескольких параметров, такое тоже возможно. Метод `ServiceOverrides` позволяет вам передавать объекты с безымянными типами, определяющие, какие параметры необходимо переопределить. Если вы не хотите использовать объекты с безымянными типами, другие перегруженные методы `ServiceOverrides` позволят вам передавать массив специализированных экземпляров типа `ServiceOverrides`, или `IDictionary`.

Применяя безымянные типы, вы сопоставляете параметры, которые хотите переопределить, с именованными регистрациями. В первом случае вы соотносите *имя параметра* `entrée` с регистрацией, названной `entrée`. В этом случае имя параметра совпадает с именем регистрации, но в соблюдении этого правила нет необходимости. Другие параметры преобразуются похожим образом.

ВНИМАНИЕ

Хотя использование в этом случае безымянных типов может показаться строгим типированием, эти типы являются очередной группой волшебных строк. В итоге такая группа преобразуется к словарю имен и значений. Имена свойств безымянного типа должны соответствовать именам параметров соответствующего конструктора. Если вы измените имя параметра в конструкторе, метод ServiceOverride перестанет работать, до тех пор пока вы не исправите его, поэтому считайте, что соответствие имен — это необходимость.

Поскольку метод `ServiceOverrides` основывается на соответствии текстов имен параметров и сконфигурированных переопределенных методов, не слишком надейтесь на него. Если вы вынуждены использовать его только для того, чтобы справиться с неопределенностью, более качественным решением будет написание API, решающего эту задачу. Часто это приводит к улучшению общей структуры программы.

В следующем разделе вы узнаете, как можно переработать текущую версию класса `ThreeCourseMeal` к более общей реализации и в то же время избавиться от «неотъемлемой» неопределенности. Вы можете сделать это, разрешив работу с произвольным количеством компонентов — но здесь необходимо понимать, как контейнер Castle Windsor подключает списки и последовательности.

10.3.2. Подключение последовательностей

В подразделе 6.4.1 мы говорили о том, как внедрение конструктора позволяет предупреждать нарушение принципа единственной ответственности. Тогда вы должны были понять, что чрезмерно активное внедрение конструкторов следует рассматривать не как недостаток самого паттерна внедрения конструктора. Скорее наоборот — нам на руку, что в такой ситуации мы можем заметить проблемы подобной структуры.

Что касается контейнеров внедрения зависимостей и неопределенности, можно заметить похожее соотношение. Контейнеры внедрения зависимостей, в общем, плохо справляются с неопределенностью. Хотя мы можем создать хороший контейнер вроде Castle Windsor для того, чтобы справиться с этой проблемой, это решение зачастую оказывается довольно неуклюжим. Нередко это свидетельствует о том, что мы можем улучшить структуру нашего собственного кода.

СОВЕТ

Если конфигурирование какой-либо части вашего API с помощью контейнера Castle Windsor затруднительно, подумайте, смогли бы вы сделать ваш API проще и понятнее. Это не только облегчит конфигурирование с помощью контейнера Castle Windsor, но и положительно скажется на общем дизайне программы.

Вместо того чтобы добровольно ограничивать себя возможностями контейнера Castle Windsor, нам следует принять используемые в нем соглашения и позволить ему вести нас вперед, к более качественному и последовательному дизайну. В этом разделе мы рассмотрим пример, демонстрирующий, как можно переработать наш код, чтобы избавиться от неопределенности, а также изучить принцип работы контейнера Castle Windsor с последовательностями, массивами и списками.

Рефакторинг кода

В подразделе 10.3.1 вы видели класс `ThreeCourseMeal` и узнали, как его «неотъемлемая» неопределенность заставила нас избавиться от автоматического подключения и вместо этого явно использовать `ServiceOverride`. Это должно было натолкнуть вас на мысль, что дизайн API следует пересмотреть.

Простое обобщение приведет нас к созданию реализации IMeal, которая принимает произвольное количество экземпляров ICourse вместо фиксированного числа, как это было с классом ThreeCourseMeal:

```
public Meal(IEnumerable<ICourse> courses)
```

Обратите внимание на то, что вместо требования трех отдельных экземпляров класса ICourse в конструкторе, одна зависимость от экземпляра IEnumerable<ICourse> позволяет вам предоставить любое количество объектов классу Meal — от нуля до... множества! Это решит проблему неопределенности, поскольку теперь существует лишь одна зависимость. Кроме того, этот шаг также улучшает API и реализацию, поскольку теперь существует один общий класс, который может смоделировать различные ситуации.

Основываясь на регистрации компонентов, приведенной в листинге 10.6, вы можете ожидать автоматического разрешения класса IMeal, если зарегистрируете его таким образом:

```
container.Register(Component
    .For<IMeal>()
    .ImplementedBy<Meal>());
```

Однако когда вы попробуете разрешить IMeal, контейнер сгенерирует исключение. Это исключение не назовешь очевидным, но оно объясняется тем, что контейнеру не было указано, как именно разрешать IEnumerable<ICourse>. Рассмотрим еще несколько различных возможных способов конфигурации.

Конфигурирование массивов

Контейнер Castle Windsor довольно хорошо работает с массивами. Поскольку массивы представляют собой реализацию типа IEnumerable<T>, вы можете явно сконфигурировать массив для параметра конструктора courses. Это можно сделать практически так же, как было показано в листинге 10.7. В листинге 10.8 вы увидите те же параметры, но уже определенные как службы.

Листинг 10.8. Явное определение массива служб

```
containér.Régistér(Componént
    .For<IMéal>()
    .ImpléméntédBy<Méal>()
    .SérvicéOvérridés(néw
        {
            coursés = néw[]
                {
                    "éntr é",
                    "mainCoursé",
                    "déssért"
                }
        }));
```

1 Переопределение параметра courses

Как и в листинге 10.7, вы используете метод ServiceOverride, когда хотите переопределить автоматическое подключение конкретными параметрами. В этом случае

вы хотите явно конфигурировать параметр конструктора `courses` (1) класса `Meal`. Так как этот параметр имеет тип `IEnumerable<ICourse>`, вы должны определить последовательность служб `ICourse`.

Поскольку массив реализует интерфейс `IEnumerable<T>`, вы можете определить массив именованных служб. Это возможно сделать путем создания массива *имен* служб. Данные имена идентичны тем, которые были назначены каждой регистрации в листинге 10.6, контейнер Castle Windsor преобразует этот массив имен служб к массиву экземпляров класса `ICourse` во время работы программы. Эти действия аналогичны действиям, производимым в листинге 10.7 с одним лишь отличием — Fluent registration API воспринимает и преобразует массивы имен служб к массивам служб.

Хотя преобразование класса `ThreeCourseMeal` к классу `Meal` выглядит правильным ходом, похоже, что в плане конфигурирования ничего не изменилось. Возможно ли улучшить и этот аспект?

Конечно, упростить конфигурирование возможно, но это чревато потерей управления программой. Как показывает рис. 10.6, необходимость выбирать службы заданного типа есть не всегда, в некоторых случаях могут понадобиться они все.

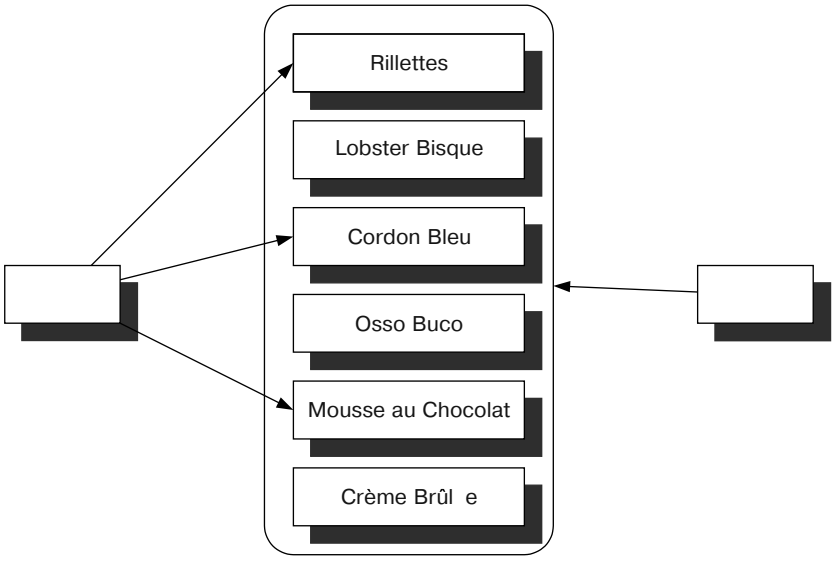


Рис. 10.6. Есть еще несколько способов работы с зависимостями, представленными в виде массивов. В ситуации, описанной справа, нам нужно разрешить *все* службы, сконфигурированные в контейнере. В ситуации, описанной слева, нам нужно лишь несколько таких служб

Пример, который мы рассмотрели, соответствует ситуации, когда нам приходилось вручную выбирать из явно созданного списка именованных служб, входящего в состав концептуально большего списка всех сконфигурированных служб заданного типа. В других случаях мы могли предпочесть более простое соглашение, согласно которому мы бы использовали *все* доступные службы требуемого типа. Рассмотрим способы достижения этой цели.

Разрешение последовательностей

Контейнер Castle Windsor по умолчанию не разрешает массивы или объекты класса `IEnumerable<T>`. Это может показаться несколько удивительным, поскольку вызов метода `ResolveAll` возвращает массив:

```
IIngredient[] ingredients = container.ResolveAll<IIngredient>();
```

Однако если вы попробуете указать контейнеру разрешить компонент, который представляет собой массив служб, сгенерируется исключение. Правильный способ справиться с этой проблемой — зарегистрировать в контейнере встроенный тип `CollectionResolver`, например так:

```
container.Kernel.Resolver.AddSubResolver(
    new CollectionResolver(container.Kernel));
```

Такая регистрация позволит контейнеру разрешать последовательности зависимостей, например `IEnumerable<T>`. Теперь вы можете разрешать класс `Meal` без явного использования метода `ServiceOverrides`. При такой регистрации:

```
container.Register(Component
    .For<IMeal>()
    .ImplementedBy<Meal>());
```

Вы можете разрешить `IMeal` с помощью класса `CollectionResolver`:

```
var meal = container.Resolve<IMeal>();
```

Это создаст экземпляр типа `Meal` со всеми службами `ICourse` из контейнера.

Потребители, использующие списки зависимостей, являются наиболее логичным способом использования нескольких регистраций одной абстракции. Но перед тем как перейти к следующей теме, рассмотрим последний случай, который может показаться несколько удивительным: случай, при котором в дело вступают множественные регистрации.

10.3.3. Подключение декораторов

В подразделе 9.1.2 мы рассмотрели, как паттерн проектирования «Декотатор» может пригодиться при реализации сквозных аспектов приложения. По определению декораторы предоставляют несколько типов одной абстракции. У нас есть, по крайней мере, две реализации абстракции: сам декоратор и «декорированный» тип. Если наложить декораторы друг на друга, реализаций может получиться еще больше.

Это еще один пример множественной регистрации одной службы. В отличие от предыдущих разделов, эти регистрации не являются концептуально одинаковыми, они представляют собой взаимозависимости. В этом разделе вы увидите, как сконфигурировать контейнер Castle Windsor так, чтобы работать с этим паттерном.

Явное подключение декораторов

Контейнер Castle Windsor требует от нас регистрации всех компонентов, которые мы хотим использовать. Когда приходится работать с декораторами, мы должны регистрировать как декоратор, как и «декорируемый» тип. Поскольку оба типа

реализуют один и тот же интерфейс, создается неопределенность, которую можно исправить. Как показывает листинг 10.9, это можно сделать явно.

Листинг 10.9. Явное конфигурирование декоратора

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Breading>()
    .ServiceOverrides(new
        {
            ingredient = "cutlet"
        }));
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<VealCutlet>()
    .Named("cutlet"));
```

Тип `Breading` можно рассматривать как надстройку над типом `VealCutlet`. Зарегистрируем `Cotoletta`. Когда вы разрешите `Cotoletta`, вам понадобится ссылка на тип `Breading`, который должен содержать тип `VealCutlet`. Сначала вы регистрируете тип `Breading`. Помните, в контейнере Castle Windsor первая регистрация всегда имеет приоритет. Вы явно используете метод `ServiceOverrides` для конфигурирования того, какая именованная служба должна быть использована в качестве параметра конструктора `ingredient`. Обратите внимание на то, что вы создаете ссылку на компонент, который называется `cutlet`, хотя к этому моменту этот компонент еще не зарегистрирован. Это возможно потому, что порядок регистраций для работы неважен. Вы можете зарегистрировать компоненты *до* регистрации их зависимостей, и эта конструкция будет работать, но только если к моменту разрешения служб все компоненты будут подходящим образом зарегистрированы.

Это означает, что вы должны зарегистрировать класс `VealCutlet` перед разрешением `IIngredient`. Затем вы регистрируете его с именем `cutlet`. Это имя соответствует имени службы, переданной параметру конструктора `ingredient` в предыдущей конфигурации.

Хотя явное конфигурирование декораторов возможно и иногда необходимо, контейнер Castle Windsor сам понимает паттерн «Декоратор» и обеспечивает менее явный способ выполнить то же самое действие.

Неявное подключение декораторов

Контейнер Castle Windsor позволяет нам неявно конфигурировать декораторы, регистрируя их в правильном порядке. Помните то, что первая регистрация имеет приоритет, и именно ее тип будет возвращен методом `Resolve`. Поэтому сначала мы должны регистрировать самый «внешний» декоратор.

По определению декоратор имеет зависимость от другого экземпляра того же типа. Если мы не определяем явно, какую регистрацию использовать, есть вероятность создания циркулярной ссылки. Однако контейнер Castle Windsor не допускает таких ошибок. Вместо этого он выбирает *следующую* регистрацию подходящего типа. Это означает, что вместо листинга 10.9 вы можете написать такой код:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Breading>());
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<VealCutlet>());
```

Нет необходимости явно называть компоненты или использовать метод `ServiceOverrides` для конфигурирования зависимостей. Когда вы разрешаете `IIngredient`, контейнер Castle Windsor автоматически связывает класс `Breading` со следующей доступной службой `IIngredient`, которая является классом `VealCutlet`.

ПРИМЕЧАНИЕ

Следующим членом в логическом ряду, содержащем паттерн «Декоратор», является паттерн «Перехват» (Interception). Контейнер Castle Windsor отлично работает с этим паттерном, но, поскольку в подразделе 9.3.3 содержится полномасштабный пример, я не буду повторять его здесь и просто сошлюсь на него.

Контейнер Castle Windsor позволяет нам работать с несколькими компонентами различными способами. Мы можем зарегистрировать компоненты как альтернативы друг другу, тогда они разрешатся как последовательности или как иерархия декораторов. Во многих случаях контейнер Castle Windsor определит, что нужно делать, но мы всегда можем использовать метод `ServiceOverrides` для явного определения того, как compose-службы, если нужно управлять ими более явно.

Может также возникнуть ситуация, когда нам необходимо работать с API, отличающимся от внедрения конструкторов. К этому моменту вы увидели, как конфигурировать компоненты в контейнере Castle Windsor (включая определение стилей жизненных циклов). Но до сих пор мы не позволяли контейнеру связывать зависимости, неявно подразумевая, что все компоненты используют внедрение конструктора. Поскольку такая ситуация довольно редка, в следующем разделе мы рассмотрим работу с классами, объекты которых создаются особых образом.

10.4. Конфигурирование сложных API

До этого момента мы изучали способы конфигурирования компонентов, использующих внедрение конструктора. Одним из множества преимуществ внедрения конструкторов является тот факт, что контейнер внедрения зависимостей, например, Castle Windsor, может легко понять, как создать все классы графа зависимостей.

Ситуация усложняется, когда поведение API становится менее определенным. В этом разделе вы увидите, как работать с примитивными аргументами конструкторов, статическими фабриками и внедрением свойств. Все эти вопросы требуют особого внимания. Начнем с рассмотрения классов, которые принимают в качестве параметров конструктора примитивные типы, такие как строки или целые числа.

10.4.1. Конфигурирование примитивных зависимостей

Все работает хорошо до тех пор, пока мы внедряем абстракции в потребителей. Но ситуация становится более запутанной, когда конструктор зависит от примитивного типа, такого как строка, число или перечисление. Такая зависимость часто возникает в том случае, когда реализуется доступ к данным и конструктор класса принимает в качестве параметра строки соединения. Мы рассмотрим и более общий пример, касающийся всех строк и чисел.

Концептуально не имеет особого смысла регистрировать как компонент контейнера строку или число, а в контейнере Castle Windsor это и вовсе не сработает. Если мы попробуем разрешить компонент с примитивной зависимостью, сгенерируется исключение, даже если примитивный тип был зарегистрирован ранее.

В качестве примера рассмотрим следующий конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере параметр `Spiciness` является перечислением:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

ВНИМАНИЕ

Перечисления уже не используются и должны быть переработаны в полиморфические классы. Однако в этом примере они будут как раз кстати.

Вам понадобится явно указать контейнеру Castle Windsor, как разрешить параметр конструктора `spiciness`. Листинг 10.10 демонстрирует, как это сделать, используя синтаксис, очень похожий на метод `ServiceOverrides`, но уже с другим методом.

Листинг 10.10. Передача примитивного типа в качестве значения конструктора

```
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<ChiliConCarne>()
    .DependsOn(new
        {
            spiciness = Spiciness.Hot
        }));
```

Вместо метода `ServiceOverrides`, который переопределяет автоматическое связывание, вы можете использовать метод `DependsOn`, позволяющий передавать объекты для конкретных зависимостей. В этом случае вы передаете `Spiciness.Hot` в качестве параметра конструктора `spiciness`.

ПРИМЕЧАНИЕ

Различие между методами ServiceOverrides и DependsOn заключается в том, что при использовании метода DependsOn мы передаем *непосредственно объекты*, используемые в качестве заданного параметра или свойства, а при использовании метода ServiceOverrides передаются имена служб, которые будут разрешены для заданного параметра или свойства.

ВНИМАНИЕ

Как и в случае с методом ServiceOverrides, метод DependsOn полагается на соответствие между именем параметра и именем анонимного свойства, переданного методу DependsOn. Если мы переименуем параметр, то нам следует также отредактировать вызов метода DependsOn.

Когда нам нужно передать примитивное значение, например строку соединения, мы можем явно определить значение в коде (или получить его из конфигурации приложения) и назначить его, используя метод `DependsOn`. Достоинство использования метода `DependsOn` заключается в том, что нам не нужно явно вызывать конструктор или передавать прочие зависимости, поскольку для этого лучше подходит автоматическое подключение. Недостатком использования этого метода является его неустойчивость по отношению к рефакторингу.

Существует более надежная альтернатива, позволяющая нам явно вызвать конструктор. Она также может быть использована для работы с классами, не имеющими традиционных конструкторов.

10.4.2. Регистрирование компонентов с помощью блоков кода

Экземпляры некоторых классов не могут быть получены с помощью общедоступного (`public`) конструктора. Вместо этого для создания объектов вы должны использовать определенного рода фабрику. Это всегда проблематично для контейнеров внедрения зависимостей, поскольку по умолчанию они следят за `public`-конструкторами.

Рассмотрим этот пример — конструктор `public`-класса `JunkFood`:

```
internal JunkFood(string name)
```

Даже несмотря на то, что класс `JunkFood` использует ключевое слово `public`, его конструктор имеет модификатор `internal`. Очевидно, экземпляры класса `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 контейнера Castle Windsor этот API является проблематичным, поскольку в статических фабриках не существует определенных и прочно

установленных соглашений. Чтобы объединить контейнер и фабрики, мы можем использовать блок кода, как показано в листинге 10.11.

Листинг 10.11. Конфигурирование метода фабрики

```
container.Register(Component
    .For<IMeal>()
    .UsingFactoryMethod(() =>
        JunkFoodFactory.Create("chicken meal")));
```

Вы можете задействовать метод `UsingFactoryMethod` для определения блока кода, создающего подходящий экземпляр. В этом случае потребуется вызвать метод `Create` класса `JunkFoodFactory` с желаемым параметром.

Данный блок кода будет выполнен в подходящее время соответственно стилю жизненного цикла компонента. В этом случае, поскольку вы явно не определили стиль жизненного цикла, им по умолчанию является Singleton. Метод `Create` будет вызван лишь один раз, независимо от того, сколько раз вы разрешите `IMeal`. Если бы вы сконфигурировали компонент так, что он использовал бы стиль Transient, метод `Create` вызывался бы каждый раз при разрешении `IMeal`.

Помимо выполнения более экзотической инициализации объектов, чем использование обычных `public`-конструкторов, применение блоков кода представляет собой альтернативный подход — более безопасный с точки зрения типов и более подходящий для передачи примитивов, чем применение метода `DependsOn`, показанного в подразделе 10.4.1:

```
container.Register(Component
    .For<ICourse>()
    .UsingFactoryMethod(() =>
        new ChiliConCarne(Spiciness.Hot)));
```

В этом случае вы используете блок кода для явного создания нового экземпляра класса `ChiliConCarne` с желаемым параметром `Spiciness`. Это повышает безопасность типов, но полностью убирает автоматическое подключение для рассматриваемого типа.

СОВЕТ

Существуют более продвинутые способы переопределения метода UsingFactoryMethod, позволяющие вам разрешать зависимости из контейнера. Это может быть полезным, когда вы хотите задействовать метод UsingFactoryMethod для явного назначения лишь одного из нескольких параметров, но при этом необходимо передавать все прочие параметры, чтобы могла выполниться компиляция.

Метод `UsingFactoryMethod` — это хороший инструмент для тех случаев, когда вам нужно работать с классами, объекты которых не могут быть созданы с помощью `public`-конструкторов. До тех пор пока существуют `public`-API, которые вы можете использовать для создания экземпляров желаемых классов, вы можете применять метод `UsingFactoryMethod` для явного определения блока кода, который создаст требуемый объект.

Последнее распространенное «отклонение» от внедрения конструктора, которое мы рассмотрим здесь, — это внедрение свойств.

10.4.3. Подключение с внедрением свойств

Внедрение свойств — это менее определенная форма внедрения зависимостей, поскольку компилятор не заставляет вас назначать значение записываемому свойству. Несмотря на это, контейнер Castle Windsor умеет работать с внедрением свойств и, если может, назначает значения записываемым свойствам.

Рассмотрим класс `CaesarSalad` (салат «Цезарь»):

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

Распространенным заблуждением является то, что в салате «Цезарь» есть мясо цыпленка. Сам по себе салат «Цезарь» является *салатом*, но поскольку мясо цыпленка улучшает его вкус, многие рестораны предлагают добавить его в салат в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует эту ситуацию, предоставляя записываемое свойство, называющееся `Extra`.

Если вы зарегистрируете *только* класс `CaesarSalad` без упоминания `Chicken` (цыплята), свойство `Extra` не будет назначено:

```
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<CaesarSalad>());
```

Основываясь на этой регистрации, разрешение `ICourse` вернет экземпляр класса `CaesarSalad` без `Chicken`. Однако вы можете изменить результат разрешения, добавив в контейнер свойство `Chicken`:

```
container.Register(Component
    .For<IIngredient>()
    .ImplementedBy<Chicken>());
container.Register(Component
    .For<ICourse>()
    .ImplementedBy<CaesarSalad>());
```

Теперь при разрешении `ICourse` итоговым свойством `Extra` экземпляра класса `CaesarSalad` будет экземпляр класса `Chicken`. Так и есть, контейнер Castle Windsor проверяет новый экземпляр на наличие записываемых свойств и назначает ему значение, если может предоставить компонент, соответствующий типу свойства.

СОВЕТ

Когда вам нужно явно управлять назначением значений свойствам, вы можете использовать метод ServiceOverrides.

В этом разделе вы узнали, как работать с API, не сводящимися к простому внедрению конструктора. Вы можете адресовать примитивный конструктор с помощью метода `DependsOn` или путем использования метода `UsingFactoryMethod`, который также поддерживает методы работы с фабриками, и прочие альтернативы `public` конструкторам. Внедрение свойств также поддерживается контейнером Castle Windsor.

10.5. Резюме

Обзор контейнера Castle Windsor, содержащийся в этой главе, представляет лишь малую толику возможностей одного из самых зрелых и обширных имеющихся контейнеров внедрения зависимостей. Поскольку повсеместно используются швы, мы можем настроить их по собственному желанию. В нашем распоряжении есть также множество других особенностей.

В этой главе мы сфокусировались на наиболее часто используемых фрагментах API контейнера Castle Windsor. Материал, представленный здесь, охватывает основные вопросы применения контейнера, а также содержит советы о том, на какие более продвинутые особенности следует обратить внимание. Это все, что вам нужно знать, если ваша база кода построена в соответствии с паттернами внедрения зависимостей и соглашениями. Располагая этим знанием, вы сможете использовать контейнер Castle Windsor повсеместно в вашем приложении.

Даже «скелет» контейнера внедрения зависимостей Castle Windsor кажется колоссальной конструкцией. Он поддерживает практически любую функцию, которая может вам потребоваться. Как один из наиболее старых контейнеров внедрения зависимостей для .NET, из года в год он становится только лучше. Причем он поддерживает многие новые идеи и современные конструкции. Однако, возможно, самый значительный недостаток контейнера Castle Windsor заключается в том, что большой набор функций реализуется с помощью местами неоднородного API. Хотя и несложно начать работу с классом `WindsorContainer`, более сложные сценарии иногда трудно реализовать, если только вы не знаете наизусть весь API. К счастью, поскольку форум поддержки контейнера Castle Windsor активен и модерируется готовыми помочь разработчиками, вы, скорее всего, быстро получите ответ на любой ваш вопрос.

Хотя сравнительно сложный API может кому-то показаться обескураживающим, с контейнером Castle Windsor, как и с любым другим контейнером внедрения зависимостей, легко начать работу. Всего лишь создайте экземпляр класса `WindsorContainer`, сконфигурируйте его и разрешайте компоненты из него.

Существует несколько способов сконфигурировать контейнер: конфигурирование в коде, использование XML и конфигурирование, основанное на соглашениях. В некоторых случаях для достижения оптимального результата можно применять их все.

В вашем распоряжении есть также множество стилей жизненных циклов — Singleton, Transient и Web Request Context. Если существующих стилей недостаточно, вы можете создать свой, но такая ситуация будет возникать довольно редко.

Поддержка множества компонентов для одной абстракции является, возможно, одним из слабейших мест контейнера Castle Windsor. Контейнер может работать с массивами и делает это лучше, чем при применении других типов последовательностей или списков, но мы можем справиться с этим недостатком относительно легко. Точный способ достижения этой цели зависит от того, хотим мы разрешить все элементы последовательности или же нам нужен набор компонентов одной службы.

Хотя мы предпочитаем пользоваться автоматическим подключением, метод `ServiceOverrides` позволяет явно сконфигурировать то, как зависимости назначаются компонентам.

Иногда компоненты не используют внедрения конструкторов, а применяют вместо этого внедрения свойств или требуют отдельные классы фабрик. Такие сценарии также обрабатываются с помощью различных методов.

Поскольку контейнер Castle Windsor является одним из самых разносторонних, нет особых причин не применять его, но не забывайте о других контейнерах, которые также хороши. В следующей главе мы рассмотрим еще один зрелый и сложный контейнер внедрения зависимостей: StructureMap.

11 StructureMap

Меню:

- знакомство с StructureMap;
- управление жизненным циклом;
- работа с несколькими компонентами;
- конфигурирование сложных API.

В предыдущей главе мы рассмотрели контейнер внедрения зависимостей Castle Windsor. Мы сделали это для того, чтобы увидеть, как могут применяться принципы и паттерны, описанные в частях 1–3. В этой главе мы исследуем другой контейнер внедрения зависимостей: StructureMap.

Контейнер StructureMap является старейшим контейнером внедрения зависимостей для .NET — он существует дольше, чем другие. Несмотря на свой возраст, он все еще активно разрабатывается и имеет множество современных особенностей, поэтому его следует считать зрелым, но не устаревшим. Он также один из самых востребованных контейнеров внедрения зависимостей[1].

В этой главе мы рассмотрим, как контейнер StructureMap может быть использован для реализации принципов и паттернов, изложенных в частях 1–3. На рис. 11.1 представлена структура этой главы. Эта глава состоит из четырех разделов. В первом демонстрируется API контейнера StructureMap. Данный раздел обязательно нужно прочесть, если вы хотите перейти к следующим разделам. Каждый из последующих разделов можно прочитать независимо от предыдущего, хотя четвертый раздел использует несколько методов, представленных в третьем разделе. Названия этих методов отчасти говоря сами за себя, поэтому вы сможете прочесть четвертый раздел, не читая третий. Но, с другой стороны, вам все равно может понадобиться обратиться к третьему разделу.

В первом разделе вы познакомитесь с контейнером StructureMap и узнаете, как сконфигурировать и разрешать его компоненты. Три последующих раздела работают с паттернами, требующими большего внимания. Вы можете прочесть их по порядку или же пропустить некоторые и рассмотреть только те, которые вас заинтересовали.

---

[1] Я снова хотел бы отметить, что не существует научной статистики, касающейся использования контейнеров внедрения зависимостей, все подобные утверждения основаны на интернет-опросах и т. д. Просто поверьте мне.

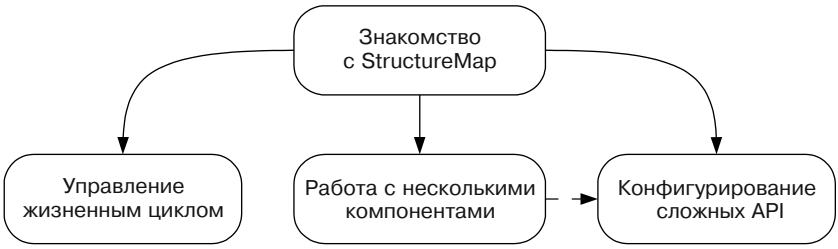


Рис. 11.1. Структура данной главы

Изучение этой главы позволит вам начать работу с контейнером, а также справиться с наиболее распространенными проблемами, с которыми вы можете столкнуться при повседневном использовании этого контейнера. Глава не является полным обзором контейнера StructureMap — на это потребовалась бы целая книга.

Вы можете прочесть данную главу отдельно от прочих глав части 4, чтобы научиться работать с контейнером StructureMap, или же вы можете прочесть эту часть до конца, чтобы сравнить контейнеры ВЗ. Цель этой главы заключается в том, чтобы показать, как StructureMap работает и реализует паттерны и принципы, описанные в предыдущих десяти главах.

11.1. Знакомство со StructureMap

В этом разделе вы узнаете, где можно достать StructureMap, что он собой представляет и то, как начать его использовать. Мы также рассмотрим самые популярные конфигурации и поместим настройки конфигурации в пакеты, которые сможем переиспользовать. В табл. 11.1 предоставляется основная информация, которая скорее всего понадобится вам для того, чтобы начать работу с контейнером.

Таблица 11.1. Первый взгляд на контейнер Castle Windsor

Вопрос	Ответ
Где достать контейнер?	Перейдите по ссылке http://structuremap.github.com/structuremap/index.html и нажмите ссылку Download the Latest Release (Загрузить последний релиз). Из Visual Studio 2010 вы также можете получить его с помощь NuGet. Имя пакета называется structuremap
Что нужно загрузить?	Вы можете скачать ZIP-архив со скомпилированными бинарными файлами. Вы также можете скачать текущую версию исходного кода и скомпилировать его самостоятельно. Бинарные файлы представляют собой DLL-библиотеки, которые вы можете поместить, куда пожелаете, и ссылаться на них из своего кода
Какие платформы поддерживаются?	.NET 3.5 SP1, .NET 4
Сколько это стоит?	Нисколько. Это проект с открытым исходным кодом
Где можно получить помощь?	Нет гарантии, что вы получите поддержку, но вам, скорее всего, могут помочь на официальном форуме по адресу http://groups.google.com/group/structuremapusers
На какой версии основана глава?	2.6.1

Как и в случае с контейнером Castle Windsor, для работы с контейнером StructureMap вам нужно лишь следовать простому ритму, проиллюстрированному на рис. 11.2.

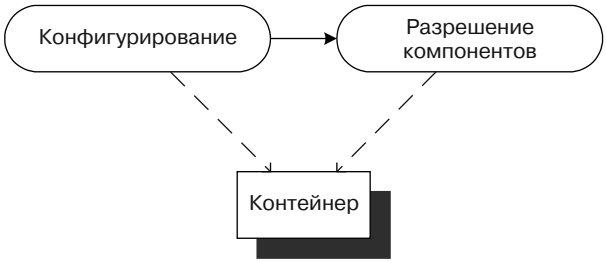


Рис. 11.2. Общий принцип использования контейнера StructureMap прост: сначала необходимо сконфигурировать контейнер, а затем разрешать компоненты из него. В большинстве случаев следует создать экземпляр класса Container и полностью сконфигурировать его перед тем, как начать разрешать компоненты из него. Компоненты разрешаются из сконфигурированного экземпляра

КОНТЕЙНЕР ИЛИ OBJECTFACTORY?

Ранние версии контейнера StructureMap использовали класс `ObjectFactory` как единый контейнер для всего приложения. Он использовался следующим образом:

```
SauceBéarnaise sauce =
    ObjectFactory.GetInstance<SauceBéarnaise>();
```

Одной из проблем, возникающих при использовании статической фабрики, является тот факт, что она вынуждает нас применять ее как локатор сервисов (Service Locator). Использование класса `ObjectFactory` теперь не рекомендуется, лучше применять экземпляры контейнеров. На сайте контейнера StructureMap (и во многих других местах) приводится множество примеров, задействующих класс `ObjectFactory` для демонстрации различных особенностей контейнера, но мы должны рассматривать их как пережитки старых версий.

В остальной части главы мы будем игнорировать факт существования `ObjectFactory` и сфокусируемся лишь на экземплярах контейнера.

ВНИМАНИЕ

Еще не так давно API контейнера StructureMap часто менялся. Можно было часто обнаружить в Интернете примеры кода, в которых использовались методы или классы, недоступные в текущей версии; наиболее вероятно то, что их переименовали или переработали. Хотя все примеры кода компилировались и работали в момент написания этой главы, некоторые части API могли измениться с того времени.

Когда вы закончите читать этот раздел, вы усвоите общий шаблон использования контейнера StructureMap, а также сможете начать применять его в тех ситуациях, где все компоненты следуют подходящим паттернам внедрения зависимостей, например при внедрении конструктора (Constructor Injection). Мы начнем с простейшего сценария и рассмотрим, как можно разрешать объекты с помощью контейнера StructureMap.

11.1.1. Разрешение объектов

Основная цель любого контейнера DI — разрешение компонентов. В этом подразделе мы рассмотрим API, позволяющий разрешать компоненты с помощью контейнера StructureMap.

Если вы помните принцип разрешения компонентов с помощью контейнера Castle Windsor, вы должны знать, что этот контейнер требует от вас регистрации *всех* соответствующих компонентов перед тем, как вы сможете разрешить их. Для контейнера StructureMap этот принцип не работает; если вы запрашиваете конкретный тип, имеющий конструктор по умолчанию, в конфигурировании нет необходимости. Наиболее простой способ использования контейнера StructureMap:

```
var container = new Container();
SauceBéarnaise sauce = container.GetInstance<SauceBéarnaise>();
```

Имея экземпляр класса `StructureMap.Container`, вы можете использовать обобщенный метод `GetInstance` для получения объекта конкретного класса `SauceBéarnaise`. Поскольку этот класс имеет конструктор по умолчанию, контейнер StructureMap автоматически определяет, как создать его экземпляр. Явное конфигурирование контейнера не требуется.

ПРИМЕЧАНИЕ

Метод GetInstance<T> эквивалентен методу контейнера CastleWindsor Resolve<T>.

Поскольку контейнер StructureMap поддерживает автоматическое подключение, даже при отсутствии конструктора по умолчанию контейнер сможет создавать объекты без конфигурирования до тех пор, пока используемые параметры конструктора являются конкретными типами и все дерево параметров имеет типы-потомки с конструкторами по умолчанию.

В качестве примера рассмотрим следующий конструктор `Mayonnnaise` (майонез):

```
public Mayonnaise(EggYolk eggYolk, OliveOil oil)
```

Хотя здесь приведен несколько упрощенный рецепт майонеза, оба класса `EggYolk` (яичный желток) и `OliveOil` (оливковое масло) являются конкретными классами с конструкторами по умолчанию. Хотя сам по себе класс `Mayonnaise` не имеет конструктора по умолчанию, контейнер StructureMap все равно создаст его безо всякой конфигурации:

```
var container = new Container();
var mayo = container.GetInstance<Mayonnaise>();
```

Этот метод работает потому, что контейнер StructureMap способен определить, как создать все требуемые параметры конструктора. Однако когда мы предоставим слабо связанные компоненты, мы должны сконфигурировать контейнер StructureMap путем преобразования абстракций к конкретным типам.

Преобразование абстракций к конкретным типам

Хотя способность контейнера StructureMap к автоматическому подключению конкретных типов может пригодиться в некоторых случаях, слабое связывание обычно

требует преобразования абстракций к конкретным типам. Создание объектов, основанное на подобных преобразованиях, — это основная функция любого контейнера внедрения зависимостей, но вы все равно должны определить преобразования.

В этом примере вы преобразуете интерфейс `IIngredient` к конкретному классу `SauceBéarnaise`, что позволяет вам 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 выражение `ConfigurationExpression`, используя блок кода (подробнее об этом рассказано во врезке «Вложенные замыкания» далее). Конфигурационное утверждение читается практически как предложение (или как рецепт кулинарной книги): *«Для IIngredient использовать SauceBéarnaise»*. Метод `For` позволяет определять абстракции, а метод `Use` дает возможность определять конкретный тип, который реализует абстракцию.

Строго типизированный API, предоставленный классом `ConfigurationExpression`, помогает предотвратить ошибки в конфигурации, поскольку метод `Use` имеет родовое ограничение (Generic Constraint), которое указывает, что тип, определенный в аргументе `type`, должен наследовать от типа абстракции, определенного в аргументе метода `For`. Предыдущий пример кода компилируется, поскольку класс `SauceBéarnaise` реализует интерфейс `IIngredient`.

Во многих случаях строго типизированный API — это все, что нам требуется, и поскольку он обеспечивает желаемую проверку во время компиляции, мы должны использовать ее так часто, как только это возможно. Однако бывают ситуации, когда нам нужно применять слабо типизированный API для разрешения служб. Это также возможно.

Разрешение слабо типизированных служб

Иногда мы не можем использовать общий API, поскольку не знаем подходящего типа во время разработки. Все, что у нас есть, — это экземпляр класса `Type`, но мы хотим получить экземпляр этого типа. Вы видели пример этой ситуации в разделе 7.2, когда обсуждали класс ASP.NET MVC `DefaultControllerFactory`. Релевантный метод следующий:

```
protected internal virtual IController GetControllerInstance(
    RequestContext requestContext, Type controllerType);
```

Поскольку у вас есть только экземпляр типа `Type`, вы не можете использовать общие классы и должны обратиться за помощью к слабо типизированным API. К счастью, контейнер StructureMap предоставляет слабо типизированную перегрузку метода `GetInstance`, позволяющего вам реализовывать метод `GetControllerInstance`, например, так:

```
return (IController)this.container.GetInstance(controllerType);
```

Слабо типизированный перегруженный метод `GetInstances` дает возможность передавать аргумент `controllerType` непосредственно контейнеру StructureMap, но также требует явного преобразования возвращаемого значения к `IController`.

Не важно, какую версию метода `GetInstance` вы будете использовать, контейнер StructureMap гарантирует то, что будет возвращен объект требуемого типа или же сгенерируется исключение, если зависимости не могут быть удовлетворены. Когда все требуемые зависимости будут подобающим образом сконфигурированы, контейнер StructureMap сможет автоматически связать требуемый тип.

В предыдущем примере `this.container` — это экземпляр класса `StructureMap.IContainer`. Чтобы разрешать требуемый тип, все слабо связанные зависимости должны быть заранее сконфигурированы. Существует много способов сконфигурировать контейнер StructureMap, в следующем подразделе приведены наиболее распространенные из них.

11.1.2. Конфигурирование контейнера

Как говорилось в разделе 3.2, есть несколько концептуально отличающихся друг от друга способов сконфигурировать контейнер внедрения зависимостей. Они представлены на рис. 11.3. Конфигурация в коде — это строго типизированный явный способ. Способ с применением XML, с другой стороны, отличается поздним связыванием, но при этом также является явным. Авторегистрация, в отличие от них, опирается на соглашения, которые могут быть как строго, так и слабо типизированными.

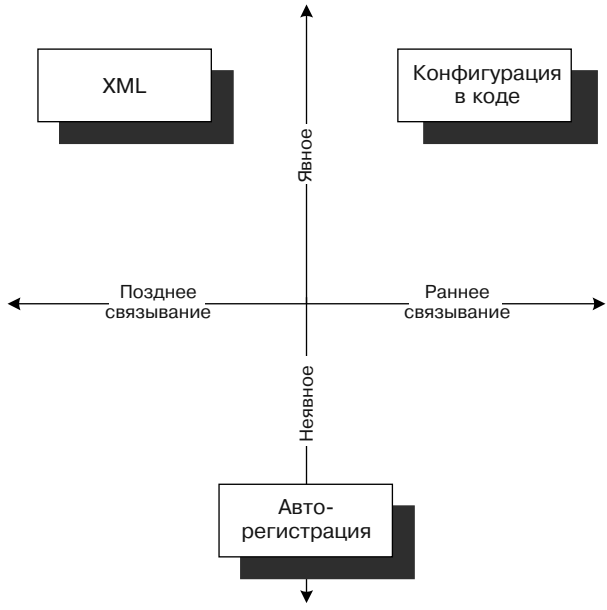


Рис. 11.3. Концептуально разные способы конфигурации

Как и прочие контейнеры внедрения зависимостей с богатой историей, контейнер StructureMap сначала конфигурировался с помощью XML. Но прошло совсем немного времени, и многие команды разработчиков поняли, что определение регистрации типов с помощью XML очень ненадежно. В наше время предпочитается

строго типизированное конфигурирование. Так, сконфигурировать контейнер возможно с помощью конфигурирования в коде или же, что более эффективно, с использованием традиционной авторегистрации.

Контейнер StructureMap поддерживает все три подхода и даже позволяет объединять их в одном и том же контейнере. В этом отношении он предоставляет нам все, что только может понадобиться. В этом разделе мы рассмотрим, как можно использовать каждый из этих трех типов конфигурирования контейнера.

Конфигурация в коде

В подразделе 11.1.1 вы уже видели краткий пример строго типизированного конфигурационного API контейнера StructureMap. Рассмотрим его более детально.

Существует несколько точек входа в конфигурационный API. Вы уже наблюдали, как этот интерфейс явно вызывается в методе `Configure`:

```
var container = new Container();
container.Configure(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
```

Альтернативным способом конфигурирования является определение точно такого же блока кода при создании экземпляра класса `Container`:

```
var container = new Container(r => r
    .For<IIngredient>()
    .Use<SauceBéarnaise>());
```

Результат будет аналогичным, однако в этой главе я буду следовать одним и тем же соглашениям и предпочту использовать метод `Configure`, а не конструктор.

ВЛОЖЕННЫЕ ЗАМЫКАНИЯ

Контейнер StructureMap активно использует паттерн «Вложенное замыкание»(Nested Closure)[1]. В качестве примера рассмотрим описание метода `Configure`:

```
public void Configure(Action<ConfigurationExpression> configure);
```

Параметр `configure` — это делегат, принимающий экземпляр класса `ConfigurationExpression` в качестве входного параметра. В примерах этой главы данный параметр зачастую обозначается `r`, и я обычно передаю делегат как блок кода, выраженный с использованием параметра `r`.

Примечание: в примерах кода, расположенных на сайте контейнера StructureMap и в блоге Джереми Миллера, параметр, использованный в блоке кода, иногда называется `x` или `register`. В этой главе я выбрал название `r` (как

[1] Подробнее о вложенных замыканиях написано в статье *Jeremy Miller* Patterns in Practice: Internal Domain Specific Languages. — MSDN Magazine, January 2010. Она также доступна по адресу http://msdn.microsoft.com/ru-ru/magazine/ee291514.aspx на русском языке. Эта статья написана одним из создателей контейнера StructureMap и использует примеры из API этого контейнера, чтобы проиллюстрировать процесс разработки API, который выполняет конфигурирование с помощью блоков кода (их общеизвестное название — лямбда-выражения).

сокращение от `registry`). Хотя имя переменной `r` не назовешь очевидным, небольшая область видимости рассматриваемых блоков кода заставляет предпочесть именно это имя, а не имя подлиннее.

Класс `ConfigurationExpression` содержит множество методов, которые мы можем использовать для конфигурирования контейнера StructureMap; один из них — это метод `For`, с которым вы уже знакомы. Как вы увидите далее в этом разделе, еще одним методом, использующимся для конфигурирования контейнера, является метод `Scan`, имеющий такое описание:

```
public void Scan(Action<IAssemblyScanner> action);
```

Обратите внимание на то, что метод `Scan` принимает в качестве входного параметра делегат. Когда вы создаете блок кода, определяющий метод `Scan`, возникает ситуация, в которой один блок кода находится в другом блоке кода — отсюда и произошло название «вложенное замыкание».

В отличие от контейнера Castle Windsor, преобразование `IIngredient` к `SauceBéarnaise` показанным ранее способом не препятствует вам разрешать тип `SauceBéarnaise`. Так и есть, компоненты `sauce` и `ingredient` будут соответствующим образом разрешены в следующем фрагменте кода:

```
container.Configure(r =>
    r.For<IIngredient>().Use<SauceBéarnaise>());
var sauce = container.GetInstance<SauceBéarnaise>();
var ingredient = container.GetInstance<IIngredient>();
```

Как говорилось в подразделе 10.1.2, преобразование `IIngredient` к `SauceBéarnaise` с помощью контейнера Castle Windsor заставляло конкретный класс (`SauceBéarnaise`) «исчезнуть», и вам приходилось использовать *переадресацию типов*, чтобы разрешить оба типа. В этих дополнительных действиях нет необходимости, если вы используете контейнер StructureMap, который позволяет разрешать оба типа, `IIngredient` и `SauceBéarnaise`. В обоих случаях возвращаемые объекты имеют тип `SauceBéarnaise`.

В реальных приложениях нам всегда необходимо преобразовать более чем одну абстракцию, поэтому мы должны сконфигурировать множественные преобразования. Это можно сделать с помощью всего одного вызова метода `Configure` или же использовать множественные последовательные вызовы. Эти два примера эквивалентны:

```
containér.Configuré(r =>
{
    r.For<IIngrédiént>()
        .Usé<SaucéB arnaisé>();
    r.For<ICoursé>()
        .Usé<Coursé>();
});
```

```
containér.Configuré(r => r
      .For<IIngrédiént>()
      .Usé<SaucéB arnaisé>());
containér.Configuré(r => r
      .For<ICoursé>()
      .Usé<Coursé>());
```

Пример справа использует два последовательных вызова для конфигурирования контейнера, пример слева передает блок кода с большим количеством утверждений одному вызову метода `Configure`. Результатом выполнения обоих примеров является регистрация корректных преобразований интерфейсов `ICourse` и `IIngredient`. Однако многократное конфигурирование одной абстракции приведет к интересным результатам:

```
container.Configure(r =>
    r.For<IIngredient>().Use<SauceBéarnaise>());
container.Configure(r =>
    r.For<IIngredient>().Use<Steak>());
```

В этом примере вы регистрируете интерфейс `IIngredient` дважды. Если вы разрешите `IIngredient`, вы получите экземпляр класса `Steak`. Последняя конфигурация имеет приоритет, но и предыдущие конфигурации не забываются. Контейнер StructureMap обрабатывает множественные конфигурации одной абстракции, мы вернемся к этой теме в подразделе 11.3.1.

Существуют более продвинутые варианты конфигурирования контейнера StructureMap, но вполне возможно сконфигурировать все приложение, используя лишь методы, описанные здесь. Чтобы обойтись без излишнего явного управления конфигурацией, можно использовать более традиционный подход — автоматическую регистрацию.

Автоматическая регистрация

Во многих случаях большинство регистраций будут похожими друг на друга. Такие регистрации тяжело обслуживать, а явная регистрация каждого компонента часто является контрпродуктивной.

Рассмотрим библиотеку, содержащую множество реализаций `IIngridient`. Вы можете зарегистрировать каждый класс отдельно, но в результате все равно получите множество похожих вызовов метода `Register`. Более того, всякий раз, когда вы будете добавлять реализацию `IIngredient`, вы должны будете явно зарегистрировать ее внутри контейнера, если хотите, чтобы она была доступна для дальнейшей работы. Было бы более продуктивно поместить все реализации `IIgredient` в заданную сборку, которая будет зарегистрирована.

Это становится возможным благодаря методу `Scan`, который представляет собой еще один пример активного использования делегатов с контейнером StructureMap. Метод `Scan` доступен в классе `ConfigurationExpression`, доступ к которому есть внутри блока кода. Здесь мы видим в действии паттерн «Вложенное замыкание». Следующий пример конфигурирует все реализации интерфейса `IIngredient` одним махом:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
    }));
```

Метод `Scan` вложен в блок кода конфигурирования. Переменная `s` представляет собой экземпляр класса `IAssemblyScanner`, мы можем ее использовать для определения того, как должна сканироваться сборка и конфигурироваться типы.

Экземпляр класса `IAssemblyScanner` предоставляет несколько методов. Их можно применять для определения того, какие сборки следует сканировать и как сконфигурировать типы этих сборок. Мы можем использовать обобщенный метод `AssemblyContainingType` для нахождения сборки по представленному типу, но существует несколько других методов, позволяющих нам предоставить экземпляр класса `Assembly` или даже добавить все сборки из заданного файла.

Обширный набор методов дает возможность определять, какие типы нужно добавить и как их преобразовать. Задействование метода AddAllTypesOf позволяет легко добавлять типы, которые реализуют заданный интерфейс, но существует несколько других методов, позволяющих контролировать, как конфигурируются типы.

В предыдущем примере без дополнительных условий конфигурируются все реализации интерфейса IIngredient, но мы можем создать фильтры, которые позволят выбирать лишь несколько из них. Рассмотрим сканирование, основанное на соглашениях. Используя такое сканирование, вы можете добавлять только классы, имя которых начинается с Sauce:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
        s.Include(t => t.Name.StartsWith("Sauce"));
    }));
```

Этот пример отличается от предыдущего лишь тем, что в него добавлен метод Include, представляющий собой уже третий уровень вложенного замыкания. Метод Include принимает предикат, на основе которого определяет, должен ли включаться заданный тип Type. В рассматриваемом случае тип будет включен, если название типа начинается с Sauce.

Если нам нужен полный контроль над конфигурацией, основанной на соглашениях, мы можем определить собственное соглашение, реализовав интерфейс IRegistrationConvention. Листинг 11.1 показывает наше предыдущее соглашение, реализованное как специальное (Custom).

Листинг 11.1. Реализация собственного 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 интерфейс IRegistrationConvention, в котором определен только один член. Метод Process будет вызываться контейнером StructureMap для каждого типа сборки, определенного в методе Scan, поэтому вы должны

явно предоставлять набор граничных операторов, фильтрующих все типы, которые вам не нужны.

Использование граничных операторов гарантирует, что любой тип, проходящий через них, является реализацией интерфейса `IIngredient` и его имя начинается на `Sauce`, поэтому вы можете зарегистрировать его с помощью объекта `registry`. Обратите внимание на то, что метод `Registry` предоставляется с помощью внедрения методов, и это целесообразно, поскольку `IRegistrationConvention` определяет надстройку для контейнера StructureMap.

Вы можете 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 еще можете определить сборку вне рамок соглашения. Это позволяет вам варьировать источники типов для обработки без зависимости от соглашений. `SauceConvention` определен с использованием метода `Convention`. Этот метод требует задания `IRegistrationConvention`, поскольку аргумент `type` имеет конструктор по умолчанию. Но существует также метод `With`, который принимает экземпляр класса `IRegistrationConvention`. Его вы можете создать вручную так, как захотите.

Поскольку вы можете использовать метод Scan для сканирования всех сборок в заданном каталоге, в нем же можно реализовать дополнительный функционал, где все надстройки могут быть добавлены без перекомпиляции ядра приложения. Мы рассмотрели один из способов реализации позднего связывания, еще одним является конфигурирование с помощью XML.

Конфигурирование с помощью XML

Когда нам нужна возможность изменить конфигурацию без перекомпиляции приложения, необходимо конфигурировать контейнер с помощью XML.

СОВЕТ

Используйте конфигурирование с помощью XML только для тех типов, которые вам нужно изменить без перекомпиляции приложения. Для остальных применяйте автоматическую регистрацию или конфигурирование в коде.

Мы можем использовать специализированные XML-файлы для конфигурирования контейнера StructureMap или встроить конфигурацию в стандартный конфигурационный файл приложения. Однако, что удивительно, последний вариант напрямую не поддерживается, поэтому сначала взглянем на то, как использовать специализированный XML-файл.

Конфигурация может быть определена в XML и считана с использованием метода `AddConfigurationFromXmlFile`:

```
container.Configure(r =>
    r.AddConfigurationFromXmlFile(configName));
```

В этом примере параметр congifName — это строка, содержащая имя подходящего XML-файла. Если вы хотите использовать стандартный конфигурационный файл приложения, вы должны применять API AppDomain для определения пути файла, который задействуется в данный момент:

```
var configName =
    AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
```

ВНИМАНИЕ

Хотя статический класс ObjectFactory напрямую поддерживает считывание конфигурации из файла App.config, эта возможность не поддерживается для экземпляров контейнеров. Для решения этой проблемы рекомендуется использовать API AppDomain[1].

В дополнение к указыванию контейнеру StructureMap на файл вы можете передать XML-конфигурацию как XmlNode:

```
container.Configure(r =>
    r.AddConfigurationFromNode(xmlNode));
```

Это позволит вам получить XML-конфигурацию из случайных мест, а не только из файлов. Такими местами могут быть базы данных или встроенные ресурсы.

Независимо от источника XML, схема работы остается точно такой же. Рассмотрим простую конфигурацию, преобразующую интерфейс IIngredient к типу 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 подходящее сборке имя типа, как для абстракции, так и для реализации — контейнер StructureMap называет их плагинами и подключаемыми типами.

Если вы хотите встроить этот фрагмент XML в конфигурационный файл приложения, вы также должны зарегистрировать элемент StructureMap как раздел конфигурации:

```
<configSections>
  <section name="StructureMap"
           type="StructureMap.Configuration.
              ➥StructureMapConfigurationSection, StructureMap"/>
</configSections>
```

Конфигурирование с помощью XML — это хороший вариант, когда вам нужно изменить конфигурацию одного или более компонентов без перекомпиляции приложения. Однако, поскольку этот вариант довольно неустойчив, он подходит только для этих случаев, и вы должны использовать автоматическую регистрацию или конфигурирование в коде как основные средства конфигурации контейнера.

---

[1] Это подтверждено в беседе в Twitter между Джереми Миллером и мной: http://twitter.com/jeremyd-miller/statuses/18134210141.

СОВЕТ

Помните, что последняя конфигурация имеет приоритет? Вы можете применять такое поведение для перезаписи напрямую заданной конфигурации. Чтобы сделать это, вы должны помнить о необходимости считывать XML-конфигурацию после того, как будут сконфигурированы прочие компоненты.

В этом разделе мы рассмотрели основные конфигурационные API контейнера StructureMap. Хотя возможно написать один большой блок неструктурированного кода, лучше всего разбить конфигурацию на модули. Для этих целей можно использовать реестры.

11.1.3. Размещение конфигурации

Иногда приходится размещать логику конфигурации в группах, которые можно переиспользовать, и даже когда повторное использование не является основной целью, целесообразно дополнительно структурировать код, особенно если нужно сконфигурировать большое и сложное приложение.

С помощью контейнера StructureMap мы можем разместить конфигурацию в реестрах, которые представляют собой классы, наследующие от конкретного класса `Registry`. На рис. 11.4 показано отношение между классом `Registry` и методом `Configure`, использованным в подразделе 11.1.2. Напоминаю, что в этой главе экземпляр класса `ConfigurationExpression` имеет имя `r`. Класс `ConfigurationExpression` — это класс-потомок конкретного класса `Registry`.

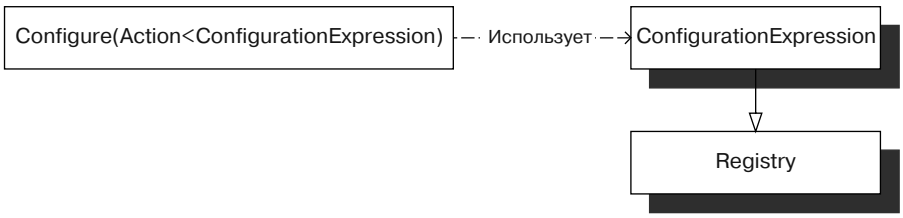


Рис. 11.4. Метод Configure класса Container принимает в качестве входного параметра делегат, который работает с ConfigurationExpression

Когда ранее в этой главе мы использовали метод `Configure`, мы создавали экземпляр класса `ConfigurationExpression` с именем `r`. Большая часть методов, которые мы вызывали у этой переменной, определена в классе `Registry`.

REGISTRY ИЛИ CONFIGURATIONEXPRESSION?

Большая часть конфигурационного API (например, методы `For` и `Scan`) доступна при наследовании непосредственно от класса `Registry`. Но мы не можем использовать методы, определенные в классе `ConfigurationExpression`. Какую функциональность мы теряем в этой ситуации?

В классе `ConfigurationExpression` определены только пять методов, они функционально делятся на две категории:

- считывание конфигурации из XML;
- добавление реестра.

Скорее всего, нам не понадобится добавлять реестры без участия класса `Registry`, поэтому отсутствие этих методов не кажется проблемой.

Конфигурация, определенная в XML, — это совершенно другой способ решения задачи. Мы выражаем определенную часть конфигурации приложения либо через XML, либо через конфигурационный API, но не применяем оба способа сразу. В свете этого невозможность определять источники XML из класса `Registry` не является ограничением.

В таком случае возможно ли для реестра наследовать от класса `ConfigurationExpression` вместо `Registry`? К сожалению, нет, поскольку конструктор класса `ConfigurationExpression` является внутренним.

Таким образом, класс реестра не может наследовать от класса `ConfigurationExpression`. Необходимо наследовать только от класса `Registry`.

Для реализации реестра нам необходимо создать класс, наследующий от класса `Registry`. Листинг 11.2 демонстрирует, как сконфигурировать класс по умолчанию `ICourse`, а также добавляет тип `IIngredient` из сборки. Он использует тот же API, который мы уже применяли в подразделе 11.1.2, однако теперь API размещен в отдельном классе.

Листинг 11.2. Реализация реестра

```
public class MenuRegistry : Registry
{
    public MenuRegistry()
    {
        this.For<ICourse>().Use<Course>();
        this.Scan(s =>
        {
            s.AssemblyContainingType<Steak>();
            s.AddAllTypesOf<IIngredient>();
        });
    }
}
```

Класс `MenuRegistry` наследует от класса `Registry` и определяет полную конфигурацию конструктора. У вас есть возможность получить из класса доступ к той части API класса `Registry`, которая определена как общедоступная, поэтому вы можете использовать методы `For` и `Scan` тем способом, который описан в подразделе 11.1.2 (1), — в этом случае единственным различием будет то, что вы реализуете не анонимный делегат, а анонимный конструктор. Вместо блока кода и вездесущей переменной `r`, к которой вы, возможно, уже привыкли к этому моменту, вы можете получить доступ к API с помощью переменной `this`.

После создания класса `MenuRegistry` вы теперь можете добавить его экземпляр в контейнер с помощью метода `Configure`:

```
container.Configure(r =>
    r.AddRegistry<MenuRegistry>());
```

Эта обобщенная версия метода AddRegistry требует, чтобы реализация Registry имела конструктор по умолчанию. Существует также необобщенная версия метода AddRegistry, принимающая в качестве входного параметра экземпляр класса Registry, давая нам полный контроль нам ее созданием.

ПРИМЕЧАНИЕ

Методы AddRegistry находятся в числе тех пяти методов, которые определены непосредственно в классе ConfigurationExpression, они недоступны другим классам кроме 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 последовательно добавить более одного реестра.

СОВЕТ

Реестры позволяют вам размещать в пакетах и структурировать конфигурационный код вашего контейнера. Используйте его вместо строкового конфигурирования, это сделает корень дерева зависимостей более удобочитаемым.

С помощью реестров мы можем сконфигурировать контейнер StructureMap, используя автоматическую регистрацию или конфигурирование в коде, а XML-конфигурация должна импортироваться непосредственно с помощью метода Configure. Мы можем объединить два подхода, получая одну часть конфигурации из XML, а другую — из одного или более реестров:

```
container.Configure(r =>
{
    r.AddConfigurationFromXmlFile(configName);
    r.AddRegistry<MenuRegistry>();
});
```

Как только контейнер будет сконфигурирован, вы можете начать разрешать его службы так, как это было описано в подразделе 11.1.1.

В этом разделе мы рассмотрели контейнер внедрения зависимостей StructureMap в общих чертах и изучили его основные механизмы: конфигурирование контейнера и последовательное разрешение его служб. Разрешение служб может быть без труда выполнено с помощью единственного вызова метода GetInstance, поэтому вся сложность заключается лишь в конфигурации контейнера. Это может быть сделано несколькими различными способами, в том числе с помощью императивного кода и XML. До сих пор мы рассматривали лишь самые простые API, впереди нас ждут более продвинутые техники. Одна из наиболее важных тем — это управление жизненным циклом компонента.

11.2. Управление жизненным циклом

В главе 8 мы обсудили управление жизненным циклом, в том числе наиболее распространенные стили жизненных циклов, такие как Singleton и Transient. Контей-

нер StructureMap поддерживает многие виды жизненных циклов и позволяет вам конфигурировать жизненный цикл всех служб. Жизненные циклы, показанные в табл. 11.2, доступны как часть API.

Таблица 11.2. Стили жизненных циклов контейнера StructureMap

Название	Комментарии
PerRequest	Название, применяемое в контейнере StructureMap для стиля PerGraph. Это стандартный стиль жизненных циклов. Экземпляры класса не отслеживаются контейнером
Singleton	Стандартный стиль Singleton
HttpContext	Название, применяемое в контейнере StructureMap для стиля Web Request Context
ThreadLocal	Создается один экземпляр на поток
Hybrid	Комбинация стилей HttpContext и ThreadLocal. HttpContext используется всегда, когда это только возможно (например, когда контейнер находится в веб-приложении), а ThreadLocal используется как «запасной аэродром»
HttpSession	Создается один экземпляр для каждой сессии HTTP. Используйте с осторожностью
HybridHttp-Session	Комбинация стилей HttpSession и ThreadLocal. HttpSession применяется, когда это только возможно (например, когда контейнер находится в веб-приложении), а ThreadLocal используется как «запасной аэродром»
Unique	Название, применяемое в контейнере StructureMap для стиля Transient

Реализации различных стилей жизненных циклов в контейнере StructureMap эквивалентны общим стилям жизненных циклов, описанным в главе 8, поэтому я не буду подробно их рассматривать в этой главе.

СОВЕТ

Стиль жизненных циклов, применяемый по умолчанию в контейнере, — это PerGraph. Как мы говорили в подразделе 8.3.3, в нем хорошо сбалансированы эффективность и безопасность. Когда вы используете потокобезопасные службы, стиль Singleton подойдет лучше, но вы должны явно конфигурировать такие службы.

В этом разделе вы увидите, как определить стили жизненных циклов для компонентов как в коде, так и в XML. В качестве более продвинутого сценария мы также рассмотрим реализацию пользовательского жизненного цикла, чтобы показать, что мы не ограничены лишь встроенными жизненными циклами. После прочтения этого раздела вы сможете использовать стили жизненных циклов контейнера StructureMap в вашем собственном приложении.

Начнем с рассмотрения конфигурирования стилей жизненных циклов для компонентов.

11.2.1. Конфигурирование стилей жизненных циклов

В этом подразделе мы рассмотрим способы управления стилями жизненных циклов компонентов с помощью контейнера StructureMap. Эти стили конфигурируются во время конфигурирования контейнера, вы можете определить их с помощью кода или XML. Мы изучим оба способа.

Конфигурирование стилей жизненных циклов в коде

Стили жизненных циклов конфигурируются как часть Configure API, который используется для общего конфигурирования компонентов. Применять его для конфигурирования стилей несложно:

```
container.Configure(r =>
    r.For<SauceBéarnaise>().Singleton());
```

Этот фрагмент кода конфигурирует конкретный класс `SauceBéarnaise` как `Singleton`. Поэтому при запросе класса `SauceBéarnaise` каждый раз возвращается один и тот же объект. Если вы хотите преобразовать абстракцию к конкретному классу, имеющему специфический жизненный цикл, объявление стиля жизненных циклов должно следовать между вызовами методов `For` и `Use`:

```
container.Configure(r =>
    r.For<IIngredient>().Singleton().Use<SauceBéarnaise>());
```

В этом фрагменте кода `IIngredient` преобразуется к типу `SauceBéarnaise`, а также меняет свой стиль на Singleton. Существуют и другие методы, похожие на Singleton, позволяющие вам объявлять один из многих других стилей, но не все стили жизненных циклов имеют собственный метод. Все стили жизненных циклов могут быть сконфигурированы с использованием обобщенного метода `LifecycleIs`. В качестве примера рассмотрим стиль Unique, не имеющий собственного метода. Он может быть сконфигурирован следующим образом:

```
container.Configure(r => r
    .For<SauceBéarnaise>()
    .LifecycleIs(new UniquePerRequestLifecycle()));
```

Метод `LifecycleIs` в качестве параметра принимает экземпляр класса `ILifecycle`, поэтому вы можете передать ему любой класс, реализующий этот интерфейс. Как вы увидите в подразделе 11.2.2, таким же способом мы будем конфигурировать компоненты собственным стилем жизненных циклов.

Все встроенные стили жизненных циклов контейнера StructureMap имеют соответствующие реализации интерфейса `ILifecycle`, кроме стиля Per Graph, поскольку он задается по умолчанию. Этот стиль конфигурируется неявно при отсутствии явного указания другого стиля. Все конфигурации, которые вы видели в разделе 11.1, использовали стиль Per Graph.

СОВЕТ

Отсутствие явного указания стиля приведет к тому, что компонент будет сконфигурирован как использующий стиль Per Graph. Передача методу LifestyleIs нулевого указателя приведет к таким же результатам.

Если мы пишем какой-либо код для общих целей, принимающий экземпляр типа `ILifecycle` и передающий его методу `LifecycleIs`, мы можем использовать его для конфигурирования компонента, применяющего стиль Per Graph. Нулевой указатель также подразумевает использование стиля Per Graph, поэтому результат работы следующих двух примеров будет одинаковым:

```
containér.Configuré(r => r
    .For<IIngrédiént>()
    .LifécycléIs(null)
    .Usé<SaucéB arnaisé>());
```

```
containér.Configuré(r => r
    .For<IIngrédiént>()
    .Usé<SaucéB arnaisé>());
```

СОВЕТ

Хотя подразумевается, что для стиля Per Graph возможно использовать нулевой указатель, лучшим решением будет полностью избегать объявления стиля жизненных циклов.

В то время как API позволяет работать с методом `Configure` и `ConfigurationExpression` дает возможность явно объявлять стили жизненных циклов, основанных на соглашениях, API Scan такого не допускает. В интерфейсе `IAssemblyScanner` нет метода, позволяющего объявлять стиль жизненных циклов для нескольких компонентов сразу.

Однако мы можем реализовать простой интерфейс `IRegistrationConvention`, который сделает это. Вы можете использовать соглашение `SingletonConvention` следующим образом:

```
container.Configure(r =>
    r.Scan(s =>
    {
        s.AssemblyContainingType<Steak>();
        s.AddAllTypesOf<IIngredient>();
        s.Convention<SingletonConvention>();
    }));
```

Обратите внимание на то, что такая же конфигурация была использована как первый пример автоматического конфигурирования в подразделе 11.1.2. Добавилась лишь строка кода, которая обеспечивает соглашение `SingletonConvention`. Это соглашение показано в листинге 11.3.

Листинг 11.3. Реализация соглашения об объявлении стилей жизненных циклов

```
public class SingletonConvention : IRegistrationConvention
{
    public void Process(Type type, Registry registry)
    {
        registry.For(type).Singleton();
    }
}
```

Как вы помните из предыдущего разговора о соглашении `IRegistrationConvention`, в листинге 11.1 метод `Process` вызывался для каждого включенного типа при операции сканирования сборки. В этом случае единственное, что вам нужно сделать, — объявить стиль жизненных циклов для каждого из них, используя метод `Singleton`. Этим вы сконфигурируете каждый тип так, что он будет применять стиль `Singleton`.

При использовании конфигурирования в коде мы можем сконфигурировать компоненты, применяя любые стили жизненных циклов тем способом, который сочтем нужным. Хотя этот способ конфигурирования является самым гибким, иногда может понадобиться сконфигурировать компоненты с помощью XML для обеспечения позднего связывания. Стили жизненных циклов можно объявить и таким способом.

Конфигурирование стилей жизненных циклов с помощью XML

Когда мы определяем компоненты с помощью XML, желательно конфигурировать стили жизненных циклов в том же месте. Это легко делается с помощью схемы, представленной в подразделе 11.1.2. Вы можете использовать опциональный атрибут `Scope` для объявления стиля:

```
<DefaultInstance PluginType="Ploeh.Samples.MenuModel.IIngredient,
    ➥Ploeh.Samples.MenuModel"
                 PluggedType="Ploeh.Samples.MenuModel.Steak,
    ➥Ploeh.Samples.MenuModel"
                 Scope="Singleton" />
```

По сравнению с примером из подраздела 11.1.2, этот фрагмент кода отличается лишь тем, что в нем появляется атрибут `Scope`, который конфигурирует экземпляр класса как Singleton. Поскольку в предыдущем примере этот атрибут не использовался, экземпляр класса автоматически начинал использовать стиль Per Graph.

Как в коде, так и с помощью XML, стили жизненных циклов сконфигурировать довольно легко. Во всех случаях эта операция выполняется в декларативной манере. Хотя конфигурировать стили достаточно просто, следует помнить, что некоторые из них создают объекты с длинным жизненным циклом, занимающие память на протяжении всего времени своего существования.

Предотвращаем утечки памяти

Как и другие контейнеры ВЗ, контейнер StructureMap создает для нас объектные графы. Но он не отслеживает за нас созданные объекты. Он может отслеживать их только для собственных целей, но это зависит лишь от жизненного цикла объекта. Например, для реализации жизненного цикла объекта, имеющего стиль Singleton, контейнер должен хранить ссылку на созданный объект. Такой принцип применяется и для объектов со стилем `HttpContext`, где все экземпляры классов хранятся в списке `HttpContext.Current.Items`. Однако по окончании HTTP-запроса все эти экземпляры выходят из области видимости и становятся целями для сборщика мусора.

С другой стороны, объекты со стилями жизненных циклов Per Graph и Transient не отслеживаются контейнерами после создания. Как было показано в листингах 8.7 и 8.8, экземпляры классов создаются и возвращаются без отслеживания их жизненного цикла внутри контейнера. Этот подход имеет преимущества и недостатки.

Поскольку контейнер StructureMap не сохраняет экземпляры объектов без необходимости, риск возникновения утечки памяти снижается. При использовании контейнера Castle Windsor утечки памяти гарантированы, если вы забудете вызвать метод `Release` для всех разрешенных графов объектов. У контейнера StructureMap такого недостатка нет, поскольку объекты будут автоматически обработаны сборщиком мусора при выходе из области видимости.

Недостатком принципа работы с объектами в контейнере Castle Windsor является тот факт, что объекты, готовые к удалению, не могут быть стопроцентно удалены. Поскольку нельзя явно разрешить граф объектов, мы не можем удалить ни один из таких объектов. Как говорилось в подразделе 6.2.1, это означает, что важность удаляемых API в службах неудаляемых автоматически значительно возрастает.

Короче говоря, контейнер StructureMap имеет фиксированное поведение и позволяет объектам подпасть под сборку мусора, как только они выйдут из области видимости в нашем коде. Но он требует, чтобы и наши классы также проявляли строго определенное поведение. Мы не может доверить контейнеру или вызывающему коду удаление какой-либо службы, поэтому необходимо ограничить использование удаляемых объектов одним методом.

Встроенные стили жизненных циклов контейнера StructureMap представляют собой довольно обширную коллекцию, которая удовлетворяет большинство повседневных нужд. Однако в некоторых случаях нам может понадобиться специализированный стиль жизненных циклов. Контейнер StructureMap позволяет нам создать такой собственный стиль жизненных циклов.

11.2.2. Разработка собственного стиля жизненных циклов

В большинстве случаев можно обойтись использованием обширной коллекции встроенных стилей жизненных циклов, предоставляемых контейнером StructureMap, но для решения некоторых особенных задач можно создать и собственный стиль жизненных циклов. В этом разделе вы научитесь делать это. После краткого обзора патерна «Шов», благодаря которому реализуется эта возможность, большую часть времени мы потратим на разработку примера.

Понимание API стилей жизненных циклов

В подразделе 11.2.1 вы уже взглянули на API, применяемый в контейнере StructureMap для создания стилей жизненных циклов. Метод `ILifestyleIs` принимает экземпляр интерфейса `ILifestyle`, который моделирует поведение стиля по отношению к контейнеру:

```
public interface ILifecycle
{
    string Scope { get; }
    void EjectAll();
    IObjectCache FindCache();
}
```

Из трех методов самым главным является метод `FindCache`. Он возвращает кэш, который используется контейнером StructureMap для поиска и вставки в него объектов с заданным стилем жизненных циклов. Интерфейс `ILifestyle` чаще всего служит аналогом абстрактной фабрики экземпляров типа `IObjectCache`, содержащих реализацию стиля. Этот интерфейс гораздо более сложен, но реализовать его не так трудно:

```
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();
}
```

Большая часть методов этого интерфейса реализует операции поиска, передачи или удаления объекта, основываясь на параметрах `Type` (тип) и `Instance` (экземпляр). На рис. 11.5 проиллюстрировано, как контейнер StructureMap взаимодействует с реализацией интерфейса `IObjectCache`.

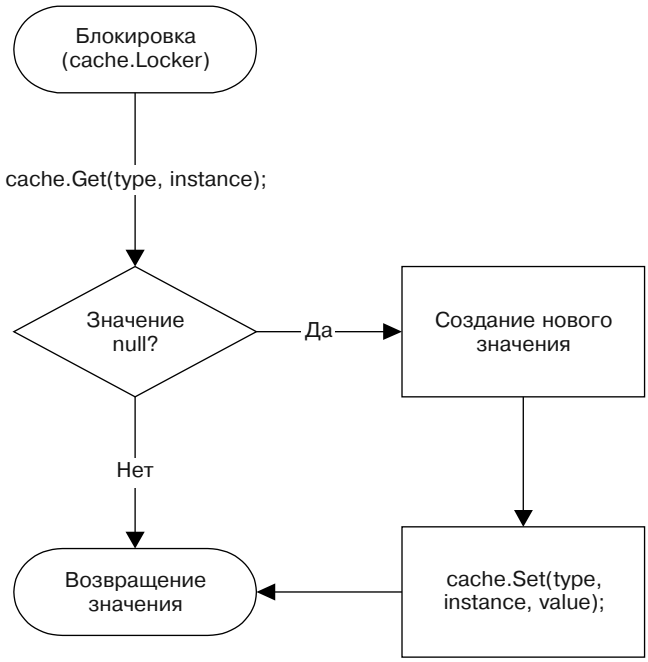


Рис. 11.5. Взаимодействие контейнера StructureMap с интерфейсом IObjectCache

Контейнер StructureMap сначала пробует получить требуемый экземпляр класса с помощью метода `Get`. Если этот метод возвращает нулевой указатель, контейнер создает требуемый экземпляр класса и добавляет его в кэш с помощью метода `Set`, перед тем как его вернуть. Рассмотрим его работу на примере.

Разработка кэширующего стиля жизненных циклов

В этом примере вы разработаете такой же кэширующий стиль жизненных циклов, который уже создавали для контейнера Castle Windsor в подразделе 10.2.3. Этот стиль кэширует и повторно использует экземпляры классов перед тем, как разрешить их.

ВНИМАНИЕ

Приведенный код игнорирует потоковую безопасность. Реальная реализация интерфейса ILifestyleManager должна быть потокобезопасной, поскольку, скорее всего, в какой-то момент времени разрешать объекты из контейнера попытаются несколько потоков.

Начнем с самого простого — листинг 11.4 показывает реализацию интерфейса ILifecycle.

Листинг 11.4. Реализация интерфейса ILifestyle

```
public partial class CacheLifecycle : ILifecycle
{
    private readonly LeasedObjectCache cache;

    public CacheLifecycle(ILease lease)
    {
        if (lease == null)
        {
            throw new ArgumentNullException("lease");
        }
        this.cache = new LeasedObjectCache(lease);
    }

    public void EjectAll()
    {
        this.FindCache().DisposeAndClear();
    }

    public IObjectCache FindCache()
    {
        return this.cache;
    }

    public string Scope
    {
        get { return "Cache"; }
    }
}
```

Класс CacheLifestyle реализует интерфейс ILifestyle. Он использует внедрение конструктора для получения экземпляра интерфейса ILease. Интерфейс ILease — это локальный вспомогательный интерфейс, который вы применяете для реализации класса CacheLifestyle. Он уже был создан нами в подразделе 10.2.3 и не имеет ничего общего с контейнером StructureMap или каким-либо другим контейнером внедрения зависимостей.

ПРИМЕЧАНИЕ

Пример реализации интерфейса ILease приведен в подразделе 10.2.3.

Вместо сохранения экземпляра ILease непосредственно в поле private, вы немедленно помещаете его ❶ в свою реализацию интерфейса IObjectCache, которая называется LeasedObjectCache. Этот кэш вы возвращаете ❷ с помощью метода FindCache.

ПРИМЕЧАНИЕ

Сравните сложность конструктора, показанного в листинге 11.4, со сложностью конструктора из листинга 10.2. Эта разница четко иллюстрирует преимущество внедрения конструктора над внедрением методов.

Хотя CacheLifestyle реализует интерфейс ILifestyle, настоящая реализация предоставляется пользовательским классом LeasedObjectCache, реализующим интерфейс IObjectCache.

Контейнер StructureMap предоставляет реализацию интерфейса IObjectCache, которая называется MainObjectCache. К сожалению, эта реализация не имеет виртуальных членов, которые мы могли бы переопределить для реализации нашего собственного стиля жизненных циклов. Вместо этого мы можем декорировать ее нашими реализациями интерфейса IObjectCache, LeasedObjectCache. В листинге 11.5 показан конструктор класса LeasedObjectCache.

Листинг 11.5. Конструктор класса 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 класса LeasedObjectCache вы используете стандартное внедрение конструктора, чтобы внедрить экземпляр класса ILease. Класс LeasedObjectCache является декоратором класса MainObjectCache, поэтому вы создаете его экземпляр и помещаете его в поле private. Обратите внимание на то, что поле objectCache имеет тип IObjectCache, поэтому легко можете расширить класс LeasedObjectCache, добавив перегруженный конструктор, который позволит вам внедрять любые реализации интерфейса IObjectCache.

Комбинация декорированного интерфейса IObjectCache и члена, имеющего тип ILease, значительно упрощает реализацию класса LeasedObjectCache. Листинг 11.6 показывает реализацию методов Get и Set, остальная часть реализации следует тому же шаблону.

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 контейнер StructureMap вызывает метод `Set`, сначала вам необходимо убедиться в том, что кэш не содержит никаких устаревших объектов. Когда этот метод возвращает значение, вы можете быть уверены в том, что если декорированный кэш содержит запрошенный экземпляр класса, вы можете безопасно возвратить его.

Наоборот, когда вызывается метод `Set`, вы мгновенно передаете метод декорированному кэшу объектов. Поскольку вы понимаете, что контейнер StructureMap использует кэш `IObjectCache` так, как это показано на рис. 11.5, вы знаете, что метод `Set` вызывается только в том случае, если контейнер создал новый объект. Это происходит в том случае, если в кэше не оказалось требуемого объекта. Это означает, что объект, переданный в параметре `value`, представляет собой только что созданный объект, поэтому вы можете безопасно обновить экземпляр класса `lease`.

Вспомогательный метод `CheckLease` вызывается многими реализациями интерфейса `IObjectCache` тем же способом, что и метод `Get`. Он очищает декорированный кэш, если истек срок существования объекта `lease`.

Теперь вы знаете, как реализовать собственный стиль жизненных циклов и любую зависимость, которую он может иметь. Все, что вам осталось узнать, — как его использовать.

Конфигурирование компонентов с применением собственного стиля жизненных циклов

Применять класс `CacheLifecycle` для конфигурирования компонента довольно просто. Это делается точно так же, как и конфигурирование при использовании любого другого стиля жизненных циклов:

```
var lease = new SlidingLease(TimeSpan.FromMinutes(1));
var cache = new CacheLifecycle(lease);
container.Configure(r => r
    .For<IIngredient>()
    .LifecycleIs(cache)
    .Use<SauceBéarnaise>());
```

В этом фрагменте контейнер конфигурируется так, что он использует стиль жизненных циклов `CacheLifestyle`. Объект, реализующий интерфейс `IInterface`, существует в течение одной минуты. Во время этого временного промежутка вы можете запрашивать столько графов объекты, сколько захотите, и всегда будете получать один и тот же экземпляр типа `SauceBéarnaise`, если граф будет содержать

экземпляр класса IInterface. Как только минута подойдет к концу, все последующие запросы будут получать новый экземпляр класса SauceBéarnaise.

Следует отметить, что способ реализации класса CacheLifestyle позволяет связать несколько объектов с одним объектом класса lease, например, так:

```
container.Configure(r =>

{
    r.For<IIngredient>().LifecycleIs(cache).Use<Steak>();
    r.For<ICourse>().LifecycleIs(cache).Use<Course>();
});
```

Сконфигурированные в этом фрагменте кода экземпляры классов ICourse и IIngredient будут синхронно удаляться и возобновляться. Иногда подобный результат является желаемым, иногда нет. Альтернативой этому способу применения стиля является использование двух раздельных экземпляров класса CacheLifestyle. Как показано в листинге 11.7, второй способ позволяет назначать разные временные промежутки.

Листинг 11.7. Использование различных экземпляров класса CacheLifestyle для каждого объекта 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>());
```

Задержка первого кэша равна одному часу. Независимо от того, как много или мало раз вам понадобится IIngredient, в этом временном промежутке вы будете получать тот же самый объект. Когда час пройдет, старый экземпляр будет удален, а новый будет универсально использоваться в течение следующего часа.

Время существования объекта класса ICourse равно 15 минут. В течение этих 15 минут вы будете получать один и тот же объект, но как только они истекут, будет возвращаться новый объект. Стоит отметить, что когда время существования экземпляра типа ICourse истечет, экземпляр типа IIngredient продолжит существование, поскольку время его существования больше. Хотя эти объекты имеют один стиль жизненных циклов, время жизни у них разное.

В листинге 11.7 использовались различные задержки при одном и том же типе SlidingLease. Такой подход не является обязательным — можно было бы использовать две совершенно разные реализации интерфейса ILease для каждого объекта.

Реализация собственного стиля жизненных циклов для контейнера StructureMap — не самая сложная задача. Она может показаться таковой только на бумаге, но на практике вы быстро поймете, что необходимо реализовать всего два класса, а *самый* сложный метод (`CheckLease`) состоит из одной условной конструкции и двух строк кода.

Несмотря на это, реализовывать собственный стиль жизненных циклов вам потребуется довольно редко. Обширный набор встроенных стилей вполне достаточен для решения повседневных задач.

На этом мы закончим нашу экскурсию по управлению жизненными циклами компонентов контейнера StructureMap. Компоненты могут быть сконфигурированы с использованием разных стилей, это верно даже в том случае, когда мы реализуем одну и ту же абстракцию. Теперь необходимо рассмотреть работу с несколькими компонентами.

11.3. Работа с несколькими компонентами

У контейнеров внедрения зависимостей возникают проблемы с функциональными неопределенностями. При использовании внедрения конструкторов исходный конструктор предпочитается переопределенным, поскольку всегда очевидно, какой конструктор применять, если нет другого выбора. Это происходит и в случаях, когда абстракции преобразовываются к конкретным типам. Если мы попытаемся преобразовать несколько конкретных типов к одной абстракции, возникнет неопределенность.

Несмотря на неприятный побочный эффект в виде неопределенностей, зачастую необходимо работать с несколькими реализациями одного интерфейса. Это могут быть следующие ситуации:

- различные конкретные типы должны использоваться разными потребителями;
- зависимости являются последовательностями;
- применяются декораторы.

В этом разделе мы рассмотрим все перечисленные случаи и увидим, как контейнер StructureMap справляется с каждым из них. Когда мы закончим, вы сможете регистрировать и разрешить компоненты даже в тех случаях, когда используется несколько реализаций одной абстракции.

Сначала взглянем на то, как мы можем предоставить более полное управление, чем то, которое предоставляет автоматическое подключение.

11.3.1. Выбираем одного кандидата из нескольких

Автоматическое подключение — очень удобный и мощный инструмент, но он предоставляет нам мало возможностей управления. До тех пор, пока абстракции явно преобразовываются к конкретным типам, никаких проблем не возникает, но как только мы предоставим несколько реализаций одного интерфейса, проявится неопределенность.

Сначала вспомним, как контейнер StructureMap справляется со множественными регистрациями одной абстракции.

Конфигурирование нескольких реализаций одного плагина

Как было показано в подразделе 11.1.2, вы можете конфигурировать несколько плагинов для одной службы:

```
container.Configure(r =>
{
    r.For<IIngredient>().Use<SauceBéarnaise>();
    r.For<IIngredient>().Use<Steak>();
});
```

В этом примере классы `Steak` и `SauceBéarnaise` регистрируются с плагином `IIngredient`. Последняя регистрация имеет приоритет, поэтому при разрешении `IIngredient` с помощью метода `container.GetInstance<IIngredient>()` вы получите экземпляр класса `Steak`. Однако вызов метода `container.GetAllInstances<IIngredient>()` возвратит список `IList<IIngredient>`, содержащий как экземпляр класса `Steak`, так и объект класса `SauceBéarnaise`. Так и есть, последовательные конфигурации не забываются, но получить доступ к ним труднее.

ПРИМЕЧАНИЕ

Последняя конфигурация заданного типа имеет приоритет. Это определяет тип по умолчанию для возвращаемого объекта.

Если существуют сконфигурированные экземпляры плагина, которые не могут быть разрешены при вызове метода `GetAllInstances`, контейнер StructureMap сгенерирует исключение, объясняющее, что существуют зависимости, которые не могут быть удовлетворены. Это согласуется с поведением метода `GetInstance`, но отличается от поведения контейнера Castle Windsor или MEF.

В листинге 11.8 показано, как можно предоставить подсказки, которые в дальнейшем помогут при выборе одного компонента из нескольких.

Листинг 11.8. Именование экземпляров типа Instances

```
container.Configure(r =>
{
    r.For<IIngredient>()
        .Use<SauceBéarnaise>()
        .Named("sauce");
    r.For<IIngredient>()
        .Use<Steak>()
        .Named("meat");
});
```

Вы можете дать каждому сконфигурированному объекту уникальное имя, которое в дальнейшем может быть использовано для отличения каждого из них от прочих похожих объектов.

Основываясь на именах объектов, определенных в листинге 11.8, вы можете разрешить как экземпляр класса `Steak`, так и объект класса `SauceBéarnaise` следующим образом:

```
var meat = container.GetInstance<IIngredient>("meat");
var sauce = container.GetInstance<IIngredient>("sauce");
```

Обратите внимание на то, что вы передаете тот ключ, который был использован в качестве имени при конфигурировании.

Учитывая, что вы должны всегда разрешать службы в одном корне компоновки, скорее всего, на этом уровне вы не столкнетесь с проблемой неопределенности.

СОВЕТ

Если вы обнаружите, что вызываете метод GetInstance, передавая ему определенный ключ, подумайте, смогли бы ли вы изменить свой подход так, чтобы не возникало неопределенности.

Вы можете использовать именованные экземпляры для того, чтобы потом с легкостью отделить их из множества альтернатив при конфигурировании зависимостей для заданного плагина.

Конфигурирование именованных зависимостей

Не менее полезным, чем автоматическое подключение, может быть переопределение нормального поведения для получения полного контроля над зависимостями. Это также может быть полезным, чтобы справиться с неопределенным API. В качестве примера рассмотрим такой конструктор:

```
public ThreeCourseMeal(ICourse entrée,
    ICourse mainCourse, ICourse dessert)
```

В этом случае у вас есть три зависимости с одинаковым типом, каждая из которых представляет собой концепцию, *отличную от других*. В большинстве случаев вы можете захотеть преобразовать каждую из зависимостей к отдельному типу. Листинг 11.9 показывает, как вы могли бы зарегистрировать преобразования типа `ICourse`.

Листинг 11.9. Конфигурирование именованных зависимостей

```
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, преобразовывая тип `Rilettes` к регистрации, названной `entrée`, `CordonBleu` — к регистрации, названной `mainCourse`, и `MousseAuChocolat` к регистрации, названной `dessert`.

Основываясь на этой конфигурации, вы можете зарегистрировать класс `ThreeCourseMeal` так, как показано в листинге 11.10.

Листинг 11.10. Переопределение автоматического связывания

```
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")));
```

Фрагмент кода конфигурации начинается как обычно, в данном случае выполняется преобразование интерфейса `IMeal` к конкретному классу `ThreeCourseMeal`. Но далее вы расширяете конструкцию методом `Ctor`. Метод `Ctor` (сокращенно от constructor, «конструктор») позволяет вам определять, как должен преобразовываться параметр конструктора заданного типа. Если существует только один параметр заданного типа, вы можете использовать переопределенный метод, позволяющий не передавать имя параметра. Однако из-за того, что конструктор класса `ThreeCourseMeal` принимает три параметра типа `ICourse`, нам необходимо идентифицировать параметр по его имени — `entrée`.

Метод `Ctor` возвращает объект, позволяющий вам определять, какое значение будет назначено параметру конструктора. Метод `Is` дает возможность использовать `IInstanceExpression<ICourse>` для выбора именованного объекта — еще один пример применения паттерна вложенного замыкания. Вы можете повторить эту идиому программирования и для следующих двух параметров.

ПРИМЕЧАНИЕ

В этом примере я давал конфигурируемым объектам имена параметров, представляющих их, но это не является обязательным требованием. Можно назвать объекты так, как вам захочется, в то время как имена параметров, очевидно, связаны с именами настоящих параметров конструктора.

ВНИМАНИЕ

Идентификация параметров по их именам — это удобный способ работы с ними, но он плохо приспособлен к рефакторингу. Если переименовать параметр, то можно сломать конфигурацию (в зависимости от используемого инструмента рефакторинга).

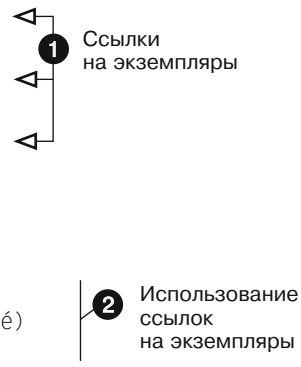
Переопределение автоматического подключения с помощью явного преобразования параметров к именованным объектам — это универсальное решение. Мы можем сделать это даже при конфигурировании именованных объектов, когда объекты находятся в одном выражении, а сам конструктор — совершенно в другом, поскольку имя — единственный параметр, связывающий именованный экземпляр с параметром. Этот подход можно применять всегда, но он довольно ненадежен, если необходимо управлять большим количеством имен.

Подключение ссылок на экземпляры

Когда существует возможность определить экземпляры объектов и конструктор в одном выражении, можно решить одну задачу более изящно. Новый способ продемонстрирован в листинге 11.11.

Листинг 11.11. Использование ссылок на экземпляры для переопределения автоматического подключения

```
containér.Configuré(r =>
{
    var éntr é =
        r.For<ICoursé>().Usé<Rilléttés>();
    var mainCoursé =
        r.For<ICoursé>().Usé<CordonBléu>();
    var déssért =
        r.For<ICoursé>().Usé<MousséAuChocolat>();

    r.For<IMéal>()
        .Usé<ThréécourséMéal>()
        .Ctor<ICoursé>("éntr é").Is(éntr é)
        .Ctor<ICoursé>("mainCoursé").Is(mainCoursé)
        .Ctor<ICoursé>("déssért").Is(déssért);
});
```



До сих пор мы игнорировали тот факт, что типичная последовательность вызовов методов `For/Chain` возвращает результат — так как не могли извлечь из этого явления пользу. Возвращаемые значения имеют тип `SmartInstances<T>`, и их можно использовать как ссылки на конфигурации, которые мы выполнили ❶. Вместо имен объектов, которые использовались в листинге 11.10, можно непосредственно задействовать подобные ссылки внутри одного из переопределенных методов `Is` ❷, соотнося каждую локальную переменную с соответствующим именованным параметром конструктора.

Хотя эта особенность позволяет нам избавиться от именования объектов, пресловутые «волшебные строки» (Magic Strings), определяющие параметры конструктора, все еще остаются. Этот API зависит от текстового ассоциирования конфигурации и имен параметров, поэтому он очень ненадежен. Лучше по возможности избегать его использования. Если вы вынуждены применять его только для того, чтобы справиться с неопределенностью, более качественным решением будет написание специального API, решающего эту проблему. Часто это приводит к улучшению общей структуры программы.

В следующем подразделе будет описан более однозначный и гибкий подход, позволяющий работать с произвольным количеством параметров конструктора. Сейчас вы узнаете, как контейнер StructureMap связывает списки и последовательности.

11.3.2. Подключение последовательностей

В подразделе 10.3.2 мы обсуждали, как переработать явно заданный класс `ThreeCourseMeal` в пригодный для решения более общих задач класс `Meal` со следующим конструктором:

```
public Meal(IEnumerable<ICourse> courses)
```

В этом подразделе мы рассмотрим, как можно сконфигурировать контейнер StructureMap для связывания экземпляров класса `Meal` с соответствующими

зависимостями от типа ICourse. Когда мы закончим, станет ясно, какие варианты вам доступны при конфигурировании объектов и ряда зависимостей.

Автоматическое подключение последовательностей

Контейнер StructureMap без проблем работает с различными последовательностями. Если мы хотим использовать все сконфигурированные объекты заданной надстройки, следует применять автоматическое подключение. Для примера, основываясь на сконфигурированных в листинге 11.9 объектах типа ICourse, вы можете сконфигурировать плагин IMeal следующим образом:

```
container.Configure(r => r.For<IMeal>().Use<Meal>());
```

Обратите внимание на то, что эта строка содержит стандартное преобразование абстракции к конкретному типу. Контейнер StructureMap автоматически распознает конструктор типа Meal и определяет, что корректный курс действий — это разрешение всех экземпляров класса ICourse. Когда вы разрешите IMeal, вы получите экземпляр класса Meal, содержащий все объекты типа ICourse из листинга 11.9: Rilettes, CordonBleu и MousseAuChocolat.

ПРИМЕЧАНИЕ

Простота автоматического подключения последовательностей в контейнере StructureMap выгодно отличается от ситуации из подраздела 10.3.2, в котором показана сложность включения той же функциональности в контейнере Castle Windsor.

Контейнер StructureMap автоматически обрабатывает последовательности и, пока мы не укажем выполнить какое-то иное действие, делает то, что мы от него ожидаем. А именно, он разрешает последовательность зависимостей ко всем зарегистрированным объектам того типа. Только когда мы явно укажем выбрать лишь несколько объектов из большого набора всех сконфигурированных объектов, он изменит свое поведение. Посмотрим, как можно этого добиться.

Выбор нескольких экземпляров из большого набора

Контейнер StructureMap по умолчанию внедряет все экземпляры объектов. Такая стратегия является корректной, но, как показывает рис. 11.6, иногда возникает необходимость выбрать только несколько сконфигурированных объектов из большого набора.

Когда мы ранее позволяли контейнеру StructureMap выполнить автоматическое связывание всех сконфигурированных объектов, это соответствовало ситуации, изображенной на рис. 11.6, *справа*. Если мы хотим сконфигурировать экземпляр так же, как это было показано на рис. 11.6, *слева*, мы должны явно определять, какие объекты следует использовать.

Когда существует возможность конфигурирования зависимостей и потребителя за один вызов метода Configure, мы можем использовать ссылки на объекты, как это было показано в листинге 11.11. Листинг 11.12 показывает эквивалентную конфигурацию для ситуации, когда конструктор ожидает последовательности зависимостей.

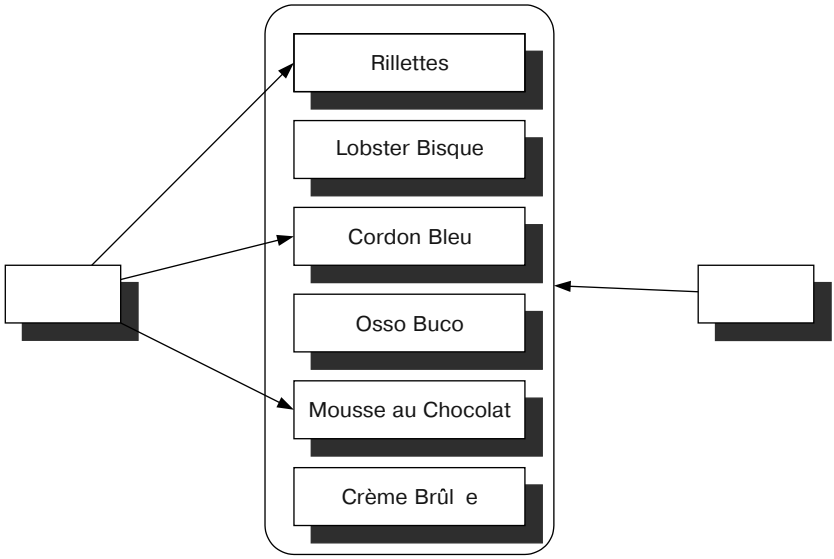


Рис. 11.6. В ситуации, продемонстрированной слева, мы хотим явно выбрать только некоторые зависимости из большого списка всех сконфигурированных объектов. Это отличается от ситуации, показанной справа, где мы хотим получить их все

Листинг 11.12. Использование ссылок на объекты для внедрения последовательности

```
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);
});
```

Как и в листинге 11.11, вы назначаете переменную каждому экземпляру типа `Instance`, возвращенному методом `Use`. Обратите внимание на то, что конфигурируются *четыре* экземпляра `ICourse`, несмотря на то что вы используете лишь три из них. Однако вам может понадобиться преобразование от типа `ICourse` к типу `LobsterBisque` для некоторых других целей, не продемонстрированных здесь. Поскольку вы не используете итоговую переменную `entrée`, вы можете полностью ее опустить, но я решил оставить ее для того, чтобы код оставался последовательным.

Поскольку конструктор класса `Meal` принимает в качестве входного параметра `IEnumerable<ICourse>`, вы можете использовать метод `EnumerableOf` для записи последовательности экземпляров `ICourse`, явно определенных в методе `Contains`. Туда вы передаете три объекта класса `Instance`, которые хотите применить.

Этот подход удобен в тех случаях, когда у нас есть возможность сконфигурировать все важные для нас экземпляры ICourse в том блоке, где находится конфигурация IMeal. Это возможно не всегда, может случиться так, что конфигурация экземпляров класса ICourse располагается в нескольких реестрах из различных сборок. В этом случае мы по-прежнему можем обратиться к ним по имени. В листинге 11.13 показан один вариант реализации этой возможности.

Листинг 11.13. Внедрение именованных объектов в последовательность

```
container.Configure(r => r
    .For<IMeal>()
    .Use<Meal>()
    .EnumerableOf<ICourse>().Contains(i =>
    {
        i.TheInstanceNamed("entrée");
        i.TheInstanceNamed("mainCourse");
        i.TheInstanceNamed("dessert");
    }));
```

Имея несколько именованных объектов, похожих на те, что мы создали в листинге 11.9, можно обратиться к каждому подобному объекту по имени при конфигурировании интерфейса IMeal. Как и в листинге 11.12, вы используете цепочку методов IEnumerableOf/Contains для записи последовательности зависимостей. На этот раз у вас нет переменных для этих объектов, поэтому вы должны искать их по имени. Переопределенный метод Contains позволяет вам передавать вложенное замыкание, указывающее, какие именованные объекты вы хотите внедрить в экземпляр класса Meal.

Контейнер StructureMap умеет работать с последовательностями. До тех пор пока нам не нужно выбрать лишь несколько объектов из списка всех плагинов заданного типа, контейнер автоматически выполняет правильные команды. Автоматическое подключение работает не только с единичными объектами, но с последовательностями, контейнер преобразовывает последовательность ко всем сконфигурированным объектам соответствующего типа.

Потребители, которые ожидают последовательность зависимостей, являются наиболее очевидными пользователями множества экземпляров одной абстракции. Но перед тем как мы закроем эту тему, необходимо рассмотреть последний — и, возможно, несколько удивительный — случай, когда в дело вступает множество экземпляров.

11.3.3. Подключение декораторов

В подразделе 9.1.2 мы разобрали случаи, когда паттерн проектирования «Декоратор» полезен при реализации сквозной функциональности. По определению декораторы представляют несколько типов одной абстракции. Как минимум, у нас есть две реализации абстракции — сам декоратор и «декорированный» тип. Если использовать декоратор на декораторе, то абстракций будет больше.

Это еще одна ситуация, в которой существуют несколько регистраций одной службы. В отличие от предыдущих разделов, эти регистрации не являются кон-

цептуально одинаковыми, они представляют собой зависимости друг от друга. В этом разделе вы увидите, каким образом можно сконфигурировать контейнер StructureMap, чтобы корректно обработать этот паттерн. Существует множество способов сконфигурировать декоратор, мы рассмотрим три различных варианта достижения конечного результата. Каждый из них имеет свои преимущества и недостатки.

Декорирование с помощью ссылок на экземпляры объектов

Рассмотрим, как можно сконфигурировать класс `Breading`, являющийся декоратором `IIngredient`. Он использует внедрение конструктора для получения объекта, который требуется декорировать.

```
public Breading(IIngredient ingredient)
```

Для получения `Cotoletta` вы должны декорировать `VealCutlet` (еще одна реализация `IIngredient`) с помощью класса `Breading`. Одним из способов достижения этой цели является использование внутри одного вызова метода `Configure` ссылок на экземпляры:

```
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 значение, возвращаемое методом `Use`, для получения ссылки на объект. Переменная `cutlet` представляет собой сконфигурированное ассоциирование `IIngredient` с `VealCutlet`, и вы можете применять это для того, чтобы объявить этот объект подходящим для использования его в качестве параметра конструктора для класса `Breading`. Поскольку последняя конфигурация имеет приоритет, экземпляр класса `Breading` становится экземпляром, возвращаемым по умолчанию.

Когда вы укажете контейнеру разрешить `IIngredient`, он возвратит объект, основанный на экземпляре по умолчанию. Таким объектом будет объект `Breading`, поскольку вы предоставили дополнительное указание — он должен разрешить объект `cutlet` для параметра `IIngredient` класса. Конечным в итоге будет получен экземпляр класса `Breading`, содержащий объект `VealCutlet`.

Передача объектов безопаснее, чем передача строк, поэтому вышеописанный вариант следует выбирать в том случае, когда декоратор и «декорируемый» тип конфигурируются в одном вызове метода. Однако это возможно не всегда.

Декорирование с помощью именованных экземпляров

Иногда нам приходится работать с именами экземпляров, поскольку мы конфигурируем декоратор и «декорируемый» тип в вызовах различных методов. Бывает и так, что эти методы находятся в разных реестрах, которые расположены в различных сборках. В таких случаях мы не можем передать объект, поэтому нам следует работать со строками даже несмотря на то, что это менее безопасно.

Предположим, что мы уже сконфигурировали тип VealCutlet следующим образом:

```
container.Configure(r => r
    .For<IIngredient>()
    .Use<VealCutlet>()
    .Named("cutlet"));
```

Поскольку вы знаете имя объекта (cutlet), вы можете использовать его для конфигурирования класса Breading:

```
container.Configure(r => r
    .For<IIngredient>()
    .Use<Breading>()
    .Ctor<IIngredient>()
    .Is(i => i.TheInstanceNamed("cutlet")));
```

Как мы уже делали в листингах 11.10 и 11.13, здесь используется переопределенный метод Is, позволяющий предоставить блок кода, который идентифицирует именованный экземпляр. Опять же, здесь задействуется паттерн «Вложенное замыкание».

Если вы сравните два предыдущих примера, вам не покажется, что они похожи. В обоих случаях вы использовали метод Ctor<T>, чтобы передать параметр конструктору. Единственное различие заключается в способе идентификации параметра с помощью метода Is.

Последовательность методов Ctor/Is несколько лучше — мы можем применять ее для определения только одного параметра конструктора, даже если рассматриваемый конструктор принимает большее количество параметров. Параметры, для которых эта последовательность не использовалась, будут автоматически подключены, согласно стандартным алгоритмам контейнера StructureMap. Это довольно полезно, если необходимо явно сконфигурировать только один параметр из нескольких предложенных.

Однако это решение не является строго типизированным. Нет гарантии того, что рассматриваемый конструктор имеет параметр идентифицированного типа. Ранее такой параметр мог существовать, но затем дизайн поменялся, и теперь конструктор принимает параметры другого типа. Компилятор не знает об этом, поскольку он «верит нам на слово», когда мы вызываем метод Ctor с аргументом определенного типа.

Альтернативный подход предлагает более строго типизированное решение.

Декорирование с помощью делегатов

Вместо того чтобы ссылаться на параметр конструктора по имени или типу, мы можем написать строго типизированный блок кода, который *использует* конструктор. Хотя он имеет недостатки, которые мы рассмотрим позже, его преимуществом является то, что он предлагает строго типизированное решение. Поэтому он потенциально более безопасен.

Это звучит довольно отвлеченно, поэтому рассмотрим, как сконфигурировать тип Cotoletta таким способом:

```
container.Configure(r => r
    .For<IIngredient>().Use<VealCutlet>()
    .EnrichWith(i => new Breading(i)));
```

Метод `EnrichWith` является членом обобщенного класса `SmartInstance<T>`, его возвращает метод `Use`. В этом случае вы вызываете метод `Use` для аргумента `VealCutlet`, который возвращает объект типа `SmartInstance<VealCutlet>`. Метод `EnrichWith` принимает делегат, который принимает в качестве входного параметра `VealCutlet` и возвращает объект.

Вы можете соотнести делегат с блоком кода, который принимает в качестве входного параметра `VealCutlet`. Компилятор определяет, что переменная `i` является экземпляром класса `VealCutlet`, поэтому теперь вы можете реализовать блок кода, вызвав конструктор класса `Breading` с этой переменной типа `VealCutlet`.

Когда вы укажете компилятору разрешить тип `IIngredient`, он сначала создаст экземпляр класса `VealCutlet`, а затем передаст его в качестве входного параметра в блок кода, который вы определили с помощью метода `EnrichWith`. Когда этот блок кода будет исполняться, объект `VealCutlet` передастся конструктору класса `Breading`, и будет возвращен экземпляр типа `Breading`.

Преимущество данного подхода заключается в том, что в блоке кода вы пишете код, который *использует* конструктор класса `Breading`. Это такая же строка кода, как и другие, поэтому она проверяется компилятором. Теперь вы с большой долей уверенности можете утверждать, что, если скомпилируется метод `Configure`, тип `VealCutlet` будет корректно «декорирован».

Хотя строгое типизирование безопаснее, оно также требует больше времени на обслуживание, чем нестрогое. Если вы впоследствии решите добавить еще один параметр конструктора в классе `Breading`, код блока больше не скомпилируется и вы должны будете самостоятельно искать источник проблемы. В этом не было бы необходимости, если бы вы использовали метод `Ctor<T>`, поскольку контейнер StrucrureMap смог бы определить новый параметр с помощью автоматического подключения.

Итак, существует несколько различных способов сконфигурировать декораторы. Строго типизированный подход немного безопаснее, но может потребовать более длительного обслуживания. Слабо типизированный API более гибок и дает возможность контейнеру StructureMap обрабатывать изменения в нашем API, но из-за этого приходится частично пожертвовать безопасностью типов.

ПРИМЕЧАНИЕ

В этом разделе мы не рассмотрели паттерн «Перехват» во время исполнения. Хотя контейнер StructureMap имеет швы (Seams), позволяющие его использовать, у контейнера нет встроенной поддержки для динамических посредников. Можно применить эти швы для использования другой библиотеки (например, Castle Dynamic Proxy) для вызова таких классов, но поскольку эти библиотеки не являются частью контейнера StructureMap, мы не будем рассматривать их в этой главе.

Контейнер StructureMap позволяет работать с несколькими объектами разными способами. Мы можем сконфигурировать объекты как альтернативы друг другу или же как иерархические декораторы. Во многих случаях контейнер StructureMap сам определит, что именно нужно сделать, но мы всегда можем явно определить, как будут создаваться службы, если нам нужно явно управлять ими.

Могут также возникнуть ситуации, когда нам необходимо будет работать с API, которые не полностью сводятся к внедрению конструктора. К этому моменту вы узнали, как сконфигурировать объекты, настроить стили их жизненных циклов и как работать с несколькими компонентами. Но до сих пор мы позволяли контейнеру связывать зависимости, неявно подразумевая, что все компоненты используют внедрение конструктора. Поскольку такая ситуация довольно редка, в следующем разделе мы рассмотрим работу с классами, которые должны инстанцироваться особым образом.

11.4. Конфигурирование сложных API

До этого момента мы изучали способы конфигурирования компонентов, использующих внедрение конструктора. Одним из множества преимуществ внедрения конструкторов является тот факт, что контейнер внедрения зависимостей, например Castle Windsor, легко понимает, как создать все классы графа зависимостей.

Это действие становится тем менее прозрачным, чем менее определенным является поведение API. В этом разделе вы увидите, как работать с примитивными аргументами конструкторов, статическими фабриками и внедрением свойств. Все эти вопросы требуют от нас особого внимания. Начнем с рассмотрения классов, которые принимают в качестве параметров конструктора примитивные типы, например строки или целые числа.

11.4.1. Конфигурирование примитивных зависимостей

Все работает хорошо до тех пор, пока мы внедряем абстракции. Но ситуация становится более запутанной, когда конструктор зависит от примитивного типа, такого как строка, число или перечисление. Такая зависимость часто возникает в том случае, когда реализуется доступ к данным и конструктор класса принимает в качестве параметра строки соединения. Мы рассмотрим и более общий пример, касающийся любых строк и чисел.

Концептуально не имеет особого смысла регистрировать как компонент контейнера строку или число, а в контейнере StructureMap это и вовсе не сработает. Если мы попробуем разрешить компонент с примитивной зависимостью, сгенерируется исключение, даже если примитивный тип был зарегистрирован ранее.

В качестве примера рассмотрим следующий конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере параметр `Spiciness` является перечислением:

```
public enum Spiciness
{
    Mild = 0,
    Medium,
    Hot
}
```

ВНИМАНИЕ

Учтите, что перечисления уже не используются и должны быть переработаны в полиморфические классы. Однако в этом примере они нам очень пригодятся.

Вам понадобится явно указать контейнеру StructureMap, как именно разрешить параметр конструктора `spiciness`. Следующий листинг показывает, для этой цели можно использовать метод `Ctor<T>`:

```
container.Configure(r => r
    .For<ICourse>()
    .Use<ChiliConCarne>()
    .Ctor<Spiciness>()
    .Is(Spiciness.Hot));
```

В разделе 11.3 вы уже видели, как метод `Ctor<T>` может быть использован для переопределения автоматического связывания одного из параметра конструктора. В этом примере неявно утверждается, что конструктор класса `ChiliConCarne` имеет лишь один параметр, `Spiciness`. В противном случае вызов метода `Ctor<spiciness>()` будет неопределенным, а вам придется передавать имя параметра.

Метод `Ctor<T>` возвращает `SmartInstance<T>` различными способами. Существует пять переопределенных вариантов метода `Is`, один из которых позволяет получить экземпляр подходящего типа. Аргументом `T` в этом примере является тип `Spiciness`, поэтому вы передаете параметр `Spiciness.Hot` как конкретное значение.

Как говорилось в разделе 11.3, использование метода `Ctor<T>` имеет свои преимущества и недостатки. Если нам нужна более строго типизированная конфигурация, вызывающая конструктор или статическую фабрику, этот способ оптимален.

11.4.2. Создание компонентов с помощью блоков кода

Экземпляры некоторых классов не могут быть получены с помощью общедоступного (`public`) конструктора. Вместо этого для создания объектов вы должны использовать определенного рода фабрику. Это всегда проблематично для контейнеров внедрения зависимостей, поскольку по умолчанию они требуют общедоступный конструктор.

Рассмотрим этот пример — конструктор `public`-класса `JunkFood`:

```
internalJunkFood(stringname)
```

Даже несмотря на то, что класс `JunkFood` применяет ключевое слово `public`, его конструктор имеет модификатор `internal`. Очевидно, экземпляры класса `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 контейнера StructureMap этот API является сложным, поскольку в статических фабриках не существует определенных и прочно установленных соглашений. Мы можем использовать блок кода, как показано в следующем листинге:

```
container.Configure(r => r
    .For<IMeal>()
    .Use(() =>
        JunkFoodFactory.Create("chicken meal")));
```

Теперь цепочка методов `For/Use` будет идентична. Однако в этой версии вы используете другой перегруженный метод `Use`. Он позволяет передавать `Func<IMeal>`, это возможно путем сообщения блока кода, вызывающего статический метод `Create` класса `JunkFoodFactory`.

СОВЕТ

Если вы хотите разрешить класс ChiliConCarne, описанный в подразделе 11.4.1, и при этом он вам нужен в строго типизированном виде, вы можете использовать именно этот переопределенный метод Use для прямого вызова конструктора.

Когда вы закончите писать код для создания экземпляра, задумайтесь, будет ли этот подход лучше, чем простой вызов конструктора из кода? Применение блока кода внутри конструкции `For/Use` дает вам некоторые преимущества:

- вы ассоциируете тип `IMeal` с `JunkFood`;
- вы все еще можете сконфигурировать стиль жизненных циклов. Хотя блок кода вызывается для создания объекта, он будет вызываться не *всякий раз*, когда потребуется экземпляр класса. Пока вы используете стиль жизненных циклов Unique, иногда будет возвращаться кэшированный экземпляр.

Итак, существует пять различных вариантов переопределения метода `Use`. Мы можем использовать обобщенную версию для определения конкретного типа, но другие переопределения позволят нам передать конкретный экземпляр или блок кода, создающий конкретный экземпляр.

Последнее распространенное «отклонение» от внедрения конструктора, которое мы рассмотрим здесь, — это внедрение свойств.

11.4.3. Подключение с внедрением свойств

Внедрение свойств — это менее определенная форма внедрения зависимостей, поскольку компилятор не требует задавать значение записываемому свойству. Несмотря на это, контейнер StructureMap умеет работать с внедрением свойств и в случае наличия такой возможности задает значения записываемым свойствам.

Рассмотрим класс `CaesarSalad` (салат «Цезарь»):

```
public class CaesarSalad : ICourse
{
    public IIngredient Extra { get; set; }
}
```

Распространенным заблуждением является то, что в салате «Цезарь» есть мясо цыпленка. Сам по себе салат «Цезарь» является *салатом*, но, поскольку мясо цыпленка улучшает его вкус, многие рестораны предлагают добавить его в салат в качестве дополнительного ингредиента. Класс `CaesarSalad` моделирует эту ситуацию, предоставляя записываемое свойство, называющееся `Extra`.

Если вы сконфигурируете только класс `CaesarSalad` без явной адресации свойства `Extra`, это свойство не будет назначено. Вы все еще можете разрешить экземпляр, но свойство будет иметь значение по умолчанию, которое ему назначит конструктор (если это вообще произойдет).

Существует несколько способов сконфигурировать класс `CaesarSalad` так, что значение свойства `Extra` будет назначено соответствующим образом. Одним из таких способов является использование ссылок на объекты, такая ситуация возникала уже несколько раз в этой главе:

```
container.Configure(r =>
{
    var chicken = r.For<IIngredient>().Use<Chicken>();
    r.For<ICourse>().Use<CaesarSalad>()
        .Setter<IIngredient>().Is(chicken);
});
```

Из предыдущих примеров вы можете вспомнить, что метод `Use` возвращает экземпляр, который может быть записан как переменная. В листинге 11.10 и множестве последующих примеров вы использовали метод `Ctor`, чтобы обозначить параметр конструктора определенного типа. Метод `Setter<T>` работает точно так же, только для свойств. Вы можете передать экземпляр `chicken` в метод `Is`, чтобы контейнер StructureMap назначил свойство при создании объекта.

Когда вы разрешаете тип `ICourse`, основываясь на этой конфигурации, вы получите экземпляр типа `CaesarSalad` с объектом `chicken`, назначенным его свойству `Extra`. Это позволит вам управлять отдельными свойствами определенных типов. API, который в большей степени ориентирован на соглашения, позволяет указать контейнеру StructureMap применять все свойства заданного типа для внедрения свойств. В качестве примера мы можем указать, что все настраиваемые свойства `IIngredient` должны быть инжектированы в подходящие объекты.

В случае с классом `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>());
```

Метод `FillAllPropertiesOfType` позволяет указать, что всем записываемым свойствам типа `IIngredient` должно быть задано значение. Контейнер StructureMap будет использовать значение по умолчанию, сконфигурированное для типа `IIngredient`, поэтому когда вы разрешите тип `ICourse`, вы получите экземпляр класса `CaesarSalad`, в свойстве `Extra` которого записано значение `chicken` (то есть салат «Цезарь» с курицей).

Метод `FillAllPropertiesOfType` заполнит все записываемые (Writable) свойства определенного типа. Поэтому если другие конкретные классы также будут иметь записываемые свойства данного типа, то в них также будут внедрены сконфигурированные экземпляры. Это может быть удобно, если мы будем следовать соглашению, которое регламентирует использование внедрения свойств для определенных типов.

В этом разделе вы узнали, как работать с API, несводимыми к простому внедрению конструктора. Вы можете использовать один из нескольких переопределенных методов `Use` и `Is` для определения конкретных экземпляров или блоков кода, которые будут использованы для создания экземпляров. Вы также узнали, что внедрение свойств может быть явно сконфигурировано при оформлении экземпляров или действовать как соглашение для определенного типа.

11.5. Резюме

В этой главе мы рассмотрели контейнер внедрения зависимостей StructureMap и его особенности. Мы связали этот материал с принципами и паттернами, представленными ранее в книге. Контейнер StructureMap является старейшим контейнером внедрения зависимостей для .NET, но его возраст не снижает его качества. Это объясняется тем, что при его написании применялись вложенные замыкания, безопасное для типов конфигурирование API, а также основанное на соглашениях сканирование типов.

Использование вложенных замыканий — это одна из наиболее характерных черт контейнера. Чтобы воспользоваться ими в полной мере, необходимо уметь работать с делегатами и блоками кода.

Начать работать с контейнером StructureMap очень просто. Он поддерживает автоматическое подключение и сам определяет, как создавать конкретные типы, даже если они не были явно сконфигурированы. Это означает, что вы можете сосредоточиться на преобразовании абстракций к конкретным типам и, когда вы закончите свою работу, сможете разрешать графы объектов. API, сканирующий типы, позволяет сконфигурировать множество служб. Для этого требуется написать всего несколько строк кода, поскольку при конфигурировании используется подход, основанный на соглашениях.

Хотя необязательно конфигурировать конкретные службы, может понадобиться эта возможность при изменении стиля жизненных циклов. По умолчанию используется стиль Per Graph, поэтому, когда вы работаете с потокобезопасными службами, потенциально возможно увеличить эффективность, сконфигурировав их как Singleton. Это действие должно быть явным, хотя оно может быть выражено во время сканирования типов с применением специального соглашения о регистрации.

Контейнер не всегда отслеживает экземпляры объектов, поэтому он не предлагает API для разрешения графа объектов. Это фактически предотвращает утечки памяти в обычных случаях, но с другой стороны, практически неизбежно приводит к утечкам при применении удаляемых (Disposable) зависимостей. Из-за этой осо-

бенности важно реализовывать все зависимости так, чтобы они работали с удаляемыми типами лишь внутри своей логики.

Контейнер StructureMap умеет работать с последовательностями зависимостей. Когда класс зависит от последовательности экземпляров одного типа, контейнер StructureMap автоматически свяжет объект со всеми объектами типа «зависимость». Контейнер имеет понятное поведение, задаваемое по умолчанию. Поэтому явные шаги приходится предпринимать только в тех случаях, когда нам нужно выбрать лишь несколько доступных объектов.

Хотя возможно явно сконфигурировать декораторы, контейнер StructureMap не имеет соглашения для их подключения, также ему недостает возможностей динамического перехвата. Однако здесь применяются швы, которые могут быть использованы для того, чтобы добавить в контейнер динамический API-посредник, если нам понадобится одна из вышеперечисленных особенностей.

Поскольку контейнер StructureMap сильно зависит от вложенных замыканий, вас не должно удивлять то, что многие методы конфигурирования имеют перегруженные аналоги. Такие аналоги позволяют передать блок кода, который сработает при создании объекта. Хотя в этих перегруженных методах нет необходимости, если регистрируемые классы используют внедрение конструкторов, мы можем использовать их для одного или нескольких классов, объекты которых создаются нестандартным образом.

Контейнер StructureMap — это полнофункциональный контейнер внедрения зависимостей, предлагающий множество продвинутых возможностей. Его стандартное поведение великолепно, и он может быть с легкостью использован — особенно, если ему необходимо будет применить автоматическое подключение конкретных типов или последовательностей. С другой стороны, он не обеспечивает динамической поддержки паттерна «Перехват», а также не может избавляться от удаляемых зависимостей. Эти небольшие недостатки обусловлены принципами проектирования. Если вы не будете реализовывать удаляемые службы и предпочтете явное использование декораторов, а не применение «Перехвата», то контейнер StructureMap отлично вам подойдет. Дело в том, что он задействует эти ограничения, чтобы упростить остальные свои черты для пользователя.

Марк Симан

Внедрение зависимостей в .NET

Перевели с английского А. Барышнев, Е. Зазноба

Заведующий редакцией	*Д. Виницкий*
Ведущий редактор	*Е. Каляева*
Научный редактор	*О. Сивченко*
Художник	*Л. Адуевская*
Корректор	*О. Андросик*
Верстка	*А. Барцевич*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 22.07.13. Формат 70×100/16. Усл. п. л. 37,410. Тираж 1500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.