Внедрение зависимостей на платформе .NET

Марк Симан
Стивен ван Дерсен

ВТОРОЕ ИЗДАНИЕ

MANNING

Dependency Injection

Principles, Practices, and Patterns

STEVEN VAN DEURSEN
MARK SEEMANN

MANNING
SHELTER ISLAND

Марк Симан
Стивен Ван Дерсен

Внедрение зависимостей на платформе .NET

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2021

ББК 32.973.2-018
УДК 004.42
С37

Симан Марк, Дерсен Стивен ван

С37 Внедрение зависимостей на платформе .NET. 2-е издание. — СПб.: Питер, 2021. — 608 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1166-4

Парадигма внедрения зависимостей (DI) в течение минувшего десятилетия де-факто стала одной из доминирующих на платформе .NET и теперь обязательна к изучению для всех .NET-разработчиков. Это переработанное и дополненное издание классической книги «Внедрение зависимостей в .NET». Вы научитесь правильно внедрять зависимости для устранения жесткой связи между компонентами приложения. Познакомитесь с подробными примерами и усвоите основы работы с ключевыми библиотеками, необходимыми для внедрения зависимостей в .NET и .NET Core.

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-1617294730 англ.
ISBN 978-5-4461-1166-4

Краткое содержание

Часть IV. DI-контейнеры

Оглавление

Часть II. Каталог

Часть III. Чистое внедрение зависимостей

Часть IV. DI-контейнеры

Предисловие Марка Симана

С компанией Microsoft связан особый феномен эхокамеры. Microsoft — очень крупная организация, окруженная системой сертифицированных партнеров, увеличивающей ее размер на несколько порядков. Возможно, тем, кто встроен в эту систему, трудно оценить ее границы. Вполне вероятно, что при поиске решения проблемы, связанной с продукцией или технологией Microsoft, найдется ответ, предполагающий использование еще большего количества продуктов Microsoft. Неважно, что именно вы прокричите в эхокамере, ответ будет один — *Microsoft*!

К моменту моего поступления на работу в Microsoft в 2003 году я уже прочно был встроен в эту систему, проработав несколько лет на сертифицированных партнеров Microsoft, и это мне понравилось! Вскоре меня отправили на внутреннюю техническую конференцию в Новом Орлеане для изучения самых передовых технологий компании.

Сегодня я уже и не вспомню, что было на тех занятиях по продуктам Microsoft, которые я посещал, но последний день все же в памяти остался. В тот день, осознав, что никакие занятия не смогут утолить жажду сведений о новых технологиях, я думал в основном о том, как прилечу обратно в Данию. Поэтому хотел выбрать такое место, где можно было бы просмотреть электронную почту. Я пришел на занятие, которое посчитал не представлявшим для меня особого интереса, и включил ноутбук.

План занятия не был четко расписан и состоял из выступления нескольких докладчиков. Одним из них был бородатый парень по имени Мартин Фаулер (Martin Fowler), рассказывающий о разработке на основе тестирования (Test-Driven Development, TDD) и динамических имитациях, или мок-объектах. Об этом парне я никогда не слышал и к его речи особо не прислушивался, но кое-что из сказанного в моем сознании все же отложилось.

Вскоре после возвращения в Данию я получил задание переписать с нуля большую ETL-систему (ETL означает «извлечение, преобразование, загрузка» — extract, transform, load) и решил попробовать TDD (оказалось, совсем не зря). Все это потребовало применения динамических имитаций, а кроме того, мне нужно было управлять зависимостями. Я понял, что это довольно непростая, но захватывающая задача, и постоянно думал над ее решением.

То, что сначала было побочным эффектом интереса к TDD, стало истинной страстью. Я провел множество исследований, изучил массу интернет-публикаций на эту тему, сам написал несколько статей, экспериментировал с кодом и обсуждал этот вопрос со всеми, кто был готов меня выслушать. Все чаще мне приходилось искать вдохновение и рекомендации вне эхокамеры Microsoft. Меня даже стали связывать с движением ALT.NET, хотя я никогда не проявлял особой активности в этом направлении. Я делал массу всевозможных ошибок, но постепенно у меня сложилось целостное понимание внедрения зависимостей — Dependency Injection (DI).

Когда представители издательства Manning обратились ко мне с предложением написать книгу о внедрении зависимостей в .NET, моей первой реакцией было: «А нужно ли это?» Я чувствовал, что все концепции, которые требуются разработчику для понимания DI, уже описаны в многочисленных публикациях в блогах. Можно ли к этому что-то добавить? Честно говоря, я думал, что DI в .NET была уже весьма заезженной темой.

Но, поразмыслив, я все же понял, что, хотя сведений множество, они сильно разбросаны по разным источникам, в которых слишком много разночтений. До первого издания этой книги не было публикаций, посвященных DI, в которых была бы предпринята попытка дать целостное описание данной технологии. Подумав еще, я пришел к выводу, что Manning предлагает мне решить весьма непростую задачу и дает прекрасную возможность собрать и систематизировать все, что я знал о DI.

В результате появились эта книга и предшествующее ей первое издание. Для представления и описания обширной терминологии и методологической основы внедрения зависимостей в книге используются среда .NET Core и язык C#, но я надеюсь, что ценность издания выйдет далеко за рамки платформы. Я думаю, что сформулированный здесь язык паттернов носит универсальный характер. Кем бы вы ни были, разработчиком в среде .NET или пользователем другой объектно-ориентированной платформы, я надеюсь, что эта книга поможет вам существенно повысить свою квалификацию разработчика программных продуктов.

Благодарности

Можно посчитать благодарности своеобразным клише, но человеческую природу не переделаешь. В процессе написания книги возникало множество поводов поблагодарить других людей, и всем им хотелось бы сказать спасибо.

Начнем с того, что на работу над книгой уходило все наше свободное время и мы почувствовали, насколько обременительным проект был для наших жен. Жена Марка Сесилия постоянно находилась рядом с ним, и он ощущал ее активную поддержку на протяжении всего процесса. Самое главное, что она понимала, насколько важен для мужа этот проект. Они так и идут по жизни рука об руку, и Марк старается выкраивать побольше времени на общение с ней и с их детьми Линой и Ярлом. Жена Стива Джудит создала ему все условия, необходимые для завершения этого грандиозного начинания, и очень обрадовалась, когда все закончилось.

На более профессиональном уровне хочется поблагодарить издательство Manning, предоставившее нам такую возможность. Инициатор проекта — Майкл Стивенс (Michael Stephens). Ведущими редакторами, зорко следившими за качеством текста, были Дэн Махарри (Dan Maharry), Марина Майклс (Marina Michaels) и Кристина Тейлор (Christina Taylor). Они помогали нам, выискивая слабые места в исходном тексте, подвергая его конструктивной критике.

В роли ведущего технического редактора выступала Карстен Стробек (Karsten Strøbæk), она вычитывала массу чернового материала и давала ценные отзывы. Карстен участвовала в выпуске Марком первого издания, выступая в роли технического корректора. В нынешнем издании техническим корректором был Крис Хенеган (Chris Heneghan), исправивший много закравшихся в исходный текст досадных ошибок и опечаток.

После того как текст был готов, начался технологический процесс. Им управлял Энтони Калькара (Anthony Calcara). В роли редактора на этой стадии выступал Фрэнсис Буран (Frances Buran), а Николь Биад (Nichole Beard) следил за качеством графического оформления книги и представленных в ней схем.

Далее хочется упомянуть различных специалистов, читавших исходный текст на разных этапах его разработки, которым мы благодарны за комментарии и замечания. Это Аджай Бхосале (Ajay Bhosale), Бьорн Нордблом (Björn Nordblom), Джемре

Менгу (Cemre Mengu), Деннис Селлингер (Dennis Sellinger), Эмануэле Ориджи (Emanuele Origgi), Эрнесто Карденас Кангауала (Ernesto Cardenas Cangahuala), Густаво Гомес (Gustavo Gomes), Игорь Кочетов (Igor Kochetov), Джереми Кейни (Jeremy Caney), Джастин Коулстон (Justin Coulston), Миккель Арентофт (Mikkel Arentoft), Паскуале Зирполи (Pasquale Zirpoli), Роберт Моррисон (Robert Morrison), Серджио Ромеро (Sergio Romero), Шон Лэм (Shawn Lam) и Стивен Бирн (Stephen Byrne), Иван Мартинович (Ivan Martinovic).

Мы получали отзывы и от многих участников проводимой издательством Manning программы раннего доступа — Manning Early Access Program (MEAP), которые задавали нам различные вопросы, помогавшие выискивать в тексте книги слабые места.

Особую благодарность мы выражаем Джереми Кейни (Jeremy Caney), который сначала был участником программы MEAP, но затем вошел в число научных редакторов. От него поступило громадное количество отзывов, касающихся как изложения, так и смыслового наполнения текста. Глубоко понимая технологию DI и методы разработки программных продуктов, он оказал нам бесценную помощь.

Особой благодарности заслуживает и Рик Слаппендель (Ric Slappendel). Рик подсказал нам, как с использованием DI создаются UWP-приложения. Его знание WPF, UWP и XAML сэкономило нам уйму времени и бессонных ночей и позволило полностью составить раздел 7.2 и сопровождающие его примеры кода. Без помощи Рика в книгу, наверное, так и не вошли бы страницы, где рассматривается UWP.

Мы также благодарны за участие в работе над книгой Алексу Мейер-Гливсу (Alex Meyer-Gleaves) и Тревису Иллигу (Travis Illig), которые просмотрели ранние версии главы 13 и дали отзывы об использовании новой конфигурации Autofac и поддержке паттерна «Декоратор».

И наконец, спасибо Могенс Хеллер Грэб (Mogens Heller Grabe), любезно позволившей нам воспользоваться изображением ее фена, включенного в розетку.

Об этой книге

Книга главным образом посвящена внедрению зависимостей — Dependency Injection (DI). Она повествует и о среде .NET, но ей здесь отводится гораздо более скромная роль. Хотя во всех примерах используется код на языке C#, основную часть обсуждаемых в книге методов можно с легкостью реализовать и на других языках и платформах. Фактически мы почерпнули множество базовых принципов и паттернов при изучении книг, где в качестве примеров приводился код на языках Java или C++.

Технология внедрения зависимостей представляет собой набор взаимосвязанных паттернов и принципов. Это, скорее всего, не конкретная технология, а образ мышления, связанный с разработкой кода. Конечной целью использования технологии DI является обеспечение легкого в сопровождении программного продукта под эгидой парадигмы объектно-ориентированного программирования (ООП).

Все понятия, приводимые в книге, относятся к объектно-ориентированному программированию. Задача, решаемая за счет технологии DI (простота сопровождения кода), универсальна, но предлагаемое решение дается в рамках объектно-ориентированного программирования на статически типизированных языках C#, Java, Visual Basic .NET, C++ и т. д. Вы не сможете применить DI к процедурному программированию, и эта технология вряд ли станет разумным решением на функциональных или динамических языках программирования.

Если рассматривать технологию DI отвлеченно от всего остального, то ее масштабы невелики, но в действительности она тесно переплетена с целым комплексом принципов и паттернов, используемых при объектно-ориентированном проектировании программных продуктов. Несмотря на то что книга от начала до конца сконцентрирована на технологии DI, в ней в свете тех перспектив, которые раскрываются благодаря использованию DI, рассматриваются многие из сопутствующих тем. Цель книги — не просто довести до вас специфику применения технологии DI, но и поднять вашу квалификацию разработчика объектно-ориентированных программных средств на более высокий уровень.

Для кого предназначена эта книга

Хотелось бы сказать, что книга предназначена для всех разработчиков, использующих среду .NET. Но в наши дни .NET-сообщество объединяет под своими знаменами широкий круг разработчиков веб-приложений, программ для настольных компьютеров, смартфонов, специалистов по созданию полнофункциональных интернет-приложений, сопряженных информационных систем, автоматизации делопроизводства, систем управления контентом и даже создателей компьютерных игр. Хотя среда .NET имеет объектно-ориентированный характер, объектно-ориентированным кодом занимаются не все разработчики.

Эта книга об ООП, и читатели как минимум должны быть заинтересованы в объектном ориентировании и понимать, что представляют собой интерфейсы. Конечно, неплохо, чтобы у них за плечами были хотя бы несколько лет профессиональной деятельности, а также знание паттернов проектирования или SOLID-принципов. Мы не думаем, что начинающие могут почерпнуть здесь что-то полезное для себя, поскольку книга предназначена главным образом для опытных разработчиков и проектировщиков программных средств.

Все примеры написаны на языке C#, следовательно, читатели, работающие с другими языками, использующимися в среде .NET, должны разбираться и в коде на C#. Книга может быть полезной и для читателей, знакомых с объектно-ориентированными языками, не применяющимися в среде .NET, поскольку материал, имеющий конкретное отношение к .NET-платформе, имеет довольно небольшой объем. Мы сами изучили множество книг о паттернах с примерами на Java, откуда неизменно черпали немало информации, поэтому надеемся, что верным будет и обратное утверждение.

Структура издания

Содержимое книги разбито на четыре части. В идеале хотелось бы, чтобы сначала вы прочитали ее от корки до корки, после чего использовали в качестве справочного пособия, но мы готовы к тому, что у вас другие приоритеты. Поэтому основная часть всех глав написана таким образом, чтобы можно было сразу же погрузиться в чтение той или иной главы и приступить к изучению книги с выбранного места.

Основным исключением из этого является часть I. В ней содержится общее введение в технологию DI, которое лучше всего изучить последовательно. Часть II представлена каталогом паттернов и им подобных сведений, а вот в части III, самой большой, технология DI исследуется с трех различных позиций. Часть IV книги содержит каталог из трех библиотек DI-контейнеров.

Итак, часть I представляет собой общее введение в технологию внедрения зависимостей. Если вы не знакомы с DI, то чтение нужно начинать именно с этого места, а если она вам уже известна, то ознакомление с содержимым этой части книги все равно пойдет вам на пользу, так как в ней вводится множество общих положений и терминов, используемых в остальных частях книги. В главе 1 дается общее представление о технологии DI, а также рассматриваются цели ее приме-

нения и преимущества, получаемые от этого. В главе 2 приводится пространный и развернутый пример кода с сильной связанностью, а в главе 3 объясняется, как тот же самый пример можно реализовать с помощью технологии DI. В отличие от других частей книги главы части I лучше читать последовательно, причем с самого начала.

Часть II представляет собой каталог паттернов, антипаттернов и примеров проблемного кода. Именно здесь можно найти рекомендации по способам реализации DI и предупреждение об опасностях, которых следует остерегаться. В главе 4 представлен каталог паттернов проектирования DI, а в главе 5 — каталог антипаттернов проектирования. В главе 6 рассматриваются обобщенные решения наиболее часто возникающих проблем. Поскольку все главы имеют вид каталога, в каждой из них содержится набор практически не связанных друг с другом разделов, предназначенных как для последовательного, так и для раздельного чтения.

В части III технология DI представлена с трех разных ракурсов: компоновки объектов (Object Composition), управления временем жизни (Lifetime Management) и перехвата (Interception). В главе 7 рассматриваются способы реализации DI в качестве надстройки над существующими средами выполнения приложений — ASP.NET Core и UWP, а также при использовании консольного приложения. В главе 8 речь идет о способах управления временем жизни зависимостей, позволяющих не допустить утечки ресурсов. Хотя структура этой главы менее строгая, чем предыдущих глав, ее основная часть может служить каталогом широко известных жизненных циклов зависимостей. Остальные три главы содержат описание способов компоновки приложений со сквозной функциональностью. В главе 9 раскрываются основы перехвата с применением декораторов, а в главах 10 и 11 углубленно анализируются понятия аспектно-ориентированного программирования. Именно здесь мы пожинаем плоды предшествующей работы, поэтому во многих отношениях считаем это место кульминационным моментом всей книги.

Часть IV представляет собой каталог библиотек DI-контейнеров. Она начинается с выяснения в главе 12 того, что представляют собой DI-контейнеры и как они вписываются в общую картину. В каждой из остальных трех глав весьма подробно рассматривается конкретный контейнер: Autofac, Simple Injector и Microsoft.Extensions.DependencyInjection. Для экономии места описание контейнеров в этих главах дается в довольно сжатой форме, и вы можете проявить интерес только к одному или двум наиболее подходящим контейнерам. По многим признакам эти три главы могут рассматриваться как весьма большой набор приложений.

Чтобы не рассматривать относящиеся к технологии DI принципы и паттерны в контексте конкретных API контейнеров, основной материал книги, исключая часть IV, не содержит ссылок на конкретный контейнер. Именно поэтому контейнерам отведена часть IV. Мы надеемся, что обобщенный характер раскрытия темы этой книги позволит ей оставаться актуальной как можно дольше.

Понятия, раскрываемые в первых трех частях, применимы и к тем библиотекам контейнеров, описание которых не вошло в часть IV. Есть неплохие контейнеры, находящиеся в свободном доступе, которые, к сожалению, мы не смогли рассмотреть. Но все же надеемся, что в книге есть множество полезного материала, который можно предложить для изучения пользователям тех библиотек, которыми поддерживаются эти контейнеры.

Стандарт оформления кода и материалы для скачивания

В этой книге приводится множество примеров кода. Основная его часть на C#, но в разных местах есть небольшое количество на XML и JSON. Исходный код в листингах и тексте оформляется моноширинным шрифтом, чтобы отделить его от обычного текста.

Весь исходный код для этой книги написан на C# в среде Visual Studio 2017. Приложения ASP.NET Core написаны на версии ASP.NET Core v2.1.

Лишь немногие из технических приемов, рассматриваемых в этой книге, зависят от современных особенностей языка. Мы стремились выдерживать разумный баланс между консервативными и современными стилями программирования. Когда мы пишем код для работы, мы гораздо чаще используем современные особенности языка, но здесь по большей части самыми новыми возможностями являются обобщения и LINQ-запросы. Нам ни в коем случае не хотелось бы навести вас на мысль, что технология внедрения зависимостей может применяться только при использовании самых современных языков.

Написание примеров кода для книги сопряжено со специфическими трудностями. По сравнению с монитором современного компьютера в книге можно разместить лишь очень короткие строки кода. Очень хотелось написать код в лаконичном стиле с короткими, но малопонятными именами методов и переменных. Такой код почти не похож на настоящий и понятный, даже если под рукой IDE-среда и отладчик, а в книге уловить в нем какой-либо смысл практически невозможно. По нашему мнению, использование как можно более легких для прочтения имен играет очень важную роль. Чтобы все поместилось на странице, иногда приходилось прибегать к необычным разрывам строк. Весь представленный код должен пройти компиляцию, но порой имеет немного необычный вид.

В коде применяется ключевое слово `var` из языка C#. В нашем профессиональном коде, где строку не нужно вписывать в ширину книжной страницы, зачастую используется иной стиль программирования, без `var`. Здесь для экономии места в тех случаях, когда мы считаем, что явное объявление затрудняет чтение кода, используем ключевое слово `var`.

В качестве синонима для слова «тип» в книге часто применяется слово «класс». В среде .NET классы, структуры, интерфейсы, перечисления и т. п. по своей сути являются типами, но поскольку слово «тип» в обычном языке имеет слишком много разнообразных значений, его использование зачастую затрудняет понимание текста.

Основная часть приведенных в книге примеров кода относится к общему для всей книги примеру интернет-магазина с внутренними вспомогательными управляющими приложениями. Возможно, это не самый впечатляющий пример из встречавшихся в книгах по программированию, но он был выбран по нескольким причинам.

- Большинству читателей эта предметная область хорошо понятна. Возможно, она кому-то покажется скучной, но зато не станет отвлекать от освоения технологии DI.

- Признаться, мы так и не нашли более подходящей предметной области, достаточно разнообразной по своему содержанию, куда бы вписывались всевозможные сценарии, намеченные нами к рассмотрению в книге.

В качестве примеров мы создали довольно большой объем кода, основная часть которого не вошла в книгу. Фактически почти весь он был написан с прицелом на использование принципа разработки на основе тестирования — Test-Driven Development (TDD), но поскольку книга посвящена не TDD, модульные тесты в ней обычно не приводились.

Исходный код всех примеров из этой книги можно скачать на веб-сайте издательства Manning: www.manning.com/books/dependency-injection-principles-practices-patterns. Так же смотрите репозиторий авторов книги: https://github.com/DependencyInjection-2nd-edition/codesamples. Инструкции по компиляции и запуску кода находятся в файле README.md в корневом каталоге пакета для скачивания.

Об авторах

Стивен ван Дерсен (Steven van Deursen) — фрилансер из Нидерландов, занимающийся разработками и проектированием в среде .NET и получивший богатый опыт работы в этой сфере начиная с 2002 года. Он живет в Неймегене, занимается программированием и зарабатывает этим на жизнь. Помимо программирования Стивен увлекается боевыми искусствами, любит поесть и, конечно же, выпить хорошего виски.

Марк Симан (Mark Seemann) — программист, проектировщик программных средств и лектор, живет в столице Дании Копенгагене. Разработкой программных средств он занимается с 1995 года, а технологией TDD увлекается с 2003-го, в том числе шесть лет был консультантом, разработчиком и проектировщиком в компании Microsoft. В настоящий момент Марк занимается разработкой ПО. Он увлекается чтением, рисованием, игрой на гитаре, любит хорошее вино и изысканную еду.

Об иллюстрации на обложке

На обложку книги помещено изображение под названием «Женщина из Водняна», где показана жительница небольшого городка, расположенного на хорватском полуострове Истрия, омываемом водами Адриатического моря. Эта репродукция из альбома традиционных хорватских костюмов середины XIX века, составленного Николой Арсеновичем и опубликованного Этнографическим музеем хорватского города Сплита в 2003 году. Иллюстрации были любезно предоставлены библиотекарем Этнографического музея Сплита, который находится в Римском квартале средневекового центра города, там же, где и руины дворца уединения римского императора Диоклетиана, построенного приблизительно в 304 году нашей эры. Альбом содержит превосходные цветные изображения жителей различных регионов Хорватии, дополненные описаниями костюмов и сцен из повседневной жизни. Воднян — это важный в культурном и историческом отношении город, расположенный на возвышенности с прекрасным видом на Адриатическое море и известный своими многочисленными храмами и драгоценными предметами культа. Женщина на обложке одета в длинную черную льняную юбку и короткий черный жакет поверх белой льняной рубашки. Жакет отделан синей вышивкой, а весь костюм завершает синий льняной фартук. На женщине также широкополая черная шляпа, шарф в цветочек и большие серьги в форме колец. Ее элегантный костюм выдает в ней городскую, а не сельскую жительницу. Народные костюмы в окружающей сельской местности более колоритны, сшиты из шерстяной ткани и украшены богатой вышивкой.

За последние 200 лет стиль одежды и жизнь изменились и различия между регионами, столь существенные в былые времена, постепенно стерлись. Сейчас трудно отличить друг от друга людей с разных континентов, не говоря уже о жителях деревень или городов, разделенных всего несколькими километрами. Возможно, культурное разнообразие заменили более разнообразная личная жизнь и, конечно же, многообразная и бурно развивающаяся жизнь в мире высоких технологий.

Издательство Manning воздает должное изобретательности и инициативности компьютерного бизнеса, выбирая для книжных обложек подобные приметы богатого разнообразия региональной жизни двухвековой давности, воскрешаемые в памяти благодаря иллюстрациям из старых книг и коллекций.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть I

Общее представление о месте, занимаемом внедрением зависимостей

Внедрение зависимостей (Dependency Injection, DI) — одна из концепций объектно-ориентированного программирования, чаще остальных подвергаемая неправильному толкованию. В отношении терминологии, цели и механизмов царит неопределенность. Как в действительности следует называть этот паттерн проектирования? Внедрением зависимостей? Инверсией зависимостей (Dependency Inversion)? Или инверсией управления (Inversion of Control)? А может, подключением стороннего кода (Third-Party Connect)? Является ли целью DI исключительно поддержка модульного тестирования, или же у нее более широкая область применения? Равнозначны ли понятия DI и локации сервисов (Service Location)? Есть ли необходимость в DI-контейнерах?

Теме DI посвящено огромное количество интернет-публикаций, журнальных статей, презентаций, используемых на различных конференциях, однако весь этот информационный поток грешит использованием противоречивой терминологии и плохими рекомендациями. Наблюдается повсеместная неразбериха, привносимая даже такими большими и влиятельными игроками, как компания Microsoft.

Так быть не должно. В этой книге представлена ясная терминология, не допускающая двоякого толкования. В подавляющем числе случаев мы принимали и уточняли уже существующую терминологию, определенную до нас другими людьми, но временами мы вносили и собственный скромный вклад, заполняющий имеющиеся пробелы. Все это очень помогло при определении области применения (границ) DI.

Одна из основных причин рассогласованности и плохих рекомендаций кроется в сильной размытости границ DI. Где заканчивается DI и начинается другая объектно-ориентированная концепция? Похоже, провести четкую разделительную линию между DI и другими аспектами создания качественного объектно-ориентированного кода не представляется возможным. Чтобы говорить о DI, приходится вникать в особенности и других концепций, таких как SOLID, Clean Code (чистый код) и даже Aspect-Oriented Programming (аспектно-ориентированное программирование). Достоверное изложение вопросов, касающихся DI, без этого попросту невозможно.

Первая часть книги поможет вам разобраться в том, какое место занимает DI по отношению к другим аспектам разработки программных средств, то есть получить общее представление.

В главе 1 будет дан краткий обзор DI, охватывающий цели технологии, принципы и получаемые преимущества, а также будет раскрыто содержание остальной части книги — обрисована общая картина, без описания подробностей. Если вам необходимо понять, в чем состоит суть DI и чем оно может заинтересовать вас, то начать нужно с изучения этой главы. При этом глава 1 допускает, что у вас могут отсутствовать даже поверхностные знания о DI. Тем же, кто уже имеет представление о DI, мы настоятельно советуем не пропускать этот информативный блок. Вполне может оказаться, что для любого читателя в ней найдется много интересного.

А вот главы 2 и 3 целиком отведены под один большой пример, который предназначен для того, чтобы дать более конкретное представление о DI. Чтобы показать отличие DI от более традиционного стиля программирования, в главе 2 демонстрируется обычная реализация приложения онлайн-торговли, имеющего

сильное связывание, которое затем, в главе 3, последовательно переделывается под использование DI.

В этой части DI будет рассматриваться в общих понятиях. Это значит, что мы не будем использовать так называемый DI-контейнер (DI Container). Возможность применения DI без использования контейнера существует. DI-контейнер — полезный, но вспомогательный инструмент. Поэтому в частях I, II и III DI-контейнеры фактически полностью игнорируются, и разговор о внедрении зависимостей ведется без их упоминания. Затем, в части IV, мы вернемся к вопросу о DI-контейнерах, чтобы проанализировать применение трех специализированных библиотек.

Часть I определяет содержание всей остальной книги. Она предназначена для тех читателей, кто только начинает знакомство с DI. Однако из беглого изучения ее глав могут извлечь определенную пользу и читатели с опытом практического применения DI, усвоив при этом терминологию, повсеместно используемую в настоящем издании. После прочтения части I книги должно сложиться четкое представление о терминологии и общих понятиях, пусть даже при этом некоторые подробности останутся все еще немного расплывчатыми. Не стоит пугаться — по мере погружения в тему все встанет на свои места, а части II, III и IV ответят на те вопросы, которые, скорее всего, появятся после прочтения части I.

1 Основы внедрения зависимостей: что, зачем и как

В этой главе

- Развеивание распространенных мифов о внедрении зависимостей.
- Осмысление цели внедрения зависимостей.
- Оценка преимуществ от внедрения зависимостей.
- Умение распознать условия, подходящие для внедрения зависимостей.

Вы наверняка слышали, что сделать беарнский соус (sauce béarnaise) достаточно сложно. Даже те, кто регулярно занимается приготовлением пищи, предпочитают не рисковать, заранее зная, что попытка будет обречена на провал. И, надо признать, напрасно, поскольку соус восхитителен. (Обычно его подают вместе со стейком, но он также отлично сочетается с белой спаржей, вареными яйцами и другими блюдами.) Некоторые исхитряются и прибегают к его заменителям вроде готовых соусов или быстрорастворимых сухих смесей, но по вкусу ничто не сравнится с оригиналом.

Беарнский соус представляет собой эмульсию из яичного желтка и сливочного масла, приправленную эстрагоном, кервелем, луком-шалотом и уксусом. В нем нет

воды. Приготовить его нелегко, а испортить можно на раз. Соус может свернуться или расслоиться, и, случись что-либо из этого, все придется выбрасывать. Процесс приготовления занимает около 45 минут, и первая попытка, оказавшаяся неудачной, уже не оставит времени на вторую. В то же время приготовить беарнский соус под силу любому шеф-повару. Освоение этого блюда входит в программу их профессионального обучения, так что они уверены в том, что беарнский соус готовить несложно.

Чтобы приготовить эту замысловатую приправу, не обязательно переквалифицироваться в профи. Всякий, кто учился этому, хотя бы раз терпел фиаско. Но стоит лишь немного вникнуть в нюансы, как соус станет получаться не хуже, чем у мастеров своего дела. *Внедрение зависимостей (DI)* представляется чем-то вроде беарнского соуса. Принято считать, что это довольно мудреная наука и если попытка воспользоваться этой технологией провалится, то времени на второй шанс может и не остаться.

ОПРЕДЕЛЕНИЕ

> Внедрение зависимостей представляет собой набор принципов и приемов проектирования программных продуктов, позволяющий разрабатывать слабосвязанный код.

Несмотря на все кажущиеся трудности технологии DI, научиться ее использованию ничуть не сложнее, чем освоить приготовление беарнского соуса. В процессе обучения можно допускать ошибки, но после того, как техника будет отточена, сбоев уже не возникнет.

Веб-сайт Stack Overflow, где собраны вопросы и ответы, касающиеся разработки программного обеспечения, предложил пользователям любопытный ответ на довольно непростой вопрос: «Как объяснить пятилетнему ребенку, что такое внедрение зависимостей?» Автором варианта, набравшего самую высокую оценку, стал Джон Манч (John Munsch), который привел удивительную по точности аналогию, доступную воображению юного исследователя[1]: *«Когда ты ищешь в холодильнике, чем бы полакомиться, могут случиться непредвиденные и неприятные вещи. Ты можешь забыть закрыть дверь холодильника, можешь взять из него то, что не разрешают брать мама с папой. Можешь даже наткнуться на просроченные продукты.*

Когда в следующий раз ты чего-то захочешь, просто скажи: "Я хочу после обеда съесть что-то сладкое". Когда ты сядешь за стол, мы дадим тебе все, что нужно».

С точки зрения объектно-ориентированной разработки программных средств это означает следующее: для предоставления необходимых служб сотрудничающие классы (пятилетний ребенок) должны зависеть от инфраструктуры (родителей).

---

[1] См. статью How to explain Dependency Injection to a 5-year old? Джона Манча (John Munsch) и др. (2009), https://stackoverflow.com/questions/1638919/.

ПРИМЕЧАНИЕ

> В терминологии DI часто встречаются понятия «компоненты» и «службы». Последние обычно представлены абстракцией, то есть определением чего-то предоставляющего службу. Реализацию абстракции часто называют компонентом, то есть классом, содержащим некое поведение. Поскольку и «служба», и «компонент» — слишком «избитые» понятия, вместо них в книге повсеместно будут использоваться такие термины, как «абстракция» и «класс».

Эта глава достаточно линейна по своей структуре. Сначала мы вводим понятие DI, включая цели и преимущества этой технологии. Хотя мы и приводим примеры, в целом эта глава содержит меньше кода, чем любая другая. Прежде чем приступить к введению в DI, рассмотрим основное назначение этой технологии — обеспечение сопровождаемости кода. Без этого невозможно понять истинное назначение внедрения зависимостей. Затем мы разберем пример (Hello DI!), после чего будут рассмотрены сфера применения данной технологии и получаемые от нее преимущества. Завершив изучение данной главы, вы будете готовы к усвоению более сложных понятий, излагаемых далее.

Большинству разработчиков DI может показаться весьма отсталым способом создания исходного кода, но, чтобы освоить технологию DI, сначала нужно разобраться с ее предназначением.

1.1. Создание сопровождаемого кода

Для чего служит DI? Эта технология является не столько самоцелью, сколько средством достижения результата. В конечном счете предназначением большинства методов программирования является предоставление работоспособного программного продукта наиболее эффективным способом. При этом один из аспектов — написание сопровождаемого кода.

Довольно быстро обнаружится (если только вы не пишете прототипы или приложения, которые никогда не доводятся до первого выпуска), что существующий код нуждается в расширении и сопровождении. В целом эффективность работы с таким кодом определяется степенью его сопровождаемости.

Отличным способом повышения сопровождаемости является *ослабление связывания*. Это было известно еще в 1994 году, когда «Банда четырех» работала над книгой «Паттерны проектирования»: *«программируйте в соответствии с интерфейсом, а не с реализацией»*[1].

Этот важный совет является не итогом, а, скорее, предпосылкой к написанию книги по паттернам проектирования. Слабое связывание придает коду расширяемость, а та, в свою очередь, делает его сопровождаемым. DI — не что иное, как технология, обеспечивающая слабое связывание. Более того, существует масса неверных представлений о DI, которые порой мешают правильному пониманию этой

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 38.

технологии. Чтобы по-настоящему понять механизм DI, вы должны забыть все, что (как вы полагаете) уже знаете.

1.1.1. Устоявшиеся мифы о DI

Возможно, вам еще не приходилось сталкиваться с DI или что-либо слышать об этой технологии, и это замечательно. Пропустите этот раздел и переходите сразу к подразделу 1.1.2. Но раз уж вы читаете эту книгу, то, скорее всего, данная тема уже была на слуху или встречалась в постах блогов.

Возможно, вы также обратили внимание, что все разговоры на данную тему сопровождались массой весьма агрессивных мнений. В этом подразделе мы рассмотрим четыре наиболее распространенных заблуждения о DI, появившихся за последние годы, и докажем их беспочвенность. Разберем следующие ложные представления:

- DI актуально лишь для позднего связывания (late binding).
- DI подходит лишь для модульного тестирования (unit testing).
- DI — раздутая разновидность абстрактной фабрики (Abstract Factory).
- DI не обходится без DI-контейнера.

Это весьма распространенные мифы. И, прежде чем приступить к изучению DI, их нужно развенчать.

Позднее связывание

В данном контексте *позднее связывание* относится к возможности замены частей приложения без повторной компиляции кода. Одним из примеров может послужить приложение, включающее дополнения от стороннего производителя (такое как Visual Studio). Другим примером может стать стандартное программное средство, поддерживающее различные среды выполнения.

Предположим, что имеется приложение, способное работать на нескольких движках баз данных (например, поддерживающее как Oracle, так и SQL Server). Для реализации этой функции вся остальная часть приложения взаимодействует с базой данных через интерфейс. Код приложения предоставляет различные реализации этого интерфейса для обращения к Oracle и к SQL Server соответственно. В данном случае для управления используемой реализацией в конкретной установке можно воспользоваться параметром конфигурации.

Мнение о том, что DI имеет отношение исключительно к подобному сценарию, является весьма распространенной ошибкой. В этом нет ничего удивительного, поскольку DI действительно допускает такой сценарий применения. Но считать это прямой аналогией было бы неправильно. Тот факт, что DI допускает позднее связывание, еще не означает, что данная технология имеет отношение только к сценариям позднего связывания. На рис. 1.1 показано, что позднее связывание является всего лишь одним из множества аспектов применения DI.

Позднее (динамическое) связывание

Рис. 1.1. Позднее связывание обеспечивается с использованием DI, но предположение о том, что эта технология применима только для сценариев позднего связывания, сродни тому, чтобы довольствоваться вертикальным кадром, а не панорамным снимком

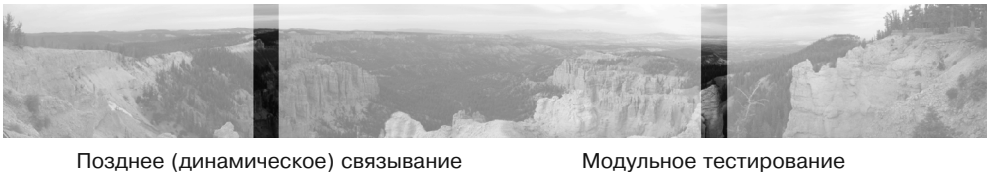
Если вы привыкли думать, что технология DI имеет отношение лишь к сценариям позднего связывания, то с этим заблуждением следует расстаться. Эта технология имеет куда более широкий спектр применения.

Модульное тестирование

Бытует и такое мнение, что DI имеет отношение лишь к поддержке модульного тестирования. Это в корне неверно, хотя DI, несомненно, является важной частью поддержки модульного тестирования. По правде говоря, мы решили освоить технологию DI, чтобы предолеть трудности, связанные с некоторыми аспектами *разработки на основе тестирования* (Test-Driven Development, TDD). За это время удалось не только разобраться в технологии DI, но и узнать, что другие специалисты используют ее для поддержки ряда сценариев, с которыми нам также уже приходилось сталкиваться.

Даже если вам еще не доводилось создавать модульные тесты (самое время заняться ими!), то технология DI будет по-прежнему актуальна благодаря всем тем преимуществам, которые она предлагает. Утверждение о том, что технология DI применима лишь для поддержки модульного тестирования, сродни утверждению, что она годится исключительно для поддержки позднего связывания. Хотя, как показано на рис. 1.2, это совершенно иной взгляд на данную тему, он настолько же узок, как и на рис. 1.1. Мы старались показать вам полную картину.

Если вы думали, что технология DI предназначалась только для модульного тестирования, избавьтесь от этого предположения. Диапазон применения DI гораздо шире модульного тестирования.



Позднее (динамическое) связывание Модульное тестирование

Рис. 1.2. Возможно, вы предполагали, что модульное тестирование является единственной целью применения DI. Аналогичный пример отказа от широкого видения в угоду узкому взгляду

Раздутая абстрактная фабрика

Наверное, самым опасным заблуждением будет считать, что технология DI включает в себя некую разновидность абстрактной фабрики общего назначения, которой можно воспользоваться для создания экземпляров зависимостей, необходимых вашему приложению.

Абстрактная фабрика

Абстрактная фабрика — это, как правило, абстракция, содержащая несколько методов, каждый из которых позволяет создавать объект определенного рода[1]. Чаще всего паттерн абстрактной фабрики используется для создания инструментария пользовательского интерфейса (UI) или клиентских приложений, предназначенных для запуска на нескольких платформах. Для достижения высокой степени повторного использования кода на всех платформах можно, к примеру, определить абстракцию `IUIControlFactory`, позволяющую создавать для потребителей конкретные разновидности элементов управления, в частности текстовых полей и кнопок:

```
public interface IUIControlFactory
{
    IButton CreateButton();
    ITextBox CreateTextBox();
}
```

Для каждой операционной системы (OS) могут быть различные реализации этой абстракции `IUIControlFactory`. В данном случае существует только два фабричных метода, но, в зависимости от приложения или инструментария, их может быть гораздо больше. Важно отметить, что абстрактная фабрика задает предопределенный перечень фабричных методов.

Во введении к этой главе прозвучало, что «для предоставления необходимых служб сотрудничающие классы *должны зависеть от инфраструктуры*». Каким было ваше первое мнение об этом предложении? Возможно, инфраструктура представилась вам некой разновидностью службы, которую можно запросить для получения нужных зависимостей? Если так, то вы не одиноки. Многие разработчики и проектировщики считают DI сервисом, которым можно воспользоваться для поиска других сервисов. Его называют локатором сервисов (Service Locator), но это — полная противоположность DI.

Локатор сервисов часто называют раздутой абстрактной фабрикой, поскольку в сравнении с обычной абстрактной фабрикой список разрешаемых типов не определен и может быть бесконечен. Обычно в нем имеется один метод, позволяющий создавать всевозможные типы, что очень похоже на следующий код:

```
public interface IServiceLocator
{
    object GetService(Type serviceType);
}
```

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 78.

ВАЖНО

Если DI видится вам в качестве локатора сервисов (то есть в виде универсальной фабрики), то с этим заблуждением стоит распрощаться. Технология DI является противоположностью локатора сервисов; она представляет собой способ структурирования кода, позволяющий полностью избавиться от запроса зависимостей. Вместо этого вы требуете их предоставления от потребителя.

DI-контейнеры

С предыдущим заблуждением тесно связано и представление о том, что технологии DI требуется DI-контейнер. Если вы придерживались предыдущего неверного представления о том, что DI применяется в качестве локатора сервисов, то из этого нетрудно было прийти к выводу, что DI-контейнер может взять на себя роль локатора сервисов. Такое вполне возможно, но этим использование DI-контейнера отнюдь не ограничивается.

DI-контейнер является дополнительной библиотекой, упрощающей создание классов при подключении приложения, но это не обязательно. Когда приложения создаются без DI-контейнера, это называется чистым внедрением зависимостей (Pure DI). Возможно, придется поработать чуть дольше, но в остальном не нужно будет идти на компромиссы ни с какими принципами DI.

ОПРЕДЕЛЕНИЕ

Чистая технология DI представляет собой практику применения DI без DI-контейнера[1].

ВАЖНО

Неправильно полагать, что технологии DI требуется DI-контейнер. DI представляет собой набор принципов и паттернов, а DI-контейнер является полезным, но вспомогательным инструментом.

Нам еще предстоит объяснить, что такое DI-контейнер, как и когда его следует использовать. Более подробно мы поговорим об этом в конце главы 3, а часть IV будет целиком посвящена применению DI-контейнеров.

Вам может показаться, что за разоблачением четырех мифов о DI последуют убедительные аргументы против каждого из них. Так и есть. В определенном

[1] В первом издании этой книги, «Внедрение зависимостей в .NET», используется понятие «DI для бедных». Теперь оно заменено понятием «Чистая технология DI», но не удивляйтесь, если увидите в Интернете старую терминологию. Чтобы разобраться в причине замены, обратитесь к статье Марка Симана Pure DI 2014 года, которая находится по адресу https://blog.ploeh.dk/2014/06/10/pure-di/.

смысле эту книгу можно считать одним большим аргументом против этих весьма распространенных заблуждений, и к этим темам мы еще обязательно вернемся. Например, в разделе 5.2 будет поднят вопрос, почему локатор сервисов является антипаттерном.

Наш опыт подсказывает нам, что от ложных фактов необходимо обязательно избавиться, поскольку люди зачастую пытаются скорректировать то, о чем им говорится относительно DI, и подвести все это под мнимые истины, устоявшиеся в их прежнем представлении. При этом, прежде чем они наконец поймут, что некоторые из наиболее устоявшихся у них представлений неверны, потребуется определенное время. Нам хотелось бы избавить вас от подобного опыта. По возможности прочитайте эту книгу так, будто до этого о DI вы не имели ни малейшего понятия.

1.1.2. Определение цели DI

Использование DI является не конечной целью, а средством достижения определенного результата. Технология DI позволяет применять слабое связывание, которое, в свою очередь, делает код сопровождаемым. Это весьма громкое заявление, и, хотя можно было бы отослать вас за пояснениями к устоявшимся авторитетам вроде «Банды четырех», будет вполне резонно все же объяснить его состоятельность.

Для большей доходчивости в следующем разделе разработка программного продукта и ряд паттернов проектирования сравниваются с электропроводкой. Есть все основания считать это вполне допустимой аналогией. Она даже использовалась для разъяснения сути разработки программных продуктов технически не подкованным людям.

В этой аналогии используются четыре конкретных паттерна проектирования, поскольку именно они зачастую фигурируют применительно к технологии DI. В этой книге вам часто будут встречаться примеры применения трех из них — «Декоратора» (Decorator), «Компоновщика» (Composite) и «Адаптера» (Adapter), а четвертый паттерн, «Нулевой объект» (Null Object), будет рассмотрен в главе 4. Если вы с ними еще не знакомы, ничего страшного: все окончательно прояснится по мере чтения книги.

Разработка программного продукта все еще считается относительно новой профессией, поэтому зачастую приходится придумывать способы реализации подходящей архитектуры. Но представители традиционных специальностей (например, строители) уже давным-давно все это придумали.

Заселение в дешевый отель

Остановившись в недорогом отеле, можно столкнуться с ситуацией, показанной на рис. 1.3. Отель для вашего удобства любезно предоставляет фен, но администрация, видимо, вам не доверяет и боится за сохранность прибора, поэтому он намертво присоединен к отверстию в стене. Руководство отеля решило, что замена украденного

фена обойдется в копеечку, поэтому пошло на явно ущербную реализацию оказываемой вам услуги.

Рис. 1.3. В дешевом номере отеля можно увидеть фен, напрямую вмонтированный в розетку. Это эквивалентно устоявшейся практике создания сильно связанного кода

Что будет, если фен выйдет из строя? Администрации отеля придется вызывать электрика. Чтобы исправить постоянно подключенный фен, нужно будет обесточить весь номер, временно сделав его непригодным для проживания. Затем специалисту понадобится особый инструмент, чтобы отключить старый фен и подключить новый. При благоприятном стечении обстоятельств электрик не забудет включить электропитание номера и вернется, чтобы проверить работоспособность прибора, но это если повезет... Вам это ничего не напоминает?

Именно такой подход предполагается при работе с сильно связанным кодом. Согласно этому сценарию фен сильно связан с розеткой и невозможно изменить одно, не затронув другое.

Сравнение электропроводки с паттернами проектирования

Обычно шнуры от электрических устройств не подключаются непосредственно к электропроводке, вмонтированной в стену. Вместо этого, как показано на рис. 1.4, используются вилки и розетки. Розетка определяет форму, которой должна соответствовать вилка.

В аналогии, проводимой с проектированием программного продукта, розетка — это интерфейс, а вилка со своим электроприбором — это реализация. Это означает, что в номере отеля (в приложении) имеется одна или (будем надеяться) несколько розеток и постояльцы (разработчики) по своему усмотрению могут подключать не только имеющиеся электроприборы, но и, возможно, свои собственные.

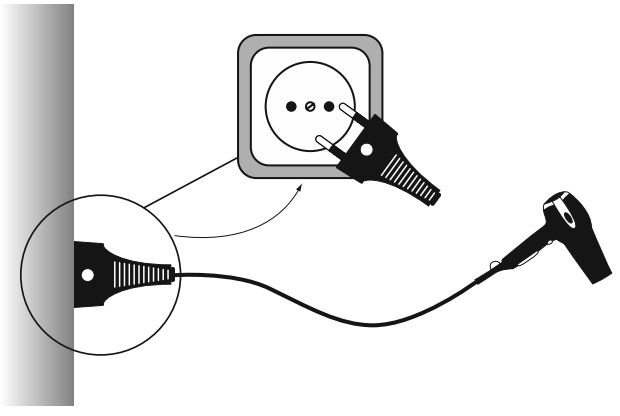


Рис. 1.4. Использование розеток и вилок определяет слабую связанность фена с проводкой в стене

В отличие от жестко встроенного фена розетки и вилки для подключения электрических приборов определяют модель со слабой связанностью. Поскольку вилка (реализация) подходит к розетке (реализует интерфейс) и может соответствовать ей по напряжению в вольтах и по частоте электрического тока в герцах (подчиняется контракту интерфейса), появляется возможность составления из техники различных конфигураций. Особый интерес вызывает то, что многие из этих самых распространенных комбинаций могут сравниваться с широко известными принципами и паттернами проектирования программных продуктов.

Начнем с того, что мы больше не привязаны к использованию одного лишь фена. Мы полагаем, что обычному читателю электроэнергия больше нужна для компьютера, а не для прибора для сушки волос. Теперь это не проблема: можно отсоединить фен и к той же розетке подключить компьютер (рис. 1.5).

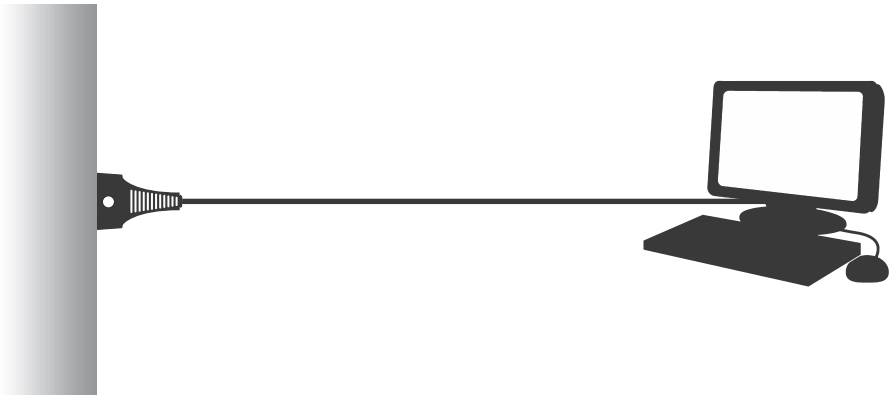


Рис. 1.5. Благодаря использованию розетки и вилки можно вместо фена с рис. 1.4 подключить компьютер. Это соответствует принципу подстановки Барбары Лисков

Принцип подстановки Барбары Лисков

Удивительно то, что идея розеток опередила появление компьютеров на десятилетия и все же им пригодилась. Создатели первых розеток даже не могли предвидеть появление персональных компьютеров, но за счет универсальности их изделия сейчас ими могут воспользоваться такие устройства, которых раньше не было и в помине.

Возможность переключения вилок (или реализаций) без необходимости менять розетку (или интерфейс) похожа на основной принцип разработки программных продуктов, который называется принципом подстановки Барбары Лисков. Он гласит, что мы должны быть в состоянии заменить одну реализацию интерфейса другой, не выводя из строя ни клиента, ни реализацию.

Что же касается технологии DI, то принцип подстановки Лисков является одним из наиболее важных принципов проектирования программных продуктов. Именно он позволяет решать возможные проблемы в будущем, даже если на данный момент предвидеть их затруднительно.

Если в компьютере пока нет необходимости, вилку можно вытащить от розетки. Даже если ничего не подключено к источнику питания, с номером ничего не случится, комната не взлетит на воздух. Иными словами, если отключить компьютер от встроенной в стену электропроводки, то ни настенная розетка, ни компьютер не выйдут из строя.

Что же касается программных средств, то клиент зачастую ожидает, что служба будет доступна. Если ее отключить, будет получено исключение `NullReferenceException`. Чтобы справиться с подобной ситуацией, можно создать реализацию ничего не выполняющего интерфейса. Этот паттерн проектирования известен как «Нулевой объект» (Null Object) и соответствует заглушке, применяемой для безопасности детей (вилке без провода, которую можно вставить в розетку). А поскольку используется слабое связывание, настоящую реализацию можно заменить чем-то абсолютно бездействующим, но не вызывающим проблем. Подобная ситуация показана на рис. 1.6.

Диапазон ваших возможных действий весьма широк. Если вы живете там, где случаются периодические отключения электричества, можно поддерживать работу компьютера, подключив его к источнику бесперебойного питания, ИБП. На рис. 1.7 показано подключение ИБП к настенной сетевой розетке и подключение компьютера к ИБП.

Компьютер и источник бесперебойного питания служат разным целям. У каждого из них своя зона ответственности (Single Responsibility), не вторгающаяся в область применения другого устройства. Вероятнее всего, ИБП и компьютер будут произведены двумя разными изготовителями, куплены в разное время и подключены отдельно друг от друга. Как показано на рис. 1.5, компьютер можно запустить без ИБП, и вполне возможно, что при сбоях электропитания вы воспользуетесь феном, подключив его к ИБП.

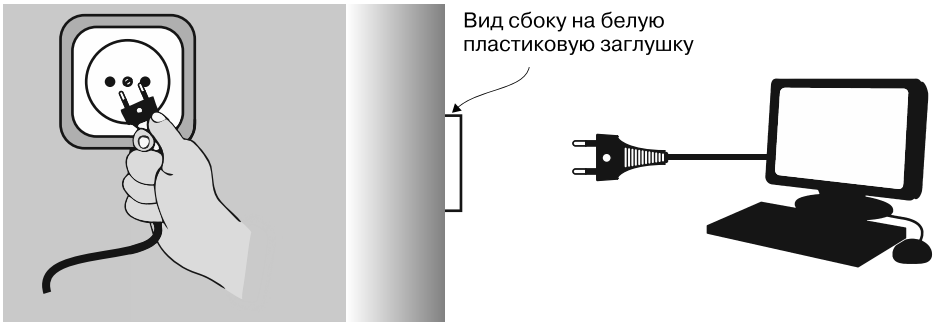


Рис. 1.6. Отсоединение компьютера и установка защитной заглушки для безопасности детей не приводит к порче ни номера, ни компьютера. Это может служить приблизительным примером применения паттерна «Нулевой объект»

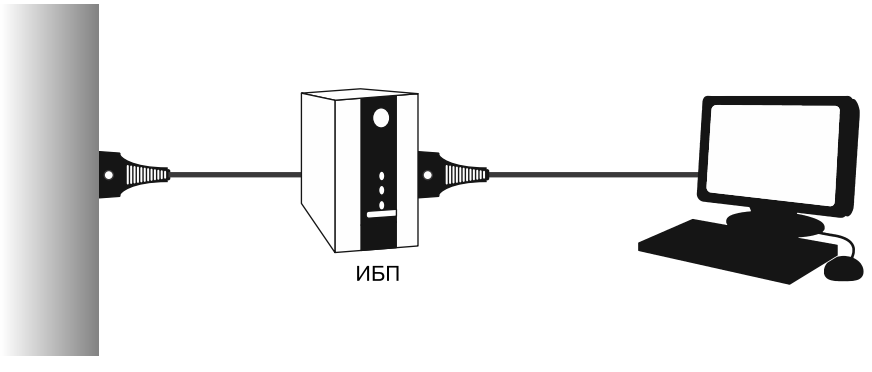


Рис. 1.7. Для поддержания работы компьютера в случае сбоя электропитания может быть использован источник бесперебойного питания. Это соответствует паттерну проектирования, который называется «Декоратор» (Decorator)

В проектировании программных продуктов такой способ перехода от одной реализации к другой с применением того же самого интерфейса известен как паттерн проектирования «Декоратор»[1]. Он позволяет постепенно вводить новые функции и сквозную функциональность (Cross-Cutting Concerns) без необходимости переписывать или изменять большие объемы уже имеющегося кода.

Еще один способ добавления новой функциональной возможности к уже существующему коду заключается в перепроектировании имеющейся реализации интерфейса с добавлением новой реализации. При объединении нескольких реализаций в одну используется паттерн проектирования «Компоновщик»[2]. На рис. 1.8 показано, как это соответствует подключению разных устройств к удлинителю.

---

1 *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 209.

2 Там же. С. 196.

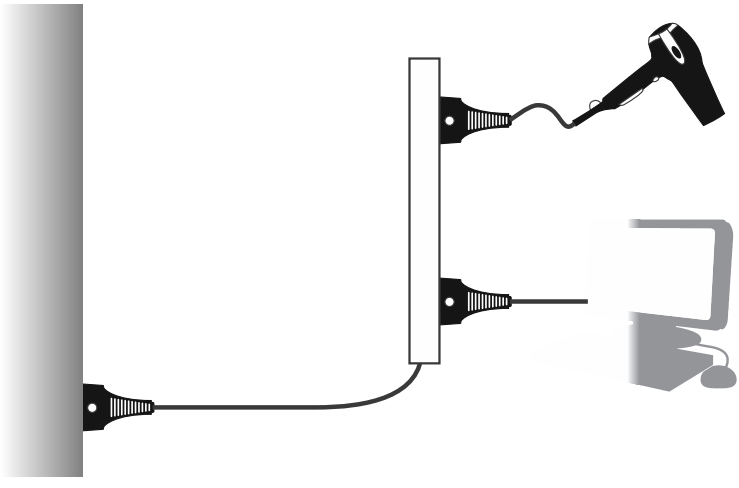


Рис. 1.8. Удлинитель позволяет подключать к одной стенной розетке сразу несколько устройств. Это можно назвать аналогией паттерна проектирования «Компоновщик»

У этого удлинителя имеется одна вилка, которую можно вставить в одну розетку, но на нем самом смонтировано несколько входов для различной бытовой техники. Это позволяет подключать и отключать фен при работающем компьютере. Аналогичным образом паттерн «Компоновщик» упрощает подключение и отключение той или иной функциональности за счет изменения набора составленных реализаций интерфейса.

И последний пример. Порой бывает, что вилка не подходит к розетке. Если вам приходилось путешествовать по миру, то вы могли заметить, что розетки везде разные. Если вы берете с собой в путешествие камеру, то для ее зарядки понадобится адаптер (рис. 1.9). Существует и паттерн проектирования с таким же названием.

Паттерн работает наподобие своего физического тезки[1]. Им можно воспользоваться для сопряжения двух взаимосвязанных, но все же разных интерфейсов друг с другом. Конкретная польза от этого проявляется при наличии уже имеющегося стороннего API, который требуется предоставить в качестве экземпляра интерфейса, используемого вашим приложением. Как и в случае с физическим адаптером, реализация паттерна проектирования может варьироваться от простого до весьма сложного.

В модели розетки и вилки удивительно то, что свою простоту и универсальность она доказывала на протяжении многих десятилетий. После создания инфраструктуры ее может использовать кто угодно, адаптируя под изменяющиеся нужды и непредвиденные требования. Любопытен и тот факт, что при переносе этой модели на разработку программных продуктов все строительные блоки уже имеются в виде принципов проектирования и паттернов.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 171.

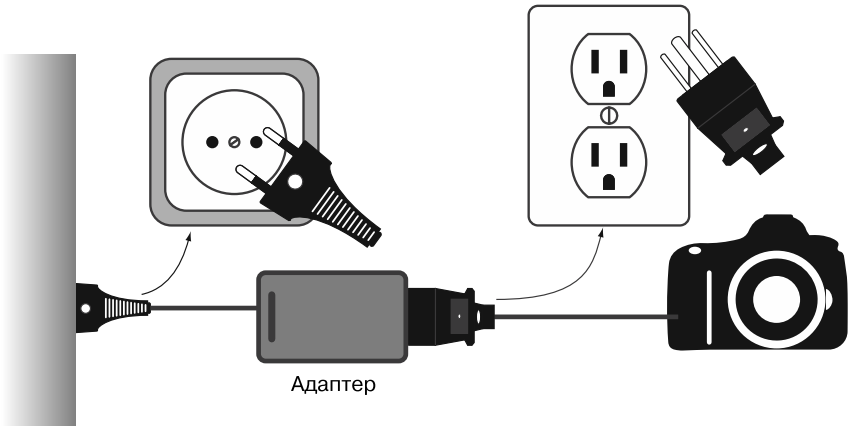


Рис. 1.9. Зачастую во время путешествий для подключения устройства к розетке приходится использовать адаптер (например, для зарядки камеры). Это можно назвать аналогией паттерна проектирования «Адаптер». Трансформация бывает простой (достаточно поменять форму вилки) или сложной (такая требует изменения переменного напряжения на постоянное)

Преимущество слабого связывания в проектировании программных продуктов такое же, как и в физической модели розетки и вилки: при наличии инфраструктуры она может использоваться кем угодно и приспосабливаться под меняющиеся потребности и непредусмотренные условия, не требуя больших изменений кода приложения и инфраструктуры. Это означает, что в идеале новое требование должно повлечь за собой всего лишь добавление нового класса, без внесения изменений в другие, уже имеющиеся в системе классы.

Эта концепция, допускающая возможность расширения приложения без изменения существующего кода, называется принципом открытости/закрытости (Open/Closed Principle). Добиться положения, при котором 100 % вашего кода будет всегда *открыто* для расширения и *закрыто* для модификаций, практически невозможно. Тем не менее слабое связывание приближает к достижению этой цели.

С каждым шагом к вашей системе становится проще добавлять новые функции и требования. Возможность включения новых функций, не касаясь уже имеющихся частей системы, означает изолированность решаемых задач. Это приводит к созданию кода, который легче понять и протестировать, что позволяет справиться со сложностями вашей системы. Именно в этом помогает слабое связывание, и именно поэтому оно способно существенно облегчить сопровождаемость кода. Более подробно принцип открытости/закрытости будет рассмотрен в главе 4.

Возможно, вам будет интересно, как выглядят эти паттерны при их реализации в коде. Об этом не стоит беспокоиться. Как уже говорилось, материал книги будет изобиловать примерами этих паттернов. Позже в главе будет показана реализация декоратора и адаптера.

Самой простой частью слабого связывания является программирование интерфейса, а не реализации. Возникает вопрос: «А откуда берутся экземпляры?» В какой-то мере на протяжении всей книги мы пытаемся найти ответ на этот вопрос, ведь это ключевой момент в понимании применения технологии DI.

Новый экземпляр интерфейса невозможно создать тем же способом, что и новый экземпляр конкретного типа. Такой код не компилируется:

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

Программа для интерфейса — Не пройдет компиляцию

Интерфейс не содержит реализации, следовательно, сделать это невозможно. Экземпляр `writer` должен быть создан с использованием иного механизма. Проблема решается с помощью DI. Наметив эту цель применения DI, можно приступить к усвоению примера.

1.2. Простой пример: Hello DI!

Рассмотрим простое консольное приложение, выводящее на экран приветствие «Hello DI!», в тех традициях, которые плотно укоренились почти во всех учебниках по программированию. Как уже упоминалось ранее в подразделе «Стандарт оформления кода и материалы для скачивания» раздела «Об этой книге», полный код доступен как часть сопутствующих книге материалов.

В этом разделе мы покажем вам, как выглядит код, и кратко опишем некоторые его основные преимущества, не вдаваясь в подробности. Более конкретные представления об особенностях кода вы получите при изучении остальных глав книги.

1.2.1. Код Hello DI!

Наверное, привычнее было бы увидеть примеры приветствия типа Hello World, написанного одной строкой кода. Здесь же мы немного усложним самый простой вариант. Для чего? Вскоре все станет ясно, но сначала посмотрим, на что похоже приветствие Hello World при использовании DI.

Взаимодействующие объекты

Чтобы разобраться в структуре программы, начнем с рассмотрения метода `Main` консольного приложения. Затем будут показаны взаимодействующие классы, но для начала взгляните на метод `Main` приложения Hello DI!:

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

Поскольку программе нужно выполнить запись в консоль, создается новый экземпляр `ConsoleMessageWriter`, в который заключается данная функциональная возможность. Эта функция записи сообщения передается классу `Salutation`, чтобы экземпляр `salutation` знал, куда следует записывать его сообщения. Поскольку те-

перь все смонтировано должным образом, логика может быть выполнена методом `Exclaim`, в результате чего сообщение будет выведено на экран.

Конструкция объектов, находящихся внутри метода `Main`, является основным примером чистой технологии DI. Для составления класса `Salutation` и его зависимости `ConsoleMessageWriter` DI-контейнер не использовался. На рис. 1.10 показаны связи взаимодействующих объектов.

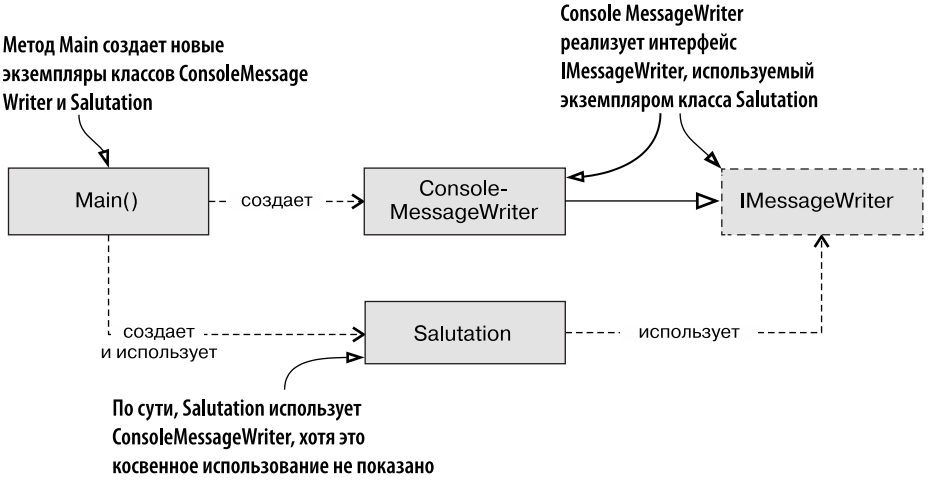


Рис. 1.10. Связи взаимодействующих объектов приложения Hello DI!

Реализация логики приложения

Основная логика приложения заключена в классе `Salutation`, показанном в листинге 1.1.

Листинг 1.1. Основная логика приложения заключена в классе 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!");
    }
}
```

Предоставляем классу Salutation зависимость IMessageWriter, применяя внедрение через конструктор

Контрольная инструкция проверяет, что предоставленный объект IMessageWriter не является нулевым

Передаем сообщение Hello DI! в зависимость IMessageWriter

Класс `Salutation` зависит от специализированного интерфейса `IMessageWriter` (определяемого далее). Он запрашивает экземпляр этого интерфейса, используя в качестве посредника его конструктор. Этот прием называется внедрением через конструктор (Constructor Injection). Граничный оператор (перевод из книги Мартина Фаулера. — *Примеч. ред.*) (Guard Clause) проверяет, что предоставленный объект `IMessageWriter` не является нулевым, и, если это так, выдает исключение[1]. После чего ранее внедренный экземпляр `IMessageWriter` используется внутри реализации метода `Exclaim` путем вызова метода `Write`. Тем самым сообщение Hello DI! передается в зависимость `IMessageWriter`.

ОПРЕДЕЛЕНИЕ

> Внедрение через конструктор (Constructor Injection) — это статическое определение списка требуемых зависимостей путем указания их конструктору класса в качестве параметров. (Внедрение через конструктор подробно рассматривается в главе 4, в которой также содержится более детальный сквозной анализ похожего примера кода.)

Выражаясь с применением терминологии DI, можно сказать, что зависимость `IMessageWriter` внедряется в класс `Salutation` с использованием аргумента конструктора. Заметьте, что классу `Salutation` абсолютно неизвестен класс `ConsoleMessageWriter`. Он взаимодействует с ним с помощью простого интерфейса `IMessageWriter`, определенного для данного случая:

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

У него могут быть и другие компоненты, но в данном простом примере нужен лишь метод `Write`. Он реализован классом `ConsoleMessageWriter`, который методом `Main` передается классу `Salutation`:

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

Класс `ConsoleMessageWriter` реализует интерфейс `IMessageWriter` путем создания оболочки для класса `Console` из библиотеки базовых классов (Base Class Library, BCL) .NET. Это простое применение паттерна проектирования «Адаптер», о котором говорилось в подразделе 1.1.2.

---

[1] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 253.

1.2.2. Преимущества, получаемые от применения DI

Сложно назвать преимуществом замену одной строки кода двумя классами и интерфейсом, в результате чего общее число строк составит 28. Ведь ту же задачу можно легко решить с использованием следующего кода:

```
private static void Main()
{
    Console.WriteLine("Hello DI!");
}
```

Технология DI может показаться неразумной, но из ее применения можно извлечь целый ряд привилегий. Чем же предыдущий пример лучше обычной одной строки кода, чаще всего используемой для реализации программы Hello World на C#? В данном примере издержки от применения DI составляют 2800 %, но по мере возрастания сложности от одной строки кода до десятков тысяч строк эти издержки уменьшаются и практически исчезают. Более сложный пример использования технологии DI показан в главе 3. Хотя он все же еще слишком прост по сравнению с настоящим приложением, можно будет заметить, что в нем навязчивость DI ощущается гораздо меньше.

Никто не станет вас осуждать за мнение о том, что предыдущий пример использования DI изобилует излишествами, но примите во внимание следующее: по своей природе классический пример Hello World является весьма простой задачей с четко определенными и довольно ограниченными требованиями. Разработка программного продукта в реальном мире не представляет собой ничего подобного. Требования изменяются и зачастую нечетко формулируются. Функции, требующие реализации, также склонны к усложнению. DI помогает решать подобные проблемы, допуская слабое связывание. При этом извлекаются преимущества, перечисленные в табл. 1.1.

Таблица 1.1. Преимущества, извлекаемые из слабого связывания. Каждое преимущество характеризуется постоянной доступностью, но в зависимости от обстоятельств оценивается по-разному

Преимущество	Описание	Когда ценится
Позднее связывание	Службы могут заменяться другими службами без перекомпиляции кода	Ценится в стандартных программных продуктах, но в меньшей степени в корпоративных приложениях с четко заданной средой выполнения
Расширяемость	Код может быть расширен и повторно использован тем способом, который ранее не был явно запланирован	Ценится всегда
Параллельная разработка	Код может разрабатываться в параллельном режиме	Ценится в больших, сложных приложениях, и меньше — в небольших, простых приложениях

Продолжение ⇩

Таблица 1.1 *(продолжение)*

Преимущество	Описание	Когда ценится
Сопровождаемость	Классы с четко выраженной ответственностью проще сопровождать	Ценится всегда
Пригодность к тестированию	Классы могут подвергаться модульному тестированию	Ценится всегда

Преимущество позднего связывания мы упомянули первым, поскольку, исходя из имеющегося опыта, именно оно превалирует в сознании большинства людей. Скорее всего, непонимание преимуществ слабого связывания со стороны проектировщиков и разработчиков возникает по причине того, что они никогда не берут в расчет другие преимущества, извлекаемые от применения данной технологии.

Позднее связывание

Когда объясняются преимущества программирования под интерфейсы и применения технологии DI, возможность замены одной службы другой является для большинства специалистов самой заметной выгодой, поэтому они склонны сравнивать преимущества и недостатки, кладя на чашу весов лишь это преимущество. Помните, мы предупреждали, что, прежде чем чему-то научиться, вам придется от многого отучиваться? Можете, конечно, заявить, что предъявляемые требования известны вам в достаточной мере и вы знаете, что, к примеру, вашу базу данных SQL Server никогда не придется ничем заменять. Но на смену старых требований приходят новые.

NoSQL, Microsoft Azure и аргументы в пользу компонуемости

Много лет назад я (рассказывает Марк) часто замечал абсолютное непонимание на лицах проектировщиков и разработчиков при попытке рассказать им о преимуществах технологии DI. Они говорили: «Ладно, ты можешь переключить компоненты доступа к реляционной базе данных на что-либо иное. А зачем? Разве есть какие-либо альтернативы реляционным базам данных?»

XML-файлы никогда не считались в широко масштабируемых корпоративных сценариях убедительной альтернативой. Но за последние несколько лет ситуация существенно изменилась.

На PDC 2008 была анонсирована облачная платформа Azure, внесшая весьма существенный вклад в пересмотр позиций по хранению данных в тех организациях, которые стойко придерживались применения исключительно продуктов разработки Microsoft. Теперь есть вполне реальная замена реляционным базам данных, и остается только спросить, желают ли разработчики, чтобы их приложение было готово к работе в облачной среде. Аргумент в пользу замены приобретает все большую силу.

Аналогичные подвижки просматриваются во всей концепции NoSQL, моделирующей приложения вокруг денормализированных данных, которые зачастую представлены документоориентированными базами данных. Повышается также важность и других концепций, например таких, как порождение событий (Event Sourcing)[1].

В подразделе 1.2.1 позднее связывание не использовалось, поскольку новый экземпляр `IMessageWriter` явным образом был разработан путем жестко запрограммированного создания нового экземпляра `ConsoleMessageWriter`. Но, изменив всего одну строку кода, можно ввести и позднее связывание:

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

Чтобы включить позднее связывание, можно заменить эту строку кода чем-то вроде листинга 1.2.

Листинг 1.2. Позднее связывание реализации IMessageWriter

```
IConfigurationRoot configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json")
    .Build();
string typeName = configuration["messageWriter"];
Type type = Type.GetType(typeName, throwOnError: true);

IMessageWriter writer = (IMessageWriter)Activator.CreateInstance(type);
```

ПРИМЕЧАНИЕ

Для наглядности в листинге 1.2 используется ряд сокращений. По факту в этом листинге просматриваются негативные особенности антипаттерна ограниченной конструкции (Constrained Construction), подробно рассмотренного в главе 5.

За счет извлечения имени типа из файла конфигурации приложения и создания из него экземпляра `Type` появляется возможность использования экземпляра `IMessageWriter` без осведомленности о конкретном типе на момент компиляции кода. Чтобы это заработало, имя типа объявляется в параметре `messageWriter`, который указывается в конфигурационном файле приложения:

```
{
  "messageWriter":
    "Ploeh.Samples.HelloDI.Console.ConsoleMessageWriter, HelloDI.Console"
}
```

Слабое связывание позволяет применить позднее связывание, так как имеется только одно место, где создается экземпляр `IMessageWriter`. Поскольку класс

[1] *Fowler M.* Event Sourcing, 2005. https://martinfowler.com/eaaDev/EventSourcing.html.

`Salutation` работает исключительно с интерфейсом `IMessageWriter`, разница для него незаметна. В примере Hello DI! позднее связывание позволит записывать сообщение не только в консоль, а, к примеру, в базу данных или файл. Появляется возможность добавлять подобные функции, даже если изначально они не планировались.

Расширяемость

Качественный программный продукт должен предоставлять возможности для внесения изменений. Со временем нужно будет добавить новые функции или расширить существующие. Слабое связывание позволит провести эффективную перекомпоновку приложения, аналогично тому, какой возможностью гибкого подхода обеспечены вы при работе с электрическими вилками и розетками.

Допустим, вы хотите повысить защищенность примера Hello DI!, разрешив записывать сообщение только аутентифицированным пользователям. Порядок добавления этой функции без внесения изменений в уже имеющуюся функциональность показан в листинге 1.3 и заключается в простом добавлении новой реализации интерфейса `IMessageWriter`.

Листинг 1.3. Расширение приложения Hello DI! защитной функцией

```
public class SecureMessageWriter : IMessageWriter
{
    private readonly IMessageWriter writer;
    private readonly IIdentity identity;

    public SecureMessageWriter(
        IMessageWriter writer,
        IIdentity identity)
    {
        if (writer == null)
            throw new ArgumentNullException("writer");
        if (identity == null)
            throw new ArgumentNullException("identity");

        this.writer = writer;
        this.identity = identity;
    }

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

Реализация интерфейса IMessageWriter наряду с его же использованием

Внедрение через конструктор, требующее экземпляр IMessageWriter

Проверка прохождения пользователем аутентификации

Если аутентификация была пройдена, запись сообщения с использованием внедренного модуля записи

ПРИМЕЧАНИЕ

Это стандартный способ применения паттерна «Декоратор», который упоминался в подразделе 1.1.2. Более подробно декораторы будут рассмотрены в главе 9.

Кроме экземпляра `IMessageWriter`, конструктору `SecureMessageWriter` требуется экземпляр `IIdentity`. Метод `Write` реализован таким образом, что сначала проводится проверка аутентификации пользователя с использованием внедренного интерфейса `IIdentity`. Если аутентификация была пройдена, задекорированному полю `writer` будет разрешено записать сообщение с помощью метода `Write`. Единственным местом, где придется вносить изменения в уже имеющийся код, является метод `Main`, поскольку доступные классы нужно составить по-другому, чем это делалось раньше:

```
IMessageWriter writer =
    new SecureMessageWriter(
        new ConsoleMessageWriter(),
        WindowsIdentity.GetCurrent());
```

ConsoleMessageWriter перехватывается декоратором SecureMessageWriter

ПРИМЕЧАНИЕ

По сравнению с кодом листинга 1.2 теперь уже используется жестко запрограммированный метод ConsoleMessageWriter.

Обратите внимание, что теперь старый экземпляр `ConsoleMessageWriter` оборачивается в новый класс `SecureMessageWriter`, то есть получается, что он *декорируется* этим классом. А класс `Salutation` не изменяется, поскольку он использует только интерфейс `IMessageWriter`. Точно так же нет необходимости ни изменять, ни дублировать функциональность, имеющуюся в классе `ConsoleWriter`. Для извлечения идентификатора пользователя, от имени которого выполняется данный код, применяется класс `System.Security.Principal.WindowsIdentity`[1].

Как уже утверждалось ранее, слабое связывание позволяет создавать код, *открытый для расширений, но закрытый для модификаций*. Единственным местом, где требуется вносить изменения в код, является точка входа в приложение. Защитные функции приложения реализованы в классе `SecureMessageWriter`, а обращение к пользовательскому интерфейсу — в классе `ConsoleMessageWriter`. Тем самым предоставляется возможность варьирования этих аспектов независимо от всего остального и составления из них комбинации по мере надобности. У каждого класса есть своя единственная ответственность (Single Responsibility).

Параллельная разработка

Разделение обязанностей позволяет разрабатывать программу в параллельном режиме. Когда проект разработки программного продукта вырастает до определенного размера, возникает необходимость в задействовании нескольких разработчиков, ведущих работу по созданию общего кода приложения одновременно. В гораздо больших масштабах появляется даже потребность в разделении коллектива разработчиков на несколько управляемых по размеру команд. На каждую команду

[1] Класс System.Security.Principal.WindowsIdentity находится в NuGet-пакете System.Security.Principal.Windows, который является частью .NET Core.

зачастую возлагается ответственность за область общего применения. Для разграничения обязанностей каждая команда разрабатывает один или несколько модулей, которые нужно будет объединить в готовое приложение. Пока области, выделенные каждой группе программистов, не будут по-настоящему независимы друг от друга, часть команд, скорее всего, будет зависеть от той функциональности, которая разрабатывается другими командами.

ОПРЕДЕЛЕНИЕ

> В объектно-ориентированном проектировании программных продуктов модулем считается группа логически связанных классов (или компонентов), в которых модуль независим от других модулей и взаимозаменяем с ними. Как правило, уровень состоит из одного или нескольких модулей.

В предыдущем примере, поскольку классы `SecureMessageWriter` и `ConsoleMessageWriter` не зависят напрямую друг от друга, они могли быть созданы командами, работающими в параллельном режиме. Им нужно было бы лишь договориться о совместном использовании интерфейса `IMessageWriter`.

Сопровождаемость

По мере того как ответственность каждого класса приобретает весьма четкое определение и ограничение, сопровождение всего приложения упрощается. Это должно быть результатом следования принципу единственной ответственности (Single Responsibility Principle), в котором утверждается, что каждый класс должен иметь только одну ответственность. Более подробно этот принцип будет рассматриваться в главе 2.

Упрощается и добавление к приложению новых функций, поскольку вполне понятно, куда именно нужно вносить изменения. Чаще всего необходимости внесения изменений в уже имеющийся код не возникает, но взамен могут понадобиться добавление новых классов и перекомпоновка приложения. Здесь опять вступает в действие принцип открытости/закрытости (Open/Closed Principle).

Упрощаются также поиск и устранение проблем, поскольку сужается круг вероятных причин их возникновения. Имея четко определенную ответственность, зачастую можно выстроить верные предположения о том, с чего начинать поиск первопричины возникшей проблемы.

Пригодность к тестированию

Приложение считается пригодным к тестированию, если к нему применимо модульное тестирование. Кого-то пригодность к тестированию волнует в последнюю очередь, а для кого-то является категоричным требованием. Мы, авторы книги, относим себя к последней категории. За свою карьеру Марку Симану не раз приходилось отказываться от предложений по трудоустройству, поскольку работодатели предполагали работу с конкретными продуктами, непригодными к тестированию.

Пригодность к тестированию

Термин «пригодный к тестированию» (testable) крайне неточен, но при этом весьма широко используется сообществом разработчиков программных продуктов, прежде всего теми из них, кто практикует модульное тестирование. В принципе, любое приложение может быть протестировано путем его прогона. Тесты могут выполняться путем использования приложения через его же пользовательский интерфейс или через любой другой интерфейс, который он предоставляет. Такие ручные тесты отнимают много времени и обходятся недешево, поэтому предпочтительнее применять автоматизированное тестирование.

Существуют различные виды автоматизированного тестирования — 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 обеспечивают быструю обратную связь касательно состояния приложения, но создавать модульные тесты можно только при должной изолированности тестируемого модуля от его зависимостей. Существует некоторая неопределенность относительно фактической степени детализации модуля, но все соглашаются с тем, что модульный тест, конечно же, не должен охватывать сразу нескольких модулей. В модульном тестировании решающее значение имеет возможность тестирования модулей в изолированном состоянии.

Приложение только тогда считается тестируемым, когда оно допускает модульное тестирование. Самый верный способ обеспечения тестируемости — разработать его с использованием TDD (Test-Driven Development).

Следует отметить, что сами по себе модульные тесты не дают гарантии работоспособности приложения. Для подобной проверки все же понадобится тестирование всей системы или другие типы сквозного тестирования.

Преимущество тестируемости является, пожалуй, самым спорным из всех перечисленных ранее. Некоторые разработчики и проектировщики по-прежнему не практикуют проведение модульного тестирования, считая это в лучшем случае чем-то, что не играет особой роли. Мы же считаем его весьма важной составляющей разработки программного продукта, именно поэтому в табл. 1.1 его сопровождает пометка «Ценится всегда». Майкл Физерс (Michael Feathers) даже определил понятие «устаревшее» для любого приложения, не предусматривающего модульное тестирование[1].

Словно по заказу, слабое связывание допускает проведение модульного тестирования, поскольку потребители следуют принципу подстановки Лисков: им безразличны конкретные типы их зависимостей. Это позволяет вводить в тестируемую систему (System Under Test, SUT) тестовых двойников, как показано в листинге 1.4.

[1] *Feathers M. C.* Working Effectively with Legacy Code. — Prentice Hall, 2004. — P. xvi.

Тестовые двойники (test doubles)

Существует общепринятая практика создавать такие реализации зависимостей, которые функционируют как дублеры уже разработанных или находящихся в стадии разработки реализаций. Такие реализации называются тестовыми двойниками (test doubles) и никогда не используются в готовом приложении. Вместо этого они служат заместителями реальных зависимостей, когда применение последних недоступно или нежелательно.

Тестовые двойники применяются, когда настоящая зависимость слишком медлительна, затратна, разрушительна или просто находится вне сферы применения проводимого теста. Вокруг тестовых двойников сформировался полноценный язык паттернов и появилось множество подтипов, таких как заглушки (stubs), имитации (mocks) и поддельные объекты (fakes)[1].

Возможность замены предполагаемых зависимостей специально предназначенными для тестов заместителями является побочным продуктом слабого связывания, но мы решили перечислить ее в качестве особого преимущества, поскольку из этого извлекается несколько иная польза. Исходя из нашего личного опыта, технология DI дает преимущества даже при проведении интеграционного тестирования. Хотя интеграционные тесты обычно обмениваются данными с реальными внешними системами (например, с базой данных), все же и им нужна определенная степень изолированности. Иными словами, в тестируемом приложении все же имеются поводы для замены, перехвата или имитации конкретных зависимостей.

Перехват текстовых сообщений

Мне (Стивену) приходилось работать над несколькими приложениями, отправлявшими CMC-сообщения с помощью стороннего сервиса. Я не хотел, чтобы наша среда тестирования отправляла эти текстовые сообщения в реальный шлюз, поскольку у них была индивидуальная тарификация и, конечно же, было нежелательно отправлять случайный спам с этими текстовыми сообщениями на мобильные телефоны.

А вот в ходе ручного тестирования текстовые сообщения отправлялись на мобильные телефоны. Но в этом случае применялся декоратор, изменявший номер телефона, отправляемый в адрес шлюза, на номер, который мог быть предоставлен тестировщиком. Таким образом, тестировщик имел возможность получать все сообщения на свой собственный телефон и проверять тестируемую систему.

В зависимости от типа разрабатываемого вами приложения заинтересованность в возможности позднего связывания может иметься, а может и отсутствовать, но интерес к тестируемости будет всегда. Некоторые разработчики не обеспокоены тестируемостью, считая, что для разрабатываемого ими приложения важную роль

[1] *Meszaros G.* xUnit Test Patterns: Refactoring Test Code. — Addison-Wesley, 2007. — P. 522.

играет позднее связывание. Как бы то ни было, технология DI предоставляет массу возможных вариантов в будущем при малых на сегодняшний день дополнительных издержках.

Пример: модульное тестирование логики Hello DI

Вернемся к коду примера Hello DI! из подраздела 1.2.1. Хотя сначала мы показали вам уже готовый код, его разработка велась по принципу TDD. Наиболее важный модульный тест показан в листинге 1.4.

ПРИМЕЧАНИЕ

Не волнуйтесь, если у вас нет опыта работы с модульными тестами. Временами они будут встречаться в книге, но опыт их применения не является обязательным условием для ее прочтения[1].

Листинг 1.4. Модульное тестирование класса Salutation

```
[Fact]
public void ExclaimWillWriteCorrectMessageToMessageWriter()
{
    var writer = new SpyMessageWriter();
    var sut = new Salutation(writer);
    sut.Exclaim();
    Assert.Equal(
        expected: "Hello DI!",
        actual: writer.WrittenMessage);
}

public class SpyMessageWriter : IMessageWriter
{
    public string WrittenMessage { get; private set; }

    public void Write(string message)
    {
        this.WrittenMessage += message;
    }
}
```

Зависимость IMessageWriter заменена заглушкой, использующей тестового шпиона SpyMessageWriter

Классу `Salutation` нужен экземпляр интерфейса `IMessageWriter`, поэтому его необходимо создать. Можно воспользоваться любой реализацией, но в модульных тестах может пригодиться тестовый двойник — в данном случае мы обходимся своей собственной реализацией тестового шпиона (Test Spy)[2].

---

[1] Возможно, вас заинтересуют книги: *Osherove R.* The Art of Unit Testing, 2nd Ed. — Manning, 2013; *Meszaros G.* xUnit Test Patterns. — Addison-Wesley, 2007.

[2] Тестовый шпион (Test Spy) является «тестовым двойником, захватывающим косвенные выходные вызовы, совершаемые в адрес другого компонента тестируемой системы для последующей проверки с помощью теста». См. книгу: *Meszaros G.* xUnit Test Patterns. — Addison-Wesley, 2007. — P. 538.

Здесь тестовый двойник используется так же, как и реализация для готового приложения. Это свидетельство того, насколько прост наш пример. В большинстве приложений тестовый двойник намного проще конкретной реализации для готового приложения, заместителем которой он является. Важная деталь — предоставление специальной тестовой реализации `IMessageWriter`, чтобы гарантировать, что в определенный момент времени тестируется только один компонент приложения. В данный момент тестируется метод `Exclaim` класса `Salutation`, поэтому нежелательно загромождать тест реализацией `IMessageWriter` для готового приложения. Для создания класса `Salutation` экземпляр тестового шпиона `IMessageWriter` передается путем внедрения через конструктор.

После прогона тестируемой системы можно вызвать метод `Assert.Equal` и проверить, соответствует ли ожидаемый результат фактическому. Если бы метод `IMessageWriter.Write` был вызван со строкой `"Hello DI!"`, то `SpyMessageWriter` сохранил бы ее в своем свойстве `WrittenMessage` и работа метода `Equal` завершилась бы успешно. Но если бы метод `Write` не был вызван или же был вызван, но с другим значением, то метод `Equal` выдал бы исключение и тест не был бы пройден.

Слабое связывание предоставляет множество преимуществ: код становится легче разрабатывать, сопровождать и расширять, и при этом повышается его пригодность к тестированию. Особой сложности это не представляет. Программы пишутся под интерфейсы, а не под конкретные реализации. Единственным серьезным препятствием является определение порядка получения экземпляров таких интерфейсов. В DI это препятствие преодолевается путем ввода зависимости из внешней среды. Предпочтительным способом является внедрение через конструктор, однако в главе 4 мы дополнительно рассмотрим и другие варианты.

1.3. Что следует, а что не следует внедрять

В предыдущем разделе были рассмотрены мотивирующие факторы, заставляющие в первую очередь обратить внимание на технологию DI. Убежденность в преимуществе слабого связывания может склонить вас к его повсеместному использованию. В целом идея неплохая. Слабое связывание оказывается особенно полезным, когда дело доходит до решения о порядке компоновки модулей. Но вам не нужно абстрагироваться от всего и делать подключаемым абсолютно все. В этом разделе будет предоставлен ряд инструментов, которые помогут вам принять решение о порядке моделирования зависимостей.

.NET BCL состоит из множества сборок. Каждый раз, когда вы пишете код, использующий тип из сборки BCL, к вашему модулю добавляется зависимость. В предыдущем разделе речь шла о важности применения слабого связывания и о том, что краеугольным камнем здесь является порядок программирования под интерфейс. Означает ли это, что вы не можете ссылаться на сборки BCL, а их типы использовать в вашем приложении напрямую? Как быть, если вам захочется воспользоваться классом `XmlWriter`, который определен в сборке `System.Xml`?

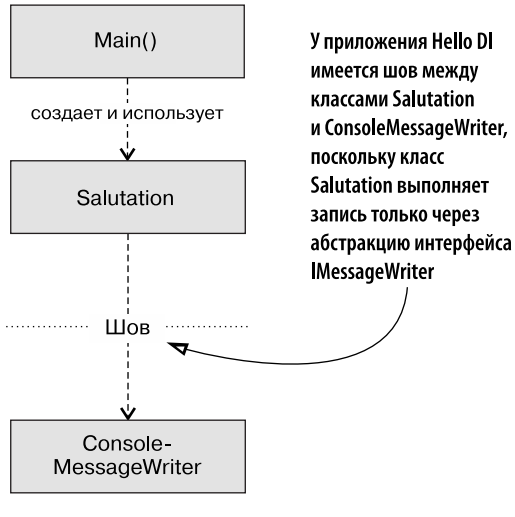
Не нужно ко всем зависимостям относиться одинаково. Множество типов, имеющихся в BCL, могут использоваться без ущерба для степени связанности, но это относится не ко всем типам. Важно знать, как различить типы, не представляющие опасности, и типы, которые могут повысить степень связанности, имеющуюся у приложения. Здесь речь в основном пойдет о последней категории типов.

В ходе изучения DI может быть полезно разделить ваши зависимости на стабильные (Stable Dependencies) и нестабильные (Volatile Dependencies). Работа по принятию решений о том, куда помещать ваши швы (Seams), не будет представлять для вас никаких сложностей. Более подробно эти понятия рассматриваются в следующих разделах.

Швы

Там, где принимается решение программировать под абстракцию, а не под конкретный тип, в приложении появляется шов. Под швом (seam) понимается место, где приложение собирается из его составных частей — подобно тому как сшиваются друг с другом части одежды[1]. Шов также является местом, где можно разбить приложение на части и изолированно поработать с модулями.

На следующем рисунке показано, что созданный в разделе 1.2 пример Hello DI! содержит шов между `Salutation` и `ConsoleMessageWriter`. Класс `Salutation` не зависит напрямую от класса `ConsoleMessageWriter`, а для записи сообщения им используется интерфейс `IMessageWriter`. Вы можете разобрать приложение по этому шву и собрать заново с другим модулем записи сообщений.



[1] *Feathers M. C.* Working Effectively with Legacy Code. — P. 29–44.

1.3.1. Стабильные зависимости

Многие модули в библиотеке BCL и за ее пределами не угрожают снижением степени модульности приложения. В них содержатся многократно используемые функции, которые можно применить, чтобы сделать свой собственный код лаконичнее. Модули BCL всегда доступны для вашего приложения, поскольку для его запуска требуется наличие среды .NET, и, поскольку они уже имеются, беспокойства, связанные с параллельной разработкой, для этих модулей неактуальны. Библиотека BCL всегда может повторно использоваться и в другом приложении.

Изначально большинство типов, определенных в BCL (за редким исключением), могут считаться безопасными или стабильными зависимостями. Мы называем их так, поскольку они уже существуют, обладают свойством обратной совместимости и их вызовы имеют вполне определенные результаты. Большинство стабильных зависимостей являются типами BCL, но стабильными могут быть и другие зависимости. К важным критериям стабильных зависимостей относятся следующие.

- Класс или модуль уже существует.
- Ожидается, что новые версии не будут содержать критических изменений.
- Интересующие типы содержат детерминированные алгоритмы.
- Вы не рассматриваете необходимость замены, заключения в оболочку, декорирования или перехвата класса или модуля другим классом или модулем.

В качестве других примеров могут рассматриваться специализированные библиотеки, заключающие в себе алгоритмы, подходящие вашему приложению. Например, при разработке приложения, связанного с химическим производством, можно ссылаться на стороннюю библиотеку, содержащую специализированные функции, соответствующие данной области.

> **Ссылка на DI-контейнер**
>
> Сами по себе DI-контейнеры могут рассматриваться в качестве либо стабильных, либо нестабильных зависимостей, что определяется тем, хотите вы их заменить или нет. Когда вы решаете сделать основой приложения конкретный DI-контейнер, вы рискуете застрять с этим выбором на весь срок службы приложения. Это еще один повод для того, чтобы ограничить использование контейнера точкой входа в приложение. На DI-контейнер должна ссылаться только точка входа.

В общем, зависимости могут считаться стабильными методом исключения. Они стабильны, если не относятся к нестабильным.

1.3.2. Нестабильные зависимости

Закладка швов в приложение требует дополнительных усилий, поэтому делать это вы должны только тогда, когда это крайне необходимо. Существует несколько причин, по которым важно изолировать зависимость за шов, но все эти причины

можно отнести к извлечению преимуществ из применения слабого связывания (см. подраздел 1.2.1).

Такие зависимости можно распознать по их склонности создавать помехи одному или нескольким из этих преимуществ. Они не являются стабильными, поскольку не обеспечивают приложениям достаточно надежную базу, оттого они и получили свое название. Зависимость следует считать изменчивой, если выполняется любой из следующих критериев.

- Зависимость вводит требование по установке и конфигурированию среды выполнения приложения. В большей степени это относится не к самим нестабильным типам среды .NET, а к той среде выполнения, которую для них нужно задействовать.

  Хорошим примером BCL-типов, относящихся к нестабильным зависимостям, могут послужить базы данных, и в первую очередь — реляционные базы данных. Если не удается скрыть реляционную базу данных за швом, вы никогда не сможете заменить ее какой-либо другой технологией. Вдобавок это затрудняет настройку и запуск автоматизированных модульных тестов. (Несмотря на то что клиентская библиотека Microsoft SQL Server является технологией, которая содержится в BCL, при ее применении задействуется реляционная база данных.) Под эту категорию подпадают и другие сторонние ресурсы, такие как очереди сообщений, веб-сервисы и даже файловые системы. Признаками такого типа зависимости являются отсутствие позднего связывания и расширения, а также невозможность проведения тестирования.
- Зависимость еще не существует или находится в стадии разработки.
- Зависимость установлена не на всех машинах организации-разработчика. Это могут быть дорогостоящие библиотеки сторонних производителей или же зависимости, которые не могут быть установлены на всех операционных системах. Наиболее характерным признаком является невозможность проведения тестов.
- Зависимость ведет себя недетерминированно. Это имеет особое значение для прохождения модульных тестов, поскольку исход всех тестов должен быть четко определен. Типичными источниками непредсказуемости являются генераторы случайных чисел и алгоритмы, зависящие от текущих даты или времени.

  Поскольку в библиотеке BCL определяются общие источники недетерминированности, такие как `System.Random`, `System.Security.Cryptography.RandomNumberGenerator` или `System.DateTime.Now`, избежать ссылки на сборку, в которой они определены, невозможно. Как бы то ни было, к ним нужно относиться как к нестабильным зависимостям, поскольку им свойственно препятствовать проведению тестов.

ВАЖНО

Нестабильные зависимости являются основным аспектом DI. Швы вводятся в приложение из-за нестабильных зависимостей, поскольку стабильным они ни к чему. Следует еще раз напомнить, что это обязывает вас проводить их компоновку с использованием технологии DI.

Теперь, когда вы понимаете разницу между стабильными и нестабильными зависимостями, область применения DI прослеживается более конкретно. Слабое связывание является весьма распространенным принципом проектирования, следовательно, технология DI (в качестве его активатора) должна присутствовать в разрабатываемом вами коде повсеместно. Никакой четкой границы между темой применения DI и качественным проектированием программных продуктов не существует, но, чтобы определить круг вопросов, рассматриваемых в оставшейся части книги, воспользуемся кратким описанием того, что она охватывает.

1.4. Область применения DI

Как уже упоминалось ранее, важным элементом DI является разделение различных обязанностей на отдельные классы. Одной из обязанностей, отстраненной от классов, является задача создания экземпляров зависимостей. Эта задача известна как компоновка объектов.

Это уже обсуждалось при рассмотрении примера Hello DI!, где класс `Salutation` был освобожден от ответственности по созданию своей зависимости. Эта обязанность была перемещена в принадлежащий приложению метод `Main`. UML-схема показана на рис. 1.11.

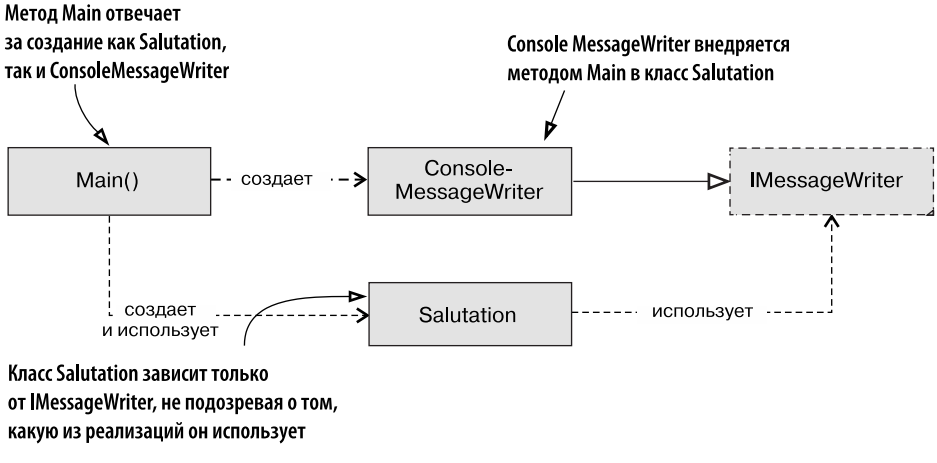


Рис. 1.11. Связи взаимодействующих объектов приложения Hello DI! (повторно)

Поскольку класс освобождается от контроля над зависимостями, он уступает не только право принятия решений по выбору конкретной реализации. Мы, как разработчики, извлекаем из этого ряд преимуществ. На первый взгляд может показаться недостатком позволить классу отдать контроль над тем, какие объекты создаются для него. Но мы этот контроль не теряем — мы просто перемещаем его в другое место.

ПРИМЕЧАНИЕ

Убирая контроль класса над его зависимостями, мы, разработчики, берем этот контроль в свои руки. В этом проявляется принцип единственной ответственности. Классы не должны заниматься созданием своих зависимостей.

Компоновка объектов не единственная удаляемая из класса область контролирования: класс также утрачивает способность контролировать время жизни объекта. Когда экземпляр зависимости внедряется в класс, потребитель не знает, когда он был создан или когда он исчезнет из области видимости. Как правило, это не волнует потребителя, но в каких-то случаях это может оказаться важным.

Технология DI дает вам возможность управлять зависимостями универсальным образом. Когда потребители непосредственно создают и настраивают экземпляры зависимостей, каждый из них может делать это по-своему. Это может не соответствовать тому, как это делают другие потребители. Не получается никакого централизованного управления зависимостями и нет никакого простого способа решения проблем сквозной функциональности.

Используя технологию DI, вы получаете возможность перехвата каждого экземпляра зависимости и совершения над ним каких-либо действий перед передачей потребителю. Тем самым обеспечивается расширяемость приложений.

С помощью DI вы можете заниматься компоновкой приложений через перехват зависимостей и управление временем их жизни. Компоновка объектов (Object Composition), перехват (Interception) и управление временем жизни (Lifetime Management) — это три измерения технологии DI. Далее последует краткий обзор каждого из них, а более подробно они будут рассматриваться в части III книги.

1.4.1. Компоновка объектов

Чтобы воспользоваться преимуществами расширяемости, позднего связывания и параллельной разработки, нужно научиться составлять классы в приложения. Это означает, что нужно создать приложение из отдельных классов путем их объединения, что очень похоже на одновременное подключение электроприборов. И, как и в случае с электрическими приборами, при введении новых требований хочется получить возможность легкого перестроения этих классов, в идеале — не внося никаких изменений в уже имеющиеся классы.

Зачастую компоновка объектов является основной мотивацией для введения DI в приложение. На самом деле изначально внедрение зависимостей служило синонимом компоновки объектов; это единственный аспект, рассматриваемый по данной теме в первоначальной статье Мартина Фаулера (Martin Fowler)[1].

Компоновка классов в приложение может выполняться несколькими способами. При рассмотрении позднего связывания для ручной компоновки приложения

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

из доступных модулей нами использовались конфигурационный файл и несколько экземпляров динамически создаваемых объектов. Можно было бы также применить конфигурацию в виде кода (Configuration as Code), воспользовавшись DI-контейнером. К этому вопросу мы еще вернемся в главе 12.

Многие называют DI инверсией управления (Inversion of Control, IoC). Порой эти два понятия применяются взаимозаменяемо, но DI является разновидностью IoC. В книге постоянно используется более конкретное понятие — DI. Если же будет подразумеваться IoC, то это будет оговорено специально.

Так все же внедрение зависимостей или инверсия управления?

Первоначально понятие инверсии управления означало любую разновидность стиля программирования, в котором ход выполнения программы контролировался общей структурой или средой выполнения[2]. Согласно данному определению инверсия управления используется в большинстве программных продуктов, разработанных в среде .NET. Когда, к примеру, пишется приложение ASP.NET Core MVC, вами создаются классы контроллеров с методами действий, но ваши методы действий будут вызываться не чем иным, как ASP.NET Core. Стало быть, управление принадлежит не вам, а самой среде.

В наши дни мы настолько привыкли работать со средами, что не считаем это чем-то особенным, но это уже другая модель, чем та, что использовалась при полном контроле над вашим кодом. Такое может случаться и для .NET-приложения, особенно для исполняемых файлов командной строки. Сразу же после вызова метода `Main` ваш код переходит под полный контроль. Им контролируется ход выполнения программы, время жизни объектов — в общем, все. Никакие специальные события не порождаются, и не вызываются никакие переопределенные компоненты.

Пока технология DI не обрела свое теперешнее имя, библиотеки, управляющие зависимостями, начали называться контейнерами инверсии управления (Inversion of Control Containers), но вскоре значение IoC постепенно сместилось в сторону данного конкретного значения: инверсия управления зависимостями (Inversion of Control Over Dependencies). Будучи ярым приверженцем систематизации, Мартин Фаулер (Martin Fowler) ввел понятие внедрения зависимостей (Dependency Injection), чтобы специально ссылаться на IoC в контексте управления зависимостями. С тех пор внедрение зависимостей вошло в широкий обиход как наиболее правильная терминология. Короче говоря, IoC — намного более широкое понятие, включающее технологию DI, но не ограничивающееся ей одной.

1.4.2. Время жизни объектов

Класс, отказавшийся от контроля над своими зависимостями, уступает не только право выбора конкретной реализации абстракции. Он также передает право контролировать моменты создания экземпляров и моменты их исчезновения из области видимости.

[1] *Fowler M.* Inversion of Control, 2005; https://martinfowler.com/bliki/InversionOfControl.html.

В среде .NET за все это берет на себя ответственность сборщик мусора. Потребитель может ввести в него свои зависимости и использовать их сколь угодно долго. По завершении работы зависимости исчезнут из области видимости. Если на них не будут ссылаться никакие другие классы, они подпадут под сборку мусора.

А что произойдет, когда один и тот же тип зависимости будет совместно использоваться сразу двумя потребителями? В листинге 1.5 показано, что можно выбрать внедрение отдельного экземпляра в каждого потребителя, а в листинге 1.6 обозначено, что можно воспользоваться альтернативным вариантом совместного использования одного и того же экземпляра сразу несколькими потребителями. Но, с точки зрения потребителя, в них нет никакой разницы. В соответствии с принципом подстановки Лисков потребитель должен относиться ко всем экземплярам данного интерфейса одинаково.

Листинг 1.5. Потребители получают свой собственный экземпляр зависимости одного и того же типа

```
IMessageWriter writer1 = new ConsoleMessageWriter();
IMessageWriter writer2 = new ConsoleMessageWriter();

var salutation = new Salutation(writer1);
var valediction = new Valediction(writer2);
```

Создаются два экземпляра одной и той же зависимости IMessageWriter

Каждый потребитель получает свой собственный закрытый экземпляр

Листинг 1.6. Потребители совместно используют экземпляр зависимости одного и того же типа

```
IMessageWriter writer = new ConsoleMessageWriter();

var salutation = new Salutation(writer);
var valediction = new Valediction(writer);
```

Создается один экземпляр

Этот самый экземпляр внедряется в двух потребителей

Поскольку зависимости могут быть совместно используемыми, отдельный потребитель не может контролировать продолжительность их жизни. Поскольку управляемый объект сможет быть выведен из области видимости и уничтожен как мусор, контроль его времени жизни проблем не создает. Но когда зависимости реализуют интерфейс `IDisposable`, ситуация, рассматриваемая в разделе 8.2, существенно усложняется. В целом управление временем жизни — это отдельный аспект DI, который играет настолько важную роль, что ему посвящена вся глава 8.

1.4.3. Перехват

Когда, как показано на рис. 1.12, контроль над зависимостями делегируется третьей стороне, право их изменения перед передачей потребляющим классам также предоставляется этой стороне. В примере Hello DI! экземпляр `ConsoleMessageWriter` изначально был внедрен в экземпляр `Salutation`. Затем, при модификации примера,

была добавлена функция защиты, для чего был создан новый класс `SecureMessageWriter`, который только делегировал дальнейшую работу классу `ConsoleMessageWriter` в том случае, если пользователь прошел аутентификацию. Это позволяет придерживаться принципа единственной ответственности. Такая возможность появилась благодаря постоянному программированию под интерфейсы; помните, что зависимости всегда должны быть абстракциями. В случае с классом `Salutation` ему все равно, что предоставлено интерфейсом `IMessageWriter`: `ConsoleMessageWriter` или `SecureMessageWriter`. В `SecureMessageWriter` может быть заключен `ConsoleMessageWriter`, выполняющий реальную работу.

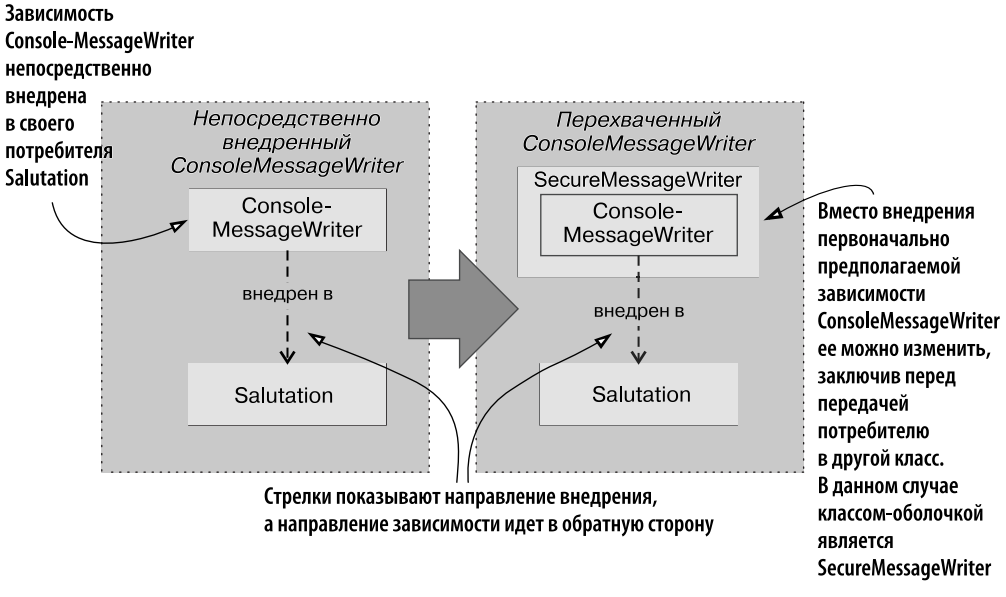


Рис. 1.12. Перехват ConsoleMessageWriter

ПРИМЕЧАНИЕ

Перехват представляет собой применение паттерна проектирования «Декоратор». Если он вам незнаком, переживать не стоит. Дополнительные сведения о нем вы найдете в главе 9, которая будет целиком посвящена перехвату.

Рассмотренные возможности перехвата ведут нас по пути к аспектно-ориентированному программированию (Aspect-Oriented Programming, AOP), к родственной теме, которая будет рассмотрена в главах 10 и 11. С помощью перехвата и AOP можно применять сквозную функциональность (Cross-Cutting Concerns), например ведение журнала, проведение контрольных проверок, управление доступом, проведение проверки допустимости и т. д., причем делать все это в четко структурированной форме, позволяющей соблюдать принцип разделения обязанностей.

1.4.4. DI в трех измерениях

Хотя вначале технология DI представляла собой серию паттернов, нацеленных на решение проблем компоновки объектов (Object Composition), понятие DI впоследствии расширилось, чтобы охватить еще и время жизни объектов (Object Lifetime) и перехват (Interception). Сегодня мы думаем о DI как о последовательном охвате всех трех измерений.

Компоновке объектов свойственно доминировать в этой картине, поскольку без гибкой компоновки не было бы и перехвата и не нужно было бы управлять временем жизни объектов. Компоновке объектов в этой главе уделялось наибольшее внимание, и данная тенденция будет прослеживаться и в оставшейся части книги, но не нужно забывать и о других аспектах. Компоновка объектов обеспечивает основу, а управление временем жизни объектов обращается к некоторым весьма важным побочным эффектам. Но в основном плоды пожинаются тогда, когда дело доходит до перехвата.

В части III каждому из кратко упомянутых здесь измерений будет посвящена отдельная глава. Но важно понимать, что на практике DI — это нечто более масштабное, нежели просто компоновка объектов.

1.5. Заключение

Внедрение зависимостей — это средство достижения цели, а не самоцель. Это наилучший способ поддержки слабого связывания — важной части сопровождаемого кода. Преимущества, которые можно извлечь из слабого связывания, становятся очевидными не сразу, но со временем, по мере усложнения используемого кода, они обязательно проявятся. Что же касается слабого связывания применительно к DI, то важно отметить, что эффективность достигается его повсеместным присутствием в коде приложения.

Сильно связанный код со временем вырождается в спагетти-код (Spaghetti Code)[1], а качественно спроектированный код со слабой связанностью может сохранять высокую степень сопровождаемости. Чтобы получить по-настоящему гибкую архитектуру[2], следует озаботиться не только слабым связыванием, но и выполнением обязательного условия программирования под интерфейс.

СОВЕТ

Технология DI должна использоваться повсеместно. Ввести слабое связывание в уже имеющийся код будет нелегко.

---

1 *Brown W. J. et al.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. — Wiley Computer Publishing, 1998. — P. 119.

2 *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — С. 221.

Технология DI — это не более чем использование коллекции принципов и паттернов проектирования. Она больше относится к способу мышления и разработки кода, нежели к инструментам и методам. Цель DI — сделать код сопровождаемым. Небольшие программы наподобие классического примера Hello World из-за малых размеров являются сопровождаемыми по самой своей природе. Поэтому в простых примерах DI, скорее всего, покажется чрезмерным усложнением. Но чем больше становится объем кода, тем очевиднее проявляются преимущества. Чтобы их продемонстрировать, следующие две главы будут посвящены более крупному и сложному примеру.

Резюме

- Внедрение зависимостей — это набор принципов и паттернов проектирования программных продуктов, позволяющий разрабатывать слабосвязанный код. А слабая связанность повышает сопровождаемость кода.
- Когда у вас есть инфраструктура со слабой связанностью, ею может воспользоваться кто угодно, подгоняя приложение под изменяющиеся нужды и ранее непредвиденные требования без необходимости внесения существенных изменений в код приложения и его инфраструктуру.
- Упрощается также поиск и устранение неполадок, поскольку сужается круг вероятных причин их возникновения.
- DI допускает позднее связывание, в результате чего появляется возможность замены классов или модулей другими классами и модулями без необходимости перекомпилирования исходного кода.
- DI упрощает расширение и повторное использование кода тем порядком, который ранее явно не был запланирован, наподобие того, какой гибкостью вы обладаете при работе с электрическими вилками и розетками.
- DI упрощает параллельную разработку одной и той же базы кода, поскольку соблюдение принципа единственной ответственности позволяет каждому члену команды разработчиков или даже целым командам более свободно работать над изолированными друг от друга частями.
- DI делает программный продукт всесторонне тестируемым, поскольку при написании модульных тестов зависимости можно заменять тестовыми реализациями.
- При практическом использовании DI сотрудничающие классы для предоставления необходимых служб должны исходить из инфраструктуры. Этого можно добиться, позволив вашим классам зависеть от интерфейсов, а не конкретных реализаций.
- Классы не должны запрашивать свои зависимости у третьей стороны. Такие действия вписываются в антипаттерн под названием «Локатор сервисов» (Service Locator). Вместо этого классы должны указывать требуемые им зависимости статически, используя параметры конструктора, то есть практиковать прием под названием «внедрение через конструктор» (Constructor Injection).

- Многие разработчики полагают, что технология DI требует особого инструментария — так называемого DI-контейнера. Это миф. DI-контейнер является полезным, но вспомогательным инструментом.
- Одним из наиболее важных принципов разработки программных продуктов является принцип подстановки Лисков. Он позволяет заменять одну реализацию интерфейса другой, не выводя из строя ни клиента, ни реализацию.
- Зависимости считаются стабильными в том случае, если они уже доступны, имеют вполне определенное поведение, не требуют настройки среды выполнения (например, под реляционную базу данных) и не нуждаются в замене, помещении в оболочку или перехвате.
- Зависимости считаются нестабильными в том случае, если они находятся в стадии разработки, не всегда доступны на всех машинах, используемых при разработке, содержат нечетко выраженное поведение или нуждаются в замене, помещении в оболочку или перехвате.
- Нестабильные зависимости являются основным аспектом DI. Они внедряются в конструктор класса.
- Если убрать контроль над зависимостями из их потребителей и переместить его в точку входа в приложение, существенно упростится применение сквозной функциональности (Cross-Cutting Concerns), а также появится возможность более эффективного управления временем жизни зависимостей.
- Чтобы добиться успеха, вам необходимо повсеместно применять DI. Все классы должны получать нужные им нестабильные зависимости за счет внедрения через конструктор. Ввести слабое связывание и DI в уже имеющийся код будет весьма затруднительно.

2 Создание кода с сильной связанностью

В этой главе

- Создание приложения с сильной связанностью.
- Оценка компонуемости данного приложения.
- Анализ дефицита компонуемости этого приложения.

Как уже упоминалось в главе 1, беарнский соус представляет собой эмульсию, приготовленную из яичного желтка и масла, но эти сведения отнюдь не сакральное знание, которое поможет вам приготовить его без ошибок. Учиться лучше всего на практике, но наглядный пример перед глазами зачастую помогает преодолеть разрыв между теорией и практикой. Перед попыткой самостоятельного приготовления беарнского соуса неплохо было бы понаблюдать, как его готовит профессиональный повар.

Во введении в DI, представленном в предыдущей главе, мы воспользовались ознакомительной экскурсией, помогающей разобраться в целях и основных принципах применения этой технологии. Но это простое объяснение не позволяет должным образом раскрыть всю ее суть. Технология DI является методом обеспечения слабого связывания, которое прежде всего выступает эффективным способом борьбы со сложностью.

Большинство программ являются сложными в том смысле, что они должны решать множество проблем одновременно. Помимо бизнес-задач, которые могут быть сложными сами по себе, программным продуктом должны также решаться вопросы безопасности, диагностики, выполнения технологических операций, производительности и расширяемости. Вместо решения всех этих задач одним скопом слабое связывание побуждает обращаться к каждой задаче по отдельности. Проще всего решать каждую из них изолированно, но в конечном итоге нужно будет скомпоновать этот непростой набор задач в одно приложение.

В этой главе будет рассмотрен более сложный пример. Будет показано, насколько легко создавать код с сильной связанностью. Мы вместе с вами проанализируем, почему код с сильной связанностью трудно поддается сопровождению. В главе 3 этот код будет полностью переписан с применением технологии DI, и в основу примера будет положен код со слабой связанностью. Если у вас есть желание сразу же перейти к изучению кода со слабой связанностью, то эту главу можно пропустить. Но если этого не делать, то после изучения данной главы вы начнете понимать, почему код с сильной связанностью создает так много проблем.

2.1. Создание приложения с сильной связанностью

Сама идея построения слабосвязанного кода не вызывает споров, но существует громадный разрыв между теорией и практикой. Перед тем как в следующей главе будет приведен порядок использования технологии DI для создания приложения со слабой связанностью, мы хотим показать вам, как все же просто сбиться с верного пути. Обычно, собираясь получить код со слабой связанностью, создают приложение, состоящее из нескольких уровней. Нарисовать схему трехуровневого приложения под силу каждому, и рис. 2.1 доказывает, что мы тоже можем это сделать.

Рис. 2.1. Стандартная трехуровневая архитектура приложения. Это самый простой и наиболее распространенный вариант многоуровневой архитектуры приложения, при котором само приложение скомпоновано из нескольких различных уровней

Конечно, нарисовать схему из трех уровней проще простого, но сделать это — все равно что заявить, что у вас к стейку будет подан беарнский соус: это декларация намерений, не дающая никаких гарантий относительно конечного результата. В конечном итоге может получиться нечто совсем иное, в чем скоро вы сможете убедиться.

Чтобы задумать и спроектировать гибкое и сопровождаемое сложное приложение, есть несколько путей, но многоуровневая архитектура приложения представляется весьма популярным и проверенным подходом. Сложность заключается в его правильной реализации. Вооружившись трехуровневой схемой, подобной той, что показана на рис. 2.1, можно приступать к созданию приложения.

2.1.1. Познакомьтесь с Мэри Роуэн

Мэри Роуэн (Mary Rowan) является профессиональным .NET-разработчиком из частной компании, являющейся сертифицированным партнером Microsoft и производящей в основном веб-приложения. Ей 34 года, и программированием она занимается уже 11 лет. Поэтому Мэри считается одним из самых опытных сотрудников компании. Кроме исполнения своих основных должностных обязанностей в качестве старшего разработчика, она часто выступает в роли наставника для младших разработчиков. В целом Мэри довольна своей работой, но ее раздражает частое затягивание этапов разработки, что заставляет ее и других сотрудников работать сверхурочно, включая выходные, чтобы уложиться в сроки.

Она догадывается о том, что должны быть другие, более эффективные способы создания программных продуктов. Стремясь узнать приемы эффективной работы, она скупает множество книг по программированию, но редко выкраивает время для их изучения, поскольку большую часть свободного времени ей приходится тратить на заботы о муже и о двух дочерях. Мэри любит ходить в горы, увлечена кулинарией и точно знает, как готовить беарнский соус.

Мэри получила задание создать новое приложение для электронной торговли с применением ASP.NET Core MVC и Entity Framework, а в качестве хранилища данных воспользоваться SQL Server. Чтобы добиться максимальной модульности, приложение должно состоять из трех уровней.

Первой реализуемой функцией должен стать простой каталог предлагаемых товаров, извлеченный из таблицы базы данных и выведенный на веб-страницу (пример показан на рис. 2.2). Если пользователь, просматривающий перечень, входит в число постоянных покупателей, цена на все товары должна быть снижена на 5 %.

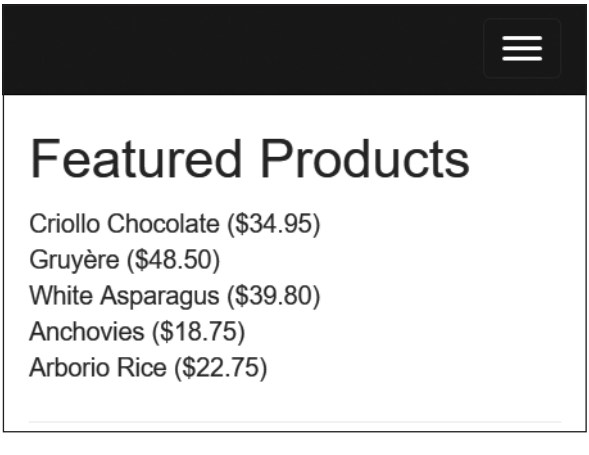


Рис. 2.2. Снимок экрана веб-приложения электронной торговли, разработку которого поручили Мэри. Это простой перечень предлагаемых товаров с указанием цены

Чтобы справиться со своей первой задачей, Мэри придется разработать следующие компоненты:

- *уровень данных* — включает таблицу базы данных `Products`, представленную всеми записями самой базы данных и классом `Product`, представляющим одну запись базы данных;
- *доменный уровень* — включает логику извлечения предлагаемых товаров;
- *уровень пользовательского интерфейса с MVC-контроллером* — обрабатывает поступающие запросы, извлекает нужные данные из уровня доменной логики и отправляет их в урезанное представление, которое в конечном итоге выводит список предлагаемых товаров.

Посмотрим, как Мэри реализует первый уровень приложения.

2.1.2. Создание уровня данных

Поскольку Мэри нужно будет извлекать информацию из таблицы базы данных, она решила начать с реализации уровня данных. Первый шаг — определение самой таблицы базы данных. Для создания указанной ниже таблицы (табл. 2.1) Мэри воспользовалась SQL Server Management Studio.

Таблица 2.1. Мэри создает таблицу Products со следующими столбцами

Имя столбца	Тип данных	Допускаются ли нулевые значения?	Первичный ключ
Id	uniqueidentifier	Нет	Да
Name	nvarchar(50)	Нет	Нет
Description	nvarchar(max)	Нет	Нет
UnitPrice	money	Нет	Нет
IsFeatured	bit	Нет	Нет

Для реализации уровня доступа к данным Мэри добавила к своему решению новую библиотеку. В листинге 2.1 показан разработанный ею класс `Product`.

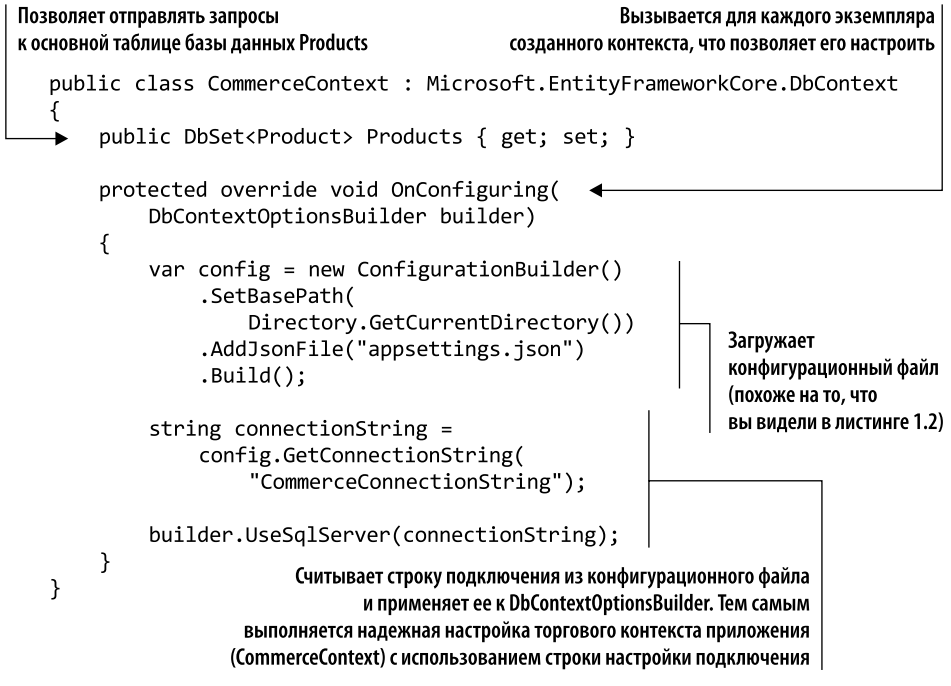
Листинг 2.1. Созданный Мэри класс Product

```
public class Product
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal UnitPrice { get; set; }
    public bool IsFeatured { get; set; }
}
```

Для доступа к данным Мэри решила воспользоваться Entity Framework. К своему проекту она добавила зависимость `Microsoft.EntityFrameworkCore.SqlServer` из пакета NuGet и реализовала сориентированный на приложение класс `DbContext`,

позволяющий ее приложению обращаться к таблице `Products` через класс `CommerceContext`, код которого показан в листинге 2.2.

Листинг 2.2. Созданный Мэри класс CommerceContext




```
public class CommerceContext : Microsoft.EntityFrameworkCore.DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(
        DbContextOptionsBuilder builder)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(
                Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json")
            .Build();

        string connectionString =
            config.GetConnectionString(
                "CommerceConnectionString");

        builder.UseSqlServer(connectionString);
    }
}
```

Поскольку класс `CommerceContext` загружает строку подключения из конфигурационного файла, этот файл нужно создать. Мэри добавляет к своему веб-проекту файл `appsettings.json` со следующим содержимым:

```
{
    "ConnectionStrings": {
        "CommerceConnectionString":
            "Server=.;Database=MaryCommerce;Trusted_Connection=True;"
    }
}
```

Краткие сведения об Entity Framework Core

Entity Framework Core является разработанной Microsoft средой отображения объектов на компоненты реляционной базы данных (Object/Relational Mapper, ORM). Она наводит мосты между моделями реляционных баз данных и объектно-ориентированным кодом наподобие того, что создается на языке C#. Эта среда позволяет разработчикам работать с абстракциями более высокого уровня, поскольку теперь уже не нужно самостоятельно создавать SQL-запросы: Entity Framework Core будет выполнять за них переход с C# на SQL.

Основным классом в Entity Framework является `DbContext`, представляющий собой паттерн элементарной операции — Unit of Work[1]. Этот паттерн состоит из локального кэша объектов, требуемых для одной коммерческой транзакции. `DbContext` позволяет получить доступ к информации, хранящейся в базе данных, подобной той, что была показана в примере таблицы `Products`.

Если среда Microsoft Entity Framework вам еще не знакома, не беда. В данном контексте детали реализации доступа к данным неважны, поэтому разобраться в примере можно даже в том случае, если вам лучше знакомы другие технологии доступа к данным[2].

ВНИМАНИЕ

CommerceContext загружает строку подключения из конфигурационного файла, заманивая вас в ловушку. Каждому новому экземпляру CommerceContext приходится считывать конфигурационный файл, даже если тот в ходе выполнения приложения нисколько не изменился. В CommerceContext не должно быть жестко запрограммированной строки подключения, но и загрузки конфигурационного значения из системы конфигурации также быть не должно. Этот вопрос будет рассмотрен в подразделе 2.3.3.

`CommerceContext` и `Product` являются открытыми типами, содержащимися в одной сборке. Мэри знает, что позже ей придется нарастить функциональность своего приложения, но создание компонента доступа к данным, необходимого для реализации первой функции приложения, теперь завершено (рис. 2.3).

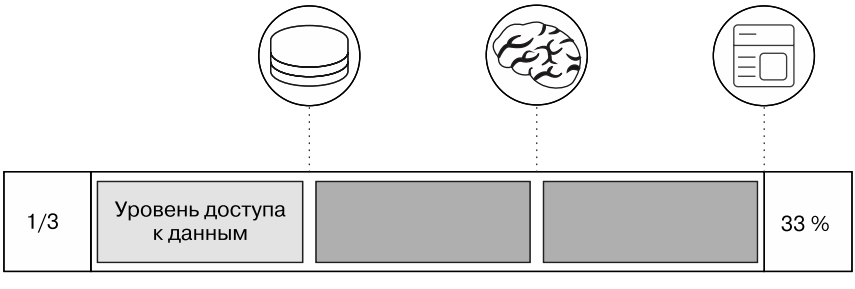


Рис. 2.3. Прогресс, достигнутый Мэри в реализации многоуровневой архитектуры, показанной на рис. 2.1

Теперь, когда уровень доступа к данным реализован, следующим логическим шагом станет *доменный уровень логики*. Этот уровень известен и под другими названиями: «уровень доменной логики», «бизнес-уровень» и «уровень бизнес-логики». На доменном уровне находится все поведение, которое должно иметься у приложения, применительно к той области, для которой оно разрабатывается.

---

1 *Фаулер М.* Шаблоны корпоративных приложений. — М.: Диалектика, 2018. — С. 205.

2 Entity Framework Core посвящена отдельная книга: *Smith J.* Entity Framework Core in Action. — Manning, 2018.

2.1.3. Создание уровня доменной логики

Если приложения не занимаются только выдачей отчетов, доменная логика присутствует в них всегда. Поначалу четкого представления о ней может и не сложиться, но по мере изучения области решаемой задачи станут проявляться изначально присущие этой логике, а также неявно выраженные правила и предположения. При отсутствии какой-либо доменной логики список товаров, показываемый классом `CommerceContext`, мог бы чисто технически использоваться непосредственно из уровня пользовательского интерфейса.

ВНИМАНИЕ

> Реализация доменной логики на уровне пользовательского интерфейса или на уровне доступа к данным ни к чему хорошему не приведет. Не создавайте себе ненужных проблем и приступайте к реализации доменного уровня с самого начала.

В требованиях к создаваемому Мэри приложению указано, что постоянные покупатели должны видеть прейскурант с 5 % скидкой. Мэри еще нужно понять, как распознать постоянного покупателя, поэтому она обратилась за советом к своему коллеге Йенсу.

Мэри: Мне нужно реализовать разрабатываемую бизнес-логику, чтобы постоянный покупатель получал 5%-ную скидку.

Йенс: Проще простого. Нужно лишь умножить цену на 0,95.

Мэри: Спасибо, конечно, но вопрос был не об этом. Я хотела узнать, как определить постоянного покупателя?

Йенс: Понятно. А это веб-приложение или программа для настольного компьютера?

Мэри: Это веб-приложение.

Йенс: Хорошо, тогда можно воспользоваться свойством `User` объекта `HttpContext` и проверить, принадлежит ли текущему пользователю роль `PreferredCustomer`.

Мэри: Не тот случай, Йенс. Код должен быть на доменном уровне. Это же библиотека, в ней нет объекта `HttpContext`.

Йенс: Ох. *(Задумывается на минутку.)* В таком случае для определения ценности покупателя нужно воспользоваться `HttpContext` среды ASP.NET. А затем можно передать эту ценность в доменную логику в качестве булева значения.

Мэри: Ну, я не знаю...

Йенс: Можно будет также разделить обязанности, поскольку твоей доменной логике не придется заниматься вопросами обеспечения безопасности. Не забывай про принцип единственной ответственности! Именно здесь и нужно проявить соответствующую гибкость!

Мэри: Возможно, ты прав.

Йенс дал совет, основываясь на своих технических познаниях в ASP.NET. Разговор его раздражал, и, чтобы Мэри от него поскорее отвязалась, он обрушил на нее множество заумных слов.

Дело в том, что Йенс не понимает, о чем говорит.

- *Он неправильно применяет концепцию разделения обязанностей.* При всей важности избавления доменной логики от обязанности обеспечения безопасности ее перемещение на уровень представления не поможет разделению обязанностей.
- *Гибкость он упомянул только потому, что недавно услышал, как кто-то рьяно отстаивал преимущества ее применения.*
- *Он совершенно не понимает, в чем заключается смысл принципа единственной ответственности.* Несмотря на то что предоставляемый гибкими Agile-методологиями быстрый цикл отклика может оказать помощь в соответствующем совершенствовании вашего программного продукта, сам по себе принцип единственной ответственности не зависит от выбранной методологии разработки программных средств.

Принцип единственной ответственности

Как уже говорилось в главе 1, принцип единственной ответственности (Single Responsibility Principle, SRP) гласит, что у каждого класса должна быть только одна ответственность, или, выражаясь яснее, у класса должна быть только одна причина для внесения изменений[1].

Если поместить SQL-инструкции в представление, содержащее HTML-разметку, то трудно будет кому-то не согласиться, что изменения в разметке будут случаться в другое время, с другой периодичностью и по другим причинам, нежели изменения в SQL-инструкциях. Наши SQL-инструкции изменяются при изменении нашей модели данных или же при необходимости настройки производительности. А наша разметка изменяется, когда требуется изменить внешний вид веб-приложения. Это разные вещи, меняющиеся по разным причинам. Поэтому непосредственное размещение SQL-инструкций в представлении является *нарушением SRP-принципа.*

Но чаще всего усмотреть в классе несколько причин для изменений не так-то и легко. В этом деле часто помогает взгляд на SRP через призму связности кода (code cohesion). *Связность* определяется как функциональное родство элементов класса или модуля. Чем ниже такое родство, тем слабее связность и тем выше риск того, что классом нарушается SRP-принцип.

Возможность выявления нарушений SRP-принципа — это одно, а вот определение того, должно ли это нарушение быть исправлено, — совершенно иное. Неразумно применять SRP-принцип в случае отсутствия «симптоматики». Ненужное разбиение классов, не создающих проблем с их сопровождаемостью, может внести дополнительную сложность. Искусство разработки программных продуктов заключается в управлении их сложностью.

---

[1] *Martin R. C.* Agile Principles, Patterns and Practices in C#. — Pearson Education, 2007. — P. 115.

К сожалению, Мэри последовала неразумным советам Йенса, создала новый проект библиотеки на C# и добавила к нему класс под названием `ProductService`, показанный в листинге 2.3. Чтобы класс `ProductService` прошел компиляцию, она должна добавить в него ссылку на созданный ею уровень обращения к данным, поскольку там определен класс `CommerceContext`.

Листинг 2.3. Созданный Мэри класс ProductService

```
public class ProductService
{
    private readonly CommerceContext dbContext;

    public ProductService()
    {
        this.dbContext = new CommerceContext();
    }

    public IEnumerable<Product> GetFeaturedProducts(
        bool isCustomerPreferred)
    {
        decimal discount =
            isCustomerPreferred ? .95m : 1;

        var featuredProducts =
            from product in this.dbContext.Products
            where product.IsFeatured
            select product;

        return
            from product in
                featuredProducts.AsEnumerable()
            select new Product
            {
                 Id = product.Id,
                 Name = product.Name,
                 Description = product.Description,
                 IsFeatured = product.IsFeatured,
                 UnitPrice =
                     product.UnitPrice * discount
            };
    }
}
```

Создание нового экземпляра CommerceContext для последующего использования

Получение всех товаров из базы данных, отфильтрованных по признаку рекомендуемых

Создание перечня товаров со скидкой на основе скидочного процента для конкретного покупателя

Мэри рада, что ей удалось поместить технологию доступа к данным (Entity Framework Core), конфигурацию и доменную логику в класс `ProductService`. Она делегировала сведения о покупателе вызывающему коду путем передачи параметра `isCustomerPreferred` и воспользовалась его значением для вычисления скидки на все товары.

Последующая доработка может включать замену жестко запрограммированного значения (`.95`) настраиваемым числом, но пока достаточно будет и этой реализации. Мэри близка к решению задачи. Осталось только реализовать уровень пользовательского интерфейса. Мэри решила, что это может подождать до завтра.

На рис. 2.4 отображен прогресс, достигнутый Мэри в реализации архитектуры, представленной на рис. 2.1.

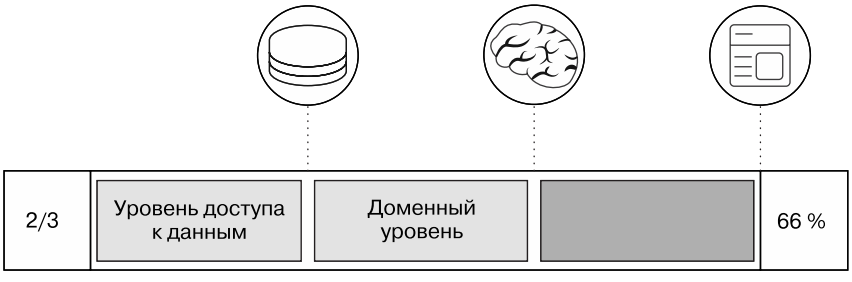


Рис. 2.4. По сравнению с тем, что было показано на рис. 2.3, теперь у Мэри есть реализация уровня доступа к данным и доменного уровня. Уровень пользовательского интерфейса по-прежнему ждет своей реализации

Мэри не понимает, что, позволив классу `ProductService` зависеть от класса `CommerceContext`, относящегося к уровню доступа к данным, она допустила сильную связанность доменного уровня с уровнем доступа к данным. Что с этим не так, мы объясним в разделе 2.2.

2.1.4. Создание уровня пользовательского интерфейса

На следующий день Мэри продолжила разработку приложения электронной торговли, добавив к своему решению применение среды ASP.NET Core MVC. Не переживайте, если вы еще незнакомы со средой ASP.NET Core MVC. Хитросплетения работы среды MVC пока не являются предметом нашего обсуждения. Нам важен вопрос потребления зависимостей, который, по большому счету, нейтрален к используемой платформе.

Краткие сведения о среде ASP.NET Core MVC

Среда ASP.NET Core MVC получила свое название от паттерна проектирования «Модель — представление — контроллер» (Model — View — Controller)[1]. В данном контексте важнее всего понять, что при поступлении веб-запроса его обрабатывает контроллер, по возможности используя при этом модель предметной области, и формирует ответ, который в итоге отображается представлением.

Контроллер — это, как правило, обычный класс, являющийся производным от абстрактного класса `Controller`. В нем имеется один или несколько методов действий, обрабатывающих запрос; например, в классе `HomeController` обычно имеется метод под названием `Index`, который обрабатывает запрос на представление страницы по умолчанию. Когда управление возвращается из метода действия, получившаяся модель передается представлению через экземпляр `ViewResult`.

[1] *Фаулер М.* Шаблоны корпоративных приложений. — М.: Диалектика, 2018. — С. 347.

В листинге 2.4 показано, как Мэри реализовала метод `Index` своего класса `HomeController`, чтобы извлечь предлагаемые товары из базы данных и передать их представлению. Чтобы сделать этот код компилируемым, ей пришлось добавить в него ссылки на уровень доступа к данным и на доменный уровень. Это происходит потому, что класс `ProductService` определен на доменном уровне, а класс `Product` — на уровне доступа к данным.

Листинг 2.4. Метод Index в классе контроллера по умолчанию

```
Определение принадлежности покупателя
к категории постоянных клиентов
  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();
  }
```

Создание ProductService из доменного уровня

Сохранение перечня в созданном контроллером обобщенном словаре ViewData для дальнейшего использования представлением

Получение перечня предлагаемых товаров (определенного на уровне доступа к данным) из ProductService

В качестве части жизненного цикла среды ASP.NET Core MVC свойство `User` класса `HomeController` автоматически заполняется нужным объектом пользователя, поэтому Мэри использует его для определения принадлежности текущего пользователя к категории постоянных покупателей. Вооружившись этой информацией, она может вызвать доменную логику для получения перечня предлагаемых товаров.

ПРИМЕЧАНИЕ

Когда Мэри создавала свой доменный уровень, она снова получила код с сильной связанностью. В данном случае HomeController сильно связан с ProductService. Если ProductService был бы стабильной зависимостью, то ничего страшного не произошло бы, но, как известно из главы 1, ProductService является нестабильной зависимостью. Нестабильность обусловливается тем, что в этом классе вводится требование по установке и настройке реляционной базы данных.

В создаваемом Мэри приложении прейскурант должен выводиться представлением под названием `Index`. Разметка для этого представления показана в листинге 2.5.

Листинг 2.5. Разметка представления Index

```
<h2>Featured Products</h2>
<div>
@{
    var products =
        (IEnumerable<Product>)this.ViewData["Products"];

    foreach (Product product in products)
    {
        <div>@product.Name (@product.UnitPrice.ToString("C"))</div>
    }
}
</div>
```

Получение товаров, сведения о которых заполнены контроллером

Перебор товаров в цикле, форматирование их поля UnitPrice и вывод их в формате HTML Перебор товаров в цикле, форматирование их поля UnitPrice и вывод их в формате HTML

Среда ASP.NET Core MVC позволяет создавать стандартный код HTML с небольшими фрагментами императивного кода, встроенного для доступа к объектам, созданным и переданным контроллером, который создал представление. В данном случае метод `Index` класса `HomeController` передает перечень предлагаемых товаров ключу под названием `Products`, который используется Мэри в представлении для вывода прейскуранта. На рис. 2.5 показано, что теперь Мэри реализовала всю архитектуру, ранее представленную на рис. 2.1.

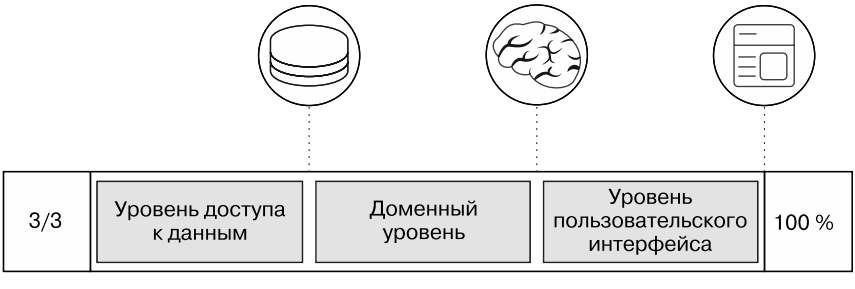


Рис. 2.5. Теперь Мэри реализовала все три уровня приложения

Когда готовы все три уровня, приложение теоретически считается работоспособным. Но убедиться в этом Мэри может только на практике.

2.2. Оценка приложения с сильной связанностью

Итак, Мэри реализовала все три уровня и настало время убедиться, работает ли приложение. Она нажала клавишу F5, и появилась веб-страница, показанная на рис. 2.2. Функция отображения предлагаемых товаров (Featured Products) готова, и Мэри стала действовать увереннее, почувствовав готовность к реализации следующей функции приложения. В конце концов, она следовала устоявшимся

передовым приемам проектирования и создала трехуровневое приложение... или все же не создала?

Удалось ли Мэри разработать по-настоящему многоуровневое приложение? Увы, нет, хотя у нее были самые лучшие намерения. Она создала три проекта Visual Studio, соответствующие трем уровням запланированной архитектуры. Если не присматриваться, то все это действительно похоже на желаемую многоуровневую архитектуру, но вскоре вы увидите, что код обладает сильной связанностью.

На платформе Visual Studio подобная разработка решений и проектов происходит легко и естественно. Если нужно привлечь функциональные возможности из другой библиотеки, можно просто добавить ссылку на нее и написать код, создающий новые экземпляры тех типов, которые определены в других библиотеках. Но при каждом добавлении ссылки вы добавляете зависимость.

2.2.1. Оценка схемы зависимостей

При работе с решениями в Visual Studio легко потерять из виду важные зависимости. Это связано с тем, что Visual Studio отображает все их разом со всеми другими ссылками проекта, которые могут указывать на сборки в библиотеке .NET Base Class Library (BCL). Чтобы разобраться в том, как модули в созданном Мэри приложении связаны друг с другом, можно нарисовать схему зависимостей (рис. 2.6).

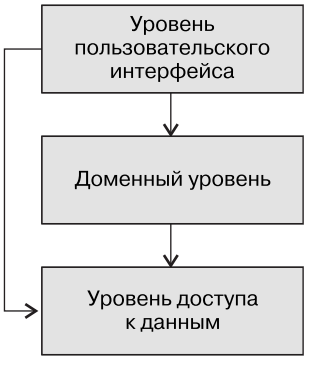


Рис. 2.6. Схема зависимостей приложения, созданного Мэри, показывающая зависимость модулей друг от друга. Стрелки указывают направления зависимостей модулей

Самым значительным заключением из увиденного на рис. 2.6 является то, что уровень пользовательского интерфейса зависит как от доменного уровня, так и от уровня доступа к данным. Похоже, что в ряде случаев пользовательский интерфейс может обойтись без доменного уровня. Это требует дальнейшего изучения.

2.2.2. Оценка компонуемости

Основная цель создания трехуровневого приложения состоит в разделении ответственности. Хотелось бы отделить нашу доменную модель от уровней доступа к данным и пользовательского интерфейса, чтобы ни одна из этих ответственностей не засоряла доменную модель. В больших приложениях возможность изолированной работы над каждой областью играет довольно важную роль. Для оценки реализации, созданной Мэри, можно задать простой вопрос: возможно ли изолированное использование каждого модуля?

Теоретически мы должны иметь возможность для произвольного составления модулей. Может потребоваться создание новых модулей для объединения уже имеющихся модулей с новыми, ранее непредвиденными способами, но в идеале у нас должна быть возможность проделать все это без необходимости что-либо изменять в существующих модулях. Можно ли использовать модули из созданного

Мэри приложения новыми впечатляющими способами? Рассмотрим ряд наиболее вероятных сценариев.

ПРИМЕЧАНИЕ

В следующем исследовании рассматривается возможность замены модулей, но имейте в виду, что этот прием используется для оценки компонуемости. Даже при отсутствии намерений по замене модулей такое исследование позволяет вскрыть потенциальные проблемы, относящиеся к связанности. Если обнаружится, что код имеет сильную связанность, то все преимущества слабой связанности будут утрачены.

Создание нового пользовательского интерфейса

Если созданное Мэри приложение станет успешным, у заинтересованных проектом сторон может появиться желание, чтобы она создала в среде Windows Presentation Foundation (WPF) полнофункциональную клиентскую версию. Возможно ли это при повторном использовании доменного уровня и уровня доступа к данным?

При исследовании схемы зависимостей, показанной на рис. 2.6, можно сразу убедиться, что от веб-интерфейса пользователя не зависит ни один из модулей, поэтому его можно удалить, а вместо него воспользоваться интерфейсом пользователя, созданным в среде WPF. Создание полнофункциональной клиентской версии на WPF — это создание нового приложения, которое делит основную часть своей реализации с исходным веб-приложением. На рис. 2.7 показано, как WPF-приложение вынуждено перенять зависимости веб-приложения. Исходное веб-приложение может оставаться в неизменном виде.

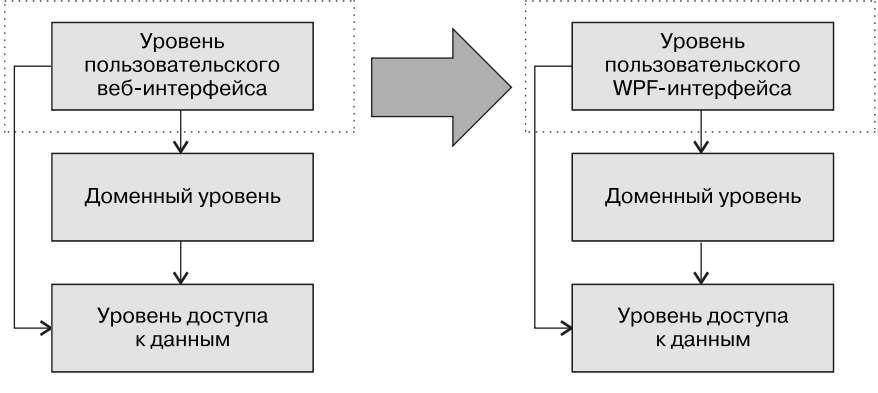


Рис. 2.7. Возможность замены пользовательского веб-интерфейса пользовательским WPF-интерфейсом обусловливается тем, что от пользовательского веб-интерфейса не зависит ни один из модулей. Заменяемая часть выделена пунктирным прямоугольником

Созданная Мэри реализация допускает замену уровня пользовательского интерфейса. Рассмотрим еще одну интересную перекомпоновку.

Создание нового уровня доступа к данным

Специалисты по анализу рынка, работающие вместе с Мэри, пришли к выводу, что для оптимизации прибыли ее приложение должно быть доступно в качестве облачного приложения, размещенного на Microsoft Azure. В Azure данные должны храниться в сервисе Azure Table Storage Service, обладающем широкими возможностями масштабирования. Этот механизм хранения данных основывается на гибких контейнерах данных, содержащих неограниченный объем информации. Данный сервис не обеспечивает наличие конкретной схемы базы данных, и в нем не соблюдается ссылочная целостность.

Хотя самые распространенные технологии доступа к данным в среде .NET основываются на ADO.NET Data Services, протоколом, используемым для связи со службой Table Storage Service, является HTTP. Такой тип базы данных иногда называют базой данных формата «ключ — значение», и у нее совершенно другое устройство, нежели у реляционной базы данных, доступной с помощью Entity Framework Core.

Чтобы приложение электронной торговли стало облачным, уровень доступа к базе данных нужно заменить модулем, использующим Table Storage Service. Возможно ли это?

Из схемы зависимостей, показанной на рис. 2.6, известно, что и уровень пользовательского интерфейса, и доменный уровень зависят от уровня доступа к данным, основанным на применении Entity Framework. При попытке удаления уровня доступа к данным решение перестанет компилироваться без реорганизации всех остальных проектов, поскольку требуемая зависимость будет утрачена. В большом приложении с десятками модулей можно также попытаться удалить некомпилируемые модули, чтобы посмотреть, что останется. В случае с приложением, разработанным Мэри, вполне очевидно, что нам придется удалить все модули, не оставив абсолютно ничего, что и показано на рис. 2.8.

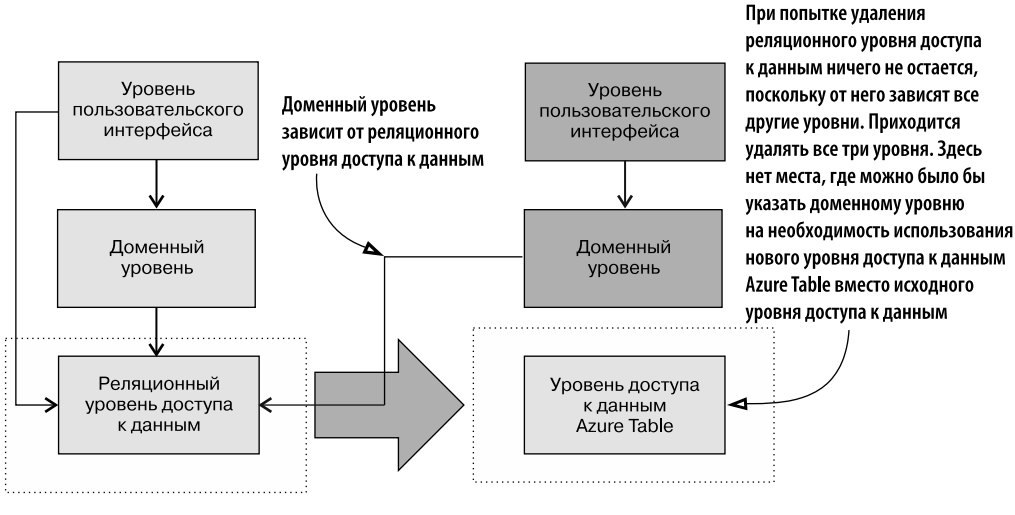


Рис. 2.8. Попытка заменить реляционный уровень доступа к данным

Хотя можно было бы разработать уровень доступа к данным среды Azure Table, подражающий интерфейсу (API), предоставляемому исходным уровнем доступа к данным, невозможно найти способ его применения без вторжения в другие части приложения. Приложение даже и близко не обладает той степенью компонуемости, которую хотелось бы иметь заказчикам проекта. Включение облачных возможностей, максимизирующих прибыль, требует серьезной переработки приложения, поскольку ни один из существующих модулей не годится для повторного использования.

Оценка других комбинаций

Приложение можно было бы проанализировать и при других комбинациях модулей, но, поскольку уже известно, что оно не в состоянии поддерживать необходимый сценарий, в этом нет никакого смысла, как, собственно, и в проверке абсолютно всех комбинаций.

Можно, к примеру, задаться вопросом, возможно ли заменить доменную модель какой-либо иной реализацией. Но в большинстве случаев это вызвало бы недоумение, поскольку в доменной модели заключено ядро приложения. Без этой модели теряется сам смысл существования большинства приложений.

2.3. Анализ недостающей компонуемости

Почему в созданной Мэри реализации не удалось достичь нужной степени компонуемости? Может, из-за того, что пользовательский интерфейс имеет прямую зависимость от уровня доступа к данным? Рассмотрим эту возможность более подробно.

2.3.1. Анализ схемы зависимостей

Почему пользовательский интерфейс зависит от библиотеки доступа к данным? Всему виной сигнатура метода данной доменной модели:

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

Предоставляет клиентам тип, определенный в библиотеке доступа к данным

Метод `GetFeaturedProducts` класса `ProductService` возвращает последовательность товаров, а класс `Product` определен на уровне доступа к данным. Чтобы пройти компиляцию, любой потребитель метода `GetFeaturedProducts` должен ссылаться на уровень доступа к данным. Можно изменить сигнатуру метода, чтобы вернуть тип, определенный в доменной модели. Это тоже будет правильно, но проблему не решит.

Предположим, что мы разрываем зависимость между пользовательским интерфейсом и библиотекой доступа к данным. Измененная схема зависимостей теперь будет выглядеть так (рис. 2.9).

Сможет ли Мэри после внесения этого изменения заменить уровень доступа к реляционной базе данных на какой-либо уровень, в котором заключен доступ к сервису Azure Table? К сожалению, нет, поскольку доменный уровень по-прежнему зависит от уровня доступа к данным. А пользовательский интерфейс, в свою очередь, все еще зависит от доменной модели. Если попытаться удалить исходный уровень доступа к данным, то от приложения ничего не останется. Первопричина проблемы кроется где-то в другом месте.

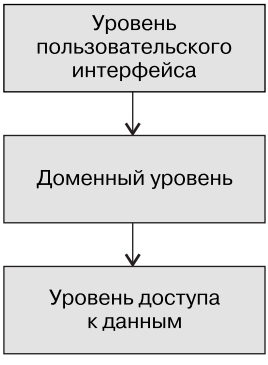


Рис. 2.9. Схема зависимостей при гипотетической ситуации, где зависимость пользовательского интерфейса от уровня доступа к данным удалена

2.3.2. Анализ интерфейса доступа к данным

Доменная модель зависит от уровня доступа к данным, поскольку на нем определена вся модель данных. Возможно, для реализации уровня доступа к данным вполне разумным решением было бы использование Entity Framework. Но с точки зрения слабого связывания ее непосредственное использование в доменной модели таковым не является.

Нежелательный в этом плане код может попасть в класс `ProductService`. Конструктор создает новый экземпляр класса `CommerceContext` и присваивает его приватному полю:

```
this.dbContext = new CommerceContext();
```

Тем самым класс `ProductService` получает сильную связь с уровнем доступа к данным. Нет никакого рационального способа перехватить этот фрагмент кода и заменить его чем-то другим. Ссылка на уровень доступа к данным жестко закодирована в классе `ProductService`!

Для извлечения объектов `Product` из базы данных в реализации метода `GetFeaturedProducts` используется `CommerceContext`:

```
var featuredProducts =
    from product in this.dbContext.Products
    where product.IsFeatured
    select product;
```

Ссылка на `CommerceContext` в `GetFeaturedProducts` усиливает жестко заданную зависимость, но к этому моменту ущерб уже нанесен. Нам нужен более подходящий способ компоновки модулей без такой сильной связанности. Если вернуться к преимуществам применения технологии DI, рассмотренным в главе 1, то можно увидеть, что в созданном Мэри приложении отсутствует следующее.

- Позднее связывание — сильная связанность доменного уровня с уровнем доступа к данным не позволяет развернуть две версии версий одного и того же приложения, где одна из них подключена к локальной базе данных SQL Server,

а другая размещается на Microsoft Azure, используя хранилище данных Azure Table Storage. Иными словами, загрузка нужного уровня доступа к данным с помощью позднего связывания невозможна.

- Расширяемость — поскольку все классы в приложении сильно связаны друг с другом, подключение сквозной функциональности (Cross-Cutting Concerns), например функции обеспечения безопасности из главы 1, становится дорогостоящим. Это требует внесения изменений во многие классы системы. Следовательно, такая конструкция с сильной связанностью не обладает достаточной расширяемостью.
- Сопровождаемость — кроме того, что добавление сквозной функциональности потребует радикальных изменений во всем приложении, каждое такое добавление, вероятнее всего, будет усложнять затрагиваемые им классы, затрудняя и чтение их кода. Следовательно, приложение станет не таким сопровождаемым, как бы этого хотелось Мэри.
- Параллельная разработка — если придерживаться предыдущего примера с добавлением сквозной функциональности, весьма нетрудно будет понять, что необходимость внесения радикальных изменений практически в весь объем кода не позволяет нескольким разработчикам работать над одним и тем же приложением в параллельном режиме. Возможно, вам (как и нам) уже приходилось сталкиваться с весьма болезненными конфликтами объединения при передаче своей работы системе управления версиями. Качественно спроектированная система со слабой связанностью сокращает количество возможных конфликтов объединения. Когда над созданным Мэри приложением начинает трудиться все больше разработчиков, работать, не наступая друг другу на пятки, становится все труднее.
- Тестируемость — мы уже установили, что в сложившейся ситуации замена уровня доступа к данным невозможна. Однако тестирование кода без базы данных является обязательным условием проведения модульных тестов. Но даже для проведения интеграционных тестов Мэри потребуется замена некоторых частей кода, а текущая конструкция ее затрудняет. Стало быть, разработанное Мэри приложение непригодно к тестированию.

А теперь можно задаться вопросом, как должна выглядеть желаемая схема зависимостей. Чтобы добиться максимальной многократности использования кода, желательно иметь минимальное количество зависимостей. С другой стороны, если в приложении вообще не будет зависимостей, оно станет практически бесполезным.

Какие зависимости вам потребуются и куда они должны быть направлены, зависит от предъявляемых к приложению требований. Но поскольку уже установлено, что намерений заменять доменный уровень совершенно иной реализацией у нас нет, можно предположить, что безопасная зависимость от него может быть у других уровней. Желаемая схема зависимостей, показанная на рис. 2.10, имеет весьма существенный недостаток, который будет мешать предстоящему в следующей главе созданию приложения со слабой связанностью.

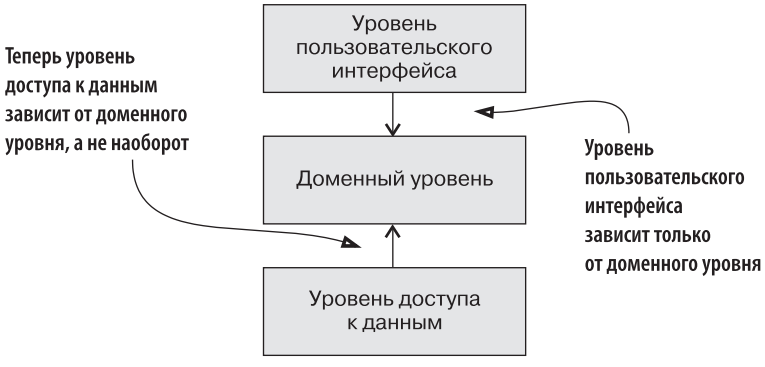


Рис. 2.10. Желаемая схема зависимостей

На рисунке показан разворот зависимости между доменным уровнем и уровнем доступа к данным. Подробности его осуществления будут изложены в следующей главе.

2.3.3. Прочие вопросы

В созданном Мэри коде есть и другие требующие решения проблемы, которые не хотелось бы оставлять без внимания.

- Похоже, что основная доменная модель реализована на уровне доступа к данным. Наряду с технической проблемой, заключающейся в ссылке доменного уровня на уровень доступа к данным, есть и концептуальная проблема, суть которой в том, что такой класс, как `Product`, определен на уровне доступа к данным. Открытый класс `Product` принадлежит доменной модели.
- По совету Йенса Мэри решила внедрить в пользовательский интерфейс код, определяющий принадлежность пользователя к постоянным покупателям. Но порядок такого определения должен быть частью бизнес-логики, следовательно, его реализация должна быть выполнена в доменной модели. Аргументация Йенса насчет разделения обязанностей и принципа единственной ответственности не оправдывает факт размещения кода в неправильном месте. Следовать принципу единственной ответственности в рамках одной библиотеки вполне допустимо, и это будет ожидаемым подходом.
- Мэри загрузила строку подключения из конфигурационного файла внутри класса `CommerceContext` (показанного в листинге 2.2). С точки зрения ее потребителей, зависимость от этого настроечного значения полностью скрыта. Как уже упоминалось при обсуждении кода листинга 2.2, в этой скрытности содержится ловушка.

  При всей важности настройки скомпилированного приложения полагаться на файлы конфигурации должны только файлы полностью готового приложения. Более гибким вариантом для многократно используемых библиотек будет не самостоятельное считывание ими конфигурационных файлов, а обязательное

свойство их настраиваемости со стороны вызывающего кода. В конце концов, первичным вызывающим кодом является точка входа в приложение. В этом месте все представляющие интерес настроечные данные могут быть считаны непосредственно из конфигурационного файла при запуске и по мере необходимости предоставлены базовым библиотекам. Мы хотим, чтобы конфигурация, требуемая `CommerceContext`, предоставлялась в явном виде.

- Похоже, что представление (показанное в коде листинга 2.5) содержит слишком много функций. Оно выполняет приведение строк и их конкретное форматирование. Эти функции должны быть перемещены в базовую модель.

2.4. Заключение

Код с сильной связанностью написать на удивление легко. Даже когда Мэри с явным намерением создавала трехуровневое приложение, оно превратилось в практически монолитный кусок спагетти-кода[1]. (Когда речь заходит о разбиении на уровни, этот процесс сравнивают с приготовлением лазаньи.)

Легкость написания кода с сильной связанностью обусловлена, в частности, тем, что к этому нас уже подталкивают языковые функции и используемые инструменты. Когда нужен новый экземпляр объекта, можно воспользоваться ключевым словом `new`. Если нет ссылки на требуемую сборку, среда Visual Studio упрощает ее добавление. Но при каждом использовании ключевого слова `new` вами вводится сильное связывание. Как уже говорилось в главе 1, не все случаи сильного связывания несут негативный оттенок, но вы должны стремиться к предотвращению сильных связанностей с нестабильными зависимостями.

Итак, у вас уже должно складываться представление о причинах проблематичности кода с сильной связанностью, но мы еще не показали вам способы избавления от его проблем. В следующей главе будет показан способ создания приложения, допускающего более гибкую компоновку с теми же функциями, что и у программы, созданной Мэри. Одновременно с этим будут рассмотрены и другие вопросы, упомянутые в разделе 2.3.3.

Резюме

- Сложные программные продукты должны решать множество разнообразных задач, таких как обеспечение безопасности, проведение диагностики, выполнение возложенных функций, достижение высокой производительности и обеспечение возможностей расширения.
- Слабое связывание побуждает к изолированной реализации всех обязанностей приложения, но в конечном итоге этот весьма непростой набор обязанностей все равно придется скомпоновать.

---

[1] *Brown W. J. et al.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. — Wiley Computer Publishing, 1998. — P. 119.

- Создать код с сильной связанностью довольно легко. Хотя не всякая сильная связанность наносит вред, подобной связанности с нестабильными зависимостями следует избегать.
- В приложении, созданном Мэри, способов замены уровня доступа к данным каким-либо другим уровнем доступа к данным не было, поскольку от этого уровня зависел доменный уровень. Сильная связанность, имеющаяся в приложении Мэри, не позволила воспользоваться преимуществами, предоставляемыми слабой связанностью: поздним связыванием, расширяемостью, сопровождаемостью, тестируемостью и параллельной разработкой.
- От конфигурационных файлов должно зависеть только полностью готовое приложение. Другие части приложения не должны запрашивать значения из конфигурационного файла, их настройкой должен заниматься вызывающий код.
- Принцип единственной ответственности гласит, что у каждого класса должна быть только одна причина для внесения изменений.
- Принцип единственной ответственности должен рассматриваться с точки зрения связности. Связность определяется как функциональная взаимосвязь элементов класса или модуля. Чем ниже степень родства, тем ниже связность; а чем ниже связность, тем больше вероятность нарушения классом принципа единственной ответственности.

3 Создание кода со слабой связанностью

В этой главе

- Изменение созданного Мэри приложения электронной торговли с приданием ему слабой связанности.
- Анализ аспектов, придающих приложению слабую связанность.
- Оценка получившегося приложения со слабой связанностью.

Когда готовят стейк на гриле, прежде чем разрезать мясо на кусочки, важно дать ему отлежаться. За это время соки перераспределяются, что делает стейк более сочным. Если нарезать мясо слишком рано, все соки из него вытекут и оно станет слишком сухим и менее вкусным. Перед гостями за такое упущение было бы стыдно, поскольку вы собирались поразить их своими кулинарными способностями. Для любой профессии важны понимание и овладение передовыми методами работы, но не менее важно понимать первопричины негативной практики, из которой необходимо извлекать соответствующие уроки.

Понимание разницы между передовыми и порочными методами играет в обучении весьма важную роль. Именно поэтому предыдущая глава была полностью посвящена примеру и анализу кода с сильной связанностью: именно анализ и позволил вскрыть его порочность.

Если кратко, то слабая связанность дает массу преимуществ — позднее связывание, расширяемость, сопровождаемость, тестируемость и возможность параллельной

разработки. С сильной связанностью эти преимущества утрачиваются. Хотя не всякая сильная связанность нежелательна, нужно стремиться избегать сильных связанностей с нестабильными зависимостями. При этом для решения проблем, выявленных в ходе проведенного анализа, можно воспользоваться внедрением зависимостей (Dependency Injection, DI). Поскольку DI является радикальным отходом от тех способов, которыми Мэри создавала свое приложение, изменять уже существующий код мы не будем. Код будет создаваться с нуля.

ПРИМЕЧАНИЕ

Не нужно на основании этого делать вывод о невозможности реструктуризации любого существующего приложения в сторону использования внедрения зависимостей или рассматривать данное решение в качестве рекомендации переписывать любое существующее приложение с нуля. Крупные переделки слишком затратны и рискованны. Предпочтительнее неспешные, пошаговые реструктуризации. Это совершенно не говорит о том, что реструктуризации даются легко. Дело это отнюдь не простое. Исходя из нашего опыта, чтобы выполнить реструктуризацию, придется приложить немало усилий[1].

Начнем с краткой оценки приложения, созданного Мэри. Наряду с этим наметим подходы к его переработке и определим, как должен будет выглядеть желаемый конечный результат нашей работы.

3.1. Создание нового варианта приложения электронной торговли

В результате анализа приложения Мэри, проведенного в главе 2, выяснилось, что нестабильные зависимости имели сильную связанность между различными уровнями. Как показано на схеме зависимостей, воспроизведенной на рис. 3.1, от уровня доступа к данным зависит как доменный уровень, так и уровень пользовательского интерфейса.

В этой главе мы постараемся развернуть зависимость между доменным уровнем и уровнем доступа к данным. Иначе говоря, вместо того, чтобы доменный уровень зависел от уровня доступа к данным, уровень доступа к данным, как показано на рис. 3.2, будет зависеть от доменного уровня.

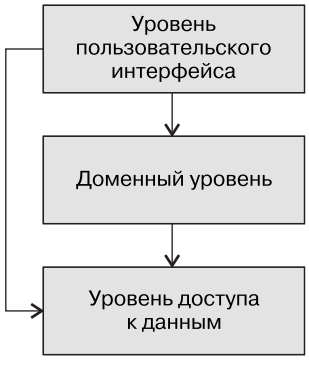


Рис. 3.1. Схема зависимостей в созданном Мэри приложении, показывающая, как модули зависят друг от друга

[1] По теме реструктуризации есть целая книга: *Feathers M. C.* Working Effectively with Legacy Code. — Prentice Hall, 2004.

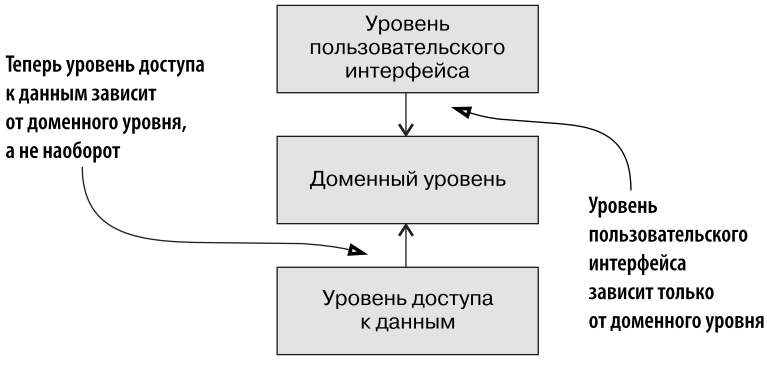


Рис. 3.2. Схема зависимостей для приложения Мэри с желаемым разворотом зависимости

При создании этого разворота зависимости допускается замена уровня доступа к данным без необходимости полной переработки приложения. (Это радикальный отход от метода, используемого Мэри при создании приложения.) Попутно будут применены несколько паттернов. Затем будет использовано внедрение через конструктор, рассмотренное в главе 1. И наконец, будет также применено внедрение зависимостей через метод класса (Method Injection) и корень композиции (Composition Root) в качестве места разрешения зависимостей, которые будут рассмотрены по ходу дела.

Данный подход приведет к созданию еще нескольких классов, поскольку мы сосредоточимся на разделении обязанностей в приложении. Мэри определила четыре класса, а мы определим девять классов и три интерфейса. На рис. 3.3. показан несколько более углубленный взгляд на приложение с классами и интерфейсами, создаваемыми в этой главе.

На рис. 3.4 представлен порядок взаимодействия основных классов, имеющихся в приложении. В конце главы будет еще раз рассмотрена более детальная версия этой схемы.

При создании программного продукта предпочтительнее начинать с самого значимого места — той части, что наиболее видима для потребителей. Как и в приложении электронной торговли, созданном Мэри, зачастую это пользовательский интерфейс. С него-то мы и начнем свой путь, наращивая функциональные возможности до тех пор, пока не будет готова одна из функций, после чего перейдем к другой. Такая технология «от потребителя» помогает сконцентрироваться на требуемой функциональности, не усложняя решение.

В главе 2 Мэри применила обратный подход. Она начала с уровня доступа к данным и пошла от него, работая по принципу «шиворот-навыворот», то есть от себя к потребителю. Сказано, возможно, грубовато, но вскоре вы увидите, что противоположный подход даст более быстрое осознание того, каким именно получается создаваемый продукт. Вот почему мы начнем создавать приложение в обратном порядке, с уровня пользовательского интерфейса, после чего создадим доменный уровень и лишь последним — уровень доступа к данным.

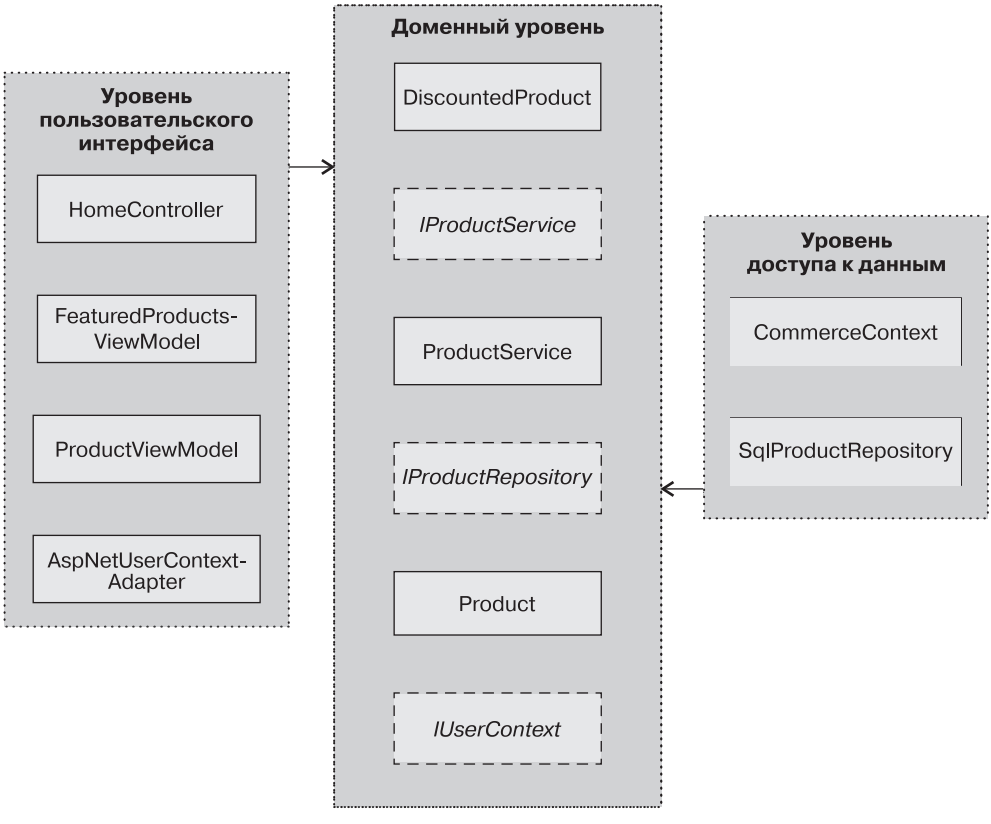


Рис. 3.3. Классы и интерфейсы, которые будут в приложении к концу главы. Интерфейсы обозначены пунктирными линиями

ПРИМЕЧАНИЕ

Технология «от потребителя» тесно связана с принципом YAGNI — You Aren't Gonna Need It («Вам это не понадобится»). Этот принцип подчеркивает, что должны быть реализованы только самые необходимые функции и эти реализации должны быть максимально простыми.

Поскольку мы практикуем разработку на основе тестирования (Test-Driven Development, TDD), то, как только наш подход «от потребителя» потребует создания нового класса, мы начнем с написания модульных тестов. Хотя для создания рассматриваемого примера были написаны модульные тесты, TDD не требуется для реализации и использования DI, поэтому показывать эти тесты в книге мы не намерены. Если вам интересно, то в сопровождающем книгу исходном коде эти тесты есть. Вернемся непосредственно к нашему проекту и начнем с создания пользовательского интерфейса.

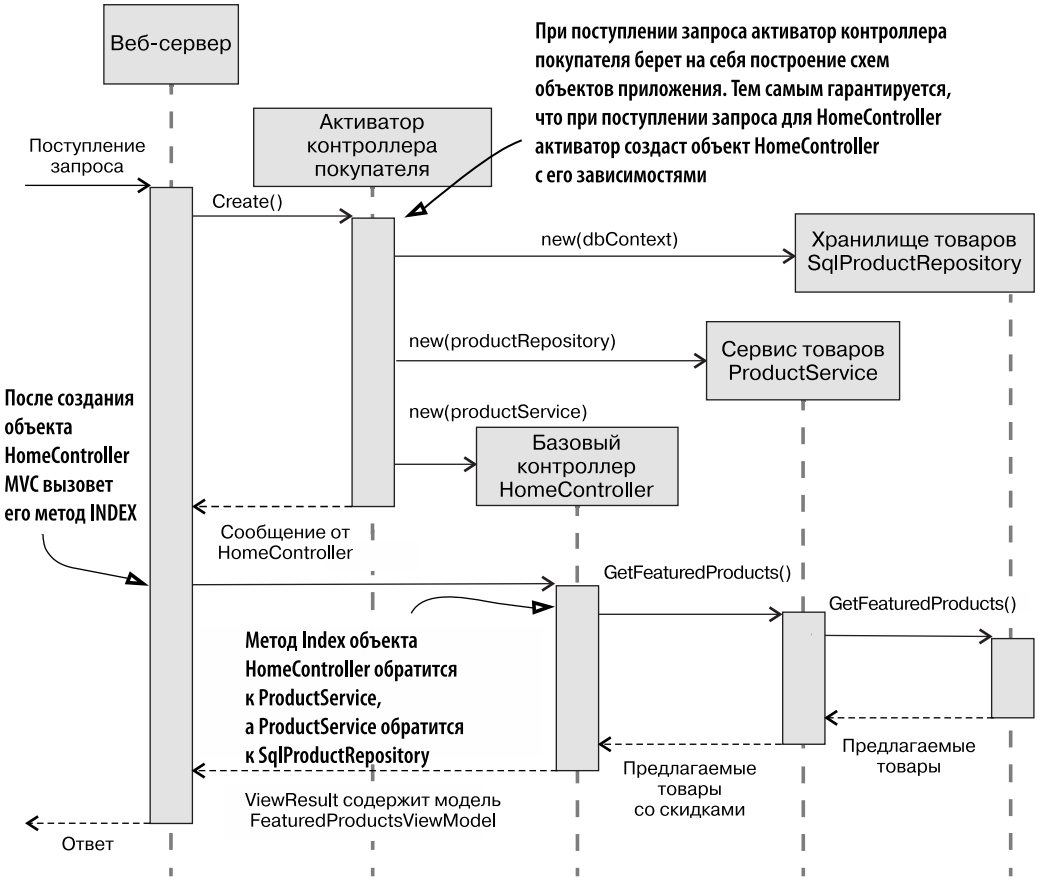


Рис. 3.4. Циклограмма, демонстрирующая взаимодействие между элементами, задействованными в реализации технологии DI в приложении электронной торговли, создаваемом в этой главе

3.1.1. Создание пользовательского интерфейса, более поддающегося сопровождению

Определяя, каким будет список предлагаемых товаров, Мэри решила создать приложение, извлекающее соответствующие элементы данных из базы и отображающее их в виде списка (который еще раз показан на рис. 3.5). Поскольку нам известно, что участников проекта больше заинтересует визуальный результат, то пользовательский интерфейс представляется вполне подходящим местом для старта разработки проекта.

Первое, что нужно сделать, открыв Visual Studio, — это добавить к решению новое приложение ASP.NET Core MVC. Поскольку прейскурант должен попасть

на начальную страницу, первым делом изменим содержимое файла `Index.cshtml`, включив в него разметку, показанную в листинге 3.1[1].

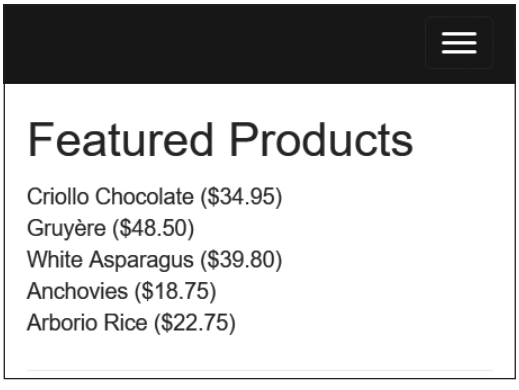


Рис. 3.5. Снимок экрана веб-приложения электронной торговли

Листинг 3.1. Разметка представления Index.cshtml

```
@model FeaturedProductsViewModel

<h2>Featured Products</h2>
<div>
    @foreach (ProductViewModel product in this.Model.Products)
    {
        <div>@product.SummaryText</div>
    }
</div>
```

Обратите внимание, насколько чище выглядит код листинга 3.1 по сравнению с исходной разметкой, созданной Мэри (листинг 3.2).

Листинг 3.2. Исходная разметка созданного Мэри представления Index из главы 2

```
<h2>Featured Products</h2>
<div>
@{
    var products = (IEnumerable<Product>)this.ViewData["Products"];

    foreach (Product product in products)
    {
        <div>@product.Name (@product.UnitPrice.ToString("C"))</div>
    }
}
</div>
```

[1] Представление для действия Index, принадлежащего HomeController, обычно находится в проекте веб-приложения в каталоге /Views/Home/Index.cshtml.

Первым усовершенствованием станет отказ от приведения словарного элемента к последовательности товаров с целью получения возможности выполнения итерации. Достигается оно весьма просто за счет использования специальной MVC-директивы `@model`. Это будет означать, что свойство страницы `Model` относится к типу `FeaturedProductsViewModel`. За счет использования директивы `@model` MVC обеспечит приведение значения, возвращаемого из контроллера к типу `FeaturedProductsViewModel`. Вторым усовершенствованием станет извлечение всей отображаемой строки товара непосредственно из свойства `SummaryText`, принадлежащего объекту `ProductViewModel`.

Оба улучшения связаны с применением моделей, характерных для представлений, в которых заключено поведение представления. Эти модели относятся к так называемым обычным старым объектам CLR (Plain Old CLR Objects, POCO)[1]. Схематично их структура показана в листинге 3.3.

Листинг 3.3. Классы FeaturedProductsViewModel и ProductViewModel

```
public class FeaturedProductsViewModel
{
    public FeaturedProductsViewModel(
        IEnumerable<ProductViewModel> products)
    {
        this.Products = products;
    }

    public IEnumerable<ProductViewModel> Products
        { get; }
}

public class ProductViewModel
{
    private static CultureInfo PriceCulture = new CultureInfo("enUS");

    public ProductViewModel(string name, decimal unitPrice)
    {
        this.SummaryText = string.Format(PriceCulture,
            "{0} ({1:C})", name, unitPrice);
    }

    public string SummaryText { get; }
}
```

В FeaturedProductsViewModel содержится список экземпляров ProductViewModel. Оба класса относятся к POCO, что позволяет подвергать их модульному тестированию

Чтобы логика вывода на экран была инкапсулирована, свойство SummaryText берется из двух значений — name и unitPrice

Использование моделей представлений упрощает представление, что хорошо, поскольку представления сложнее поддаются тестированию. Вдобавок это облегчает разработчику пользовательского интерфейса работу над приложением.

---

[1] Обычный старый объект CLR (Plain Old CLR Object, POCO), или, как его иногда называют, Plain Old C# Object, — это простой класс, созданный на языке C# либо на другом языке для общеязыковой среды выполнения (Common Language Runtime, CLR), свободный от зависимостей внешней среды.

ПРИМЕЧАНИЕ

Вы заметили недочеты в созданной Мэри исходной разметке? Хотя вызов UnitPrice.ToString("C") приводит к форматированию десятичного значения в значение валюты, данное форматирование происходит на основе культурных предпочтений пользователей, предоставляемых приложению их браузерами. Следовательно, посетитель из США видит знак доллара, а из Дании — значок датской кроны. Если бы обе валюты имели одинаковые значения, с этим можно было бы смириться, но дело обстоит иначе. Получится, что датчане получат товары по цене, составляющей некую долю от их предполагаемой стоимости. Именно поэтому в модели ProductViewModel имеется прямое указание на информацию о той или иной культуре.

Чтобы код, показанный в листинге 3.1, заработал, `HomeController` должен возвратить представление с экземпляром `FeaturedProductsViewModel`. На первом этапе это действие может быть реализовано внутри `HomeController`:

```
public ViewResult Index()
{
    var vm = new FeaturedProductsViewModel(new[]
    {
        new ProductViewModel("Chocolate", 34.95m),
        new ProductViewModel("Asparagus", 39.80m)
    });

    return this.View(vm);
}
```

Создание модели представления с жестко закодированным списком товаров со скидками

Заключение модели представления в MVC-объект ViewResult с помощью вспомогательного MVC-метода View

Мы жестко закодировали список товаров со скидками внутри метода `Index`. В качестве конечного данный результат нас не устраивает, но он позволяет веб-приложению функционировать без ошибок и дает нам возможность продемонстрировать участникам проекта пусть незаконченный, но вполне работоспособный пример приложения (в виде своеобразной заглушки), чтобы они могли высказать свои мнения.

ВНИМАНИЕ

С точки зрения технологии DI POCO-объекты, DTO-объекты и модели представлений, подобные FeaturedProductsViewModel и ProductViewModel, реального интереса не представляют[1]. В них не содержится никакого поведения, которое вы могли бы перехватить, заменить или имитировать. Это всего лишь объекты данных. Тем самым обеспечивается безопасность их создания в вашем коде и не возникает никакого риска сильной связанности вашего кода с этими объектами данных. Объекты содержат данные, со-

[1] Объект переноса данных (Data Transfer Object, DTO) используется для переноса данных между процессами.

ответствующие ходу выполнения приложения, которые проходят потоком через систему после таких давно созданных классов, как HomeController и ProductService.

На данном этапе реализована только заглушка уровня пользовательского интерфейса; полной реализации доменного уровня и уровня доступа к данным по-прежнему нет. Одним из преимуществ начала разработки приложения с уровня пользовательского интерфейса является наличие части программного продукта, которую можно запустить и протестировать. Сравните это с тем, что было сделано Мэри на сопоставимой стадии. Она достигла результата, позволяющего запустить приложение, значительно позже. Веб-приложение, использующее заглушку, изображено на рис. 3.6.

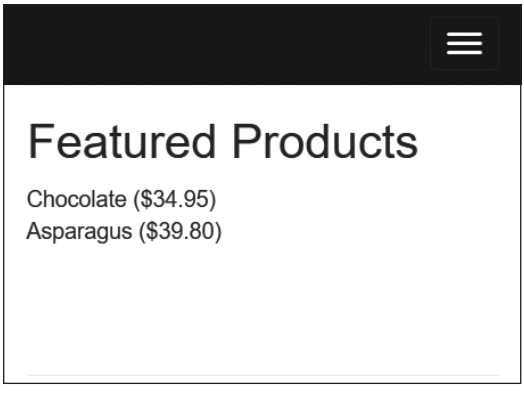


Рис. 3.6. Снимок экрана веб-приложения электронной торговли. Здесь список товаров жестко закодирован

Чтобы наш `HomeController` выполнил свои обязанности и сделал что-либо привлекательное, ему нужен список предлагаемых товаров с доменного уровня. К этим товарам должна применяться скидка. В главе 2 Мэри заключила соответствующую этому логику в свой класс `ProductService`, и мы поступим точно так же.

Метод `Index` объекта `HomeController` для извлечения предлагаемых товаров, преобразования полученных данных в экземпляр `ProductViewModel` и с последующим добавлением результата в `FeaturedProductsViewModel` будет использовать экземпляр `ProductService`. Но, если взглянуть со стороны `HomeController`, объект `ProductService` является нестабильной зависимостью, поскольку относится к такой зависимости, которая находится в стадии разработки и которой еще нет. Если нужно протестировать `HomeController` изолированно, разработать в параллельном режиме `ProductService` или же в будущем заменить или перехватить его, нужно ввести шов (seam).

Следует напомнить, что при анализе реализации, выполненной Мэри, наличие такой нестабильной зависимости считалось чуть ли не смертным грехом. Если ее

допустить, то возникнет сильная связанность с используемым типом. Чтобы избежать этой сильной связанности, мы введем интерфейс и воспользуемся технологией, которая называется внедрением через конструктор (Constructor Injection); как и кем создается экземпляр, `HomeController` неважно (листинг 3.4).

Листинг 3.4. Класс HomeController

```
public class HomeController : Controller
{
    private readonly IProductService productService;

    public HomeController(
        IProductService productService)
    {
        if (productService == null)
            throw new ArgumentNullException(
                "productService");

        this.productService = productService;
    }

    public ViewResult Index()
    {
        IEnumerable<DiscountedProduct> products =
            this.productService.GetFeaturedProducts();

        var vm = new FeaturedProductsViewModel(
            from product in products
            select new ProductViewModel(product));

        return this.View(vm);
    }
}
```

Конструктор указывает, что желающие воспользоваться классом должны предоставить экземпляр интерфейса IProductService

Контрольный оператор не позволяет поставляемому экземпляру иметь значение null, выдавая исключение

Внедренная зависимость может быть сохранена впрок и безопасно использована другими компонентами класса HomeController

Модель представления формируется из списка предлагаемых товаров

Мы изменили ProductViewModel, чтобы принять DiscountedProduct вместо строки и десятичного значения

Сохраненная зависимость productService. Обратите внимание, что GetFeaturedProducts возвращает коллекцию DiscountedProduct, а не Product. Класс DiscountedProduct определен на доменном уровне

Как уже говорилось в главе 1, внедрение через конструктор – это статическое определение списка требуемых зависимостей путем указания их конструктору класса в качестве параметров. Именно это `HomeController` и делает. В своем открытом конструкторе он определяет, какие зависимости требуются ему для корректной работы.

Когда впервые зашла речь о технологии внедрения через конструктор, нам было трудно понять реальную выгоду от ее применения. Неужели обязанность управления зависимостью перекладывается на какой-то другой класс? Да, так и есть — и в этом весь смысл. В многоуровневом приложении можно возложить эту обязанность всецело на самое начало кода приложения в корне композиции (Composition Root).

Корень композиции

Как уже говорилось в подразделе 1.4.1, нам хотелось бы иметь возможность провести компоновку наших классов в приложение способом, похожим на одновременное подключение к сети электроприборов. Такой уровень модульности может быть достигнут путем централизации создания наших классов в одном месте. Мы называем это место точкой входа.

Точка входа располагается как можно ближе к точке входа в приложение. В большинстве типов приложений .NET Core точкой входа является метод `Main`. Внутри точки входа можно принять решение о компоновке приложения вручную, то есть об использовании чистого внедрения зависимостей (Pure DI), или же делегировать компоновку DI-контейнеру. Более подробно точка входа будет рассмотрена в главе 4.

Поскольку конструктор с аргументом добавляется в `HomeController`, создание `HomeController` без этой зависимости станет невозможным. Вот почему такое действие и было предпринято. Но это означает, что главный экран приложения будет испорчен, поскольку среде MVC неизвестно, как должен быть создан наш `HomeController`, если только эта среда не получит соответствующий инструктаж.

Фактически создание `HomeController` не является обязанностью уровня пользовательского интерфейса; за это отвечает точка входа[1]. Поэтому мы считаем, что создание уровня пользовательского интерфейса завершено, а к созданию `HomeController` мы вернемся чуть позже. На рис. 3.7 показано текущее состояние реализации архитектуры, изображенной на рис. 3.2.

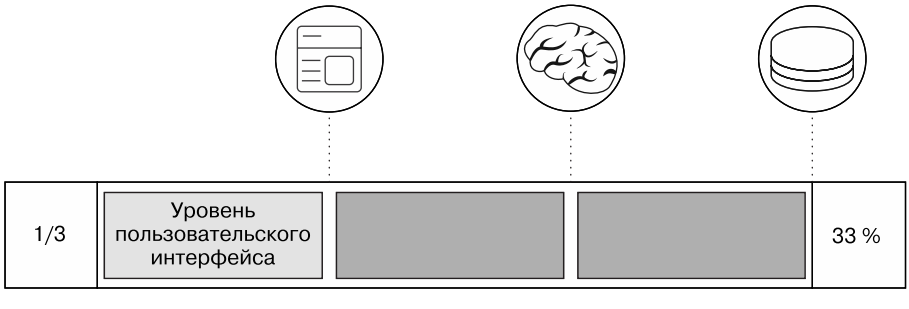


Рис. 3.7. На данной стадии реализован только уровень пользовательского интерфейса; к доменному уровню и уровню доступа к данным мы еще не приступали

Перемещаемся к следующему этапу по воссозданию нашего приложения электронной торговли и приступаем к реализации доменной модели.

[1] Согласно более подробным объяснениям в главе 4 точка входа не является частью уровня пользовательского интерфейса, даже притом, что она может быть помещена с ним в одну и ту же сборку.

3.1.2. Создание независимой доменной модели

Доменная модель является простой классической библиотекой C#, добавляемой к решению. Она будет содержать POCO-объекты и интерфейсы. С помощью POCO-объектов будет смоделирован домен, а интерфейс обеспечит абстракции, которые послужат в качестве наших основных внешних точек входа в доменную модель. Ими будет предоставлено соглашение, согласно которому доменная модель будет взаимодействовать с предстоящей реализацией уровня доступа к данным.

`HomeController`, представленный в предыдущем разделе, еще не скомпилирован, поскольку у нас нет определения абстракции `IProductService`. В этом разделе к приложению электронной торговли будет добавлен новый проект доменного уровня и ссылка на проект доменного уровня из проекта MVC, как это и было сделано Мэри. Тогда все будет в порядке, а анализ схемы зависимостей будет отложен до раздела 3.2, чтобы можно было составить о ней более полное представление. В листинге 3.5 показана абстракция `IProductService`.

Листинг 3.5. Интерфейс IProductService

```
public interface IProductService
{
    IEnumerable<DiscountedProduct> GetFeaturedProducts();
}
```

`IProductService` представляет собой основу текущего доменного уровня в том смысле, что «перебрасывает мосты» к уровню пользовательского интерфейса и к уровню доступа к данным. Он служит связующим звеном всех уровней исходного приложения.

Единственным компонентом абстракции `IProductService` является метод `GetFeaturedProducts`. Он возвращает коллекцию экземпляров `DiscountedProduct`. Каждый экземпляр `DiscountedProduct` содержит имя `Name` и цену за единицу товара `UnitPrice`. Это простой POCO-класс, в чем можно убедиться, изучив листинг 3.6, и это определение дает нам все необходимое для выполнения компиляции нашего решения на Visual Studio.

Листинг 3.6. POCO-класс DiscountedProduct

```
public class DiscountedProduct
{
    public DiscountedProduct(string name, decimal unitPrice)
    {
        if (name == null) throw new ArgumentNullException("name");

        this.Name = name;
        this.UnitPrice = unitPrice;
    }

    public string Name { get; }
    public decimal UnitPrice { get; }
}
```

Краеугольным камнем реализации технологии DI является принцип программирования на основе интерфейсов, а не на основе конкретных классов. Благодаря ему можно заменять одну конкретную реализацию другой. Прежде чем продолжить, постараемся вникнуть в роль интерфейсов в раскрытии данной темы.

ВНИМАНИЕ

Программирование на основе интерфейсов не означает, что все классы должны реализовывать какой-либо интерфейс. Обычно скрывать POCO-объекты, DTO-объекты и модели представлений за интерфейсом нет никакого смысла, поскольку в них отсутствует поведение, требующее применения имитаций, перехватов или замен. Поскольку DiscountedProduct, FeaturedProductsViewModel и ProductViewModel являются моделями (представления), интерфейс в них не реализуется. Мы рассмотрим вопрос применения интерфейсов или абстрактных классов в этом разделе чуть позже.

Далее мы создадим реализацию `ProductService`. Методом `GetFeaturedProducts` этого класса `ProductService` для извлечения списка предлагаемых товаров и применения всевозможных скидок будет использоваться экземпляр `IProductRepository` и возвращаться список экземпляров `DiscountedProduct`.

Общая абстракция доступа к данным предоставляется паттерном `Repository`, поэтому абстракция `IProductRepository` будет определена в библиотеке доменной модели[1] (листинг 3.7).

Листинг 3.7. IProductRepository

```
public interface IProductRepository
{
    IEnumerable<Product> GetFeaturedProducts();
}
```

`IProductRepository` является интерфейсом к уровню доступа к данным, возвращающем сырые сущности из постоянного хранилища. В отличие от него `IProductService` относится к бизнес-логике, например, как в данном случае, к скидкам, и преобразует сущности в более конкретизированный объект. В полноценном хранилище будет больше методов для поиска и изменения товаров, но, следуя принципу «от потребителя», мы определяем только классы и компоненты, необходимые для выполнения текущей задачи. В код проще добавить функциональность, чем что-либо из него удалить.

Поскольку мы задались целью развернуть в обратную сторону зависимость между доменным уровнем и уровнем доступа к данным, интерфейс `IProductRepository` определен на доменном уровне. Реализация `IProductRepository` как части уровня доступа к данным будет рассмотрена в следующем разделе. Тем самым нашей зависимости будет позволено указывать на доменный уровень.

---

[1] Описание паттерна проектирования «Хранилище» (Repository) дано в книге: *Фаулер М.* Шаблоны корпоративных приложений. — М.: Диалектика, 2018. С. 341–346. Но сложившийся способ его применения имеет мало общего с исходным описанием паттерна. В этом примере мы последуем обычному использованию, а не упомянутому описанию Фаулера, поскольку такое использование проще распознать и легче понять.

Сущность

Термин «сущность» пришел из предметно-ориентированного проектирования (Domain-Driven Design) и относится к предметному объекту с долгосрочной идентичностью, не связанному с конкретным экземпляром объекта[1]. Возможно, это воспринимается слишком абстрактно и условно, но это значит, что сущность представляет собой объект, существующий вне произвольных битов памяти. Любой экземпляр .NET-объекта имеет адрес в памяти (идентифицируемость), но у сущности имеется идентифицируемость, существующая в пределах времени жизни процессов.

Зачастую для идентификации сущностей и гарантированной возможности их сохранения и прочтения даже после перезагрузки главного компьютера используются базы данных и первичные ключи. Предметный (доменный) объект `Product` является сущностью, так как у понятия товара продолжительность жизни превышает время существования отдельного процесса, и для его идентификации в `IProductRepository` используется идентификатор товара.

ПРИМЕЧАНИЕ

Позволяя ProductService зависеть от IProductRepository, мы разрешаем поведению быть замененным или перехваченным. Помещая это поведение в другую библиотеку, мы разрешаем заменить всю библиотеку.

Как показано в листинге 3.8, класс `Product` также реализован с допустимым минимумом компонентов.

Листинг 3.8. Сущность Product

```
public class Product
{
    public string Name { get; set; }
    public decimal UnitPrice { get; set; }
    public bool IsFeatured { get; set; }

    public DiscountedProduct ApplyDiscountFor(
        IUserContext user)
    {
        bool preferred =
            user.IsInRole(Role.PreferredCustomer);

        decimal discount = preferred ? .95m : 1.00m;

        return new DiscountedProduct(
            name: this.Name,
            unitPrice: this.UnitPrice * discount);
    }
}
```

Класс Product содержит лишь свойства Name, UnitPrice и IsFeatured, поскольку только они нужны для реализации желаемой функции приложения

Этому методу в качестве аргумента требуется IUserContext, являющийся частью доменного уровня. Чуть позже мы его определим

Метод ApplyDiscountFor применяет скидку (если таковая имеется) на основе роли пользователя и возвращает экземпляр класса DiscountedProduct

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — С. 96.

Рисунок 3.8 иллюстрирует взаимосвязь между `ProductService` и его зависимостями.

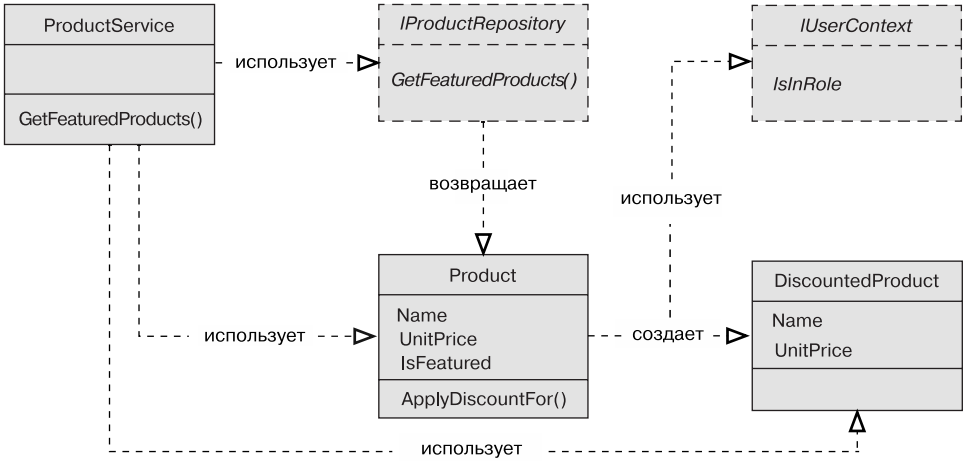


Рис. 3.8. ProductService и его зависимости

Метод `GetFeaturedProducts` класса `ProductService` должен использовать экземпляр `IProductRepository`, чтобы извлечь список предлагаемых товаров, применить скидки и возвратить список экземпляров `DiscountedProduct`. Класс `ProductService` соответствует классу с таким же именем, созданному Мэри, но теперь он является чистым классом доменной модели, поскольку в нем нет жестко закодированной ссылки на уровень доступа к данным. В листинге 3.9 показано, что, как и в случае с нашим `HomeController`, мы снова собираемся отказаться от контроля над его нестабильными зависимостями путем внедрения через конструктор.

Листинг 3.9. ProductService с внедрением через конструктор

```
public class ProductService : IProductService
{
    private readonly IProductRepository repository;
    private readonly IUserContext userContext;

    public ProductService(
        IProductRepository repository,
        IUserContext userContext)
    {
        if (repository == null)
            throw new ArgumentNullException("repository");
        if (userContext == null)
            throw new ArgumentNullException("userContext");

        this.repository = repository;
        this.userContext = userContext;
    }

    public IEnumerable<DiscountedProduct> GetFeaturedProducts()
```

Внедрение через конструктор

```
    {
        return
            from product in this.repository
                .GetFeaturedProducts()
            select product
                .ApplyDiscountFor(this.userContext);
    }
}
```

Зависимости repository и userContext извлекают список товаров и применяют скидку к каждому предлагаемому товару

Предоставление зависимости userContext методу ApplyDiscountFor с использованием внедрения через метод

Кроме `IProductRepository`, конструктору `ProductService` требуется экземпляр `IUserContext`:

```
public interface IUserContext
{
    bool IsInRole(Role role);
}

public enum Role { PreferredCustomer }
```

Это еще одно отступление от созданной Мэри реализации, где в качестве аргумента метода `GetFeaturedProducts` бралось только булево значение, показывающее, является ли пользователь постоянным покупателем. Поскольку решение об этом является частью доменного уровня, более корректно будет явно смоделировать это в виде зависимости.

Кроме этого, информация о пользователе, от имени которого выполняется запрос, является контекстно зависимой. Не хотелось бы, чтобы за сбор этой информации отвечал каждый контроллер. Это было бы сопряжено с ростом повторяемости и шансов на ошибки, а также привело бы к случайным просчетам в обеспечении безопасности.

Вместо того чтобы разрешить уровню пользовательского интерфейса предоставлять эту информацию доменному уровню, мы позволим, чтобы ее извлечение стало особенностью реализации `ProductService`. Интерфейс `IUserContext` «дает добро» `ProductService` на извлечение информации о текущем пользователе без необходимости ее предоставления со стороны `HomeController`. Контроллеру `HomeController` не нужно знать, для какой роли (или для каких ролей) разрешены цены со скидками, и `HomeController` не может случайно разрешить скидку, передав, к примеру, `true` вместо `false`. Тем самым происходит общее упрощение уровня пользовательского интерфейса.

СОВЕТ

Чтобы добиться общего упрощения системы, данные, нужные во время выполнения приложения, описывающие контекстно зависимую информацию, лучше скрыть за абстракцией и внедрить в потребителя, которому они необходимы для функционирования. Контекстная информация является метаданными о текущем запросе. Обычно это информация, на которую пользователю не должно разрешаться влиять напрямую. В качестве примеров можно привести личность пользователя (установленную при входе в систему) и текущее время системы.

Хотя в библиотеку .NET Base Class Library (BCL) включен интерфейс `IPrincipal`, представляющий собой стандартный способ моделирования пользователей приложений, по своей природе он универсален и не адаптирован под особые потребности нашего приложения. Вместо него приложению будет позволено определить абстракцию.

Метод `ProductService.GetFeaturedProducts` передает зависимость `IUserContext` методу `Product.ApplyDiscountFor`. Этот прием известен как внедрение через метод (Method Injection). Он особенно полезен в случаях, когда недолговечные объекты вроде сущностей (например, сущность `Product`, как в нашем случае) нуждаются в зависимостях. Хотя детали могут отличаться, основной прием остается все тем же. Более подробно соответствующий паттерн будет рассмотрен в главе 4. На данном этапе приложение совершенно неработоспособно. Причина в том, что осталось решить еще три проблемы.

- Отсутствует конкретная реализация `IProductRepository`. Эта проблема решается довольно легко. В следующем разделе будет показана реализация конкретного класса `SqlProductRepository` для считывания предлагаемых товаров из базы данных.
- Отсутствует конкретная реализация `IUserContext`. Она также будет показана в следующем разделе.
- Среда MVC не знает, какой конкретный тип нужно использовать. Дело в том, что в конструктор `HomeController` был введен абстрактный параметр типа `IProductService`. Эта проблема может быть решена различными способами, но для нас предпочтительнее разработать собственный активатор `Microsoft.AspNetCore.Mvc.Controllers.IControllerActivator`. Как это делается, не относится к теме данной книги, но мы коснемся этого вопроса в главе 7. Достаточно сказать, что это будет специализированная фабрика, которая создаст экземпляр конкретного `ProductService` и предоставит его конструктору `HomeController`.

На доменном уровне работа ведется только с теми типами, которые определены внутри доменного уровня, и со стабильными зависимостями библиотеки .NET BCL. Понятия доменного уровня реализованы в виде POCO-объектов. На данной стадии имеется только одно понятие — `Product`. Доменный уровень должен иметь возможность обмена данными с внешним миром (например, с базами данных). Эта потребность смоделирована в виде абстракций (например, в виде хранилищ), которые, прежде чем доменный уровень станет полезен, нам нужно заменить конкретными реализациями. На рис. 3.9 показано текущее состояние реализации архитектуры, представленной на рис. 3.2.

Нам удалось выполнить компиляцию доменной модели. Это означает, что нами создана доменная модель, не зависящая от уровня доступа к данным, который все еще предстоит создать. Но, прежде чем перейти к его созданию, хотелось бы более подробно объяснить ряд моментов.

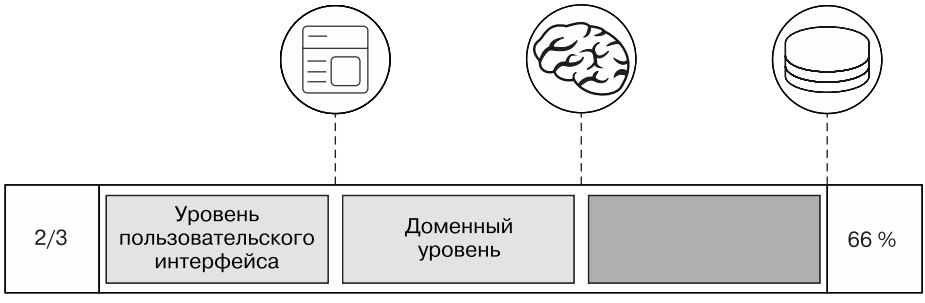


Рис. 3.9. Уровень пользовательского интерфейса и доменный уровень теперь на месте, а уровень доступа к данным еще предстоит реализовать

Принцип инверсии зависимостей

Многое из того, что мы пытаемся сделать с помощью внедрения зависимостей, имеет отношение к принципу инверсии зависимостей (Dependency Inversion Principle)[1]. Этот принцип гласит, что имеющиеся в приложениях модули более высокого уровня не должны зависеть от модулей более низкого уровня; вместо этого модули обоих уровней должны зависеть от абстракций.

Именно это мы и сделали при определении `IProductRepository`. Компонент `ProductService` является частью модуля, относящегося к более высокому доменному уровню, а реализация `IProductRepository` — назовем ее `SqlProductRepository` — является частью модуля, относящегося к более низкому уровню доступа к данным. Вместо того чтобы позволить `ProductService` зависеть от `SqlProductRepository`, мы разрешили как `ProductService`, так и `SqlProductRepository` зависеть от абстракции `IProductRepository`. Абстракция реализована в `SqlProductRepository`, а `ProductService` ее использует. Эта ситуация проиллюстрирована на рис. 3.10.

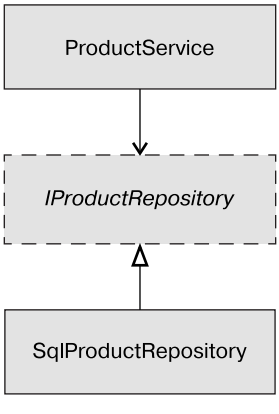


Рис. 3.10. Вместо ProductService, зависящего от SqlProductRepository, от абстракции зависят оба класса

Взаимосвязь принципа инверсии зависимостей и DI заключается в том, что этот принцип предписывает то, чего хотелось бы достичь, а DI точно определяет, как бы нам следовало этого добиться. Принцип не дает описания, как именно потребитель должен получить доступ к своим зависимостям. Однако многие разработчики не знают еще об одной интересной части принципа инверсии зависимостей.

Принцип не только предписывает слабую связанность, но и утверждает, что абстракциями должен владеть модуль, использующий абстракцию. В этом контексте «владение» означает, что потребляющий модуль контролирует форму абстракции и она распространяется с этим модулем, а не с тем модулем, который ее реализует. Потребляющий модуль должен иметь возможность определять аб-

[1] *Martin R.* Agile Principles, Patterns and Practices in C#. — Pearson Education, 2007.

стракцию так, чтобы принести наибольшую пользу самому себе.

Вы уже видели, как это было проделано нами дважды: и `IUserContext`, и `IProductRepository` именно так и определены. Они сконструированы так, чтобы подходить доменному уровню наилучшим образом, даже притом, что, как показано на рис. 3.11, их реализации возложены на уровень пользовательского интерфейса и на уровень доступа к данным соответственно.

Если дать вышестоящему модулю или уровню возможность определять свои собственные абстракции, это не только избавит его от необходимости иметь зависимость от нижестоящего модуля, но позволит упростить этот вышестоящий модуль, поскольку абстракции адаптированы под его нужды. Это возвращает нас к интерфейсу `IPrincipal` библиотеки BCL.

Согласно ранее приведенному описанию, `IPrincipal` универсален. Принцип инверсии зависимостей, напротив, подводит нас к определению абстракций, адаптированных под специальные нужды нашего приложения. Именно поэтому вместо того, чтобы позволить доменному уровню зависеть от `IPrincipal`, мы определяем свою собственную абстракцию `IUserContext`. Но это означает, что нам придется создать реализацию адаптера, позволяющую выполнять преобразование обращения от этой адаптированной под приложение абстракции `IUserContext` в обращения к среде выполнения приложения.

Если принцип инверсии зависимостей диктует обязательное распространение абстракций с их собственными модулями, не нарушает ли интерфейс доменного уровня `IProductService` этот принцип? К тому же `IProductService` потребляется уровнем пользовательского интерфейса, но реализован, как показано на рис. 3.12, доменным уровнем. Да, это нарушение принципа инверсии зависимостей.

Если бы мы были заинтересованы в исправлении этого нарушения, то нужно было бы убрать `IProductService` из доменного уровня. Но перемещение `IProductService` на уровень пользовательского интерфейса сделает наш доменный уровень зависимым от него. Поскольку доменный уровень

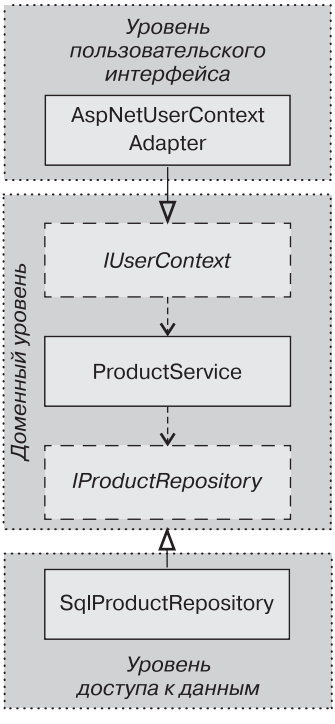


Рис. 3.11. И IUserContext, и IProductRepository являются частями доменного уровня, поскольку ими «владеет» ProductService

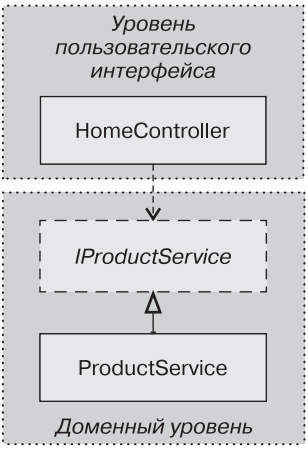


Рис. 3.12. Сделав IProductService частью доменного уровня, мы нарушили принцип инверсии зависимостей

является центральной частью приложения, нам не хотелось бы, чтобы он зависел от чего-либо еще. Кроме того, эта зависимость не позволила бы впоследствии заменить уровень пользовательского интерфейса.

Все это означает, что для устранения нарушения в нашем решении нужны еще два дополнительных проекта: один для изолированного уровня пользовательского интерфейса без точки входа (Composition Root), а другой — для абстракции `IProductService`, которой владеет уровень пользовательского интерфейса. Но сугубо из прагматичных соображений для данного примера мы решили не следовать по этому маршруту, оставив нарушение в покое. Надеемся, что вы оцените наше стремление ничего не усложнять.

Интерфейсы или абстрактные классы?

Во многих руководствах по объектно-ориентированному проектированию основной упор делается на интерфейсы как на главный механизм абстракции, а в руководстве по проектированию в среде .NET предпочтение отдается не интерфейсам, а абстрактным классам[1]. Так чем же нужно пользоваться, интерфейсами или абстрактными классами? Что касается технологии DI, то вполне допустимо будет ответить, что это не имеет значения. Главное, что программирование ведется на основании некой абстракции.

Выбор между интерфейсами и абстрактными классами важен в других контекстах, но не здесь. Вы заметите, что мы используем эти понятия взаимозаменяемо, и чтобы охватить в повествовании как интерфейсы, так и абстрактные классы, мы часто обращаемся к понятию абстракции. Это не означает, что у нас, как у авторов, нет среди них предпочтений. На самом деле есть. Когда дело доходит до создания приложений, мы, как правило, предпочитаем пользоваться интерфейсами, а не абстрактными классами, руководствуясь следующими соображениями.

- *Абстрактные классы могут быть легко использованы в качестве базовых классов*, а те, в свою очередь, могут легко превратиться в постоянно меняющиеся и неизменно разрастающиеся всемогущие объекты (God Objects)[2]. Все производные имеют сильную связанность со своим базовым классом, что может стать проблемой, когда у базового класса имеется изменчивое поведение. А интерфейсы принуждают нас твердить мантру «Композиция превыше наследования» (Composition over Inheritance)[3].

---

1 *Cwalina K., Abrams B.* Framework Design Guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries, 2nd Ed. — Addison-Wesley, 2009. — P. 88–95.

2 Под всемогущим объектом (God Object) понимается объект, обладающий чрезмерной осведомленностью или функциональностью и являющийся антипаттерном.

3 Принцип «Композиция превыше наследования» гласит, что в объектно-ориентированном программировании в классах должно быть реализовано полиморфное поведение и повторное использование кода за счет заключения в контейнеры экземпляров другого класса, реализующих требуемую функциональность (композицию), а не за счет наследования из базового или родительского класса.

- *В конкретных классах могут быть реализованы несколько интерфейсов, тогда как в среде .NET они могут быть производными только от одного базового класса.* Использование интерфейсов в качестве средств абстракции представляется более гибким приемом.
- *Определения интерфейсов в C# менее громоздки по сравнению с определениями абстрактных классов.* Пользуясь интерфейсами, можно опускать ключевые слова `abstract` и `public` из определений их компонентов. Это придает определению интерфейса более лаконичную форму.

Однако при создании многократно используемых библиотек вопрос приобретает не столь однозначную оценку, поскольку здесь уже приходится иметь дело с обратной совместимостью. В этом свете абстрактные классы могут иметь больше смысла, поскольку неабстрактные компоненты могут быть добавлены позже, а вот добавление компонентов к интерфейсу является критическим изменением. Именно поэтому в руководстве по проектированию в среде .NET предпочтение отдается абстрактным классам.

Многократно используемые библиотеки

Обычно в экосистеме .NET многократно используемые библиотеки распространяются через NuGet. Важной особенностью является то, что на момент компиляции их клиенты неизвестны. Это отличается от проекта, который повторно применяется другими проектами в одном и том же решении Visual Studio. Хотя ваши Visual Studio-решения могут содержать проекты, которые в одном и том же решении повторно используются сразу несколькими проектами, такие проекты не считаются многократно используемыми библиотеками. К примеру, проект доменного уровня может повторно использоваться сразу несколькими проектами, но это все равно не превращает его в многократно используемую библиотеку.

Внешние библиотеки изменять сложнее, поскольку у них могут иметься тысячи потребляющих их кодовых баз, и ни к одной из них разработчик библиотеки не имеет доступа. Такая многократно используемая библиотека не может быть протестирована в сопоставлении с потребляющими ее кодовыми базами.

Теперь перейдем к уровню доступа к данным и создадим реализацию для ранее определенного интерфейса `IProductRepository`.

3.1.3. Создание нового уровня доступа к данным

Мы, как и Мэри, собираемся реализовать наш уровень доступа к данным, воспользовавшись средством Entity Framework Core, поэтому для создания модели сущности последуем ее же путем, рассмотренным в главе 2. Основное отличие будет в том, что теперь `CommerceContext` является лишь составной частью реализации уровня доступа к данным, а не всем этим уровнем.

В этой модели ничего, что находится за пределами уровня доступа к данным, не будет обладать осведомленностью об используемом средстве или зависимостью от Entity Framework. Данное средство может быть заменено без каких-либо восходящих по структуре приложения эффектов. Придерживаясь данного замысла, мы можем создать реализацию `IProductRepository` (листинг 3.10).

Листинг 3.10. Реализация IProductRepository с использованием Entity Framework Core

```
public class SqlProductRepository : IProductRepository
{
    private readonly CommerceContext context;

    public SqlProductRepository(CommerceContext context)
    {
        if (context == null) throw new ArgumentNullException("context");

        this.context = context;
    }

    public IEnumerable<Product> GetFeaturedProducts()
    {
        return
            from product in this.context.Products
            where product.IsFeatured
            select product;
    }
}
```

В созданном Мэри приложении сущность `Product`, несмотря на то что была определена на уровне доступа к данным, использовалась также в качестве доменного объекта. Теперь ситуация изменилась. Класс `Product` определен на нашем доменном уровне. А наш уровень доступа к данным имеет возможность воспользоваться классом `Product` из этого уровня.

Чтобы не усложнять конструкцию, мы разрешили уровню доступа к данным не определять его собственную реализацию, а воспользоваться нашим доменным объектом. Такая возможность предоставилась нам благодаря тому, что Entity Framework Core позволяет создавать сущности, не зависящие от конкретной базы данных (persistence ignorant)[1]. Насколько эта практика разумна, во многом зависит от структуры и сложности ваших доменных объектов. Если позже будет сделан вывод, что эта общая модель накладывает нежелательные ограничения на нашу модель, мы можем внести в наш уровень доступа к данным изменения путем ввода внутренних объектов постоянного хранения, не затрагивая все остальное приложение. В этом случае для преобразования таких внутренних объектов постоянного хранения в доменные объекты нам понадобится уровень доступа к данным.

В предыдущей главе рассматривались проблемы, вызванные подразумеваемой зависимостью созданного Мэри класса `CommerceContext` от строки подклю-

[1] Понятие persistence ignorance означает, что сущности являются обычными POCO-объектами, не зависящими от каких-либо сред поддержки постоянного хранения данных.

чения. Наш новый класс `CommerceContext` превратит эту зависимость в явную, что станет еще одним отклонением от реализации, созданной Мэри. Новый класс `CommerceContext` показан в листинге 3.11.

Листинг 3.11. Более рациональная реализация класса CommerceContext

```
public class CommerceContext : DbContext
{
    private readonly string connectionString;

    public CommerceContext(string connectionString)
    {
        if (string.IsNullOrWhiteSpace(connectionString))
            throw new ArgumentException(
                "connectionString should not be empty.",
                "connectionString");

        this.connectionString = connectionString;

    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder builder)
    {
        builder.UseSqlServer(this.connectionString);
    }
}
```

Использование внедрения через конструктор в отношении требуемых зависимостей, в данном случае connectionString

Зависимость сохраняется и применяется позже в методе OnConfiguring с целью настройки CommerceContext для использования

Это почти подводит нас к завершению переделки приложения электронной торговли. Остается только решить вопрос с реализацией `IUserContext`.

3.1.4. Реализация адаптера IUserContext, учитывающего специфику среды ASP.NET Core

Последним белым пятном осталась конкретная реализация `IUserContext`. В веб-приложениях информация о пользователе, выдавшем запрос, обычно транслируется на сервер с каждым запросом. Эта информация передается с использованием cookie-файлов или HTTP-заголовков. Способ извлечения идентификационной информации о текущем пользователе во многом зависит от используемой среды выполнения. Это значит, что при создании приложения ASP.NET Core нам понадобится совершенно иная реализация, по сравнению, скажем, со службой Windows.

В реализации нашего `IUserContext` учитывается специфика среды выполнения. Нам нужно, чтобы о среде выполнения приложения ничего не было известно ни доменному уровню, ни уровню доступа к данным. В противном случае мы лишились бы возможности использования этих уровней в другом контексте. Стало быть, реализация адаптера должна быть где-то в ином месте. Поэтому идеальным местом для реализации `IUserContext` представляется уровень пользовательского интерфейса.

Возможная реализация `IUserContext` для приложения ASP.NET Core отображена в листинге 3.12.

Листинг 3.12. Реализация IUserContext для ASP.NET Core

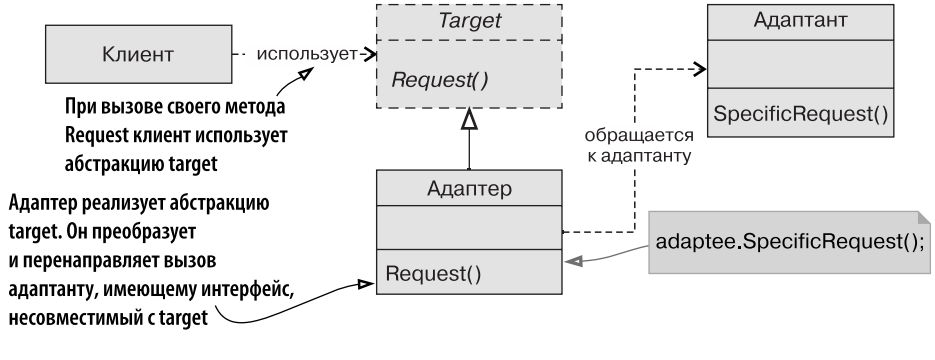
```
public class AspNetUserContextAdapter : IUserContext
{
    private static HttpContextAccessor Accessor = new HttpContextAccessor();

    public bool IsInRole(Role role)
    {
        return Accessor.HttpContext.User.IsInRole(role.ToString());
    }
}
```

Для работы `AspNetUserContextAdapter` требуется `HttpContextAccessor`, компонент, заданный средой ASP.NET Core и допускающий обращение к `HttpContext` текущего запроса, подобно аналогичной возможности использования `HttpContext.Current` в «классической» среде ASP.NET. Объект `HttpContext` применяется для доступа к информации запроса, касающейся текущего пользователя.

`AspNetUserContextAdapter` адаптирует нашу характерную для приложения абстракцию `IUserContext` под API среды ASP.NET Core. Этот класс является реализацией паттерна проектирования «Адаптер» (Adapter), рассмотренного в главе 1[1].

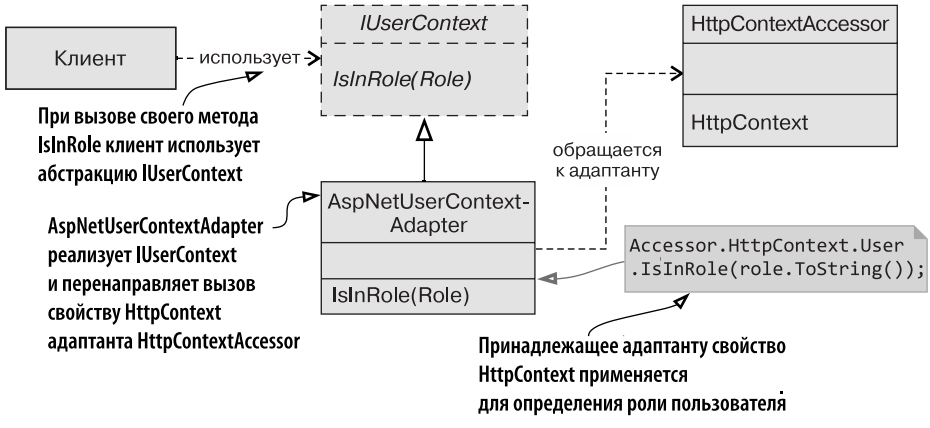
Паттерн проектирования «Адаптер»

Напомним, что паттерн проектирования «Адаптер» относится к категории структурных паттернов. Эта категория касается порядка компоновки классов и объектов с целью формирования более крупных структур. Другими паттернами этой категории являются «Компоновщик» (Composite), «Декоратор» (Decorator), «Фасад» (Facade) и «Заместитель» (Proxy). Как и адаптер для электроприборов, паттерн проектирования «Адаптер» выполняет преобразование интерфейса в ожидаемый клиентами формат. Это позволяет классам (или вилкам и розеткам) работать вместе, чего в противном случае не получилось бы из-за несовместимости их интерфейсов.



[1] Для работы адаптера ему нужна регистрация HttpContextAccessor в принадлежащем ASP.NET Core классе IServiceCollection. Эта особенность будет показана в следующей главе, в листинге 4.3.

Обычно реализации паттерна «Адаптер» довольно просты, но не стоит удивляться, если в «Адаптере» будут содержаться и сложные преобразования. Замысел заключается в том, чтобы эта сложность была скрыта от клиента.



В листинге 3.12 можно увидеть, что адаптеру `AspNetUserContextAdapter` в качестве надстройки над обращением к адаптанту приходится выполнять некоторую дополнительную работу. Благодаря этому удается упростить клиентский код, что освобождает клиента от необходимости зависеть от `HttpContext`.

С реализацией `AspNetUserContextAdapter` наша переделка приложения электронной торговли завершается. Теперь мы вплотную подошли к корню композиции (Composition Root).

3.1.5. Компоновка приложения в корне композиции

После реализации `ProductService`, `SqlProductRepository` и `AspNetUserContextAdapter` можно настроить среду ASP.NET Core MVC для создания экземпляра `HomeController`, подпитывающегося данными от экземпляра `ProductService`, который сам создан с использованием `SqlProductRepository` и `AspNetUserContextAdapter`. В конечном итоге получается граф объектов, имеющий следующий вид (листинг 3.13).

Листинг 3.13. Граф объектов приложения

```
new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        new AspNetUserContextAdapter()));
```

ОПРЕДЕЛЕНИЕ

> В объектно-ориентированном приложении группы объектов формируют сеть из своих взаимоотношений друг с другом либо с помощью прямой ссылки на другой объект, либо через цепочку промежуточных ссылок. Эти группы объектов называются графами объектов (object graphs).

Более подробно способы подключения конструкций таких графов объектов к среде ASP.NET Core мы рассмотрим в главе 7, поэтому здесь они показаны не будут. Теперь, когда все уже корректно сведено в единую систему, можно перейти на главную страницу и получить картинку, показанную на рис. 3.13.

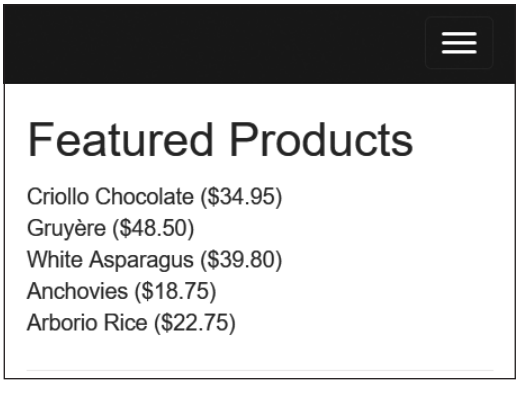


Рис. 3.13. Снимок экрана завершенного приложения

3.2. Анализ реализации со слабой связанностью

Предыдущий раздел был полон подробностей, поэтому вряд ли будет странным, если вы упустили из вида общую картину происходящего. В данном разделе мы постараемся объяснить все случившееся в более широком плане.

3.2.1. Осмысление взаимодействия компонентов

Классы на каждом уровне взаимодействуют друг с другом либо напрямую, либо в абстрактной форме. При этом они переходят границы модуля, поэтому отследить порядок их взаимодействия порой непросто. На рис. 3.14 показано взаимодействие различных зависимостей и предоставлен более подробный обзор первоначального плана, показанного на рис. 3.4.

Когда приложение запускается, код в классе `Startup` создает новый активатор контроллера покупателя и отыскивает строку подключения из конфигурационного файла приложения. Когда поступает запрос на страницу, приложение вызывает в отношении активатора контроллера метод `Create`.

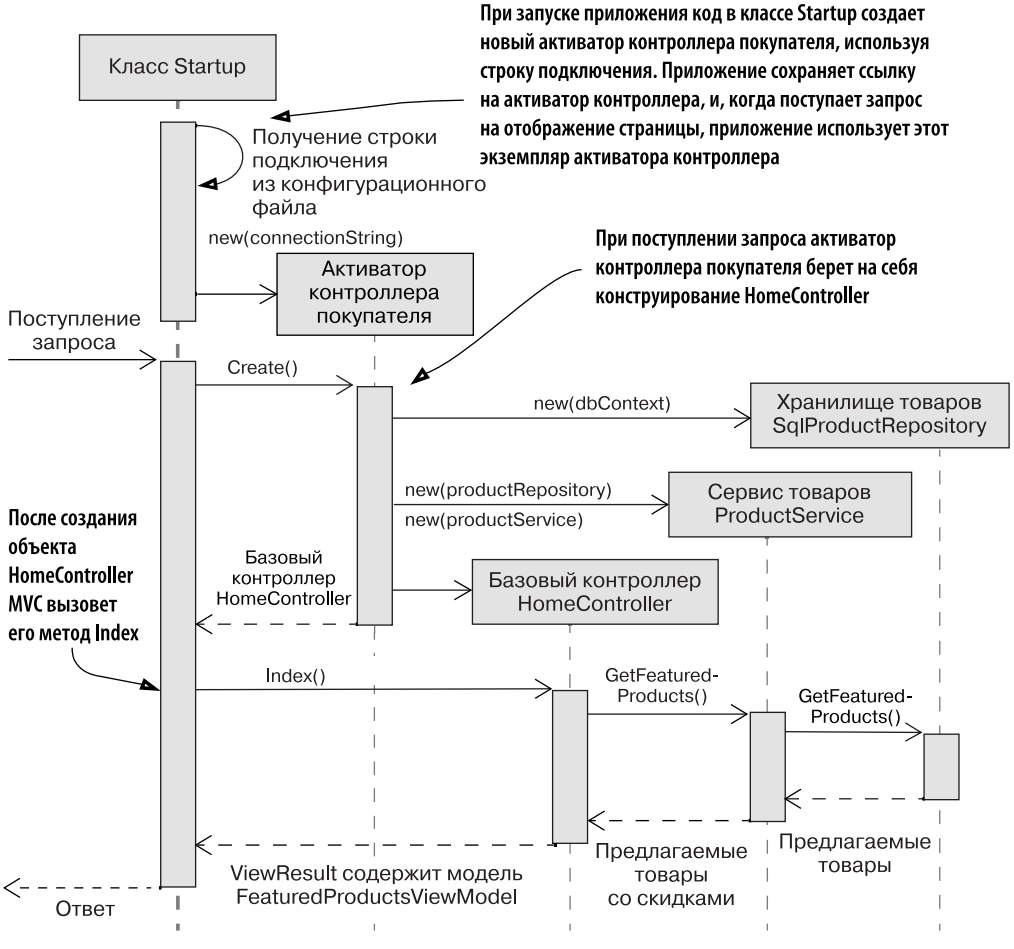


Рис. 3.14. Взаимодействие между элементами, вовлеченными в реализацию технологии DI в приложении электронной торговли

Активатор предоставляет новому экземпляру `CommerceContext` (не показанному на схеме) сохраненную строку подключения. Он вводит `CommerceContext` в новый экземпляр `SqlProductRepository`. В свою очередь, экземпляр `SqlProductRepository` вместе с экземпляром `AspNetUserContextAdapter` (не показанным на схеме) внедрен в новый экземпляр `ProductService`. Точно так же `ProductService` внедрен в новый экземпляр `HomeController`, который после этого возвращен из метода `Create`.

Затем среда ASP.NET Core MVC вызывает в отношении экземпляра `HomeController` метод `Index`, заставляя его вызвать в отношении экземпляра `ProductRepository` метод `GetFeaturedProducts`. Это приводит к вызову в отношении экземпляра `SqlProductRepository` метода `GetFeaturedProducts`. И наконец, возвращается `ViewResult` с наполненной моделью преставления предлагаемых товаров `FeaturedProductsViewModel`, а MVC находит и отображает корректное представление.

3.2.2. Анализ новой схемы зависимостей

В разделе 2.2 вы видели, как схема зависимостей может помочь в анализе и осмыслении степени гибкости, предоставляемой архитектурной реализацией. Изменила ли технология DI схему зависимостей приложения?

На рис. 3.15 показано, что схема зависимостей действительно изменилась. У доменной модели теперь нет никаких зависимостей, и она может действовать как автономный модуль. С другой стороны, теперь появилась зависимость у уровня доступа к данным, а в приложении, созданном Мэри, ее не было.

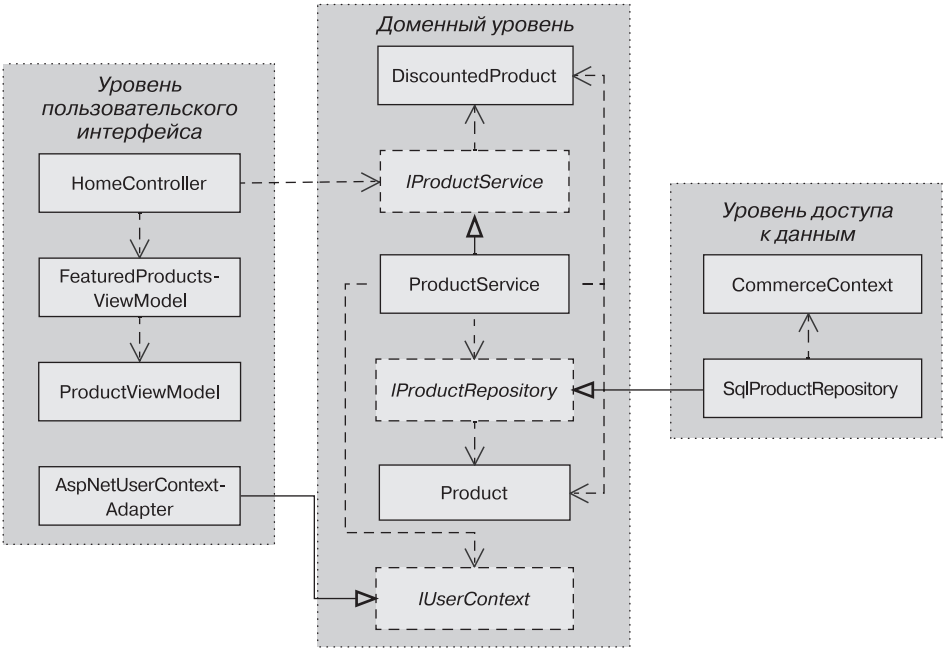


Рис. 3.15. Схема зависимостей, показывающая учебное приложение электронной торговли с применением технологии DI. Показаны все классы и интерфейсы, а также их взаимоотношения

Самое важное, что следует отметить на рис. 3.15, заключается в том, что у доменного уровня больше нет никаких зависимостей. Это должно придать нам еще больше уверенности на то, что теперь мы в состоянии дать положительные ответы на исходные вопросы о компонуемости (см. раздел 2.2).

- Можно ли заменить пользовательский интерфейс на основе веб-технологий интерфейсом на WPF-основе? Такая возможность была до переделки и по-прежнему имеется в новой конструкции. От пользовательского интерфейса на основе веб-технологий не зависит ни библиотека доменной модели, ни библиотека доступа к данным, поэтому на его место можно поставить что-нибудь другое.
- Можно ли заменить уровень доступа к реляционным данным тем, что работает с Azure Table Service? В следующей главе будет дано описание, как приложение

обнаруживает `IProductRepository` и создает его корректный экземпляр, поэтому примем пока за чистую монету следующее: уровень доступа к данным загружается с помощью позднего связывания и название типа определяется как настройка приложения в его конфигурационном файле. Можно отказаться от текущего уровня доступа к данным и вставить новый уровень при условии, что в нем также предоставляется реализация `IProductRepository`.

О контейнерах внедрения зависимостей

DI-контейнер представляет собой программную библиотеку, предоставляющую DI-функциональность и автоматизирующую многие задачи, привлекаемые к компоновке объектов, перехвату и управлению временем жизни. DI-контейнеры также известны как контейнеры инверсии управления (Inversion of Control, IoC). До этого момента мы почти не затрагивали тему DI-контейнеров. Это делалось намеренно, поскольку, как уже объяснялось в главе 1, DI-контейнер полезный, но вспомогательный инструмент. Отложим подробное рассмотрение DI-контейнеров до части IV, поскольку, по нашему мнению, обучение читателей набору принципов и паттернов, из которых состоит технология DI, а также умению различать проблемный код и антипаттерны намного важнее.

Мы создаем приложения как с DI-контейнерами, так и без них, и у вас также должна быть такая возможность. И все же мы считаем, что было бы контрпродуктивно начинать с использования DI-контейнера без того багажа знаний, который предоставляется во II и III частях данной книги. С другой стороны, после освоения принципов и практических приемов использование DI-контейнера преимущественно будет состоять из знакомства с его API. А на данный момент важно лишь получить общее представление о том, что такое DI-контейнер и как он может помочь в решении ваших задач.

Когда вышло первое издание этой книги, мы использовали DI-контейнеры буквально во всех создаваемых нами приложениях. Хотя мы знали о возможности применения технологии DI без DI-контейнера, мы полагали, что оно никогда не войдет в нашу практику. Наши представления по данному вопросу изменились, именно поэтому сейчас мы еще больше сконцентрируемся на паттернах и технологиях, относящихся к DI.

Несмотря на необходимость обращения к инфраструктуре приложения, само по себе это обращение не добавит ему деловой ценности; иногда разумнее воспользоваться библиотекой общего назначения. Это ничем не отличается от реализации регистрации данных или от доступа к ним. Регистрация данных приложения относится к разряду задач, с которыми лучше справится библиотека регистрации общего назначения. То же самое справедливо и для составления графов объектов. В части IV книги вам предстоит углубленное рассмотрение вопроса, когда DI-контейнер может быть полезен, а когда — нет.

Не нужно ожидать от DI-контейнера магического превращения кода с сильной связанностью в код со слабой связанностью. DI-контейнер может повысить сопровождаемость вашей точки входа (корня композиции), но, чтобы приложение стало сопровождаемым, оно в первую очередь должно быть спроектировано с прицелом на использование DI-паттернов и технологий. Применение DI-контейнера не гарантирует правильного использования DI и не требует этого.

В примере приложения электронной торговли, рассматриваемом в этой главе, представлен весьма ограниченный уровень сложности: в сценарии, рассчитанном только на чтение данных, имеется одно-единственное хранилище. До сих пор мы старались, чтобы приложение было как можно проще и меньше, чтобы постепенно подвести вас к усвоению основных понятий и принципов. Поскольку одна из основных целей применения DI — управление сложностью, то, чтобы полностью оценить эффективность этой технологии, нам нужно сложное приложение. В ходе изложения материала книги мы будем расширять пример приложения электронной торговли, чтобы в полной мере продемонстрировать различные аспекты технологии DI.

Неужели DI не позволяет получить общее представление?

Разработчики, начинающие применять технологию DI, часто жалуются, что они перестают видеть структуру приложения, поскольку не сразу понятно, что чем вызывается. Хотя то, что с применением DI мы переместили эти сведения из отдельно взятых классов, — абсолютная правда, код листинга 3.13 доказывает, что мы вообще-то не должны терять эту информацию. В листинге 3.13 показан пример чистой технологии DI. Когда соблюдаются нормы чистого внедрения зависимостей, точка входа обычно содержит эту информацию последовательным образом. А что еще лучше, она предоставляет вам вид на весь граф объектов, а не только вид на непосредственные зависимости класса, получаемый при применении кода с сильной связанностью.

А вот переход от чистой технологии DI к использованию DI-контейнера может привести к тому, что этот обзор будет потерян. Причина в том, что в DI-контейнерах для построения графа объектов во время выполнения программы используется отображение, а не указание на построение этого графа во время компиляции с использованием языка программирования[1]. Но, когда приложение спроектировано грамотно, обнаруживается, что этот изъян становится менее чувствительным[2]. Мы поняли, что количество требуемых переходов от класса к его зависимостям и обратно уменьшилось, а удобство сопровождения приложения повысилось.

И все же разницу между чистой технологией DI и DI-контейнером нужно учитывать, поскольку она может повлиять на ваши предпочтения в выборе одной из этих технологий. Более подробно вопрос о том, когда следует воспользоваться DI-контейнером, а когда остановить свой выбор на чистой технологии DI, будет рассмотрен в разделе 12.3.

Этой главой завершается часть I книги, целью которой было ознакомление с технологией DI и получение о ней общего представления. В этой главе были по-

---

1 Некоторые DI-контейнеры позволяют проводить визуализацию графа объектов, но это действие относится только ко времени выполнения программы, а не к просмотру кода.

2 «Проектирование с высоким качеством» — понятие субъективное, но следование SOLID-принципам доказало, что является важным инструментом в создании качественно спроектированных приложений. Акроним SOLID будет разобран в главе 10.

казаны примеры внедрения через конструктор. В качестве паттернов, имеющих отношение к DI, были представлены внедрение через метод и точка входа приложения. В следующей главе вам предстоит погружение в эти и другие паттерны проектирования.

Резюме

- Реструктуризация существующих приложений для повышения возможностей их сопровождения и получения конструкции с меньшей степенью связанности дается нелегко. С другой стороны, полная переделка зачастую сопряжен с повышенным риском и более высокими затратами.
- Использование моделей представления может упростить представление, поскольку входящие данные оформляются специально для просмотра.
- Ввиду того что представления хуже поддаются тестированию, чем меньше в нем объем обработки данных, тем лучше. Это также упрощает задачу разработчику пользовательского интерфейса, работающему над представлением.
- Ограничение объема нестабильных зависимостей на доменном уровне позволяет получить более низкую связанность и более высокую степень вторичного использования и тестируемости кода.
- Применение при разработке приложения подхода «от потребителя» способствует более быстрому созданию прототипов, что может сократить цикл получения от него обратной связи в виде замечаний и предложений.
- Когда требуется повысить степень модульности приложения, следует применить паттерн внедрения через конструктор и построить граф объектов в корне композиции, расположенном в непосредственной близости к точке входа приложения.
- Программирование на основе интерфейсов является краеугольным камнем технологии DI. Оно позволяет заменять, имитировать и перехватывать зависимости, не вынуждая вносить изменения в их потребители. Когда реализация и абстракция помещаются в разные сборки, появляется возможность полной замены библиотек.
- Программирование на основе интерфейсов не означает, что во всех классах должен реализовываться интерфейс. Объекты с небольшой продолжительностью жизни, такие как сущности, модели представлений и DTO-объекты, обычно не содержат поведения, требующего имитации, перехвата, декорирования или замены.
- Что касается технологии DI, то совершенно неважно, что именно используется — интерфейсы или абстрактные классы. С точки зрения общей разработки мы, авторы, обычно отдаем предпочтение интерфейсам, а не абстрактным классам.
- Многократно используемой считается библиотека, имеющая клиентов, о которых ничего не известно на момент компиляции. Многократно используемые

библиотеки обычно поставляются через NuGet. Библиотеки, вызываемые кодом в рамках одного и того же решения (Visual Studio), не считаются многократно используемыми.

- Технология DI тесно связана с принципом инверсии зависимости. Этим принципом предусматривается, что программировать нужно на основе интерфейсов и что уровень должен контролировать используемые в нем интерфейсы.
- Использование DI-контейнера может посодействовать повышению возможностей сопровождения корня композиции (точки входа) приложения, но от него не стоит ждать волшебного превращения кода с сильной связанностью в код со слабой связанностью. Чтобы приложение стало сопровождаемым, оно должно быть разработано на основе DI-паттернов и технологий.

Часть II
Каталог

В части I был дан обзор технологии DI и рассмотрены цели и преимущества ее применения. В главе 3 приводился весьма объемный пример, но мы уверены, что после прочтения первых глав у читателей все же остался круг нерешенных вопросов. Во части II мы намерены углубиться в рассматриваемую тему и ответить на некоторые из этих вопросов.

Судя по названию, в части II будут представлены полный каталог паттернов, антипаттернов и примеры проблемного кода. Некоторым не нравятся паттерны проектирования, они находят их слишком скучными или абстрактными. Нам же нравятся паттерны, поскольку они дают нам язык высокого уровня, позволяющий добиться эффективности и лаконичности кода при обсуждении конструкций программных средств. Мы намерены воспользоваться этим каталогом с целью предоставления языка паттернов для реализации технологии DI. Хотя описания паттернов должны содержать некоторые обобщения, мы конкретизировали применение каждого паттерна, воспользовавшись примерами. Все три главы можно изучать поочередно, но каждая из тем каталога изложена так, что их можно читать по отдельности.

В главе 4 содержится мини-каталог паттернов проектирования DI. Отчасти из этих паттернов можно составить нормативное руководство по способам реализации DI, но следует иметь в виду, что мы не считаем, что они сопоставимы по важности. Безусловно, самыми значимыми паттернами проектирования являются внедрение через конструктор (Constructor Injection) и корень композиции (Composition Root), а все остальные паттерны следует рассматривать как дополнительные, которые можно применять при особых обстоятельствах.

В главе 4 приводится целый набор обобщенных решений, а вот в главе 5 содержится каталог, составленный из ситуаций, которых следует избегать. Эти антипаттерны описывают распространенные, но неверные способы решения типовых задач DI. В каждом случае в антипаттерне дается описание его идентификации и способов устранения проблемы. Чтобы не попадать в их ловушку, антипаттерны важно знать и уметь распознавать, и точно так же, как в главе 4, представлены два наиболее важных паттерна, самым важным антипаттерном следует назвать «Локатор сервисов» (Service Locator), являющийся противоположностью DI.

По мере применения технологии DI к задачам программирования из реальной жизни вы столкнетесь с рядом трудностей. Пожалуй, у всех нас были моменты, когда казалось, что мы понимаем инструмент или прием, и все же мы думали: «В теории это может сработать, но мой случай — особенный». Когда мы ловим себя на этой мысли, становится понятно, что нам еще следует многому научиться.

На протяжении всей нашей карьеры приходилось наблюдать, что определенный круг проблем появляется снова и снова. У каждой из них есть обобщенное решение, которое можно применить для подведения своего кода к одному из DI-паттернов из главы 4. В главе 6 содержится каталог этих распространенных проблем или примеров проблемного кода и соответствующих решений.

Мы надеемся, что это будет самая полезная часть книги, поскольку она самая продолжительная. Надеемся, что после прочтения этих глав вы еще будете обращаться к ним спустя месяцы и даже годы.

4 Паттерны внедрения зависимостей

В этой главе

- Составление графа объектов с помощью корня композиции.
- Статическое объявление требуемых зависимостей с помощью внедрения через конструктор.
- Пересылка зависимостей за пределы корня композиции с помощью внедрения через метод.
- Объявление необязательных зависимостей с помощью внедрения через свойство.
- Принятие решения по используемому паттерну.

Как и у всех других профессионалов, повара используют собственный жаргон, позволяющий им изъясняться насчет сложностей приготовления блюд на языке, зачастую малопонятном для непосвященных. Не поможет даже то, что большинство используемых ими понятий основано на французском языке (тем более если вы не владеете этим иностранным языком). Отличным примером того, как поварами используется их профессиональная терминология, могут послужить соусы. В главе 1 был упомянут беарнский соус, но на подробностях его классификации мы не останавливались.

В действительности беарнский соус является голландским, где лимонный сок заменен заправкой из выпаренного уксуса, лука-шалота, кервеля и тархуна. Есть и другие соусы, основой для которых служит голландский соус, и к их числу отно-

сится муслин, любимый соус Марка (его можно получить, вмешав в голландский соус взбитые сливки).

Заметили жаргон? Вместо «легкого перемешивания взбитых сливок с соусом, не допуская его свертывания» применен термин «вмешивание», а вместо «загущения и усиления вкуса уксуса» — термин «выпаривание». Жаргон позволяет сделать язык общения лаконичнее и ярче.

В области разработки программных средств имеется собственный, весьма замысловатый сленг. Вы можете не знать, что означает кулинарный термин «водяная баня», но мы уверены, что большинство поваров будет в полном недоумении, если им сказать «струны (strings) являются неизменяемыми классами, представляющими последовательности символов Юникода». А когда речь заходит о способах структурирования кода с целью решения конкретных задач, у нас есть паттерны проектирования, дающие имена общим решениям. По аналогии с тем, как термины «голландский соус» и «вмешивание» помогают лаконично общаться на тему приготовления соуса-муслина, паттерны проектирования помогают изъясняться на тему способов структурирования кода.

В предыдущих главах уже были названы некоторые паттерны проектирования. Например, в главе 1 шел разговор о паттернах «Абстрактная фабрика» (Abstract Factory), «Нулевой объект» (Null Object), «Декоратор» (Decorator), «Компоновщик» (Composite), «Адаптер» (Adapter), «Граничный оператор» (Guard Clause), «Заглушка» (Stub), «Имитация» (Mock) и «Подделка» (Fake). Хотя на данный момент, возможно, вспомнить о каждом из них не получится, но теперь, когда речь пойдет о паттернах проектирования, у вас навряд ли возникнет чувство неловкости. Люди любят давать названия часто повторяющимся паттернам, даже если они совсем простые.

Если ваши общие познания о паттернах проектирования весьма скромны, переживать не стоит. Основная цель паттернов проектирования заключается в предоставлении подробного и полного описания конкретного способа достижения цели — рецепта, если хотите. Кроме этого вы уже видели примеры использования трех из четырех основных паттернов проектирования DI, из этой главы, к которым относятся:

- *корень композиции (Composition Root)* — дает описание, где и как нужно составлять граф объектов приложения;
- *внедрение через конструктор (Constructor Injection)* — позволяет классу статически объявить требуемые ему зависимости;
- *внедрение через метод (Method Injection)* — позволяет вам предоставить зависимость потребителю, когда для каждой операции может измениться либо зависимость, либо потребитель;
- *внедрение через свойство (Property Injection)* — позволяет клиентам в ряде случаев переопределять некоторое поведение класса по умолчанию, если это поведение по умолчанию реализовано как локальная реализация по умолчанию (Local Default).

Структура главы выстроена так, чтобы предоставить каталог паттернов. Каждый паттерн будет сопровождаться кратким описанием, примером кода, сильными и слабыми сторонами и т. д. Все четыре паттерна, представленные в этой главе, можно изучить последовательно или же прочитать про те из них, которые вам интересны. Самыми важными паттернами, которыми следует пользоваться в большинстве ситуаций, являются паттерн корня композиции, или точки входа (Composition Root), и паттерн внедрения через конструктор (Constructor Injection). Особенности других паттернов будут раскрыты по мере развития главы.

4.1. Корень композиции

Где нужно составлять графы объектов?
Как можно ближе к точке входа приложения.

Когда приложение создается из множества классов со слабой связанностью, композиция должна быть составлена как можно ближе к точке входа приложения. Для приложений многих типов точкой входа является метод `Main`. В корне композиции составляется граф объектов, впоследствии выполняющих реальную работу приложения.

ОПРЕДЕЛЕНИЕ

Корень композиции является единственным логическим местом в приложении, где модули составлены вместе.

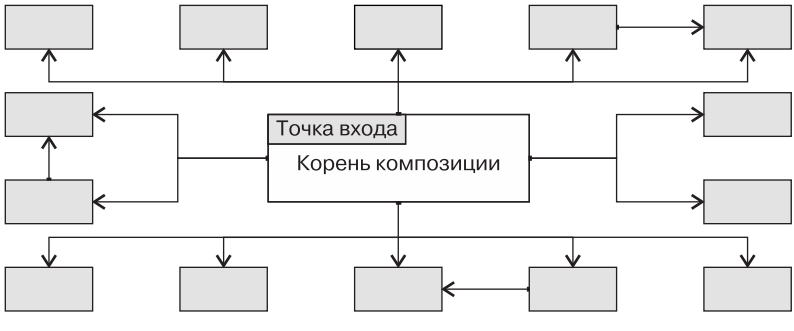


Рис. 4.1. Будучи в непосредственной близи к точке входа, корень композиции занимается составлением графа объектов, относящихся к классам со слабой связанностью. Корень композиции напрямую зависит от всех модулей, имеющихся в системе

В предыдущей главе было показано, что в большинстве классов использовалось внедрение через конструктор. Таким образом они возлагали ответственность за создание их зависимостей на своих потребителей. Эти потребители также возлагали ответственность за создание своих зависимостей на своих потребителей.

Бесконечно откладывать создание своих объектов вы не сможете. Необходимо место, где вами будет создан ваш граф объектов. Оно должно быть сконцентрировано в одной-единственной области вашего приложения. Это место называется корнем композиции.

ВНИМАНИЕ

> При использовании DI-контейнера корень композиции должен быть единственным местом, где используется этот контейнер. Применение DI-контейнера вне корня композиции приводит к возникновению антипаттерна «Локатор сервисов» (Service Locator), который будет рассмотрен в следующей главе.

В предыдущей главе в итоге получился граф объектов, показанный в листинге 3.13 (и листинге 4.1). В этом листинге также показано, что все компоненты из всех уровней приложения создаются в корне композиции.

Листинг 4.1. Граф объектов приложения из главы 3

```
new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        new AspNetUserContextAdapter()));
```

Доменный компонент

Компонент доступа к данным

Компонент пользовательского интерфейса

Если бы у вас было консольное приложение, созданное для работы именно с этим графом объектов, оно могло бы иметь вид, показанный в листинге 4.2.

Листинг 4.2. Граф объектов приложения в качестве части консольного приложения

```
public static class Program
{
    public static void Main(string[] args)
    {
        string connectionString = args[0];

        HomeController controller =
            CreateController(connectionString);

        var result = controller.Index();

        var vm = (FeaturedProductsViewModel)result.Model;

        Console.WriteLine("Featured products:");

        foreach (var product in vm.Products)
        {
            Console.WriteLine(product.SummaryText);
        }
    }

    private static HomeController CreateController(
```

Точка входа приложения

Извлечение строки подключения из предоставленных аргументов командной строки

Запрос к корню композиции приложения на создание нового экземпляра контроллера

Действует как корень композиции приложения

```
        string connectionString)
    {
        var userContext = new ConsoleUserContext();
        return
            new HomeController(
                new ProductService(
                    new SqlProductRepository(
                        new CommerceContext(
                            connectionString)),
                    userContext));
    }
}
```

Реализация IUserContext, позволяющая ProductService работать и вычислять скидки

Составление графа объектов приложения

В этом примере корень композиции отделен от метода `Main`. Но делать это не обязательно — корень композиции не является методом или классом, это концепция. Он может быть частью метода `Main` или быть разбросан по нескольким классам при условии, что все они находятся в одном и том же модуле. Выделение его в свой собственный метод помогает гарантировать консолидированность композиции и отсутствие вкраплений в нее последующей логики приложения, неизбежных в противном случае.

4.1.1. Принцип работы корня композиции

При написании кода со слабой связанностью с целью получения приложения создается множество классов. Может возникнуть соблазн скомпоновать эти классы так, что они будут разбросаны по многим разным местам в виде небольших подсистем. Однако это ограничит ваши возможности перехвата таких систем для изменения их поведения. Вместо этого нужно скомпоновать классы в одной-единственной области вашего приложения.

Рассматривая внедрение через конструктор отдельно от всего остального, задайтесь вопросом: а не откладывается ли решение о выборе зависимости в другое место? Да, так и есть, и в этом нет ничего плохого. Это означает, что вы получаете центральное место, где можно связать взаимодействующие классы.

Корень композиции действует в качестве третьей стороны, которая соединяет потребителей с их сервисами. Чем дольше откладывается решение о способе связи классов, тем дольше сохраняется свобода выбора. Поэтому корень композиции должен быть как можно ближе к точке входа приложения.

Даже модульное приложение, использующее для составления своей композиции код со слабой связанностью и поздним связыванием, имеет исходный корень, содержащий точку входа в приложение. Можно привести следующие примеры.

- Консольное приложение среды .NET Core представляет собой библиотеку (`.dll`), содержащую класс `Program` с методом `Main`.
- Веб-приложение среды ASP.NET Core также является библиотекой, содержащей класс `Program` с методом `Main`.
- Приложения UWP и WPF являются исполняемыми файлами (`.exe`) с файлом `App.xaml.cs`.

Существует множество других технологий, но у них есть одно общее свойство: один модуль содержит точку входа в приложение — это корень приложения. Пусть вас не смущает мысль, что корень композиции является частью вашего пользовательского интерфейса. Даже если поместить корень композиции в сборку, где находится и ваш уровень пользовательского интерфейса, что и будет сделано в следующем примере, корень композиции не станет частью этого уровня.

Сборки являются *артефактами разработки*: код делится на несколько сборок, чтобы его можно было развертывать по отдельности. А вот структурный уровень является *логическим артефактом*: в отдельно взятом артефакте разработки можно составлять группу из нескольких логических артефактов. Даже притом, что сборка, содержащая как корень композиции, так и уровень пользовательского интерфейса, зависит от всех остальных модулей, имеющихся в системе, сам по себе уровень пользовательского интерфейса от них не зависит.

ВНИМАНИЕ

Корень композиции не является частью уровня пользовательского интерфейса, даже притом, что он может быть помещен в ту же самую сборку.

Корень композиции не требуется помещать в тот же самый проект, куда помещен ваш уровень пользовательского интерфейса, который можно убрать из проекта корня приложения. Тогда можно будет выиграть на том, что у проекта, содержащего уровень пользовательского интерфейса, не будет никакой зависимости (например, от проекта уровня доступа к данным, как в главе 3). Этим определится невозможность случайной зависимости классов пользовательского интерфейса от классов доступа к данным. Однако недостатком такого подхода является то, что это не всегда дается легко. Например, при использовании среды ASP.NET Core MVC переместить модели контроллеров и представлений в отдельный проект будет нетрудно, а вот сделать то же самое с вашими ресурсами представлений и клиентскими ресурсами окажется весьма затруднительно.

Разделение технологии представления и корня композиции может и не принести особых преимуществ, поскольку корень композиции у каждого приложения имеет свою специфику. Повторное применение корням композиции не свойственно.

Не нужно пытаться компоновать классы в любом из других модулей, поскольку такой подход ограничит выбор вариантов. Все классы в модулях приложения должны использовать внедрение через конструктор (или в редких случаях один из двух других паттернов, рассматриваемых в этой главе), а затем возложить составление графа объектов приложения на корень композиции. Любой используемый DI-контейнер должен ограничиваться корнем композиции.

ПРИМЕЧАНИЕ

Перемещение компоновки классов за пределы корня композиции приводит к созданию либо антипаттерна «Диктатор» (Control Freak), либо антипаттерна «Локатор сервисов» (Service Locator), которые будут рассмотрены в следующей главе.

Корень композиции должен быть в приложении единственным местом, осведомленным о структуре созданных графов объектов. Код приложения не только отказывается от контроля над своими зависимостями, но и от знания о них. Централизация этих сведений упрощает разработку. Это также означает, что код приложения не может передавать зависимости другим потокам, запускаемым в параллель для текущей операции, поскольку потребитель не обладает способами определения степени безопасности подобного действия. Вместо этого при организации одновременно выполняемых операций именно на корень композиции возлагается задача создания нового графа объектов для каждой одновременно выполняемой операции.

Корень композиции, показанный в листинге 4.2, является примером чистой технологии DI. Но паттерн корня композиции применяется как для чистой технологии DI, так и для DI-контейнеров. Порядок использования DI-контейнера в корне композиции будет рассмотрен в следующем разделе.

4.1.2. Использование DI-контейнера в корне композиции

Согласно описанию из главы 3, DI-контейнер представляет собой программную библиотеку, способную автоматизировать многие задачи, вовлеченные в процесс компоновки объектов и управления их временем жизни. Но при его неправильном использовании может получиться «Локатор сервисов», так что DI-контейнер нужно применять только в качестве механизма составления графов объектов. Когда DI-контейнер рассматривается с этой позиции, его использование стоит ограничить корнем композиции. Это также оказывает весомую помощь в устранении любой связанности между DI-контейнером и всей остальной кодовой базой приложения.

ПРИМЕЧАНИЕ

> Ссылкой на DI-контейнер должен обладать только корень композиции, и ссылаться на него можно только из этого корня. (Все остальное приложение при этом не имеет ссылки на контейнер, полагаясь вместо этого на использование паттернов, рассматриваемых в этой главе.) Кроме того, на контейнер не должны ссылаться и все остальные модули. DI-контейнеры допускают применение этих паттернов и используют их для составления графа объектов приложения.

Корень композиции может быть реализован с помощью DI-контейнера. Это означает, что контейнер будет использоваться для составления всего графа объектов приложения в рамках одного вызова к его методу `Resolve`. Когда разработчикам рассказывают о такой методике работы, всегда можно упомянуть о возникающем у них чувстве дискомфорта по причине тревог за ее крайнюю неэффективность и отрицательное влияние на производительность. Но вам на этот счет волноваться не стоит. Такое почти никогда не случается, а в тех редких ситуациях, когда обнаруживаются подобные проявления, существуют способы решения проблем, которые подробнее будут рассмотрены в подразделе 8.4.2.

Не стоит волноваться и насчет потерь производительности, связанных с использованием DI-контейнера для составления больших графов объектов. Обычно это

не создает никаких проблем. В части IV книги вам предстоит углубленное изучение DI-контейнеров, где будет показан порядок использования DI-контейнера внутри корня композиции.

Когда дело касается приложений, основанных на запросах, таких как веб-сайты и сервисы, настройка контейнера производится единожды, но граф объектов готовится для каждого входящего запроса. Примером этому может послужить веб-приложение электронной торговли из главы 3.

4.1.3. Пример: реализация корня композиции с использованием чистой технологии DI

У учебного веб-приложения электронной торговли для составления графа объектов для входящих HTTP-запросов должен быть корень композиции. Как и для всех других веб-приложений среды ASP.NET Core, точкой входа является метод `Main`. Но изначально метод `Main` в приложении среды ASP.NET Core делегирует основную часть работы классу `Startup`. Это класс располагается достаточно близко к точке входа приложения, что вполне сообразуется с нашими целями, и мы им воспользуемся в качестве нашего корня композиции.

Как и в предыдущем примере с консольным приложением, мы воспользуемся чистой технологией DI. Это означает, что для составления графа объектов будет использован не DI-контейнер, а, как показано в листинге 4.3, привычный код на C#.

Листинг 4.3. Класс Startup приложения электронной торговли

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        this.Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(
        IServiceCollection services)
    {
        services.AddMvc();
        services.AddHttpContextAccessor();

    var connectionString =
        this.Configuration.GetConnectionString(
            "CommerceConnection");

    services.AddSingleton<IControllerActivator>(
        new CommerceControllerActivator(
            connectionString));
    }
    ...
}
```

Среда ASP.NET Core вызывает этот конструктор при запуске приложения

По соглашению ASP.NET вызывает этот метод. Предоставляемый экземпляр IServiceCollection позволяет оказать влияние на исходные сервисы, известные среде ASP.NET

Добавление к среде сервиса, извлекающего текущий HttpContext

Загрузка из конфигурационного файла принадлежащей приложению строки подключения к базе данных

Замена IControllerActivator по умолчанию на тот, что создает графы объектов

Если вы не знакомы со средой ASP.NET Core, то вот простое объяснение: без класса `Startup` просто не обойтись; именно он является связующим звеном. Наибольший интерес представляет класс `CommerceControllerActivator`, код которого вскоре будет показан. Именно в нем заключена вся настройка приложения.

Чтобы подключить MVC-контроллеры к приложению, нужно воспользоваться соответствующим швом в ASP.NET Core MVC, который называется `IControllerActivator` (более подробно он будет рассмотрен в разделе 7.3). А пока будет вполне достаточно понять, что для объединения со средой ASP.NET Core MVC следует создать адаптер для вашего корня композиции и сообщить о нем этой среде.

ПРИМЕЧАНИЕ

> Любая качественно спроектированная среда предоставляет соответствующие швы для перехвата создания типов среды. Обычно они оформлены в виде фабричных абстракций наподобие принадлежащего среде MVC активатора IControllerActivator.

Метод `Startup.ConfigureServices` запускается только один раз. В результате этого ваш класс `CommerceControllerActivator` является единичным экземпляром, проинициализированным только один раз. Поскольку настройка среды ASP.NET Core MVC выполняется с помощью собственного активатора `IControllerActivator`, MVC вызывает его метод `Create` для создания нового экземпляра контроллера для каждого входящего HTTP-запроса (подробности можно посмотреть в разделе 7.3). Код `CommerceControllerActivator` можно увидеть в листинге 4.4.

Листинг 4.4. Реализация IControllerActivator, принадлежащего приложению

```
public class CommerceControllerActivator : IControllerActivator
{
    private readonly string connectionString;

    public CommerceControllerActivator(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public object Create(ControllerContext ctx)
    {
        Type type = ctx.ActionDescriptor
            .ControllerTypeInfo.AsType();

        if (type == typeof(HomeController))
        {
            return
                new HomeController(
                    new ProductService(
                        new SqlProductRepository(
                            new CommerceContext(
                                this.connectionString)),
                        new AspNetUserContextAdapter()));
```

Среда ASP.NET Core MVC вызывает этот метод для каждого запроса

Создание соответствующего графа объектов, если MVC требуется HomeController

```
            }
            else
            {
                throw new Exception("Unknown controller.");
            }
        }
}
```

Приложение электронной торговли на данный момент имеет только один контроллер. У каждого нового добавляемого вами контроллера будет свой собственный блок if

Обратите внимание на то, как создание `HomeController` в этом примере практически идентично графу объектов из главы 3, который показан в листинге 4.1. Когда среда MVC вызывает метод `Create`, вами определяется тип контроллера и на основе этого типа создается соответствующий граф объектов.

В подразделе 2.3.3 уже говорилось, что от конфигурационных файлов должен зависеть только корень композиции, поскольку для многократно используемых библиотек более гибким вариантом будет обязательное свойство их настраиваемости со стороны вызывающего кода. Нужно также отделить загрузку конфигурационных значений от методов, выполняющих компоновку объектов (как показано в листингах 4.3 и 4.4). Класс `Startup` из листинга 4.3 загружает конфигурацию, а класс `CommerceControllerActivator` из листинга 4.4 зависит только от конфигурационного значения, но не от системы конфигурирования. Важное преимущество такого отделения состоит в том, что оно отвязывает компоновку объектов от используемой системы конфигурирования, позволяя проводить тестирование без наличия настоящего файла конфигурации.

Как показано на рис. 4.2, корень композиции в этом примере разбросан по двум классам. Это вполне ожидаемо. Важно то, что все классы находятся в одном и том же модуле, который в данном случае является корнем приложения.

Самое главное на этом рисунке то, что эти два класса являются единственными во всем учебном приложении, занятыми составлением графа объектов. Во всем остальном коде приложения используется только паттерн внедрения через конструктор.

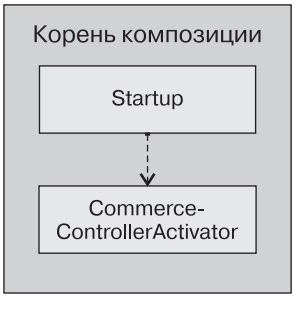


Рис. 4.2. Корень композиции разбросан по двум классам, но они определены внутри одного и того же модуля

4.1.4. Кажущийся взрыв зависимостей

Разработчики часто жалуются, что корень композиции заставляет точку входа приложения зависеть от всех остальных сборок, имеющихся в приложении. В их прежних кодовых базах с сильной связанностью точки входа нуждались только в зависимости от уровня, расположенного непосредственно под ними. Это кажется шагом назад, поскольку технология DI предназначена для снижения требуемого числа зависимостей. Они видят, что использование DI вызывает взрыв зависимостей в точке входа их приложения, — или же им так кажется.

Жалоба исходит из факта недопонимания разработчиками, как функционируют зависимости проекта. Чтобы получить достаточное представление о предмете их

беспокойства, взглянем на схему зависимостей приложения, созданного Мэри в главе 2, и сравним ее со схемой зависимостей приложения из главы 3, где используется код со слабой связанностью (рис. 4.3).

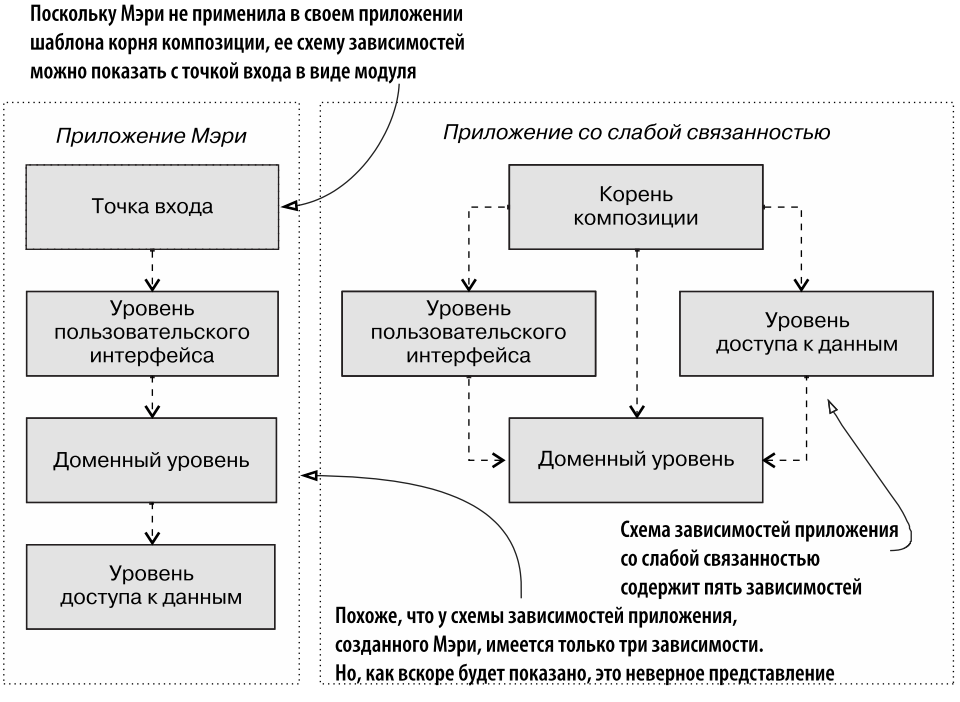


Рис. 4.3. Сравнение схемы зависимостей приложения, созданного Мэри, с приложением со слабой связанностью

На первый взгляд все действительно выглядит так, будто в приложении со слабой связанностью имеется на две зависимости больше, чем в приложении, созданном Мэри со «всего лишь» тремя зависимостями. И все же схема дает неверное представление.

Изменения, вносимые в уровень доступа к данным, также распространяются через уровень пользовательского интерфейса, и, как уже говорилось в предыдущей главе, уровень пользовательского интерфейса не может быть развернут без уровня доступа к данным.

Хотя на схеме это не показано, есть еще зависимость между уровнем пользовательского интерфейса и уровнем доступа к данным. По факту зависимости сборки носят транзитивный характер.

ПРИМЕЧАНИЕ

Транзитивность является математическим понятием, утверждающим, что при условии, когда элемент a связан с элементом b, а элемент b связан с элементом c, то элемент a также связан с элементом c.

Это транзитивное отношение означает, что по причине зависимости созданного Мэри уровня пользовательского интерфейса от доменного уровня, а доменного уровня — от уровня доступа к данным уровень пользовательского интерфейса также зависит от уровня доступа к данным и это поведение неизменно проявится при развертывании приложения. Если посмотреть на зависимости между проектами в приложении, созданном Мэри, сложится несколько иная картина (рис. 4.4).

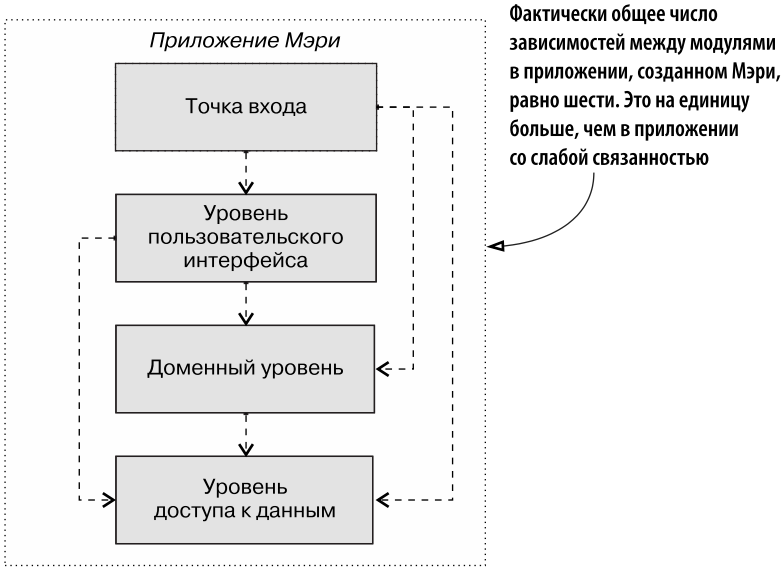


Рис. 4.4. Зависимости между библиотеками в приложении, созданном Мэри

Как видите, даже в приложении, созданном Мэри, точка входа зависит от всех библиотек. И у точки входа приложения, созданного Мэри, и у корня композиции приложения со слабой связанностью одинаковое число зависимостей. Но следует помнить, что зависимости определяются не количеством модулей, а количеством проявлений зависимости каждого модуля от другого модуля. В результате общее количество зависимостей между всеми модулями в приложении, созданном Мэри, по сути, равно шести. Это на единицу больше, чем у приложения со слабой связанностью.

Теперь представьте приложение с десятками проектов. Нетрудно предположить, насколько взрывным станет характер роста зависимостей в кодовой базе с сильной связанностью по сравнению с кодовой базой со слабой связанностью. Но, создавая код со слабой связанностью, применяющий паттерн корня композиции, можно сократить число зависимостей. Как уже было показано в предыдущей главе, это позволит вам заменять целые модули другими модулями, что в кодовой базе с сильной связанностью дается намного труднее.

Паттерн корня композиции применяется всеми разработчиками, использующими технологию DI, но корень композиции будет иметься только у проектов запуска

приложения. Корень композиции является результатом снятия ответственности за создание зависимостей с потребителей. Чтобы добиться этого, можно воспользоваться двумя паттернами: внедрением через конструктор и внедрением через свойство. Первый используется чаще и практически повсеместно. Именно поэтому ему и будет посвящен следующий раздел.

4.2. Внедрение через конструктор

Как можно гарантировать, что необходимая нестабильная зависимость будет всегда доступна разрабатываемому на данный момент классу?

Путем выдвижения требования всем вызывающим объектам предоставлять нестабильную зависимость в качестве параметра конструктора класса.

Когда классу требуется экземпляр зависимости, ее можно предоставить через конструктор класса, позволив тому сохранить ссылку для последующего использования.

ОПРЕДЕЛЕНИЕ

Внедрение через конструктор — это статическое определение списка требуемых зависимостей путем указания их конструктору класса в качестве параметров.

Сигнатура конструктора скомпилирована с типом и доступна для всеобщего обозрения. В ней ясно задокументировано, что класс нуждается в зависимостях, запрашиваемых через его конструктор. Это продемонстрировано на рис. 4.5.

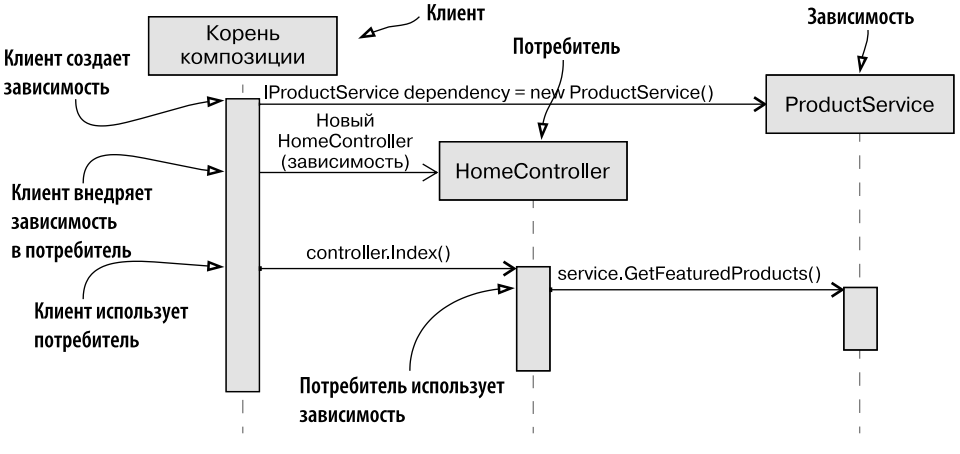


Рис. 4.5. Создание экземпляра HomeController с требуемой зависимостью IProductService с помощью внедрения через конструктор

На рисунке показано, что потребляющий класс `HomeController` нуждается для своей работы в экземпляре зависимости `IProductService`, поэтому он требует,

чтобы корень композиции (клиент) предоставил экземпляр через его конструктор. Тем самым гарантируется доступность экземпляра классу `HomeController` по мере необходимости.

4.2.1. Принцип работы внедрения через конструктор

Класс, нуждающийся в зависимости, должен предоставить открытый конструктор, получающий экземпляр нужной зависимости в качестве аргумента конструктора. Это должен быть только конструктор, находящийся в свободном доступе. Если требуется больше одной зависимости, к тому же самому конструктору должны быть добавлены дополнительные аргументы конструктора. Определение класса `HomeController`, показанного на рис. 4.5, дано в листинге 4.5.

ВАЖНО

Следует ограничить проект единственным (открытым) конструктором. Поскольку конструктор является определением зависимостей класса, особого смысла в наличии нескольких определений нет. Переопределение конструкторов приводит к неопределенности: каким из конструкторов должен воспользоваться вызывающий объект (или DI-контейнер)?

Листинг 4.5. Внедрение зависимости через конструктор

```
public class HomeController
{
    private readonly IProductService service;

    public HomeController(
        IProductService service)
    {
        if (service == null)
            throw new ArgumentNullException("service");

        this.service = service;
    }
}
```

Приватное поле экземпляра класса для хранения предоставленной зависимости

Конструктор, который статически определяет свои зависимости

Аргумент для предоставления необходимой зависимости

Контрольная инструкция, не позволяющая клиентам передавать null

Сохранение зависимости в приватном поле для последующего использования. В конструкторе не содержится никакой другой логики, проверяющей и сохраняющей его входящие зависимости

Зависимость `IProductService` является обязательным аргументом конструктора `HomeController`; любой клиент, не предоставляющий экземпляр `IProductService`, не сможет пройти компиляцию. Но, поскольку интерфейс является ссылочным типом, чтобы вызываемый код прошел компиляцию, вызывающий объект может передать в качестве аргумента `null`. Вы должны защитить класс от такого неприем-

лемого использования, применив граничный оператор (Guard Clause)[1]. Поскольку объединенные действия компилятора и граничного оператора гарантируют приемлемость аргумента конструктора при условии, что не было выдано исключение, конструктор может сохранить зависимость для последующего использования без каких-либо сведений о ее реальной реализации.

Рекомендуется помечать поле, содержащее зависимость как предназначенное только для чтения (`readonly`). Тем самым гарантируется, что после выполнения логики инициализации в конструкторе это поле не сможет подвергаться изменениям. С позиции технологии DI подобное строгое требование не выдвигается, но его соблюдение защитит вас от случайных изменений поля (например, от установки его значения в `null`) где-либо еще в коде зависимого класса.

ВАЖНО

Конструктор не нужно загружать какой-либо другой логикой, чтобы оградить его от любой работы над зависимостями. Соблюдение принципа единственной ответственности подразумевает, что компонентами должно выполняться что-то одно. Теперь, когда конструктор используется для внедрения зависимостей, его не следует загружать другими задачами. Тогда конструкция класса становится быстродействующей и надежной.

Когда управление из конструктора возвращается, новый экземпляр класса пребывает в состоянии, согласованном с соответствующим экземпляром внедренной в него зависимости. Поскольку в созданном конструктором классе хранится ссылка на эту зависимость, он может использовать зависимость из любых своих других компонентов требуемое число раз. Его компонентам не требуется проверка на `null`, поскольку присутствие экземпляра полностью гарантируется.

4.2.2. Когда следует использовать внедрение через конструктор

Внедрение через конструктор должно стать вашим выбором по умолчанию для DI. Оно относится к наиболее распространенному сценарию, где классу требуется одна или несколько зависимостей, а приемлемые локальные реализации по умолчанию (Local Defaults) недоступны.

ОПРЕДЕЛЕНИЕ

Локальная реализация по умолчанию — реализация зависимости по умолчанию, создаваемая в том же самом модуле или на том же уровне.

---

[1] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 253.

Локальная реализация по умолчанию (Local Default)

При разработке класса, имеющего зависимость, у вас, вероятно, уже есть конкретный замысел по реализации этой зависимости. Если создается доменный сервис, обращающийся к хранилищу, то, скорее всего, планируется разработка хранилища, использующего реляционную базу данных.

Было бы заманчиво создать эту реализацию в качестве используемой разрабатываемым классом по умолчанию. Но, когда такая реализация размещается в другой сборке, ее использование в качестве варианта по умолчанию означает создание жесткой ссылки на другую сборку, что фактически сводит на нет многие преимущества слабой связанности, рассмотренные в главе 1. Подобная реализация, будучи полной противоположностью локальной реализации по умолчанию, является внешней реализацией по умолчанию (Foreign Default). Класс, имеющий жесткую ссылку на внешнюю реализацию по умолчанию, применяет антипаттерн «Диктатор» (Control Freak), который будет рассмотрен в главе 5.

И наоборот, если предполагаемая реализация по умолчанию определена в той же самой библиотеке, что и потребляющий класс, такая проблема не возникнет. В случае с хранилищами такой сценарий маловероятен, но такие локальные реализации по умолчанию часто фигурируют в качестве реализаций паттерна «Стратегия»[1].

ВНИМАНИЕ

Локальная реализация по умолчанию с зависимостями становится внешней реализацией по умолчанию, когда одна из ее зависимостей сама является внешней реализацией по умолчанию. Здесь снова проявляется транзитивность.

Внедрение через конструктор относится к весьма распространенному сценарию, когда объекту требуется зависимость, а приемлемая локальная реализация по умолчанию недоступна. Дело в том, что такое внедрение гарантирует обязательное предоставление зависимости. Если зависимый класс абсолютно не может функционировать без зависимости, такая гарантия неоценима (табл. 4.1).

Таблица 4.1. Преимущества и недостатки внедрения через конструктор

Преимущества	Недостатки
Гарантированное внедрение. Простота реализации. Статическое объявление зависимостей класса	Среды, применяющие антипаттерн ограниченной конструкции, могут затруднить внедрение через конструктор

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 362.

В случаях, когда локальная библиотека может предоставить нужную реализацию по умолчанию, внедрение через свойства также может стать подходящим вариантом, но обычно так не получается. В предыдущих главах мы показали множество примеров того, как в качестве зависимостей выступают хранилища. Это вполне подходящий образец зависимостей, где локальная библиотека не в состоянии предоставить приемлемой реализации по умолчанию, поскольку нужные реализации принадлежат специализированным библиотекам доступа к данным. Помимо уже упомянутого гарантированного внедрения, этот паттерн также легко реализовать, используя структуру, представленную в листинге 4.5.

Главным недостатком внедрения через конструктор является то, что, если создаваемый класс вызывается текущей средой выполнения приложения, для поддержки этого класса может понадобиться настройка среды. Некоторыми средами, особенно старыми, предполагается, что у ваших классов будет конструктор без параметров[1]. (Это так называемый антипаттерн ограниченной конструкции, мы изучим его в следующей главе.) В этом случае среде понадобится особое содействие в создании экземпляров, когда конструктор без параметров недоступен. Объяснения порядка задействования внедрения через конструктор в наиболее распространенных средах выполнения приложений будут даны в главе 7.

Как уже говорилось в разделе 4.1, очевидным недостатком внедрения через конструктор является потребность в немедленной инициализации всего графа зависимостей. При всей своей кажущейся неэффективности данное обстоятельство вряд ли станет проблемой. В конце концов, даже в случае относительно сложного графа объектов мы обычно говорим о создании всего лишь нескольких десятков новых экземпляров объектов, а с созданием экземпляра объекта среда .NET справляется довольно быстро. Любые возможные препятствия на пути повышения производительности вашего приложения будут возникать в других местах, поэтому волнения напрасны[2].

ПРИМЕЧАНИЕ

Ранее уже говорилось, что конструкторы компонентов должны быть свободны от всей логики, за исключением контрольных проверок и хранения входящих зависимостей. Тогда создание проходит быстро и предотвращается большинство проблем со снижением производительности.

---

[1] Среда ASP.NET Web Forms вынуждала к применению для форм и пользовательских элементов управления конструкторов без параметров, но с выходом .NET 4.7.2 ситуация изменилась. Теперь в среде ASP.NET формы и элементы управления могут создаваться с применением внедрения через конструктор.

[2] В исключительно редких случаях данное обстоятельство может создавать реальную проблему, но в главе 8 будет рассмотрен способ, позволяющий отложить создание зависимости, который может стать одним из возможных вариантов решения этой проблемы. Пока просто остановимся на том, что с начальной загрузкой может возникнуть проблема.

Слишком большие графы объектов

Однажды я, Стивен, разговаривал с разработчиком, который пошел на замену DI-контейнеров из-за серьезных проблем с производительностью своего старого контейнера. После этого переключения он сообщил о весьма впечатляющем ускорении ответа на веб-запрос на 300–400 мс. Но, проанализировав работу его приложения, я понял, что в некоторых случаях созданный граф объектов содержал более 19 000 экземпляров этих объектов. Неудивительно, что у некоторых DI-контейнеров происходило существенное снижение производительности.

Для меня размер этого графа объектов был просто немыслимым. Раньше я никогда не видел ничего подобного. В системе было множество огромных классов со слишком большим числом зависимостей. Зачастую в них было от 20 и более зависимостей[1]. Даже у часто используемых классов было такое количество зависимостей, что число экземпляров объектов в графе объектов выходило из-под контроля, или, как выразился сам этот разработчик, «реальный мир иногда превосходил фантазии».

Хотя эту историю можно посчитать доказательством возникновения проблем с производительностью, ее мораль заключается в том, что качественно спроектированные системы вряд ли когда-либо будут сталкиваться с такой проблемой. В подобных системах у классов имеется всего лишь несколько зависимостей (не более 4–5), благодаря чему граф объектов не получается слишком широким. В качественно спроектированных системах графы объектов обычно становятся глубже из-за легкости применения нескольких уровней декораторов[2]. Но в конечном итоге количество объектов в графах качественно спроектированных систем не выйдет за пределы нескольких сотен. Это означает, что при нормальных условиях и качественно спроектированной системе даже более медленные DI-контейнеры обычно не вызывают никаких проблем производительности.

Теперь, когда вы уже знаете, что внедрение через конструктор является наиболее предпочтительным способом применения DI, рассмотрим некоторые известные примеры. Для этого далее будет рассмотрено внедрение через конструктор в .NET BCL.

4.2.3. Известные примеры использования внедрения через конструктор

Хотя внедрение через конструктор получает в приложениях, применяющих DI, весьма широкое распространение, в библиотеке BCL оно представлено весьма скромно. Главным образом это связано с тем, что BCL является не полноценным приложением, а набором многократно используемых библиотек. Двумя соответствующими примерами, показывающими разновидность внедрения через конструктор в BCL, являются классы `System.IO.StreamReader` и `System.IO.StreamWriter`. Оба получают в своих конструкторах экземпляр `System.IO.Stream`. Ниже представлены все

[1] Это проблемный код, который называется чрезмерным внедрением через конструктор (Constructor Over-injection); он будет рассмотрен в разделе 6.1.

[2] В главе 10 будет создан пример, содержащий несколько уровней декораторов.

связанные со Stream StreamWriter-конструкторы; StreamReader-конструкторы выглядят аналогично:

```
public StreamWriter(Stream stream);
public StreamWriter(Stream stream, Encoding encoding);
public StreamWriter(Stream stream, Encoding encoding, int bufferSize);
```

Stream — абстрактный класс, который служит абстракцией, на основе которой выполняют свои обязанности StreamWriter и StreamReader. Их конструкторам можно предоставить любую реализацию Stream, и они ею воспользуются, но при попытке «подсовывания» им null-потока они выдадут исключение ArgumentNullExceptions.

ПРИМЕЧАНИЕ

Для классов в составе многократно используемой библиотеки классов (например, BCL) наличие нескольких конструкторов зачастую имеет вполне определенный смысл. Однако для компонентов приложения никакого смысла в этом нет.

Хотя библиотека BCL предоставляет примеры, где можно увидеть практическое применение внедрения через конструктор, более наглядной всегда получается демонстрация рабочего примера. В следующем разделе будет последовательно разобран пример полной реализации.

4.2.4. Пример: добавление конвертации валюты к прейскуранту предлагаемых товаров

Начальник Мэри сказал, что ее приложение работает вполне успешно, но появились клиенты, желающие расплачиваться за товар в разных валютах. Нельзя ли создать новый код, позволяющий приложению отображать и производить расчет стоимости товаров с учетом нового запроса? Мэри вздохнула и поняла, что будет недостаточно жестко запрограммировать несколько различных валютных конвертаций. Ей понадобится создать код, обладающий достаточной гибкостью, чтобы приспособиться со временем к любой валюте. И тут опять востребована технология DI.

Мэри нуждается как в объекте для представления количества денег и их валютной принадлежности, так и в абстракции, позволяющей выполнять конвертацию денег из одной валюты в другую. Она назовет эту абстракцию ICurrencyConverter. Для простоты, как показано на рис. 4.6, класс валюты Currency будет содержать только код валюты Code, а класс Money будет составлен как из объекта класса Currency, так и из объекта класса суммы Amount.

В листинге 4.6 показаны намеченные на рис. 4.6 классы Currency и Money и интерфейс ICurrencyConverter.

Листинг 4.6. Классы Currency и Money и интерфейс ICurrencyConverter

```
public interface ICurrencyConverter
{
    Money Exchange(Money money, Currency targetCurrency);
}
```

```
public class Currency
{
    public readonly string Code;

    public Currency(string code)
    {
        if (code == null) throw new ArgumentNullException("code");

        this.Code = code;
    }
}

public class Money
{
    public readonly decimal Amount;
    public readonly Currency Currency;

    public Money(decimal amount, Currency currency)
    {
        if (currency == null) throw new ArgumentNullException("currency");

        this.Amount = amount;
        this.Currency = currency;
    }
}
```

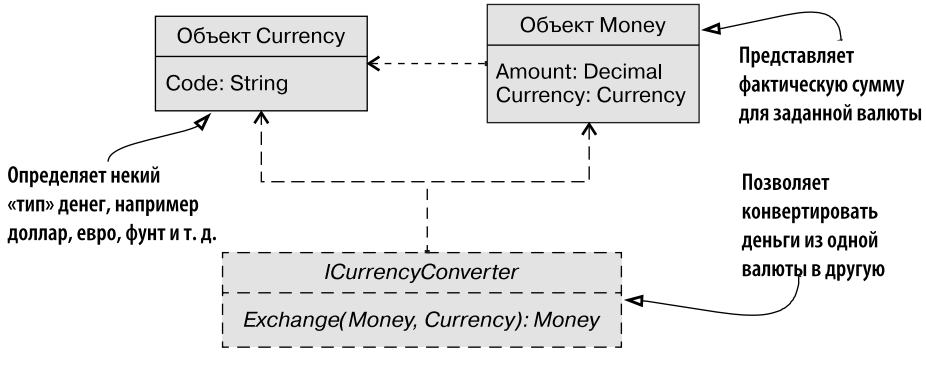


Рис. 4.6. Конвертация валют с помощью ICurrencyConverter

Вероятно, `ICurrencyConverter` представляет собой внепроцессный ресурс, такой как веб-сервис или база данных, предоставляющая коэффициент пересчета. Это означает, что уместно было бы реализовать конкретный `ICurrencyConverter` в отдельном проекте, таком как уровень доступа к данным. Следовательно, ни о какой приемлемой локальной реализации по умолчанию речи не идет.

В то же время класс `ProductService` будет нуждаться в `ICurrencyConverter`. Здесь вполне подойдет внедрение через конструктор. В листинге 4.7 показано, как зависимость `ICurrencyConverter` внедряется в `ProductService`.

Листинг 4.7. Внедрение ICurrencyConverter в ProductService

```
public class ProductService : IProductService
{
    private readonly IProductRepository repository;
    private readonly IUserContext userContext;
    private readonly ICurrencyConverter converter;

    public ProductService(
        IProductRepository repository,
        IUserContext userContext,
        ICurrencyConverter converter)
    {
        if (repository == null)
            throw new ArgumentNullException("repository");
        if (userContext == null)
            throw new ArgumentNullException("userContext");
        if (converter == null)
            throw new ArgumentNullException(“converter”);
        this.repository = repository;
        this.userContext = userContext;
        this.converter = converter;
    }
}
```

Поскольку класс `ProductService` уже имеет зависимость от `IProductRepository` и `IUserContext`, новая зависимость `ICurrencyConverter` добавляется в качестве третьего аргумента конструктора, после чего выполняется та же последовательность действий, выделенная в листинге 4.5. Граничные операторы гарантируют, что зависимости не будут иметь значение `null`, а значит, их можно безопасно сохранить для последующего использования в полях, предназначенных только для чтения. Поскольку присутствие `ICurrencyConverter` в `ProductService` гарантировано, его можно использовать отовсюду; например, как показано в листинге 4.8, в методе `GetFeaturedProducts`.

Листинг 4.8. ProductService использует ICurrencyConverter

```
public IEnumerable<DiscountedProduct> GetFeaturedProducts()
{
    Currency userCurrency = this.userContext.Currency;

    var products = this.repository.GetFeaturedProducts();

    return
        from product in products
        let unitPrice = product.UnitPrice
        let amount = this.converter.Exchange(
            money: unitPrice,
            targetCurrency: userCurrency)
        select product
            .WithUnitPrice(amount)
            .ApplyDiscountFor(this.userContext);
}
```

Добавляем к IUserContext свойство Currency для получения валюты, предпочитаемой пользователем

Теперь у товара есть свойство UnitPrice типа Money

При наличии денег (Money) и новой валюты (Currency) вызываем ICurrencyConverter для предоставления суммы в новой валюте

Обратите внимание, что поле `converter` может использоваться без потребности в предварительной проверке его доступности. Дело в том, что его присутствие гарантированно.

4.2.5. Краткое заключение

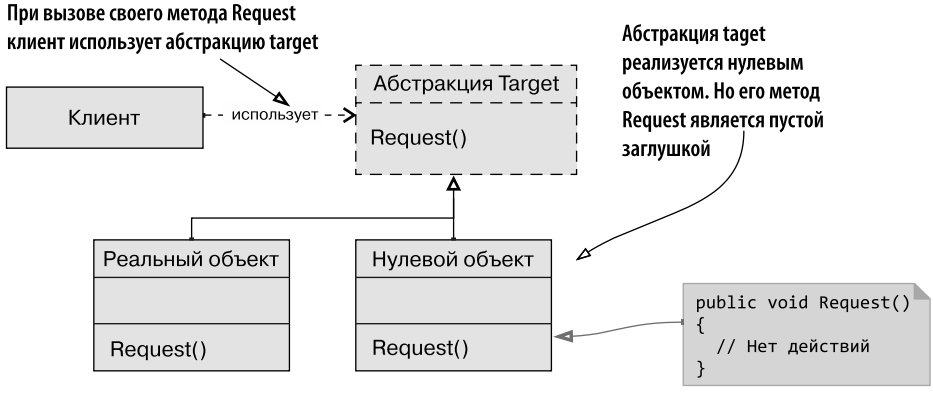
Внедрение через конструктор является наиболее общеприменяемым паттерном DI, который вдобавок легче всего поддается реализации. Он применяется, когда требуется зависимость. Если нужно сделать зависимость необязательной, можно перейти на внедрение с помощью свойства, если при этом имеется приемлемая локальная реализация по умолчанию.

ВНИМАНИЕ

Зависимости вряд ли когда-либо будут необязательными. Необязательные зависимости усложняют потребляющий компонент наличием проверок на null-значение. Вместо этого нужно сделать зависимость обязательной, а затем, на случай отсутствия подходящей доступной реализации для требуемой зависимости, создать и внедрить реализации нулевых объектов.

Паттерн «Нулевой объект»

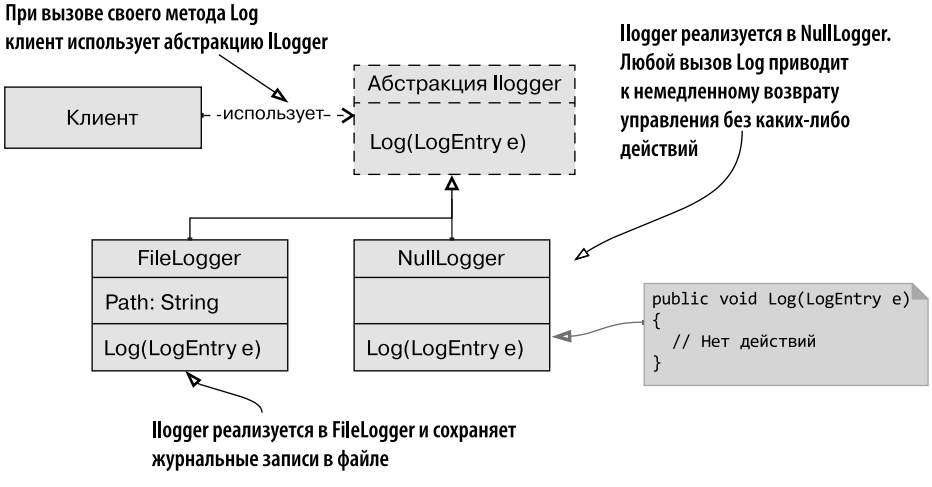
Паттерн проектирования «Нулевой объект» позволяет зависимости потребителя быть всегда доступной, даже при отсутствии какой-либо фактической реализации[1]. Путем внедрения реализации, не содержащей никаких действий — нулевого объекта, — потребитель может обходиться с зависимостью однозначно, не испытывая необходимости в усложнениях за счет добавления проверок на `null`-значения.



[1] *Martin R. C. et al.* Pattern Languages of Program Design 3. — Addison-Wesley, 1998. — P. 5.

Реализации паттерна нулевого объекта обычно пусты, за исключением тех случаев, когда нулевой объект должен возвратить значение. Тогда возвращается самое простое допустимое значение.

Время от времени приложение нуждается в создании выходных данных, позволяющих разработчикам или операторам проанализировать возникающие проблемы. Для этого довольно часто применяется абстракция ведения журнала (Logging). Несмотря на то что класс может быть разработан с прицелом на ведение журнала, приложению, в котором он выполняется, может не понадобиться выполнение регистрационных записей в том или ином классе. Хотя такому классу можно позволить проводить проверку на доступность механизма ведения журнала — например, путем проверки на `null`-значение, — надежнее все же применить решение о внедрении реализации нулевого объекта.



Следующим паттерном в этой главе станет внедрение через метод, для которого используется несколько иной подход. Он главным образом предназначен для применения в ситуации, когда уже имеется зависимость, которую нужно передать вызываемым взаимодействующим объектам.

4.3. Внедрение через метод

Как внедрить зависимость в класс, когда она разная для каждой операции?
Путем ее предоставления в качестве параметра метода.

В тех случаях, когда зависимость может варьироваться с каждым вызовом метода или же потребитель такой зависимости может варьироваться при каждом вызове, зависимость может предоставляться через параметр метода.

ОПРЕДЕЛЕНИЕ

Внедрение через метод предоставляет потребителю зависимость путем ее передачи в виде аргумента того метода, который вызван вне корня композиции.

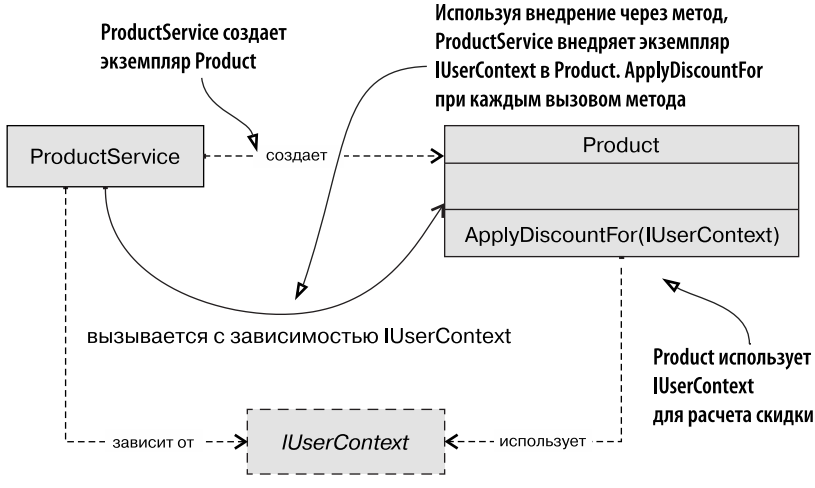


Рис. 4.7. Используя внедрение через метод, ProductService создает экземпляр Product и внедряет экземпляр IUserContext в Product.ApplyDiscountFor при каждым вызове метода

4.3.1. Принцип работы внедрения через метод

Вызывающий объект предоставляет зависимость в виде параметра метода при каждом его вызове. Пример такого подхода в приложении электронной торговли, созданном Мэри, находится в классе `Product`, где метод `ApplyDiscountFor` принимает зависимость `IUserContext`, используя внедрение через метод:

```
public DiscountedProduct ApplyDiscountFor(IUserContext userContext)
```

Зависимость IUserContext принята с использованием внедрения через метод

`IUserContext` предоставляет контекстную информацию для выполнения операции, что вполне обычно для внедрения через метод. Как показано в листинге 4.9, этот контекст зачастую будет передаваться методу вместе с правильным значением.

Листинг 4.9. Передача зависимости вместе с правильным значением

```
public decimal CalculateDiscountPrice(decimal price, IUserContext context)
{
    if (context == null) throw new ArgumentNullException("context");

    decimal discount = context.IsInRole(Role.PreferredCustomer) ? .95m : 1;

    return price * discount;
}
```

Передаваемый по значению параметр `price` представляет собой значение, с которым должен работать метод, а `context` содержит информацию о текущем контексте операции; в данном случае это информация о текущем пользователе. Зависимость предоставляется методу вызывающим объектом. Как уже неоднократно демонстрировалось, гарантию доступности контекста остальному коду тела метода обеспечивает граничный оператор.

4.3.2. Когда следует использовать внедрение через метод

Внедрение через метод отличается от других DI-паттернов тем, что внедрение происходит не в корне композиции, а динамически при вызове. Это позволяет вызывающему объекту предоставить конкретный контекст выполнения операции, что является весьма распространенным механизмом расширения, используемым в .NET BCL (табл. 4.2).

Таблица 4.2. Преимущества и недостатки внедрения через метод

Преимущества	Недостатки
Позволяет вызывающему объекту предоставлять контекст конкретной операции. Позволяет внедрять зависимости в не созданные внутри корня композиции объекты, ориентированные на данные	Ограниченная применимость. Заставляет зависимость превращаться в часть открытого API класса или его абстракции

Обычно внедрение через метод применяется в двух случаях:

- когда потребитель внедренной зависимости меняется при каждом вызове;
- когда внедряемая зависимость меняется при каждом обращении к потребителю.

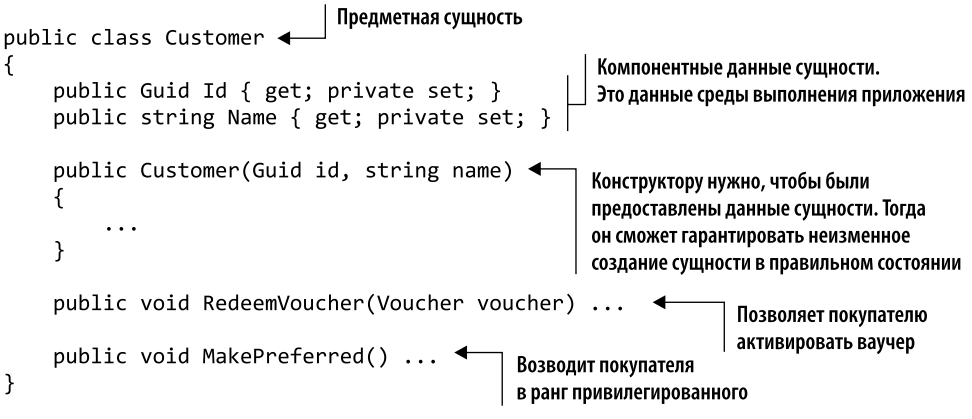
В следующих подразделах показаны примеры каждого из этих случаев. Код в листинге 4.9 является примером того, как меняется потребитель. Эта форма получила более широкое распространение, поэтому сначала будет представлен еще один подобный пример.

Пример: изменение потребителя зависимости при каждом вызове метода

Когда применяется предметно-ориентированное проектирование (Domain-Driven Design, DDD), нередко создаются предметные сущности, содержащие доменную логику, где в одном и том же классе происходит, по сути, эффективное смешивание данных среды выполнения с поведением[1]. А сущности, как правило, внутри корня композиции не создаются. Возьмем, к примеру, следующую сущность покупателя `Customer` (листинг 4.10).

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011.

Листинг 4.10. Сущность, содержащая доменную логику, но пока не имеющая зависимостей




```
public class Customer
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }

    public Customer(Guid id, string name)
    {
        ...
    }

    public void RedeemVoucher(Voucher voucher) ...

    public void MakePreferred() ...
}
```

`RedeemVoucher` и `MakePreferred` в листинге 4.10 относятся к доменным методам. В `RedeemVoucher` выполняется доменная логика, позволяющая покупателю активировать ваучер. (Возможно, и вы при покупке данной книги получили скидку, активировав ваучер.) `voucher` является объектом-значением, используемым методом. А в `MakePreferred` реализуется доменная логика, возводящая покупателя в особый ранг. Постоянный покупатель может быть возведен в статус привилегированного, что может дать ему определенные преимущества и скидки наподобие тех, что получают клиенты авиакомпаний, часто пользующиеся их услугами.

Сущности, содержащие поведение, помимо их обычного набора компонентных данных, могут легко получить широкий спектр методов, каждому из которых потребуются свои собственные зависимости. Хотя для внедрения таких зависимостей может возникнуть соблазн использовать внедрения через конструктор, это приведет к ситуации, при которой каждую сущность придется создавать со всеми ее зависимостями, даже если для конкретного случая ее использования понадобятся только некоторые из них. Тем самым усложнится тестирование логики сущности, поскольку конструктору необходимо будет предоставить все зависимости, даже если тест будет заинтересован всего лишь в некоторых из них. Как показано в листинге 4.11, внедрение через метод предлагает лучшую альтернативу.

Листинг 4.11. Сущность, использующая внедрение через метод

```
public class Customer
{
    public Guid Id { get; private set; }
    public string Name { get; private set; }

    public Customer(Guid id, string name)
    {
        ...
    }
```

```
    public void RedeemVoucher(
        Voucher voucher,
        IVoucherRedemptionService service)
    {
        if (voucher == null)
            throw new ArgumentNullException("voucher");
          if (service == null)
            throw new ArgumentNullException("service");

        service.ApplyRedemptionForCustomer(
            voucher,
            this.Id);
    }

    public void MakePreferred(IEventHandler handler)
    {
        if (handler == null)
            throw new ArgumentNullException("handler");

        handler.Publish(new CustomerMadePreferred(this.Id));
    }
}
```

За счет использования внедрения через метод в обоих доменных методах сущности, RedeemVoucher и MakePreferred, принимаются нужные зависимости — IVoucherRedemptionService и IEventHandler. Проверяются параметры и используется предоставляемая IEventHandler

Как показано далее, метод `RedeemVoucher`, принадлежащий `Customer`, в компоненте `CustomerServices` может быть вызван одновременно с передачей зависимости `IVoucherRedemptionService` (листинг 4.12).

Листинг 4.12. Компонент, использующий внедрение через метод для передачи зависимости

```
public class CustomerServices : ICustomerServices
{
    private readonly ICustomerRepository repository;
    private readonly IVoucherRedemptionService service;

    public CustomerServices(
        ICustomerRepository repository,
        IVoucherRedemptionService service)
    {
        this.repository = repository;
        this.service = service;
    }

    public void RedeemVoucher(
        Guid customerId, Voucher voucher)
    {
        var customer =
            this.repository.GetById(customerId);

        customer.RedeemVoucher(voucher, this.service);
        this.repository.Save(customer);
    }
}
```

Класс CustomerServices использует внедрение через конструктор для статического определения нужных ему зависимостей. Одной из таких зависимостей является IVoucherRedemptionService

Зависимость IVoucherRedemptionService передается уже созданной сущности Customer с помощью внедрения через метод. Customer создается внутри реализации ICustomerRepository

В листинге 4.12 из `ICustomerRepository` запрашивается только один экземпляр `Customer`. Но один и тот же экземпляр `CustomerServices` может вызываться снова и снова, используя множество покупателей и ваучеров, из-за чего одна и та же зависимость `IVoucherRedemptionService` предоставляется многим различным экземплярам `Customer`. Объект `Customer` является потребителем зависимости `IVoucherRedemptionService`. При многократном использовании зависимости происходит варьирование покупателя.

Это похоже на первый пример внедрения через метод, показанный в листинге 4.9, и на метод `ApplyDiscountFor`, рассмотренный в листинге 3.8. Бывает и наоборот, когда при сохранении покупателей меняется зависимость.

Пример: изменение внедренной зависимости при каждом вызове метода

Представим себе надстройку над приложением, рисующим графику, где всем предоставляется возможность подключения своих собственных изобразительных эффектов. Внешнему изобразительному эффекту может понадобиться информация о контексте среды выполнения, который может быть передан приложением для получения изобразительного эффекта. Это типичный вариант использования внедрения через метод. Для применения таких эффектов можно воспользоваться следующим интерфейсом:

Абстракция надстроек, представляющих изобразительные эффекты. Эти эффекты могут быть встроены в приложение путем реализации данной абстракции

```
public interface IImageEffectAddIn
{
    Bitmap Apply(
        Bitmap source,
        IApplicationContext context);
}
```

Позволяет надстройке применять ее эффект к источнику, а затем возвращает новое растровое изображение с примененным эффектом

Предоставляет контекстную информацию изобразительному эффекту за счет того, что графическое приложение применяет внедрение через метод

Принадлежащая `IImageEffectAddIn` зависимость `IApplicationContext` может изменяться с каждым вызовом метода `Apply`, предоставляя эффекту информацию о контексте, в котором вызывается операция. В качестве надстройки может использоваться любой класс, реализующий этот интерфейс. Некоторые реализации могут вообще не беспокоиться насчет контекста, а некоторые могут проявлять к нему интерес.

Как показано в листинге 4.13, для возвращения совокупного результата клиент может воспользоваться целым списком надстроек, вызывая каждую из них с исходным растровым изображением `Bitmap` и с контекстом.

Листинг 4.13. Пример клиента надстройки

```
public Bitmap ApplyEffects(Bitmap source)
{
    if (source == null) throw new ArgumentNullException("source");

    Bitmap result = source;

    foreach (IImageEffectAddIn effect in this.effects)
    {
        result = effect.Apply(result, this.context);
    }

    return result;
}
```

Приватное поле `effects` является списком экземпляров `IImageEffectAddIn`, позволяющих клиенту проходить по списку для вызова метода `Apply` каждой надстройки. При каждом вызове метода `Apply` в отношении надстройки контекст операции, представленный полем `context`, передается в качестве параметра метода:

```
result = effect.Apply(result, this.context);
```

Иногда значение и операционный контекст заключаются в одну абстракцию, работающую в качестве комбинации того и другого. Здесь важно отметить следующее: как уже было показано в обоих примерах, зависимость, внедренная через метод, становится частью определения абстракции. Обычно это востребовано в том случае, когда зависимость содержит информацию о среде выполнения, предоставляемую непосредственно вызывающими ее объектами.

В тех случаях, когда зависимость является для вызывающего объекта деталью реализации, нужно попытаться не допустить «загрязнения» абстракции; поэтому лучше будет остановить свой выбор на внедрении через конструктор. В противном случае все это может просто закончиться передачей зависимости из верхней части графа объектов приложения в самый низ, вызывая радикальные изменения.

Во всех предыдущих примерах использование внедрения через метод показывалось вне корня композиции. Это было сделано намеренно. Внутри корня композиции внедрение через метод не применяется. В корне композиции внедрение через метод может инициализировать ранее созданный класс с его зависимостями. Но подобное действие приводит к временной связанности (Temporal Coupling), и поэтому оно не приветствуется.

Проблемный код с временной связанностью

Временная связанность является типичной проблемой в проектировании API. Она возникает, когда существует подразумеваемая связь между двумя и более компонентами класса, требующая, чтобы клиенты вызывали один компонент перед другим. Это образует временную сильную связанность компонентов. Классическим

примером может послужить использование метода `Initialize`, хотя массу других примеров можно найти даже в библиотеке BCL. Например, подобное использование `System.ServiceModel.EndpointAddressBuilder` пройдет компиляцию, но даст сбой в ходе выполнения программы:

```
var builder = new EndpointAddressBuilder();
var address = builder.ToEndpointAddress();
```

Получается, что перед созданием `EndpointAddress` нужен URI. Следующий код пройдет компиляцию и будет успешно выполнен:

```
var builder = new EndpointAddressBuilder();
builder.Uri = new UriBuilder().Uri;
var address = builder.ToEndpointAddress();
```

В API нет никаких намеков на то, что это необходимо, но между свойством `Uri` и методом `ToEndpointAddress` существует временная связанность.

Как показано в листинге 4.14, *повторяющимся паттерном* при применении внутри корня композиции является использование важного метода `Initialize`.

Листинг 4.14. Пример временной связанности

```
public class Component
{
    private ISomeInterface dependency;

    public void Initialize(
        ISomeInterface dependency)
    {
        this.dependency = dependency;
    }

    public void DoSomething()
    {
        if (this.dependency == null)
            throw new InvalidOperationException(
                "Call Initialize first.");

        this.dependency.DoStuff();
    }
}
```

Методы Initialize и DoSomething нужно вызывать в определенном порядке, но эта взаимосвязь является подразумеваемой. В результате получается временная связанность

Возможность вызова DoSomething до вызова Initialize заставляет добавить эту дополнительную контрольную инструкцию, необходимую каждому открытому методу данного класса

Семантически название метода `Initialize` является подсказкой, но на структурном уровне данный API не дает никаких указаний на временную связанность. Поэтому код, похожий на этот, проходит компиляцию, но при выполнении выдает исключение:

```
var c = new Component();
c.DoSomething();
```

Теперь уже решение данной проблемы должно быть очевидным — вместо данного приема нужно применить внедрение через конструктор:

```
public class Component
{
    private readonly ISomeInterface dependency;

    public Component(ISomeInterface dependency)
    {
        if (dependency == null)
            throw new ArgumentNullException("dependency");

        this.dependency = dependency;
    }

    public void DoSomething()
    {
        this.dependency.DoStuff();
    }
}
```

ВНИМАНИЕ

Не сохраняйте зависимости, внедренные через метод. Это приводит к временной связанности, захваченным зависимостям или скрытым побочным эффектам[1]. Метод должен воспользоваться зависимостью или передать ее и не должен сохранять такую зависимость. Использование внедрения зависимости через метод встречается в .NET BCL довольно часто, поэтому обратимся к следующему примеру.

4.3.3. Известные примеры применения внедрения через метод

В .NET BCL, в частности в пространстве имен `System.ComponentModel`, предоставляется множество примеров внедрения через метод. Для реализации настраиваемой во время разработки функциональности компонентов используется средство разработки `System.ComponentModel.Design.IDesigner`. Там применяется метод `Initialize`, получающий экземпляр `IComponent`, благодаря чему становится известно, разработке какого компонента в данный момент оказывается помощь. (Следует учесть, что применение данного метода `Initialize` становится причиной временной связанности.) Средства разработки создаются реализациями `IDesignerHost`, которые также принимают экземпляры `IComponent` в качестве параметров:

```
IDesigner GetDesigner(IComponent component);
```

Это неплохой пример сценария, где информация переносится самим параметром. Компонент может переносить информацию о том, какой `IDesigner` создавать, но в то же время он является компонентом, над которым средство разработки

[1] Захваченные зависимости будут рассмотрены в подразделе 8.4.1.

должно впоследствии работать. Еще один пример в пространстве имен `System.ComponentModel` предоставляется классом `TypeConverter`. Некоторые из его методов берут экземпляр `ITypeDescriptorContext`, который согласно своему названию, передает информацию о контексте текущей операции, например сведения о свойствах типа. Поскольку таких методов там много, перечислять все нет смысла, но образцом может послужить следующий метод:

```
public virtual object ConvertTo(ITypeDescriptorContext context,
    CultureInfo culture, object value, Type destinationType)
```

В этом методе контекст операции передается в явном виде параметром `context`, а вот значение, предназначенное для преобразования, и целевой тип переданы в качестве отдельных параметров. Разработчики могут использовать или игнорировать контекстный параметр по своему усмотрению. Среда ASP.NET Core MVC также содержит несколько экземпляров внедрения зависимостей через метод. Можно, к примеру, воспользоваться интерфейсом `IValidationAttributeAdapterProvider`, чтобы предоставить экземпляры `IAttributeAdapter`. Его единственный метод имеет следующий вид:

```
IAttributeAdapter GetAttributeAdapter(
    ValidationAttribute attribute, IStringLocalizer stringLocalizer)
```

Среда ASP.NET Core позволяет устанавливать для свойств представления маркер `ValidationAttribute`. Это является удобным способом применения метаданных с описанием допустимости свойств, заключенных в модели представления.

Основываясь на `ValidationAttribute`, метод `GetAttributeAdapter` позволяет возвращать `IAttributeAdapter`, который дает возможность выводить на вебстраницу надлежащие сообщения об ошибках. Параметр `attribute` в методе `GetAttributeAdapter` является объектом, для которого должен быть создан `IAttributeAdapter`, а `stringLocalizer` является зависимостью, которая внедряется через метод.

ПРИМЕЧАНИЕ

> Рекомендуя сделать своим предпочтительным DI-паттерном внедрение зависимостей через конструктор, мы исходим из того, что вами в основном создаются приложения. С другой стороны, если вами создается среда выполнения, может пригодиться и внедрение зависимостей через метод, поскольку оно позволяет среде передавать надстройкам информацию о контексте. Это одна из причин широкого использования внедрения через метод в BCL. Но внедрение через метод может пригодиться и при разработке кода приложений.

Далее мы увидим, как Мэри использует внедрение через метод, чтобы избавиться от повторений в коде. Когда мы последний раз встречались с Мэри (в разделе 4.2), она работала над `ICurrencyConverter`: внедряла его через конструктор в класс `ProductService`.

4.3.4. Пример: добавление конвертации валюты в сущность товаров Product

В листинге 4.8 было показано, как метод `GetFeaturedProducts` вызывал метод `ICurrencyConverter.Exchange`, используя принадлежащую товару цену за единицу — `UnitPrice` и предпочитаемую покупателем валюту в приложении, разрабатываемом Мэри. Посмотрим на метод `GetFeaturedProducts` еще раз:

```
public IEnumerable<DiscountedProduct> GetFeaturedProducts()
{
    Currency currency = this.userContext.Currency;

    return
        from product in this.repository.GetFeaturedProducts()
        let amount = this.converter.Exchange(product.UnitPrice, currency)
        select product
            .WithUnitPrice(amount)
            .ApplyDiscountFor(this.userContext);
}
```

Преобразование сущностей товара `Product` из одной валюты `Currency` в другую будет повторяющейся задачей во многих частях ее приложения. По этой причине Мэри склонна перенести логику преобразования `Product` из `ProductService` и сосредоточить ее в одном месте, сделав частью сущности `Product`. Это позволит избавить другие части системы от повторения этого кода. Лучшего кандидата, чем внедрение через метод, найти для этого будет трудно. Мэри создает в `Product` новый метод `ConvertTo`, показанный в листинге 4.15.

Листинг 4.15. Сущность Product с методом ConvertTo

```
public class Product
{
    public string Name { get; set; }
    public Money UnitPrice { get; set; }
    public bool IsFeatured { get; set; }

    public Product ConvertTo(
        Currency currency,
        ICurrencyConverter converter)
    {
        if (currency == null)
            throw new ArgumentNullException("currency");
        if (converter == null)
            throw new ArgumentNullException("converter");

        var newUnitPrice =
            converter.Exchange(
                this.UnitPrice,
                currency);
```

Метод ConvertTo принимает значение Currency

Теперь зависимость ICurrencyConverter внедряется через метод

Новая цена единицы товара определяется с помощью вызова метода Exchange

```
        return this.WithUnitPrice(newUnitPrice);
    }

    public Product WithUnitPrice(Money unitPrice)
    {
        return new Product
        {
            Name = this.Name,
            UnitPrice = unitPrice,
            IsFeatured = this.IsFfeatured
        };
    }
    ...
}
```

Новый экземпляр Product создан на основе исходного Product, где UnitPrice заменяется заново созданной ценой за единицу товара

Мэри выполняет перепроектировку метода `GetFeaturedProducts` с помощью нового метода `ConvertTo` (листинг 4.16).

Листинг 4.16. GetFeaturedProducts с использованием метода ConvertTo

```
public IEnumerable<DiscountedProduct> GetFeaturedProducts()
{
    Currency currency = this.userContext.Currency;

    return
        from product in this.repository.GetFeaturedProducts()
        select product
            .ConvertTo(currency, this.converter)
            .ApplyDiscountFor(this.userContext);
}
```

Теперь ICurrencyConverter предоставляется путем внедрения через метод

Вместо ранее показанного вызова метода `ICurrencyConverter.Exchange` теперь `GetFeaturedProducts` передает `ICurrencyConverter` методу `ConvertTo` с помощью внедрения через метод. Это упрощает метод `GetFeaturedProducts` и предотвращает дублирование любого кода, когда Мэри требуется выполнить преобразование товара в каком-либо другом месте ее кодовой базы. За счет использования внедрения через метод вместо внедрения через конструктор она избавляется от потребности создания сущности `Product` со всеми ее зависимостями. Тем самым упрощаются разработка и тестирование.

ПРИМЕЧАНИЕ

Хотя определение ICurrencyConverter уже приводилось в разделе 4.2, порядок реализации класса ICurrencyConverter еще не рассматривался, поскольку для всего, что касается внедрения через метод или внедрения через конструктор, это неважно. Если вам интересно посмотреть на его реализацию, она доступна в исходном коде, сопровождающем эту книгу.

В отличие от других DI-паттернов, рассматриваемых в этой главе, внедрение через метод используется главным образом при возникновении потребности в предоставлении зависимостей уже существующему потребителю. За счет внедрения через конструктор и внедрения через свойство зависимости предоставляются потребителю во время его создания.

Последним паттерном, рассматриваемым в этой главе, будет внедрение зависимостей через свойство, позволяющее переопределять локальную реализацию по умолчанию (Local Default) класса. Внедрение через метод применялось только за пределами корня композиции, а внедрение через свойство, как и внедрение через конструктор, используется в пределах корня композиции.

4.4. Внедрение через свойство

Как можно активировать внедрение зависимостей в качестве возможного варианта в классе с удачной локальной реализацией по умолчанию?

Предоставив доступное для записи свойство, позволяющее вызывающим объектам предоставлять зависимость, если им нужно переопределить поведение по умолчанию.

Когда у класса удачная реализация по умолчанию, но все же хочется оставить его открытым для возможного последующего расширения, можно предоставить доступное для записи свойство, позволяющее клиенту предоставлять другую, отличную от реализации по умолчанию, реализацию зависимости класса. Как показано на рис. 4.8, клиенты, желающие воспользоваться классом `Consumer` в его исходном виде, могут создать экземпляр класса и использовать его без лишних размышлений, а вот клиенты, желающие изменить поведение класса, могут добиться этого, установив свойство зависимости на другую реализацию `IDependency`.

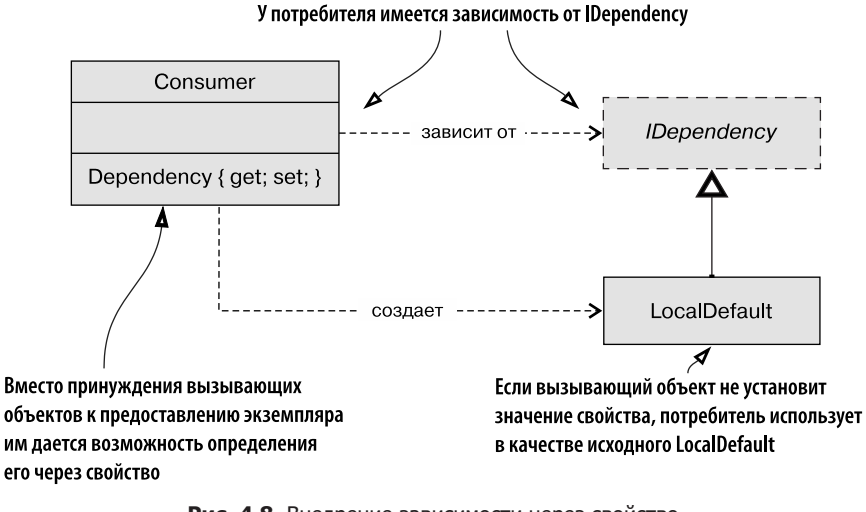


Рис. 4.8. Внедрение зависимости через свойство

ОПРЕДЕЛЕНИЕ

Внедрение зависимостей через свойство (Property Injection) позволяет заменять локальную реализацию по умолчанию с помощью свойства, открытого для установки его значения. Внедрение через свойство также известно как внедрение через метод записи значения (Setter Injection).

4.4.1. Принцип работы внедрения через свойство

Класс, использующий зависимость, должен предоставить открытое, доступное для записи свойство типа зависимости. В элементарной реализации все может быть так же просто, как в листинге 4.17.

Листинг 4.17. Внедрение зависимости через свойство

```
public class Consumer
{
    public IDependency Dependency { get; set; }
}
```

`Consumer` зависит от `IDependency`. Клиенты могут предоставлять реализации `IDependency`, устанавливая значение свойства `Dependency`.

ПРИМЕЧАНИЕ

В отличие от внедрения через конструктор помечать поле, на котором основано свойство, доступным только для чтения (readonly) нельзя, поскольку вызывающим объектам позволяется изменять свойство в любой момент жизни потребителя.

Все остальные компоненты зависимого класса могут воспользоваться внедренной зависимостью для работы по своему предназначению, например:

```
public void DoSomething()
{
    this.Dependency.DoStuff();
}
```

К сожалению, такая реализация не отличается особой надежностью. Дело в том, что свойство `Dependency` не гарантирует возвращение экземпляра `IDependency`. Подобный код выдаст исключение `NullReferenceException`, если значением свойства `Dependency` будет `null`:

```
var instance = new Consumer();

instance.DoSomething();
```

Этот вызов приведет к выдаче исключения, поскольку мы забыли установить значение для instance.Dependency

Эту проблему можно решить, позволив конструктору установить для свойства в его методе записи значение по умолчанию в сочетании с граничным оператором. Еще одно осложнение возникает, если клиенты переключают зависимость в какой-то из моментов жизни класса:

```
var instance = new Consumer();

instance.Dependency = new SomeImplementation();

instance.DoSomething();

instance.Dependency = new SomeOtherImplementation();

instance.DoSomething();
```

Установка для свойства Dependency допустимой реализации

Изменение свойства Dependency в период времени существования класса, что может стать причиной проблем для Consumer

Решить проблему можно за счет введения внутреннего флага, позволяющего клиенту устанавливать значение свойства `Dependency` только один раз в ходе инициализации[1].

Способ избавления от подобных осложнений показан в примере подраздела 4.4.4. Но прежде, чем до него добраться, мы хотим объяснить, когда будет уместно воспользоваться внедрением через свойство.

4.4.2. Когда следует использовать внедрение через свойство

Пользоваться внедрением через свойство следует только в том случае, когда разрабатываемый класс имеет удачную локальную реализацию по умолчанию, а вам все-таки нужно позволить вызывающим объектам предоставлять различные реализации зависимости класса. Важно отметить, что внедрением через свойство лучше воспользоваться, когда зависимость является необязательной. Если зависимость будет обязательной, безусловно, лучше будет воспользоваться внедрением через конструктор.

В главе 1 мы рассмотрели веские доводы для написания кода со слабой связанностью, позволяющей изолировать модули друг от друга. Но слабая связанность может также вполне успешно применяться к классам внутри одного и того же модуля. Зачастую это делается путем ввода в модуль абстракций и разрешения классам внутри этого модуля обмениваться данными через абстракции, вместо того чтобы быть тесно связанными друг с другом. В основном слабая связанность в границах модуля применяется для того, чтобы классы были открыты для их расширения и не создавали препятствий при тестировании.

---

[1] Эрик Липперт (Eric Lippert) называет это popsicle immutability. (Термин на сегодняшний день не имеет русского аналога. Мне кажется, более точно будет перевести как «неизменяемость через заморозку». Суть в том, что popsicle (фруктовый лед) создается методом замораживания готового продукта вокруг палочки и во время замораживания в продукт еще можно внести изменения. Это его отличает от обычного мороженого, когда требуемый состав определяется раз и навсегда на самом старте процесса производства. — *Примеч. науч. ред.*) См.: *Lippert E.* Immutability in C# Part One: Kinds of Immutability, 2007, https://mng.bz/y2Eq/.

ПРИМЕЧАНИЕ

Понятие открытости класса для его расширения отражено в принципе открытости/закрытости (Open/Closed Principle), который утверждает, что класс должен быть открыт для его расширения, но закрыт для изменения. Реализацию классов в соответствии с принципом открытости/закрытости можно вести, подраумевая некую локальную реализацию по умолчанию, но при этом все же предоставлять клиентам возможность расширения класса путем замены одной зависимости на другую.

Принцип открытости/закрытости

У сущностей программного кода (классов, модулей, функций и т. д.), соответствующих принципу открытости/закрытости, имеется два основных признака.

- *Они открыты для расширения*. Это означает возможность внесения изменений или расширения в поведение такой сущности. Возможно, само по себе это утверждение никого не впечатлит, поскольку если предположить, что ваша команда владеет всем приложением, то вы всегда сможете изменить поведение какой-либо части системы. Просто переходите к исходному коду и вносите в него изменения. Но интерес к этому признаку возникает в контексте следующего.
- *Они закрыты для изменения*. Это означает, что при расширении системы должна быть возможность справиться с этой задачей, не затрагивая уже существующий исходный код. Это утверждение может показаться весьма странным; как модифицировать систему, не имея возможности внести изменения в ее исходный код?

Важная часть ответа на данное противоречие предоставляется самой технологией DI. Она позволяет вам заменять или перехватывать классы, чтобы добавить или изменить поведение так, что об этом не будет известно ни потребляющему классу, ни его зависимости. Принцип открытости/закрытости подталкивает к созданию конструкции, в которой каждое требование новой функциональности может разрешаться путем создания одного или нескольких новых классов или модулей, не затрагивая уже существующие классы или модули.

Когда возможность добавления к системе новых функциональных или нефункциональных требований не требует вмешательства в уже существующие части, это означает, что решаемая задача изолирована от других частей системы. В результате получается более легкий для понимания и тестирования код, который в силу этого становится проще сопровождать. Хотя возможность расширения системы без потребности в изменении существующего кода является тем самым идеалом, к которому стоит стремиться, он все же недостижим. Всегда найдутся обстоятельства, при которых придется вносить изменения в какие-нибудь уже существующие части системы.

Как разработчик, вы должны выяснить, какого рода изменения скорее всего произойдут в приложении. Основываясь на понимании ожидаемых путей развития конкретного приложения или системы, нужно смоделировать его таким образом, чтобы достичь наибольшей сопровождаемости. Важным аспектом приближения к этому идеалу является предотвращение регулярных радикальных изменений системы.

Работа с абстракциями — одна из основных тем этой книги, и это еще не все. В главах 9 и 10 будут рассмотрены некоторые технологии, позволяющие помочь сделать приложения открытыми для расширений, но закрытыми для изменений.

ПРИМЕЧАНИЕ

Принцип открытости/закрытости тесно связан с принципом DRY[1].

СОВЕТ

Иногда нужно только предоставить точку расширения, оставив вместо локальной реализации по умолчанию пустую операцию (no-op)[2]. В таких случаях в качестве локальной реализации по умолчанию можно воспользоваться паттерном «Нулевого объекта» (Null Object).

Пока никаких настоящих примеров внедрений через свойство показано не было, поскольку этот паттерн имеет еще более узкую область применения, чем другие паттерны, особенно в контексте разработки приложений (табл. 4.3).

Таблица 4.3. Преимущества и недостатки внедрения через свойство

Преимущества	Недостатки
Простое для понимания	Не всегда просто добиться надежной реализации. Ограниченная область применения. Применимо только для многократно используемых библиотек. Вызывает появление временной связанности

Главное преимущество внедрения через свойство состоит в простоте его понимания. Зачастую применение данного паттерна встречается в виде первой попытки освоения технологии DI.

Но внешняя простота бывает обманчива, и внедрение через свойство сопряжено с рядом сложностей. Реализовать надежность применения данного паттерна весьма непросто. Клиенты могут забыть предоставить зависимость по причине ранее рассмотренной проблемы временной связанности. А что, если, кроме всего прочего, клиент предпримет попытку изменить зависимость в период времени существования класса? Это может привести к противоречивому или неожиданному поведению, поэтому вы можете захотеть защитить себя от такого развития событий.

Несмотря на недостатки, имеет смысл применять внедрение через свойство при создании многократно используемых библиотек. Это позволяет компонентам определять рациональные варианты реализации по умолчанию, упрощая таким образом работу с API библиотеки.

---

[1] Принцип DRY — Don't Repeat Yourself («Не повторяйся») — утверждает, что «каждая часть знания должна иметь в системе единственное, однозначное и общепринятое представление».

[2] NOP, no-op и NOOP представляют собой сокращения от no operation («отсутствие операции»). Это инструкция языка ассемблера, которая ничего не делает. Понятие no-op в компьютерной науке стало обозначением операции, которая ничего не делает.

ПРИМЕЧАНИЕ

Но при создании приложений внедрение через свойство нами не используется, и вы не должны этим особо увлекаться. Даже если у вас есть локальная реализация зависимости по умолчанию, лучшей альтернативой все же будет внедрение через конструктор. Оно проще и надежнее. Может создаться впечатление, что внедрение через свойство пригодится для обхода зацикленных зависимостей, но, исходя из объяснений, приведенных в главе 6, такой код можно отнести к проблемным.

При разработке приложений ваши классы подключаются к корню композиции. Внедрение через конструктор не позволяет вам забыть о предоставлении зависимости. Даже при наличии локальной реализации по умолчанию такие экземпляры могут предоставляться конструктору корнем композиции. Это упрощает класс и позволяет корню композиции контролировать значение, получаемое всеми потребителями. Это может даже быть реализация нулевого объекта.

СОВЕТ

Избегайте использования внедрения через свойство в качестве решения проблемы избыточности внедрения через конструктор. Классы, имеющие множество зависимостей, считаются неудачным кодом, и внедрение через свойство не упростит конструкцию класса. Избыточность внедрения через конструктор будет рассмотрена в разделе 6.1.

Наличие хорошей локальной реализации по умолчанию частично зависит от степени детализации модулей. Библиотека BCL поставляется в виде довольно-таки большого пакета; пока реализация по умолчанию находится внутри BCL, можно утверждать, что она также локальна. В следующем разделе мы вкратце коснемся этой темы.

4.4.3. Известные примеры использования внедрения через свойство

В библиотеке .NET BCL внедрение через свойство встречается чаще, чем внедрение через конструктор, потому, вероятно, что весьма качественные локальные реализации по умолчанию определены во многих местах, а также потому, что тем самым упрощается создание экземпляров по умолчанию большинства классов. Например, в `System.ComponentModel.IComponent` имеется открытое для записи свойство `Site`, позволяющее вам определять экземпляр `ISite`. Главным образом эта возможность используется в сценариях времени разработки (например, в среде Visual Studio) для изменения или усовершенствования компонента, когда он размещен в конструкторе. Теперь перейдем к более содержательному примеру использования и реализации внедрения через свойство.

4.4.4. Пример: внедрение через свойство, применяемое в качестве модели расширяемости многократно используемой библиотеки

В ранее приведенных примерах расширялось учебное приложение из предыдущей главы. Хотя можно было бы показать вам пример внедрения через свойство, используя учебное приложение, нам не хочется вводить вас в заблуждение, поскольку внедрение через свойство вряд ли когда-либо подойдет для разработки приложений — практически всегда более удачным выбором будет внедрение через конструктор. Вместо этого отдадим предпочтение демонстрации примера многократно используемой библиотеки. В этом случае будет рассмотрен фрагмент кода из Simple Injector.

Simple Injector является одним из DI-контейнеров, рассматриваемых в части IV настоящей книги. Он помогает создавать граф объектов приложения. Более развернуто Simple Injector будет рассмотрен в главе 14, поэтому здесь углубляться в подробности мы не будем. Исходя из задач внедрения через свойство, нам неважен принцип работы Simple Injector.

Будучи многократно используемой библиотекой, Simple Injector обращается к применению внедрения через свойство довольно часто. Расширению может подвергаться существенная часть его поведения, и используемый для этого способ основан на предоставлении реализаций его поведения по умолчанию. Simple Injector предоставляет свойства, позволяющие пользователю изменять реализацию по умолчанию. Одной из особенностей поведения, замену которой разрешает Simple Injector, является способ выбора библиотекой нужного конструктора для выполнения внедрения через конструктор[1]. В разделе 4.2 уже обсуждалось, что у класса должен быть только один конструктор. Поэтому изначально Simple Injector допускает только применение классов, имеющих всего один создаваемый открытый конструктор. В любом другом случае Simple Injector выдает исключение. Но Simple Injector позволяет вам переопределять это поведение. Эта возможность может пригодиться для конкретных узких интеграционных сценариев. Для этого в Simple Injector определяется интерфейс `IConstructorResolutionBehavior`[2]. Пользователь может определить специальную реализацию, и предоставляемые библиотекой реализации по умолчанию могут, как здесь показано, заменяться путем установки свойства `ConstructorResolutionBehavior`:

```
var container = new Container();

container.Options.ConstructorResolutionBehavior =
    new CustomConstructorResolutionBehavior();
```

---

[1] Чтобы объяснить порядок внедрения через свойство, в этом примере используется присущая DI-контейнеру возможность внедрения через конструктор. Но не стоит переживать: в примере демонстрируются свойство с локальной реализацией по умолчанию и два граничных оператора.

[2] Это реализация паттерна «Стратегия».

`Container` является центральной реализацией паттерна «Фасад»[1] в API Simple Injector. Он используется для определения взаимоотношений между абстракциями и реализациями и для построения графа объектов этих реализаций. Класс включает в себя свойство `Options` типа `ContainerOptions`. Оно содержит в себе ряд свойств и методов, позволяющих изменять поведение библиотеки по умолчанию. Одним из этих свойств является `ConstructorResolutionBehavior`. Упрощенная версия класса `ContainerOptions` с его свойством `ConstructorResolutionBehavior` имеет следующий вид[2]:

```
public class ContainerOptions
{
    IConstructorResolutionBehavior resolutionBehavior =
        new DefaultConstructorResolutionBehavior();

    public IConstructorResolutionBehavior ConstructorResolutionBehavior
    {
        get
        {
            return this.resolutionBehavior;
        }
        set
        {
            if (value == null)
                throw new ArgumentNullException("value");

            if (this.Container.HasRegistrations)
            {
                throw new InvalidOperationException(
                    "The ConstructorResolutionBehav" +
                    "ior property cannot be changed" +
                    " after the first registration " +
                    "has been made to the container.";
            }

            this.resolutionBehavior = value;
        }
    }
}
```

Назначение приватному полю resolutionBehavior локальной реализации по умолчанию DefaultConstructorResolutionBehavior

Контрольная инструкция, проверяющая на наличие значения null

Контрольная инструкция с вариацией рассматриваемого внутреннего флага, обеспечивающего неизменяемость через заморозку

Сохранение входящей зависимости в приватном поле, позволяющее переопределить локальную реализацию по умолчанию

Если в контейнере нет регистраций, свойство `ConstructorResolutionBehavior` может быть изменено несколько раз. Это важно, поскольку, когда регистрация уже проведена, Simple Injector использует указанный объект `ConstructorResolutionBehavior`

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 221.

[2] Как уже утверждалось в сноске в подразделе 4.4.1, неизменяемость через заморозку позволяет клиенту устанавливать зависимость в ходе инициализации.

для проверки возможности создания такого типа путем анализа конструктора класса. Если бы пользователь изменил поведение по разрешению конструктора после выполнения регистрации, он смог бы повлиять на корректность предыдущих регистраций.

Дело в том, что в противном случае Simple Injector мог бы использовать для компонента другой, нежели одобренный им при регистрации, конструктор. Это означает, что либо должны быть заново вычислены все предыдущие регистрации, либо после проведения регистрации пользователь должен быть лишен возможности изменить поведение. Поскольку вычисление заново может повлечь за собой скрытое снижение производительности и его труднее реализовать, Simple Injector реализует последний подход.

По сравнению с внедрением через конструктор внедрение через свойство является более запутанным. В своем примитивном виде оно может выглядеть простым, но при должной реализации оно в большинстве случаев намного сложнее.

Внедрение через свойство применяется в многократно используемой библиотеке, где зависимость носит необязательный характер и имеется ее подходящая локальная реализация по умолчанию. В тех случаях, когда есть недолговечный объект, требующий зависимость, нужно применять внедрение через метод. В остальных случаях следует использовать внедрение через конструктор.

На этом завершается рассмотрение последнего паттерна в данной главе. В следующем разделе приводится краткое напоминание и объясняется, как для решения вашей задачи выбрать правильный паттерн.

4.5. Выбор используемого паттерна

Паттерны, описанные в данной главе, являются центральной частью технологии DI. Вооружившись корнем композиции и соответствующим сочетанием паттернов внедрения, можно реализовать чистое внедрение зависимостей или же воспользоваться DI-контейнером. При использовании DI существует множество нюансов и тонкостей, которые следует освоить, но эти паттерны охватывают самую суть механики, отвечающую на вопрос: «Как мне внедрить нужные зависимости?»

Данные паттерны не взаимозаменяемы. В большинстве случаев выбором по умолчанию должен быть паттерн внедрения через конструктор, но бывают ситуации, когда более удачной альтернативой представляются другие паттерны. На рис. 4.9 показан процесс принятия решения, способный помочь подобрать подходящий паттерн, но, если вы сомневаетесь, следует выбирать внедрение через конструктор. Этот выбор будет практически безошибочным.

Первое, что нужно выяснить, — о какой зависимости идет речь: о нужной вам или о уже существующей, которую нужно передать другому сотрудничающему с вашим компоненту. В большинстве случаев это, наверное, будет зависимость, необходимая именно вам. Но в сценариях надстроек может понадобиться передать текущий

контекст надстройке. Когда зависимость меняется от операции к операции, самым подходящим кандидатом на реализацию будет внедрение через метод.

Во-вторых, требуется узнать, какой класс нуждается в зависимости. Если данные времени выполнения смешиваются с поведением в одном и том же классе, как это может быть сделано в ваших доменных сущностях, вполне подойдет внедрение через метод. В иных случаях, когда создается код приложения, а не код многократно используемой библиотеки, автоматически применяется внедрение через конструктор.

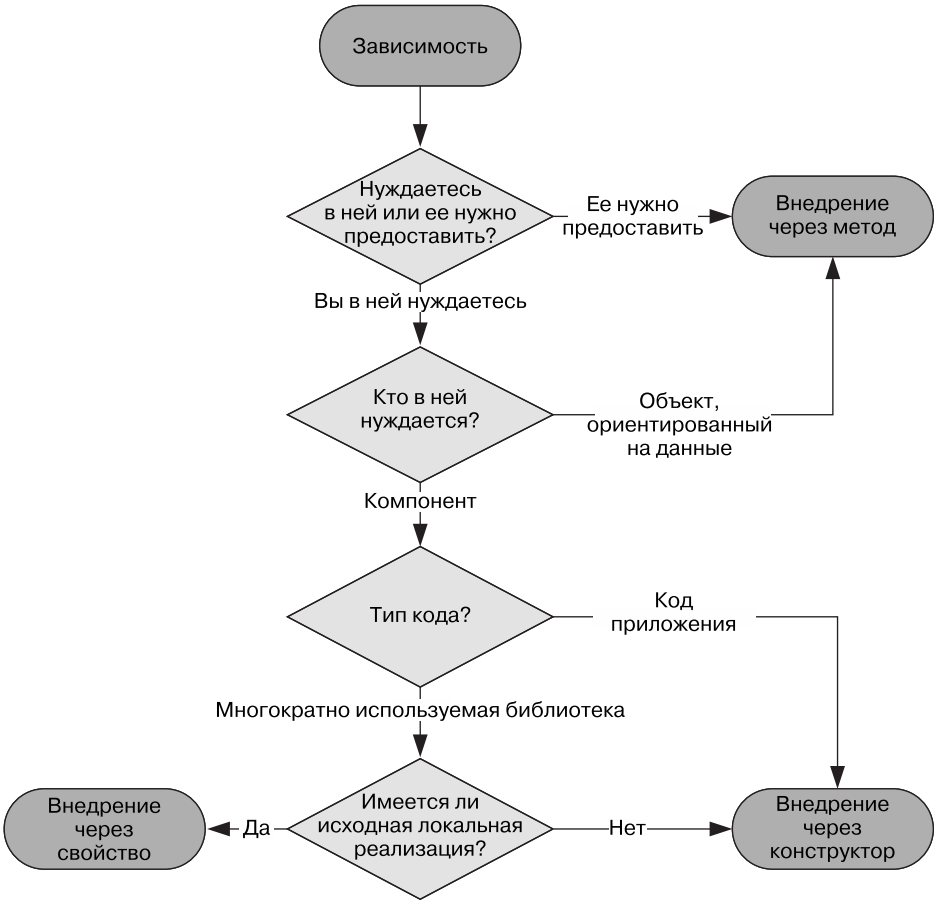


Рис. 4.9. Процесс выбора паттерна. В большинстве случаев следует выбирать внедрение через конструктор, но бывают ситуации, в которых больше подходят другие DI-паттерны

Когда дело касается написания кода приложения, нужно отказываться даже от использования локальных реализаций по умолчанию, отдавая предпочтение распо-

ложению всех установок по умолчанию в одном центральном месте приложения — в корне композиции. С другой стороны, при написании многократно используемой библиотеки применение локальной реализации по умолчанию является решающим фактором, поскольку оно дает возможность сделать явное задание зависимости необязательным — если не указана переопределяющая реализация, все берет на себя реализация по умолчанию. Этот сценарий может быть весьма эффективно реализован с внедрением через свойство.

Выбором по умолчанию для внедрения зависимостей должно быть внедрение через конструктор. Его проще будет понять и надежно реализовать, чем любые другие DI-паттерны. Применяя лишь паттерн внедрения через конструктор, можно создавать целые приложения, но сведения о других паттернах помогут в некоторых случаях поступить мудрее и выбрать не его, а другой, более подходящий паттерн. В следующей главе мы подойдем к вопросу применения DI с обратной стороны и рассмотрим неудачные способы использования DI.

Резюме

- Корень композиции является единственным логическим местом в приложении, где модули составляются вместе. Конструкция компонентов вашего приложения должна быть сосредоточена в одной его области.
- Корень композиции будет иметься только у стартовых проектов.
- Хотя корень композиции может развертываться в нескольких классах, все они должны находиться в одном и том же модуле.
- Корень композиции получает непосредственную зависимость от всех других модулей, имеющихся в системе. По сравнению с кодом с сильной связанностью код со слабой связанностью, в котором применяется паттерн корня композиции, снижает общее количество зависимостей между модулями, подсистемами и уровнями.
- Даже притом, что корень композиции может быть помещен в ту же сборку, что и пользовательский интерфейс или уровень представления, корень композиции не является частью этих уровней. Сборки являются артефактами развертывания, а уровни — логическими артефактами.
- При использовании DI-контейнера на него следует ссылаться только из корня композиции. Все остальные модули должны быть безразличны к существованию DI-контейнера.
- Использование DI-контейнера вне корня композиции приводит к применению антипаттерна «Локатор сервисов» (Service Locator).
- Обычно в качественно спроектированной системе потери производительности от использования DI-контейнера для создания больших графов объектов незначительны.

- Корень композиции должен быть единственным местом во всем приложении, хранящим сведения о структуре созданного графа объектов. Это означает, что код приложения не может передавать зависимости другим потокам, запущенным параллельно с текущей операцией, поскольку у потребителя нет возможности узнать, насколько это безопасно. Вместо этого, когда ответвляются одновременно выполняемые операции, на корень композиции возлагается создание нового графа объектов для каждой такой операции.
- Внедрение через конструктор — это статическое определение списка требуемых зависимостей путем указания их конструктору класса в качестве параметров.
- Конструктор, используемый для внедрения через конструктор, не должен делать ничего, кроме применения граничного оператора и сохранения полученных зависимостей. Вся остальная логика должна содержаться вне конструктора. Это способствует быстрому и надежному созданию графа объектов.
- Внедрение через конструктор должно быть вашим выбором по умолчанию для использования технологии DI, поскольку это наиболее надежный и простой способ ее правильного применения.
- Внедрение через конструктор хорошо подходит в тех случаях, когда зависимости носят обязательный характер. Но важно отметить, что зависимости бывают необязательными весьма редко. Необязательные зависимости усложняют компонент необходимостью проверки на `null`. Вместо этого, когда не имеется подходящей доступной реализации, в корень композиции должна внедряться реализация нулевого объекта.
- У компонентов приложения должен быть всего один конструктор. Переопределяемые конструкторы приводят к неопределенности. Для многократно используемых библиотек, подобных BCL, наличие нескольких конструкторов зачастую имеет некоторый смысл, а вот для компонентов приложения — нет.
- Внедрение через метод — это передача зависимостей при вызовах метода.
- В случаях, когда зависимость или потребитель зависимости могут меняться для каждой операции, можно применить внедрение через метод. Такое внедрение может пригодиться для сценариев надстроек, где некий контекст времени выполнения нужно передать открытому API надстройки или когда ориентированному на данные объекту требуется зависимость для выполнения конкретной операции, что часто бывает с доменными сущностями.
- Внедрение через метод не подходит для использования в корне композиции, поскольку это приводит к временной связанности.
- Метод, принимающий зависимость внедрением через метод, не должен сохранять эту зависимость. Это приводит к временной связанности, захваченным зависимостям или к скрытым побочным эффектам. Зависимости должны сохраняться только при внедрении через конструктор или свойство.
- Локальной реализацией зависимости по умолчанию (Local Default) называется ее реализация по умолчанию, созданная в том же самом модуле или на том же самом уровне.

- Внедрение через свойство позволяет библиотекам классов быть открытыми для расширения, поскольку оно позволяет вызывающим объектам изменять поведение библиотеки по умолчанию.
- Внедрение через свойство может быть простым на вид, но при должной реализации оно, как правило, оказывается более сложным по сравнению с внедрением через конструктор.
- Кроме необязательных зависимостей внутри многократно используемых библиотек, область применения внедрения через свойство весьма ограниченна, и обычно внедрение через конструктор оказывается более подходящим выбором. Внедрение через конструктор упрощает код класса, позволяя корню композиции контролировать значение, получаемое всеми потребителями, и не допускать временного связывания.

5 Антипаттерны внедрения зависимостей

В этой главе

- Создание кода с сильной связанностью при использовании антипаттерна «Диктатор» (Control Freak).
- Запрашивание зависимостей класса с применением антипаттерна «Локатор сервисов» (Service Locator).
- Придание нестабильной зависимости глобальной доступности при использовании антипаттерна «Окружающий контекст» (Ambient Context).
- Принудительное использование конкретной сигнатуры конструктора с применением антипаттерна «Ограниченная конструкция» (Constrained Construction).

Многие блюда готовятся на сковороде с маслом. Если рецепт еще не освоен, можно начать прогревать масло, отвернуться от сковороды и вчитываться в рецепт. Но, как только завершится нарезка овощей, масло начнет подгорать. Можно подумать, что дымящееся масло означает, что сковорода нагрелась и готова к приготовлению блюда. Это обычное заблуждение неопытных поваров. Когда масло начинает подгорать, оно также начинает разлагаться. Это называется достижением температуры дымообразования. Большинство подгоревших масел приобретают неприятный вкус, но это еще не все. Они образуют вредные соединения и теряют полезные антиоксиданты.

В предыдущей главе приводилось краткое сравнение паттернов проектирования с рецептами. Паттерн предоставляет понятный всем язык, которым можно воспользоваться для конструктивного обсуждения сложной концепции.

Когда концепция (или, скорее, реализация) искажается, у нас получается антипаттерн.

ОПРЕДЕЛЕНИЕ

> Под антипаттерном понимается часто встречающееся решение проблемы, порождающее явные негативные последствия, хотя наряду с ним существуют и другие доступные задокументированные решения, доказавшие свою эффективность[1].

Перегревшееся задымившее масло — типичный пример, подпадающий под понятие кулинарного антипаттерна. Это весьма распространенная ошибка. Ее допускают многие неопытные повара, поскольку им кажется, что так и нужно делать, а негативные последствия выражаются в том, что еда теряет вкус и становится вредной.

Антипаттерны в некотором смысле являются формализованным способом описания распространенных ошибок, неоднократно допускаемых людьми. В этой главе будет рассмотрен ряд наиболее распространенных антипаттернов, связанных с внедрением зависимостей. Все они в той или иной форме попадались нам в ходе профессиональной деятельности, да мы и сами порой грешили их применением.

Зачастую антипаттерны являются простыми попытками реализации DI в приложении. Но из-за своего неполного соответствия основам DI-реализации могут превратиться в решения, приносящие больше вреда, чем пользы. Изучение этих антипаттернов может дать вам представление о том, какие ловушки следует обходить при реализации своих первых проектов с применением DI. Но даже при многолетнем опыте применения DI можно запросто ошибиться.

ВНИМАНИЕ

> Эта глава отличается от других глав, поскольку основной объем демонстрируемого кода является примером того, как не следует выполнять реализацию DI. Не пытайтесь повторять это на практике!

Исправления в антипаттерны могут быть внесены путем реструктуризации кода в направлении одного из паттернов DI, представленного в главе 4. Насколько сложно будет выполнить исправление в каждом отдельно взятом случае, зависит от особенностей реализации. Для каждого антипаттерна будут предоставлены некие обобщенные рекомендации по его реструктуризации в целях получения наиболее подходящего паттерна.

---

[1] *Brown W. J. et al.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. — Wiley Computer Publishing, 1998. — P. 7.

ПРИМЕЧАНИЕ

Поскольку данная тема не является для этой книги главной, реструктуризация от антипаттерна DI к паттерну будет рассмотрена только в настоящей главе. Если есть заинтересованность в углубленном изучении возможностей переработки уже существующих приложений в сторону использования технологии DI, то этому посвящена целая книга: *Feathers M. C.* Working Effectively with Legacy Code. — Prentice Hall, 2004. Хотя речь в ней идет не только о DI, она охватывает большинство понятий, изучаемых в данной книге.

Для придания ему тестируемости иногда устаревший код требует радикальных мер. Зачастую это означает постепенное внесение изменений, позволяющих избежать случайного вывода из строя ранее работающего приложения. В некоторых случаях наиболее подходящим временным решением может стать применение антипаттерна. Несмотря на то что такой шаг может оказаться улучшением по сравнению с исходным кодом, важно отметить, что от этого он не становится паттерном; существуют и другие задокументированные и повторяемые решения, доказавшие свою эффективность. Антипаттерны, рассматриваемые в этой главе, сведены в табл. 5.1.

Таблица 5.1. Антипаттерны внедрения зависимостей

Антипаттерн	Описание
«Диктатор» (Control Freak)	В отличие от инверсии управления (Inversion of Control), управление зависимостями осуществляется напрямую
«Локатор сервисов» (Service Locator)	Зависимости для потребителей могут предоставляться неявным сервисом, но это не гарантируется
«Окружающий контекст» (Ambient Context)	Предоставляет отдельную зависимость через статический метод доступа
«Ограниченная конструкция» (Constrained Construction)	Предполагает наличие у конструкторов конкретной сигнатуры

Остальная часть главы описывает каждый антипаттерн, представляя их в порядке важности. Читать главу можно последовательно или изучать только то, что представляет интерес, поскольку разделы не связаны друг с другом. Если читать все подряд не захочется, то советуем обратиться к соответствующим разделам об антипаттернах «Диктатор» и «Локатор сервисов».

Наиболее важным считается паттерн внедрения зависимостей через конструктор, а в качестве наиболее часто встречающегося антипаттерна фигурирует «Диктатор». Он сильно мешает применению любой правильной технологии DI, поэтому, прежде чем приступать к изучению других антипаттернов, предлагаем сфокусировать внимание именно на нем. Но наибольшую опасность представляет антипаттерн «Локатор сервисов», поскольку в нем зачастую видят решение проблемы, создаваемой антипаттерном «Диктатор». Он будет рассмотрен в разделе 5.2.

5.1. Антипаттерн «Диктатор»

Что противоположно инверсии управления? Изначально понятие «инверсия управления» было придумано для противопоставления обычному положению дел, но мы не можем вести речь об антипаттерне «обычного порядка». Вместо этого для описания класса, не отказывающегося от управления своими нестабильными зависимостями, применим термин «Диктатор».

ОПРЕДЕЛЕНИЕ

Антипаттерн «Диктатор» всегда формируется в любом месте, кроме корня композиции, где код полагается на применение нестабильной зависимости. Он нарушает принцип инверсии зависимостей, рассмотренный в подразделе 3.1.2.

Антипаттерн «Диктатор» формируется, к примеру, при создании нового экземпляра нестабильной зависимости с помощью ключевого слова new. Реализация антипаттерна «Диктатор» показана в листинге 5.1.

Листинг 5.1. Пример антипаттерна «Диктатор»

```
public class HomeController : Controller
{
    public ViewResult Index()
    {
        var service = new ProductService();

        var products = service.GetFeaturedProducts();
        return this.View(products);
    }
}
```

HomeController создает новый экземпляр нестабильной зависимости, ProductService, что становится причиной возникновения кода с сильной связанностью

При каждом создании нестабильной зависимости получается неявное утверждение о намерении управлять жизненным циклом экземпляра и о том, что никто другой не получит возможности перехватывать данный конкретный объект. Хотя использование ключевого слова new в отношении нестабильной зависимости — пример проблемного кода, к его применению в отношении стабильных зависимостей это не относится[1].

ПРИМЕЧАНИЕ

Не нужно думать, что ключевое слово new вдруг стало «незаконным», но от его использования для получения нестабильной зависимости вне корня композиции нужно воздерживаться. К тому же не следует забывать и о статических классах, которые также могут стать нестабильными зависимостями. Хотя вы никогда не будете использовать ключевое слово new в отношении этих классов, зависимость от этих классов вызывает те же самые проблемы.

---

[1] *Проблемный код* (code smells) — конкретные структуры в коде, свидетельствующие о присутствующих в нем проблемах дизайна, влияющих на его качество.

Наиболее явно антипаттерн «Диктатор» проявляется, когда не предпринимается никаких усилий для ввода в код каких-либо абстракций. Несколько соответствующих примеров вы уже видели в главе 2, когда Мэри создавала свое приложение электронной торговли (см. раздел 2.1). Попытка ввода DI при таком подходе не делается. Но даже там, где разработчики что-то слышали о технологии DI и о возможности создания компоновки, антипаттерн «Диктатор» зачастую в той или иной вариации все же встречается.

В следующих подразделах будет показано несколько примеров, напоминающих код, встречавшийся нам на практике. В каждом из рассматриваемых случаев разработчики старались улучшить интерфейсы, но при этом не разбирались как следует в исходных намерениях и мотивах.

5.1.1. Пример: антипаттерн «Диктатор», полученный путем обновления зависимостей с применением ключевого слова new

Многие разработчики слышали о принципе программирования для интерфейсов, но не понимали заложенного в него глубокого смысла. Пытаясь все сделать правильно или придерживаться передового опыта, они создавали код, не имеющий особого смысла. Так, в листинге 3.9 был показан вариант `ProductService`, использующий для извлечения предлагаемых товаров экземпляр интерфейса `IProductRepository`. Напомним, как выглядел соответствующий код:

```
public IEnumerable<DiscountedProduct> GetFeaturedProducts()
{
    return
        from product in this.repository.GetFeaturedProducts()
        select product.ApplyDiscountFor(this.userContext);
}
```

Здесь важно то, что компонентная переменная `repository` представляет собой абстракцию. В главе 3 вы видели, как поле `repository` может быть заполнено внедрением через конструктор, но нам встречались и другие, более примитивные попытки. Одна из них приводится в листинге 5.2.

Листинг 5.2. Создание ProductRepository через new

```
private readonly IProductRepository repository;

public ProductService()
{
    this.repository = new SqlProductRepository();
}
```

В этом примере антипаттерна «Диктатор» в конструкторе выполняется непосредственное создание нового экземпляра, в результате чего появляется код с сильной связанностью

Поле `repository` объявлено через интерфейс `IProductRepository`, поэтому любой компонент в классе `ProductService` (например, `GetFeaturedProducts`) программируется для интерфейса. Хотя все это представляется правильным, из этого мало что получается, поскольку во время выполнения программы типом всегда будет

SqlProductRepository. Способов перехвата или изменения переменной repository нет, если только не изменить и не перекомпилировать сам код. Кроме того, если жестко запрограммировать переменную так, что у нее всегда будет конкретный тип, то из ее объявления абстракцией ничего путного не получится. Непосредственное создание зависимостей с применением ключевого слова new является одним из примеров антипаттерна «Диктатор».

Прежде чем перейти к анализу и возможным способам решения возникающих проблем, порождаемых применением антипаттерна «Диктатор», рассмотрим еще несколько примеров для получения более четкого представления о контексте и общих чертах неудачных попыток применения технологии DI. В следующем примере сразу просматривается неудачное решение. Большинство разработчиков попытается усовершенствовать свой подход.

5.1.2. Пример: антипаттерн «Диктатор», полученный в результате использования фабрик

Самой распространенной и ошибочной попыткой исправления очевидных проблем от использования ключевого слова new для обновления зависимостей является применение какой-либо фабрики. Фабрики представлены несколькими вариантами. Мы кратко рассмотрим три из них:

- конкретную фабрику;
- абстрактную фабрику;
- статическую фабрику.

Если сказать Мэри Роуэн (из главы 2), что она может располагать лишь абстракцией IProductRepository, она введет в приложение фабрику ProductRepositoryFactory, производящую необходимые ей экземпляры. Посмотрим на диалог относительно такого подхода с ее коллегой Йенсом. Он предположительно затронет все перечисленные нами варианты фабрик.

Мэри: В этом классе ProductService нам нужен экземпляр IProductRepository. Но IProductRepository — это интерфейс, поэтому мы не можем просто создавать его новые экземпляры, и наш консультант говорит, что мы не должны также создавать новые экземпляры SqlProductRepository.

Йенс: А если воспользоваться какой-либо фабрикой?

Мэри: Да, я думаю о том же, но не знаю, как это сделать. Я не понимаю, как это решит нашу проблему. Вот, посмотри...

Мэри начинает записывать код, демонстрируя свою задачу. Вот что у нее получается:

```
public class ProductRepositoryFactory
{
    public IProductRepository Create()
    {
        return new SqlProductRepository();
    }
}
```

Конкретная фабрика

Мэри: Сведения о том, как создаются экземпляры `ProductRepository`, заключены в `ProductRepositoryFactory`, но проблема этим не решается, потому что этим придется воспользоваться в `ProductService` примерно следующим образом:

```
var factory = new ProductRepositoryFactory();
this.repository = factory.Create();
```

Видишь? Теперь в `ProductService` нам нужно создать новый экземпляр класса `ProductRepositoryFactory`, но здесь по-прежнему жестко кодируется применение `SqlProductRepository`. Единственное, чего мы добились, — это переместили проблему в другой класс.

Йенс: Да, я понимаю, а нельзя ли нам решить проблему, применив вместо этого абстрактную фабрику?

Отвлечемся пока от диалога Мэри и Йенса и оценим произошедшее. Мэри совершенно права в том, что класс конкретной фабрики не решает проблему возникновения антипаттерна «Диктатор», а только уводит ее в сторону. Он усложняет код, не добавляя к нему никаких ценностей. Теперь `ProductService` напрямую управляет временем жизни фабрики, а фабрика напрямую управляет временем жизни `ProductRepository`, и возможность перехвата или замены экземпляра `Repository` по-прежнему отсутствует.

ПРИМЕЧАНИЕ

Не нужно на основе материала данного раздела делать выводы о том, что мы категорически против применения классов конкретных фабрик. Такая фабрика способна решать другие проблемы, например связанные с повторяемостью кода, заключая в себя сложную логику создания. Но в отношении DI в ней нет никакой ценности. Ее нужно использовать там, где это имеет смысл.

Совершенно очевидно, что конкретная фабрика не решит проблемы DI, и сталкиваться с ее успешным применением в данной сфере нам никогда не приходилось. Комментарии Йенса относительно абстрактной фабрики звучат многообещающе.

Абстрактная фабрика

Вернемся к диалогу Мэри и Йенса и послушаем, что сказал Йенс об абстрактной фабрике.

Йенс: А что, если создать такую абстрактную фабрику, как эта?

```
public interface IProductRepositoryFactory
{
    IProductRepository Create();
}
```

Тогда нам не придется жестко кодировать какие-либо ссылки на `SqlProductRepository` и фабрикой можно будет воспользоваться `ProductService` для получения экземпляров `IProductRepository`.

Мэри: Но теперь, когда фабрика станет абстрактной, как мы получим от нее новый экземпляр?

Йенс: Мы можем создать ее реализацию, возвращающую экземпляры `SqlProductService`.

Мэри: Да, но как мы создадим сам экземпляр?

Йенс: Мы просто воспользуемся ключевым словом `new` в `ProductService`... Хотя постой...

Мэри: Но мы опять окажемся там, откуда начали.

Мэри и Йенс быстро поняли, что применение абстрактной фабрики не изменит сложившуюся ситуацию. Сначала они ломали голову над тем, как получить экземпляр абстрактного `IProductRepository`, а теперь вместо этого им нужен экземпляр `IProductRepositoryFactory`.

> **Повсеместное злоупотребление абстрактными фабриками**
>
> Абстрактная фабрика — один из паттернов из исходной книги о паттернах проектирования[1]. Паттерн абстрактной фабрики встречается чаще, чем можно было бы себе представить. Имена используемых классов зачастую скрывают этот факт (например, не заканчиваясь термином factory).
>
> Однако, что касается технологии DI, зачастую наблюдается злоупотребление абстрактными фабриками. В главе 6 мы еще вернемся к паттерну абстрактной фабрики и увидим, почему чаще всего его применение считается неудачным кодом.

Теперь, когда Мэри и Йенс отказались от абстрактной фабрики, посчитав ее неприемлемым выбором, остался еще один ущербный вариант. Мэри и Йенс вот-вот найдут решение.

Статическая фабрика

Послушаем, как Мэри и Йенс решают, какой подход, по их мнению, должен сработать.

Мэри: Давай создадим статическую фабрику. Посмотри:

```
public static class ProductRepositoryFactory
{
    public static IProductRepository Create()
```

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 78.

```
    {
        return new SqlProductRepository();
    }
}
```

Теперь, когда класс стал статическим, нам не нужно возиться с его созданием.

Йенс: Но возвращение экземпляров `SqlProductRepository` по-прежнему жестко кодируется, так в чем тогда он нам поможет?

Мэри: Мы можем справиться с поставленной задачей с помощью конфигурационных настроек, определяющих, какой из типов `ProductRepository` создавать. Таким образом:

```
public static IProductRepository Create()
{
    IConfigurationRoot configuration = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json")
        .Build();

    string repositoryType = configuration["productRepository"];

    switch (repositoryType)
    {
        case "sql": return new SqlProductRepository();
        case "azure": return AzureProductRepository();
        default: throw new InvalidOperationException("...");
    }
}
```

Видишь? Тем самым мы можем определить, следует ли нам использовать реализацию на основе SQL Server или на основе Microsoft Azure, и нам даже не придется перекомпилировать код приложения для смены одной реализации на другую.

Йенс: Здорово! Так мы и сделаем. Теперь консультант должен быть доволен!

ПРИМЕЧАНИЕ

Задуманная Мэри и Йенсом статическая фабрика ProductRepositoryFactory считывает установки из конфигурационного файла в ходе выполнения приложения, но вспомним, что в подразделе 2.3.3 этот подход считался проблемным: основывать свою работу на считывании установок из конфигурационного файла может только готовое приложение. Другие части приложения, например ProductRepositoryFactory, не могут запрашивать значения из конфигурационного файла, а вместо этого они должны настраиваться вызывающими их объектами.

Применение такой статической фабрики не позволяет добиться удовлетворительного решения и достичь исходной цели программирования для интерфейсов. Посмотрим на граф зависимостей, показанный на рис. 5.1.

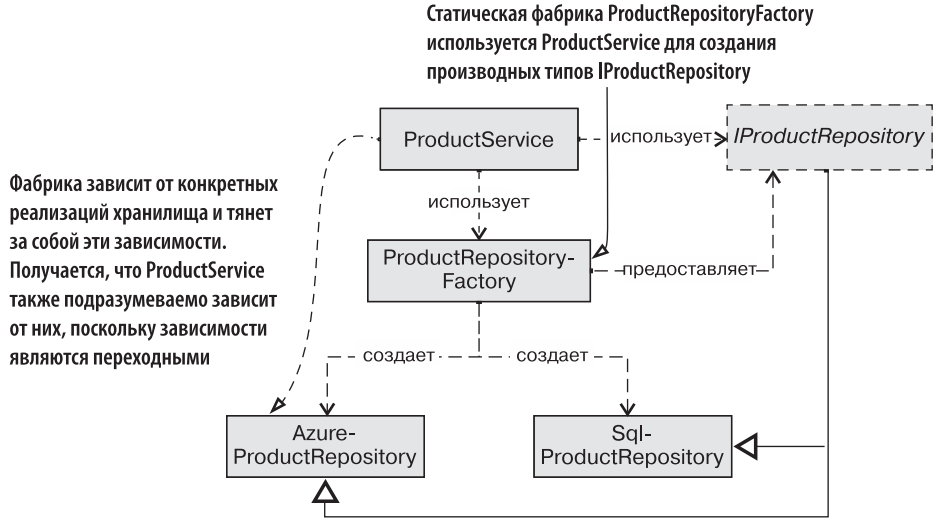


Рис. 5.1. Граф зависимостей для предложенного решения ProductRepositoryFactory

Все классы должны ссылаться на абстрактный интерфейс `IProductRepository` по следующим причинам:

- `ProductService` — потому что он является потребителем экземпляров `IProductRepository`;
- `ProductRepositoryFactory` — так как он создает экземпляры `IProductRepository`;
- `AzureProductRepository` и `SqlProductRepository` — потому что они реализуют `IProductRepository`.

`ProductRepositoryFactory` зависит как от `AzureProductRepository`, так и от `SqlProductRepository`. Поскольку класс `ProductService` находится в прямой зависимости от `ProductRepositoryFactory`, он также зависит от обоих конкретных реализаций `IProductRepository` — если вспомнить подраздел 4.1.4, такие зависимости являются транзитивными.

Поскольку `ProductService` зависит от статической фабрики `ProductRepositoryFactory`, возникают неразрешимые конструктивные проблемы. Если определить `ProductRepositoryFactory` на доменном уровне, это будет означать, что доменному уровню придется зависеть от уровня доступа к данным, потому что `ProductRepositoryFactory` создает экземпляр `SqlProductRepository`, размещаемый на этом уровне. Но уровень доступа к данным уже зависит от доменного уровня, так как `SqlProductRepository` использует такие типы и абстракции, как `Product` и `IProductRepository` из этого уровня. Это приводит к циклической ссылке между двумя проектами. Кроме того, если переместить `ProductRepositoryFactory` на уровень доступа к данным, то все равно понадобится зависимость от

доменного уровня к уровню доступа к данным, поскольку `ProductService` зависит от `ProductRepositoryFactory`. И это по-прежнему вызывает зацикливание зависимостей. Эти проблемы проектирования показаны на рис. 5.2.

Мы ничего не придумываем

Если бы мы были консультантами по данному примеру, мы совсем бы этому не обрадовались. На самом деле такое же решение было предложено в проекте, которым занимался Марк, а Стивену в прошлом также неоднократно приходилось сталкиваться с подобными конструкциями. Проект, которым занимался Марк, был достаточно масштабным и нацеленным на основную бизнес-область компании из списка Fortune 500. Ввиду сложности приложения важно было добиться его приемлемой модульности. К сожалению, Марк был привлечен к проекту слишком поздно, и его предложения были отклонены, поскольку требовали внесения существенных изменений в уже разработанную кодовую базу.

Марк перешел к работе над другими проектами, но впоследствии узнал, что команде удалось довести контракт до завершения, но проект был признан провальным и, что называется, головы полетели. Было бы неразумно говорить, что проект не удался только потому, что не была применена технология DI, но принятый подход свидетельствовал об отсутствии надлежащей архитектуры.

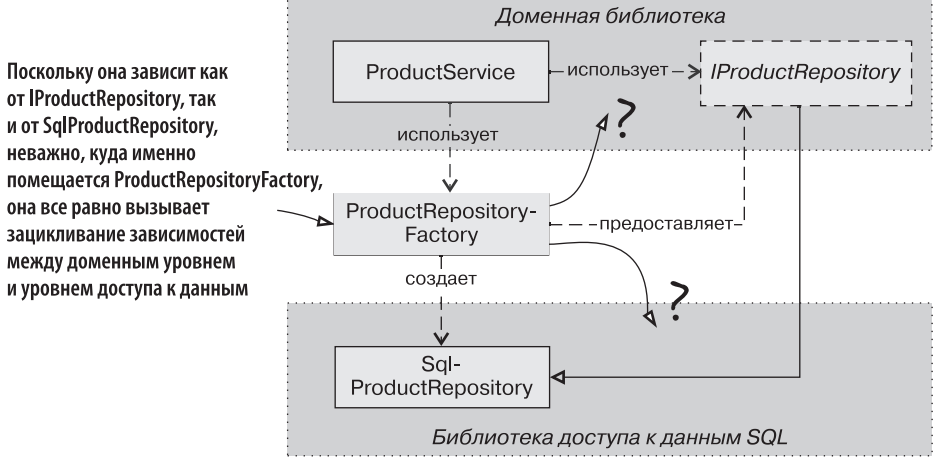


Рис. 5.2. Зацикливание зависимостей между доменным уровнем и уровнем доступа к данным, вызванное применением статической фабрики ProductRepositoryFactory

Независимо от того, как перемещаются ваши типы, единственным способом предотвращения этих зацикленных зависимостей между проектами является создание единого проекта для всех типов. Но это нежизнеспособный вариант, потому

что возникает сильное связывание доменного уровня с уровнем доступа к данным, не позволяющее производить замену уровня доступа к данным.

Вместо реализаций `IProductRepository` со слабой связанностью Мэри и Йенс остановились на модулях с сильной связанностью. Что еще хуже, фабрика всегда тащит за собой все реализации — даже те, которые не нужны! Если приложения выкладываются в облако Azure, им все равно понадобится вместе со своим приложением распространять, к примеру, библиотеку `Commerce.SqlDataAccess.dll`.

Если Мэри и Йенсу когда-либо потребуется третий тип `IProductRepository`, им придется внести изменения в фабрику и перекомпилировать свое решение. Хотя их решение может быть настраиваемым, оно не может расширяться; если отдельно взятой команде или даже компании нужно будет создать новое хранилище, у них не будет вариантов, обходящихся без доступа к исходному коду. Кроме того, будет невозможно заменить конкретные реализации `IProductRepository` реализациями, специально предназначенными для тестирования, поскольку для этого понадобится определение экземпляра `IProductRepository` динамически во время выполнения программы, а не статически в конфигурационном файле в ходе разработки приложения.

Короче говоря, статическая фабрика может показаться решением проблемы, но в реальности она ее только усугубляет. Даже в самых лучших случаях она заставляет вас ссылаться на нестабильные зависимости. Еще один вариант антипаттерна «Диктатор», рассматриваемый в следующем разделе, может быть получен при использовании переопределяемых конструкторов в сочетании с внешними реализациями по умолчанию.

5.1.3. Пример: антипаттерн «Диктатор», полученный при использовании переопределяемых конструкторов

Переопределение конструкторов довольно распространено во многих кодовых базах .NET (включая BCL). Зачастую многие переопределения предоставляют вполне разумные значения по умолчанию для одного или двух полноценных конструкторов, принимающих в качестве входных данных все соответствующие параметры. (Такой прием называется *выстраиванием цепочки конструкторов*.) Когда дело касается DI, то временами попадаются и другие применения переопределений.

В весьма распространенном антипаттерне задается переопределение конструктора, специально предназначенного для тестов, что позволяет определить зависимость в явном виде, хотя эксплуатационный код использует конструктор без параметров. Это может иметь негативные последствия, когда реализация зависимости по умолчанию представляет собой внешнюю, а не локальную реализацию. В разделе 4.4.2 уже объяснялось, что обычно стоит предоставлять все нестабильные зависимости с помощью внедрения через конструктор, даже те из них, которые могли бы быть локальной реализацией по умолчанию.

Внешняя реализация по умолчанию

Внешняя реализация по умолчанию является противоположностью локальной реализации по умолчанию. Это реализация нестабильной зависимости, применяемая по умолчанию, даже если она определена в модуле, отличном от модуля своего потребителя. Рассмотрим в качестве примера реализацию хранилища, встречавшуюся в учебном приложении электронной торговли в предыдущих главах.

Такому сервису, как `ProductService`, для работы требуется экземпляр `ProductRepository`. Зачастую при разработке таких приложений задумывается подходящая реализация: одна из тех, в которых осуществляется нужная функциональность чтения данных из реляционной базы данных и их записи в нее. Может появиться намерение воспользоваться такой реализацией по умолчанию. Проблема заключается в том, что задуманная реализация (`SqlProductRepository`) определена в другом модуле. Это вынуждает вас принять нежелательную зависимость от уровня доступа к данным.

Перетаскивание нежелательных модулей лишает многих преимуществ слабой связанности, рассмотренных в главе 1. Усложняется повторное использование модуля доменного уровня, поскольку он тянет за собой модуль доступа к данным, хотя у вас может возникать желание воспользоваться им в другом контексте. Усложняется также параллельная разработка, поскольку класс `ProductService` теперь напрямую зависит от класса `SqlProductRepository`.

В листинге 5.3 показан класс `ProductService` с конструктором по умолчанию и переопределенным конструктором. Это пример того, как не надо делать.

Листинг 5.3. ProductService с несколькими конструкторами

```
private readonly IProductRepository repository;

public ProductService()
    : this(new SqlProductRepository())
{
}

public ProductService(IProductRepository repository)
{
    if (repository == null)
        throw new ArgumentNullException("repository");

    this.repository = repository;
}
```

Конструктор без параметров передает внешнюю реализацию по умолчанию SqlProductRepository переопределенному конструктору. Это привязывает доменный уровень к уровню доступа к данным SQL

Конструктор, использующийся для внедрения, принимает требуемый IProductRepository и сохраняет его в поле repository

На первый взгляд может показаться, что этот стиль программирования является наилучшим. Он позволяет предоставлять имитируемые зависимости для проведения модульного тестирования, и в то же время класс может по-прежнему создаваться

удобным способом, не требуя при этом предоставления ему его зависимостей. Этот стиль показан в следующем примере:

```
var productService = new ProductService();
```

Позволяя `ProductService` создавать нестабильную зависимость `SqlProductRepository`, вы снова принудительно вводите сильную связанность между модулями. Хотя `ProductService` может повторно использоваться с различными реализациями `IProductRepository`, которые в ходе тестирования будут предоставлены через наиболее гибкое переопределение конструктора, этот подход исключает возможность перехвата экземпляра `IProductRepository` в приложении.

Теперь, когда вы познакомились с несколькими примерами антипаттерна «Диктатор», мы надеемся, что у вас появилось более четкое представление о том, что следует искать — появление ключевого слова `new`, которому предшествуют нестабильные зависимости. Это поможет вам обойти наиболее очевидные подводные камни. Но если нужно отвязаться от уже допущенного применения антипаттерна, то решить эту задачу поможет материал следующего раздела.

5.1.4. Анализ антипаттерна «Диктатор»

Антипаттерн «Диктатор» выступает противоположностью инверсии управления. Когда возникает непосредственное управление созданием нестабильной зависимости, получается код с сильной связанностью и утрачиваются многие (если не все) преимущества кода со слабой связанностью, перечисленные в главе 1.

«Диктатор» — самый распространенный антипаттерн DI. Он представляет собой способ создания экземпляров по умолчанию в большинстве языков программирования, поэтому может встречаться даже в тех приложениях, разработчики которых никогда даже и не рассматривали возможность применения технологии DI. От этого вполне естественного и глубоко укоренившегося способа создания новых объектов многим разработчиками очень трудно отказаться. Даже когда они начинают задумываться о применении технологии DI, для них наступают тяжелые времена, когда приходит осознание, что теперь нужен контроль за тем, где и когда нужно создавать экземпляры. Введение такого контроля может даваться нелегко, но даже если справиться с этой задачей, будут и другие, хотя и менее значимые, капканы, требующие путей обхода.

Негативные последствия применения антипаттерна «Диктатор»

Применение кода с сильной связанностью, приводящее к появлению антипаттерна «Диктатор», вызывает потенциальную утрату многих преимуществ модульного проектирования. Все это уже было рассмотрено в предыдущих разделах, но подведем итоги.

- *Несмотря на то что приложение можно настроить на применение одной из нескольких заранее сконфигурированных зависимостей, утрачивается возможность*

замены таких зависимостей по вашему желанию. Невозможно будет предоставить реализацию, созданную после того, как приложение уже было скомпилировано, и, конечно же, невозможно будет предоставить в качестве реализации конкретные экземпляры.

- *Затрудняется повторное использование модуля-потребителя, поскольку он тянет за собой зависимости, которые могут в новом контексте быть нежелательными.* В качестве соответствующего примера рассмотрим модуль, который из-за использования внешней реализации по умолчанию (Foreign Default) зависит от библиотек среды ASP.NET Core. Это затрудняет его повторное использование в качестве части приложения, которое не должно или не может зависеть от среды ASP.NET Core (например, в службе Windows или в приложении мобильного телефона).
- *Усложняется ведение параллельной разработки.* Дело в том, что приложение-потребитель имеет тесную связанность со всеми реализациями своих зависимостей.
- *Уменьшается пригодность к тестированию.* В качестве заменителей зависимости нельзя применять ее тестовые дубликаты.

Тщательно продумывая конструкцию, можно все же создавать приложения с кодом, имеющим сильную связанность и с четко определенными обязанностями, не допуская при этом снижения возможностей их сопровождения. Но даже в этом случае придется заплатить слишком высокую цену, сохраняя множество ограничений. Учитывая объем усилий, затраченных на достижение такого результата, продолжать их вкладывание в антипаттерн «Диктатор» не имеет никакого смысла. Нужно отойти от применения антипаттерна «Диктатор» и перейти к использованию настоящей технологии DI.

Переработка антипаттерна «Диктатор» в применение технологии DI

Чтобы избавиться от антипаттерна «Диктатор», нужно переработать код в направлении к одному из подходящих паттернов проектирования DI, рассмотренных в главе 4. Для начала нужно воспользоваться руководством, показанным на рис. 4.9, чтобы определить, на какой из паттернов следует нацелиться. В большинстве случаев таким паттерном будет внедрение через конструктор. Переработка будет выполняться поэтапно.

1. Убедитесь, что программирование ведется с прицелом на абстракцию. В примерах так и было; но в иных ситуациях сначала может понадобиться извлечь интерфейс и внести изменения в определения переменных.
2. Если конкретные реализации зависимости создаются сразу в нескольких местах, их нужно свести в один метод создания. Следует обеспечить выражение возвращаемого значения в виде абстракции, а не конкретного типа.

3. Теперь, когда у вас есть только одно место создания экземпляра, это создание нужно убрать из класса-потребителя, реализовав один из паттернов DI, например внедрение через конструктор.

В случае с примерами `ProductService` из предыдущих разделов наилучшим будет решение о применении внедрения через конструктор (листинг 5.4).

Листинг 5.4. Переработка с переходом от антипаттерна «Диктатор» к использованию внедрения через конструктор

```
public class ProductService : IProductService
{
    private readonly IProductRepository repository;

    public ProductService(IProductRepository repository)
    {
        if (repository == null)
            throw new ArgumentNullException("repository");

        this.repository = repository;
    }
}
```

Пока «Диктатор» является самым разрушительным антипаттерном, но, даже если с ним удастся справиться, могут возникнуть более коварные проблемы. В следующих разделах будут рассмотрены другие антипаттерны. Они менее проблематичны, чем «Диктатор», и с ними легче справиться, но нужно быть начеку и исправлять код по мере их обнаружения.

5.2. Антипаттерн «Локатор сервисов»

Иногда бывает трудно отказаться от замысла непосредственного управления зависимостями, поэтому многие разработчики выводят статические фабрики (подобные той, что была рассмотрена в подразделе 5.1.2) на новые уровни. Это приводит к появлению антипаттерна «Локатор сервисов».

ОПРЕДЕЛЕНИЕ

> Локатор сервисов предоставляет компонентам приложения, не входящим в корень композиции, доступ к неограниченному набору нестабильных зависимостей.

В наиболее часто встречающемся варианте своей реализации антипаттерн «Локатор сервисов» является статической фабрикой, которая может настраиваться на использование конкретных сервисов еще до того, как ей начнет пользоваться первый потребитель. (Но вам также могут попадаться и абстрактные «Локаторы сервисов».)

Предположительно это может произойти в корне композиции. В зависимости от конкретной реализации, локатор сервисов может настраиваться с помощью кода путем считывания конфигурационного файла или с помощью использования комбинации этих двух вариантов. В листинге 5.5 антипаттерн «Локатор сервисов» показан в действии.

Листинг 5.5. Использование антипаттерна «Локатор сервисов»

```
public class HomeController : Controller
{
    public HomeController() { }

    public ViewResult Index()
    {
        IProductService service =
            Locator.GetService<IProductService>();

        var products = service.GetFeaturedProducts();

        return this.View(products);
    }
}
```

У HomeController имеется конструктор без параметров

HomeController запрашивает экземпляр IProductService из статического класса Locator

Здесь, как обычно, используется запрошенный IProductService

Вместо статичного определения списка требуемых зависимостей в `HomeController` имеется конструктор без параметров, запрашивающий зависимости чуть позже. Тем самым эти зависимости скрываются от потребителей контроллера `HomeController`, затрудняя его использование и тестирование. На рис. 5.3 показано взаимодействие программных составляющих из листинга 5.5, где можно увидеть взаимоотношения между реализациями локатора сервисов и `ProductService`.

Много лет назад еще возникали споры относительно того, что локатор сервисов является антипаттерном. Но полемика закончилась: локатор сервисов — это антипаттерн. Но не удивляйтесь, если встретится кодовая база, где он будет разбросан повсюду.

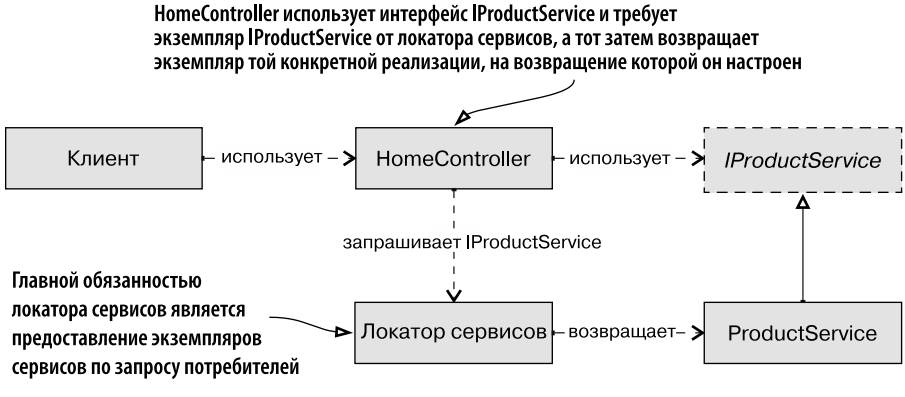


Рис. 5.3. Взаимодействие между HomeController и локатором сервисов

Наша личная история, связанная с локатором сервисов

До окончательного отказа от локатора сервисов у меня (у Марка) пару лет были с ним весьма напряженные отношения. Хотя я не могу точно припомнить, когда мне впервые попалась на глаза популярная статья с описанием локатора сервисов в качестве паттерна[1], она предоставила мне возможное решение проблемы, над которой я в ту пору размышлял: как внедрять зависимости. Согласно описанию паттерн «Локатор сервисов» представлялся ответом на все мои вопросы, и я тут же приступил к разработке многократно используемой библиотеки, основанной на применении этого паттерна, который я условно назвал «Локатор сервисов».

В 2007 году, нацелившись на создание Enterprise Library 2 (https://blogs.msdn.microsoft.com/ploeh/2007/03/15/service-locator-2-released/), я выпустил полностью переработанную библиотеку. Вскоре я от нее отказался, потому что понял, что она была основана на применении антипаттерна. У Стивена также была история, очень похожая на мою.

В 2009 году был выпущен проект с открытым кодом Common Service Locator (CSL) (https://github.com/unitycontainer/commonservicelocator). Это была многократно используемая библиотека, в которой имелась реализация паттерна «Локатор сервисов», похожая на библиотеку «Локатор сервисов», созданную Марком. Это абстракция над разрешающим API DI-контейнеров, позволяющая другим многократно используемым библиотекам выполнять разрешения зависимостей, обходясь без получения сильной зависимости от конкретного DI-контейнера. Затем разработчики приложений могли подключить свой собственный DI-контейнер.

Вдохновленный созданием CSL, я (Стивен) приступил к разработке своего собственного DI-контейнера в виде простой CSL-реализации. Поскольку это была CSL-реализация, я условно назвал свою библиотеку «Простой локатор сервисов» (Simple Service Locator). Как и в истории с Марком, мне не понадобилось много времени, чтобы понять, что «Локатор сервисов» является антипаттерном, который не следует применять ни разработчикам приложений, ни разработчикам многократно используемых библиотек. Поэтому я удалил зависимость от CSL и переименовал DI-контейнер в «Простой инжектор» (https://simpleinjector.org).

Важно отметить, что при взгляде только на статическую структуру DI-контейнер будет похож на локатор сервисов. Отличие едва уловимое и заключается не в механизмах реализации, а в способе использования. По сути, для разрешения полного графа объектов из корня композиции правильно будет воспользоваться запросом контейнера или локатора. Запрос с целью получения детализированных сервисов не из корня композиции, а из каких-либо других мест приводит к возникновению антипаттерна «Локатор сервисов». Рассмотрим пример, показывающий локатор сервисов в действии.

---

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

5.2.1. Пример: ProductService, использующий локатор сервисов

Вернемся к нашему проверенному на практике `ProductService`, которому требуется экземпляр интерфейса `IProductRepository`. Предполагая то, что нужно применить антипаттерн «Локатор», в `ProductService`, как показано в листинге 5.6, будет использоваться статический метод `GetService`.

Листинг 5.6. Использование локатора сервисов внутри конструктора

```
public class ProductService : IProductService
{
    private readonly IProductRepository repository;

    public ProductService()
    {
        this.repository = Locator.GetService<IProductRepository>();
    }

    public IEnumerable<DiscountedProduct> GetFeaturedProducts() { ... }
}
```

В этом примере метод `GetService`, чтобы указать тип запрашиваемого сервиса, реализован с использованием обобщенных параметров типа. Для указания типа, если вам это больше нравится, можно также воспользоваться аргументом `Type`.

Как показано в листинге 5.7, эта реализация класса `Locator` выполнена в максимально коротком из возможных вариантов. Можно было бы добавить граничные операторы и обработку ошибок, но нам нужно выделить основное поведение. Код также может включать в себя функционал загрузки настроек из файла, но мы оставим создание всего этого вам в качестве упражнения.

Листинг 5.7. Простая реализация локатора сервисов

```
public static class Locator
{
    private static Dictionary<Type, object> services =
        new Dictionary<Type, object>();

    public static void Register<T>(T service)
    {
        services[typeof(T)] = service;
    }

    public static T GetService<T>()
    {
        return (T)services[typeof(T)];
    }

    public static void Reset()
    {
        services.Clear();
    }
}
```

Статический класс Locator хранит все настроенные сервисы во внутреннем словаре, который отображает абстрактные типы на каждый конкретный экземпляр

Метод GetService позволяет выполнять разрешение произвольной абстракции

Клиенты, такие как `ProductService`, могут использовать метод `GetService` для запроса экземпляра абстрактного типа `T`. Поскольку этот пример кода не содержит граничных операторов и обработки ошибок, то при отсутствии записей затребованного типа в словаре этот метод выдает весьма загадочное исключение `KeyNotFoundException`. Вы сами можете составить представление о том, как можно будет добавить код для выдачи исключения с более понятным описанием.

Метод `GetService` может только возвратить экземпляр затребованного типа, если тот заранее был введен во внутренний словарь. Это может быть сделано с помощью метода `Register`. В этом примере кода также отсутствует граничный оператор, поэтому имеется возможность зарегистрировать с помощью метода `Register` значение `null`, но в более надежной реализации такое не должно быть допущено. В этой реализации также происходит неизменное кэширование зарегистрированных экземпляров, но совсем не сложно придумать реализацию, позволяющую создавать новые экземпляры при каждом вызове `GetService`. В определенных случаях, особенно при проведении модульного тестирования, важно иметь возможность перезапуска «Локатора сервисов». Она предоставляется путем использования метода `Reset`, который очищает внутренний словарь.

Такие классы, как `ProductService`, зависят от доступности сервиса в «Локаторе сервисов», поэтому важную роль здесь играет его предварительная настройка. В модульном тесте это может быть сделано с помощью тестового дубликата (test double), реализованного заглушкой (stub), как показано в листинге 5.8[1].

Листинг 5.8. Модульный тест, зависящий от локатора сервисов

```
[Fact]
public void GetFeaturedProductsWillReturnInstance()
{
    // Подготовка
    var stub = ProductRepositoryStub();

    Locator.Reset();

    Locator.Register<IProductRepository>(stub);

    var sut = new ProductService();

    // Действие
    var result = sut.GetFeaturedProducts();

    // Утверждение
    Assert.NotNull(result);
}
```

Создание заглушки для интерфейса IProductRepository

Сброс локатора Locator на его установки по умолчанию, чтобы предыдущий тест не смог повлиять на текущий

Выполнение зарегистрированной задачи для текущего теста; теперь GetFeaturedProducts будет использовать ProductRepositoryStub. Внутреннее применение Locator.GetService приводит к возникновению временной связанности Locator.Register и GetFeaturedProducts

Использование статического метода Register для настройки локатора сервисов на применение экземпляра stub

[1] Чтобы подробнее изучить тестовые дубликаты, обратитесь к изданию: *Meszaros G.* xUnit Test Patterns: Refactoring Test Code. — Addison-Wesley, 2007.

Этот пример показывает, как для настройки локатора сервисов используется статический метод `Register`. Если, как показано в примере, это сделано до создания `ProductService` этим сервисом, то для работы с хранилищем `ProductRepository` используется настроенная заглушка. В готовом приложении локатор сервисов будет настроен в корне композиции на подходящую реализацию `ProductRepository`.

Этот способ определения зависимостей из класса `ProductService` действительно работает при условии, что единственным критерием успеха является возможность использования зависимости и ее замены по желанию. Но у него есть серьезные недостатки.

5.2.2. Анализ антипаттерна «Локатор сервисов»

«Локатор сервисов» — весьма опасный антипаттерн, потому что он почти всегда работоспособен. Зависимости можно найти из потребляющих классов, и эти зависимости можно заменить другими реализациями — даже тестовыми дубликатами из модульных тестов. Применяя аналитическую модель, изложенную в главе 1, в целях вычисления тех случаев, когда локатор сервисов может соответствовать преимуществам модульной конструкции приложения, обнаружится, что он подходит в большинстве случаев.

- Можно поддерживать позднее связывание путем изменения регистрации.
- Можно разрабатывать код параллельно, поскольку программирование ведется с прицелом на использование интерфейсов, заменяя модули по своему желанию.
- Можно добиться весьма качественного разделения обязанностей, поэтому ничто не останавливает вас от написания сопровождаемого кода, просто это будет сложнее сделать.
- Зависимости можно заменять тестовыми дубликатами, обеспечивая тем самым тестируемость.

Есть только одна область, в которой локатор сервисов не соответствует требованиям, и к этому нельзя относиться легкомысленно.

Негативные эффекты от применения антипаттерна «Локатор сервисов»

Основная проблема при использовании локатора сервисов состоит в том, что он влияет на возможность повторного использования классов-потребителей. Это проявляется двояко:

- класс тащит за собой локатор сервисов как избыточную зависимость;
- класс скрывает очевидность своих зависимостей.

Посмотрим сначала на показанный на рис. 5.4 граф зависимостей для `ProductService` из примера, приведенного в подразделе 5.2.1.

Кроме ожидаемой ссылки на `IProductRepository`, `ProductService` также зависит от класса `Locator`. Это означает, что для повторного использования класса

`ProductService` необходимо снова распространить не только его и относящуюся к нему зависимость `IProductRepository`, но и зависимость от класса `Locator`, существующую только по техническим причинам. Если класс `Locator` определен в другом модуле, не в том, что `ProductService` и `IProductRepository`, то новое приложение, собирающееся использовать `ProductService`, также должно принять этот модуль.

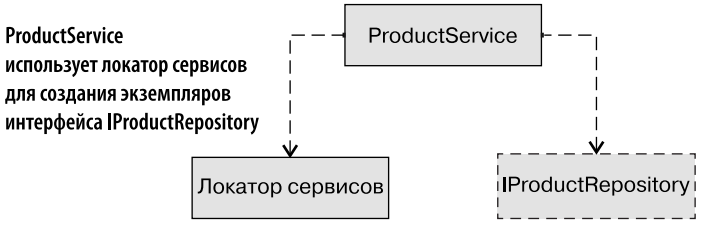


Рис. 5.4. Граф зависимостей для ProductService

Возможно, эту дополнительную зависимость от класса `Locator` и можно было бы допустить, если бы он действительно был необходим для работы технологии DI. Мы бы восприняли его как налог, уплачиваемый за получение преимуществ. Но имеются и более удачные и доступные варианты (например, внедрение через конструктор), поэтому данная зависимость представляется избыточной. Более того, в поле зрения разработчиков, желающих воспользоваться классом `ProductService`, ни эта избыточна, зависимость, ни связанное с ним дополнение в виде `IProductRepository` явным образом не видны. На рис. 5.5 показано, что среда Visual Studio не предлагает никаких подсказок по использованию этого класса.

Рис. 5.5. Единственное, о чем может сообщить принадлежащая среде Visual Studio служба IntelliSense о классе ProductService, это то, что у него имеется конструктор без параметров. Все его зависимости остаются невидимыми

Если нужно создать новый экземпляр класса `ProductService`, среда Visual Studio может только сообщить вам, что у класса имеется конструктор без параметров. Но если впоследствии вы попытаетесь запустить код, то при отсутствии регистрации экземпляра `IProductRepository` с помощью класса `Locator` будет получена ошибка времени выполнения. Это может произойти, если вы плохо знаете устройство класса `ProductService`.

ПРИМЕЧАНИЕ

Представьте себе, что создаваемый вами код поставляется в непонятном .dll-файле без документации. Просто ли будет кому-либо им воспользоваться? Можно разработать практически самодокументируемые API, и, хотя это под силу только опытным программистам, это весьма достойная цель. Проблема, связанная с локатором сервисов, состоит в том, что любой использующий его компонент дезориентирует разработчиков относительно своего уровня сложности. Через открытый API он выглядит простым, но на самом деле является сложным, что не может быть обнаружено до первой попытки его запуска на выполнение.

Класс `ProductService` далек от самодокументирования: до того, как он заработает, абсолютно невозможно сказать что-либо о том, какие зависимости должны присутствовать. Фактически в будущих версиях разработчики `ProductService` могут даже принять решение о добавлении дополнительных зависимостей. Это будет означать, что код, работающий для текущей версии, в будущей версии может дать сбой, а предупреждающей ошибки компилятора получено не будет. Локатор сервисов облегчает непреднамеренное внесение изменений, не поддерживающих обратной совместимости.

Использование обобщений может навести на мысль, что локатор сервисов обладает строгой типизацией. Но даже API, подобный показанному в листинге 5.7, имеет слабую типизацию, поскольку может быть запрошен любой тип. Возможность компиляции кода, вызывающего метод `GetService<T>`, не дает вам никаких гарантий, что он не станет направо и налево выдавать исключения в процессе выполнения приложения.

В ходе модульного тестирования возникает еще одна проблема, связанная с тем, что тестовый дубликат, зарегистрированный в одном наборе тестовых данных, приведет к созданию неудачного кода под названием «Взаимозависимые тесты» (Interdependent Tests), поскольку он останется в памяти при выполнении следующего набора тестовых данных. Поэтому необходимо после каждого теста производить перенастройку тестовых установок (Fixture Teardown), вызвав метод `Locator.Reset()`[1]. Нужно не забыть это сделать самостоятельно, на чем легко можно оступиться.

Дело совсем не в механике

Хотя локатор сервисов принимает различные формы и обличия, его общая сигнатура выглядит примерно так:

```
public T Resolve<T>()
```

Нетрудно предположить, что каждый API с этой сигнатурой является локатором сервисов, но это не так. На самом деле это та самая сигнатура, которая демонстрируется большинством DI-контейнеров. Как локатор сервисов, он определяется не статической структурой API, а той ролью, которую API играет в приложении.

Важный аспект антипаттерна «Локатор сервисов» заключается в том, что компоненты приложения запрашивают зависимости вместо статического их объявления через свой конструктор. Как уже ранее объяснялось, это сопряжено с массой недостатков. Но когда код, являющийся частью корня композиции, запрашивает зависимости, эти недостатки не проявляются.

Поскольку корень композиции уже зависит от всего остального, имеющегося в системе (о чем уже говорилось в разделе 4.1), тащить за собой дополнительную зависимость он просто не может. Ему уже по определению известно о каждой зависимости.

[1] Дополнительные сведения о перенастройке тестовых установок можно найти в издании: *Meszaros G.* xUnit Test Patterns. — P. 100.

Скрывать свои зависимости корень композиции не может — от кого же ему их скрывать? Его роль заключается в построении графа объектов; ему не нужно предоставлять эти зависимости.

Запрос зависимостей, даже если он делается через DI-контейнер, при неправильном применении становится локатором сервисов. Когда код приложения (в отличие от кода инфраструктуры) активно запрашивает сервис для получения необходимых зависимостей, он становится локатором сервисов.

ВНИМАНИЕ

DI-контейнер, заключенный в корень композиции, локатором сервисов не является. Он представляет собой компонент инфраструктуры.

Использование локатора сервисов может показаться совершенно безобидным, но оно может привести к всевозможным неприятным ошибкам времени выполнения. Как избежать этих проблем? Когда принимается решение об избавлении от локатора сервисов, нужно найти соответствующий способ. Подходом по умолчанию должно стать внедрение через конструктор, если только не станет более подходящим какой-нибудь другой паттерн DI из главы 4.

Рефакторинг антипаттерна «Локатор сервисов» и переход к применению технологии DI

Поскольку при внедрении через конструктор происходит статическое объявления зависимостей класса, то использование чистой технологии DI приведет к ошибке компиляции. А вот при использовании DI-контейнера утрачивается возможность проверки правильности кода в ходе компиляции. Но статически объявленные зависимости класса все же гарантируют возможность проверки правильности графа объектов вашего приложения путем запроса к контейнеру на создание для вас всех графов объектов. Это можно сделать при запуске приложения или же в виде составной части модульного или интеграционного теста.

В некоторых DI-контейнерах сделан даже еще один шаг вперед, позволяющий проводить более сложный анализ DI-конфигурации. Он разрешает обнаруживать все разновидности самых распространенных подводных камней. Локатор сервисов будет полностью невидим DI-контейнеру, не позволяя тем самым проводить подобные проверки от вашего имени.

Во многих случаях у класса-потребителя локатора сервисов могут быть вызовы к нему, распространяемые по всей кодовой базе. В таких случаях он действует в качестве замены инструкции `new`. Когда такое происходит, первым шагом по переработке является объединение создания каждой зависимости в одном методе.

При отсутствии у компонента поля для хранения экземпляра зависимости вы можете ввести это поле и обеспечить его использование всем остальным кодом, когда он является потребителем зависимости. Поле нужно снабдить пометкой `readonly`,

исключив возможности его изменения за пределами конструктора. Подобные действия вынуждают вас присваивать полю значение из конструктора, используя локатор сервисов. Теперь можно ввести параметр конструктора, присваивающий полю значение вместо «Локатора сервисов», который затем может быть удален.

ПРИМЕЧАНИЕ

Введение в конструктор параметра зависимости, скорее всего, приведет в негодность уже имеющихся потребителей, поэтому лучше начать с самых верхних классов и спускаться вниз по графу зависимостей.

Переработка класса, применяющего локатор сервисов, похожа на переделку класса, использующего антипаттерн «Диктатор». Дополнительные заметки по переработке реализации антипаттерна «Диктатор» в применение технологии DI содержатся в подразделе 5.1.4.

На первый взгляд локатор сервисов может показаться вполне допустимым паттерном DI, но не стоит обманываться: он может явным образом решить задачу слабой связанности, но при этом принести в жертву другие решаемые вопросы. DI-паттерны, представленные в главе 4, предлагают более подходящие варианты с меньшим количеством недостатков. Это справедливо как для антипаттерна «Локатор сервисов», так и для других антипаттернов, представленных в этой главе. Несмотря на то что они отличаются друг от друга, все эти антипаттерны имеют одну общую особенность: связанные с ними проблемы могут быть успешно решены путем использования одного из DI-паттернов, рассмотренных в главе 4.

5.3. Антипаттерн «Окружающий контекст»

Родственным локатору сервисов является антипаттерн «Окружающий контекст». Если локатор сервисов разрешает глобальный доступ к неограниченному набору зависимостей, то антипаттерн «Окружающий контекст» открывает доступ через статический метод доступа к одной строго типизированной зависимости.

ОПРЕДЕЛЕНИЕ

Антипаттерн «Окружающий контекст» с помощью статических компонентов класса предоставляет коду приложения, находящемуся вне корня композиции, глобальный доступ к нестабильной зависимости или к ее поведению.

В листинге 5.9 антипаттерн «Окружающий контекст» показан в действии.

В этом примере `ITimeProvider` представляет собой абстракцию, позволяющую извлекать текущее время системы. Поскольку может потребоваться повлиять на восприятие времени приложением (например, для тестирования), вызывать напрямую `DateTime.Now` нежелательно. Вместо того чтобы позволить потребителям вызывать `DateTime.Now` напрямую, более удачным решением может стать маскировка доступа к `DateTime.Now` за абстракцией. Заманчиво, однако, предоставить потребителям доступ к реализации этой абстракции по умолчанию через статическое свойство

или метод. Показанное в листинге 5.9 свойство `Current` открывает доступ к такой реализации `ITimeProvider`.

Листинг 5.9. Использование антипаттерна «Окружающий контекст»

Статическое свойство Current представляет окружающий контекст, открывающий доступ к экземпляру ITimeProvider. В результате зависимость ITimeProvider становится скрытой, а тестирование затрудняется

```
public string GetWelcomeMessage()
{
    ITimeProvider provider = TimeProvider.Current;
    DateTime now = provider.Now;

    string partOfDay = now.Hour < 6 ? "night" : "day";

    return string.Format("Good {0}.", partOfDay);
}
```

Окружающий контекст структурно похож на паттерн «Одиночка» (Singleton)[1]. Оба они открывают доступ к зависимости путем использования компонентов статического класса. Разница в том, что окружающий контекст позволяет своей зависимости изменяться, а паттерн «Одиночка» гарантирует абсолютную неизменчивость своего единственного экземпляра.

ПРИМЕЧАНИЕ

Паттерн «Одиночка» должен использоваться либо из корня композиции, либо когда зависимость является стабильной. С другой стороны, когда паттерн «Одиночка» используется для предоставления приложению глобального доступа к нестабильной зависимости, последствия его применения аналогичны, как говорится в подразделе 5.3.3, тем, что возникают при использовании антипаттерна «Окружающий контекст».

Доступ к текущему времени системы является общей потребностью. Немного тщательнее присмотримся к примеру `ITimeProvider`.

5.3.1. Пример: получение данных о времени через «Окружающий контекст»

Потребность в получении контроля над временем возникает по многим причинам. У большинства приложений есть бизнес-логика, зависящая от времени или хода времени. В предыдущем примере был показан простой случай вывода приветственного сообщения на основе текущего времени. Два других примера включают в себя следующее.

- *Вычисление стоимости на основе дня недели.* В некоторых бизнес-сферах вполне естественно для клиентов платить больше за обслуживание в выходные дни.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 157.

- *Отправка пользователям уведомлений с использованием разных каналов связи на основе времени суток.* Например, в рабочие часы может быть сподручнее отправлять уведомления по электронной почте, а в остальное время лучше отправлять их в виде СМС.

Поскольку востребованность работы со временем крайне высока, у разработчиков зачастую возникает потребность в упрощении доступа к такой нестабильной зависимости путем использования антипаттерна «Окружающий контекст». Пример абстракции `ITimeProvider` показан в листинге 5.10.

Листинг 5.10. Абстракция ITimeProvider

```
public interface ITimeProvider
{
    DateTime Now { get; }
}
```

Позволяет потребителям узнать текущее время системы

В листинге 5.11 показана упрощенная реализация класса `TimeProvider` для этой абстракции `ITimeProvider`.

Листинг 5.11. Реализация антипаттерна «Окружающий контекст» в виде класса TimeProvider

```
public static class TimeProvider
{
    private static ITimeProvider current =
        new DefaultTimeProvider();

    public static ITimeProvider Current
    {
        get { return current; }
        set { current = value; }
    }

    private class DefaultTimeProvider : ITimeProvider
    {
        public DateTime Now { get { return DateTime.Now; } }
    }
}
```

Статический класс, отрывающий глобальный доступ к сконфигурированной реализации ITimeProvider

Инициализация локальной реализации по умолчанию, использующей реальные системные часы

Статическое свойство, открывающее глобальный доступ по чтению и записи к нестабильной зависимости ITimeProvider

Реализация по умолчанию, использующая реальные системные часы

Используя реализацию `TimeProvider`, можно проводить модульное тестирование ранее определенного метода `GetWelcomeMessage`. Такой тест показан в листинге 5.12.

Это один из вариантов антипаттерна «Окружающий контекст». Другими часто встречающимися вариантами являются:

- *окружающий контекст, позволяющий потребителям воспользоваться поведением глобально сконфигурированной зависимости.* Возвращаясь к предыдущему примеру, `TimeProvider` может предоставить потребителям статический метод `GetCurrentTime`, скрывающий используемую зависимость за счет ее внутреннего вызова;

- *окружающий контекст, объединяющий статический метод доступа с интерфейсом в единую абстракцию.* В отношении предыдущего примера это означало бы наличие единственного основного класса `TimeProvider`, содержащего как свойство экземпляра `Now`, так и статическое свойство `Current`;
- *окружающий контекст, где вместо абстракции, определяемой пользователем, используются делегаты.* Вместо того чтобы располагать интерфейсом `ITimeProvider`, вполне достаточное описание которого содержится в нем самом, можно добиться того же самого, воспользовавшись делегатом `Func<DateTime>`.

Листинг 5.12. Модульный тест, зависящий от антипаттерна «Окружающий контекст»

```
[Fact]
public void SaysGoodDayDuringDayTime()
{
    // Подготовка
    DateTime dayTime = DateTime.Parse("20190101 6:00");

    var stub = new TimeProviderStub { Now = dayTime };

    TimeProvider.Current = stub;

    var sut = new WelcomeMessageGenerator();

    // Действие
    string actualMessage = sut.GetWelcomeMessage();

    // Утверждение
    Assert.Equal(expected: "Good day.", actual: actualMessage);
}
```

Замена реализации по умолчанию заглушкой, которая всегда возвращает указанное значение dayTime

API WelcomeMessageGenerator проявляет скрытность, поскольку его конструктор скрывает факт того, что ITimeProvider является обязательной зависимостью

Между TimeProvider.Current и GetWelcomeMessage возникает временная связанность

Окружающий контекст может возникать во многих обличиях и реализациях. Опять же предостережение насчет окружающего контекста состоит в том, что он обеспечивает либо прямой, либо косвенный доступ к нестабильной зависимости с помощью некоторого статического компонента класса. Прежде чем провести анализ и оценку возможных способов устранения проблем, вызванных окружающим контекстом, рассмотрим еще один весьма распространенный пример окружающего контекста.

5.3.2. Пример: ведение регистрационных записей с помощью окружающего контекста

Еще один весьма распространенный случай, когда разработчики стремятся сократить путь, попадая в ловушку, касается применения в приложении регистрационных записей. Любое реальное приложение требует возможности вести запись информации

об ошибках и других необычных условиях в файл или другой источник для последующего анализа. Многие разработчики чувствуют, что ведение регистрационных записей является особой деятельностью, заслуживающей «отступления от правил». Код, подобный показанному в листинге 5.13, можно найти даже в кодовой базе разработчиков, достаточно хорошо знакомых с технологией DI.

Листинг 5.13. Применение окружающего контекста при ведении журналов

```
public class MessageGenerator
{
    private static readonly ILog Logger =
        LogManager.GetLogger(typeof(MessageGenerator));

    public string GetWelcomeMessage()
    {
        Logger.Info("GetWelcomeMessage called.");

        return string.Format(
            "Hello. Current time is: {0}.", DateTime.Now);
    }
}
```

Получение нестабильной зависимости ILog с помощью статического окружающего контекста LogManager и сохранение ее в приватном статическом поле. Это скрывает зависимость и затрудняет тестирование класса MessageGenerator

Использование поля Logger для регистрации каждого вызова метода

Широкая распространенность антипаттерна «Окружающий контекст» во многих приложениях, занимающихся ведением регистрационных записей, обусловливается несколькими причинами. Во-первых, код, похожий на показанный в листинге 5.13, обычно фигурирует в качестве первого примера, показываемого в документации библиотек со средствами журналирования. Разработчики, не разобравшись, копируют эти примеры. Винить их не за что, ведь они предполагают, что разработчикам библиотек известны наиболее рациональные приемы программирования, которыми они готовы поделиться. К сожалению, бывает и иначе. Примеры в документации даются для простоты понимания, а не для распространения передового опыта, даже если он и имеется в арсенале разработчиков библиотек.

Кроме того, антипаттерн «Окружающий контекст» зачастую применяется разработчиками для систем регистрации из-за того, что им нужно входить в своем приложении практически в каждый класс. Внедрение его в конструктор может легко привести к созданию конструкторов с переизбытком зависимостей. В результате получится рассматриваемый в главе 6 проблемный код, который называется *избыточным внедрением через конструктор* (Constructor Over-injection).

Джефф Этвуд (Jeff Atwood) еще в 2008 году опубликовал в блоге большую статью, посвященную опасности регистрирования (https://blog.codinghorror.com/the-problem-with-logging/). Перечислим некоторые из его аргументов.

- Регистрирование ведет к увеличению объема кода, из-за чего становится труднее разобраться в коде самого приложения.

- При регистрировании приходится неизменно чем-то жертвовать, и в большинстве случаев оно предполагает постоянный сброс данных на диск.
- Чем больше объем регистрирования, тем труднее поиск нужной информации.
- Если данные стоит сохранять в регистрационном файле, то их стоит показать в пользовательском интерфейсе.

Во время своей работы над Stack Overflow Джефф удалил практически все регистрирование, оставив только запись необработанных исключений. Если возникла ошибка, должно быть выдано исключение.

Всецело соглашаясь с анализом Джеффа, нам все же хотелось бы подойти к данному вопросу с точки зрения проектирования. По нашему мнению, при качественно спроектированном приложении можно было бы воспользоваться регистрированием взаимоотношений основных компонентов, не засоряя им всю кодовую базу. Более подробно разработка такого приложения рассматривается в главе 10.

ПРИМЕЧАНИЕ

Наши рассуждения ни в коем случае не нужно воспринимать как полный отказ от регистрирования. Ведение регистрационных записей является жизненно важной частью любого приложения, что справедливо и для тех приложений, которые мы создаем. Сказанное нами нужно воспринимать так, что ваше приложение должно разрабатываться с прицелом на то, что журналированием будут охвачены только несколько классов вашей системы. Если за ведение регистрационных записей будет отвечать основная часть компонентов вашего приложения, сопровождать код станет сложнее.

Существует множество других примеров использования антипаттерна «Окружающий контекст», но эти два примера настолько распространены, что в компаниях, обращавшихся к нам за консультацией, нам приходилось сталкиваться с ними несчетное количество раз. (Мы даже винили себя за то, что в прошлом сами внедряли применение антипаттерна «Окружающий контекст» в практику программирования.) После рассмотрения двух самых распространенных примеров применения антипаттерна «Окружающий контекст» в следующем подразделе предстоит понять суть создаваемой им проблемы и способы, позволяющие с ней справиться.

5.3.3. Анализ антипаттерна «Окружающий контекст»

Антипаттерн «Окружающий контекст» обычно встречается, когда у разработчиков появляется повсеместно используемая сквозная функциональность в виде нестабильной зависимости. Такая общедоступность наводит разработчиков на мысль об оправданности их отхода от применения внедрения зависимостей через конструктор. Она позволяет им скрыть зависимости и избавиться от необходимости добавления зависимости к множеству конструкторов в их приложении.

Негативные последствия применения антипаттерна «Окружающий контекст»

Проблемы при использовании антипаттерна «Окружающий контекст» перекликаются с проблемами, связанными с использованием антипаттерна «Локатор сервисов». Вот основные из них.

- Зависимость имеет скрытую форму.
- Тестирование усложняется.
- Затрудняется изменение зависимости, основанной на ее контексте.
- Между инициализацией зависимости и ее использованием возникает временная связанность.

Когда зависимость скрывается за счет разрешения к ней глобального доступа через использование антипаттерна «Окружающий контекст», становится проще утаить факт наличия у класса слишком большого числа зависимостей. Это связано с использованием проблемного кода, который называется избыточным внедрением через конструктор и зачастую служит признаком нарушения принципа единственной ответственности (Single Responsibility Principle).

Когда у класса много зависимостей, это свидетельствует о том, что он берет на себя больше задач, чем нужно. Теоретически возможно наличие класса со множеством зависимостей, если есть «лишь одна причина, приводящая к изменению класса»[1]. Но чем больше класс, тем меньше вероятность, что он будет придерживаться этого принципа. Использование антипаттерна «Окружающий контекст» скрывает факт излишнего усложнения классов и необходимости реструктуризации приложения.

Использование антипаттерна «Окружающий контекст» из-за своего глобального состояния также усложняет и тестирование. Когда тест меняет глобальное состояние, как показано в коде листинга 5.12, это может повлиять на другие тесты. Такое случается, когда тесты запускаются в параллельном режиме, но могут затрагиваться даже последовательно выполняемые тесты, когда тест «забудет» вернуть назад внесенные им изменения в качестве составной части постпроцессинга и разборки. Несмотря на возможность «смягчения проблем», связанных с тестированием, для этого потребуется специально созданный антипаттерн «Окружающий контекст», а также либо глобальная, либо специально разработанная для теста логика его разборки. Все это связано с усложнениями, от которых избавлен альтернативный вариант.

Использование антипаттерна «Окружающий контекст» затрудняет предоставление разным потребителям разных реализаций зависимости. Предположим, что возникла потребность в том, чтобы часть вашей системы работала с того момента времени, который фиксируется при запуске текущего запроса, в то время как другая ее часть, возможно связанная с выполнением долговременных операций, должна

---

[1] *Мартин Р. К., Ньюкирк Дж. В., Косс Р. С.* Быстрая разработка программ. Принципы, примеры, практика. — М.: Вильямс, 2004. — С. 164.

получить зависимость, обновляемую в реальном времени[1]. В коде, показанном в листинге 5.13, именно это и происходит, то есть потребителям предоставляются разные реализации зависимости:

```
private static readonly ILog Logger =
    LogManager.GetLogger(typeof(MessageGenerator));
```

Чтобы иметь возможность предоставлять потребителям разные реализации, API `GetLogger` требуется, чтобы потребитель передавал соответствующую информацию о типе. Тем самым код потребителя неоправданно усложняется.

Применение антипаттерна «Окружающий контекст» приводит к тому, что использование его зависимости приобретает связанность на временном уровне. Если антипаттерн «Окружающий контекст» не будет проинициализирован в корне композиции, приложение будет давать сбой, только когда класс воспользуется зависимостью в первый раз. Нам бы хотелось, чтобы в таком случае наши приложения давали сбой значительно раньше.

Я использую абстракцию. Что при этом может пойти не так?

Я (Стивен) однажды работал на клиента, у которого была огромная кодовая база, в которой использовалось регистрирование на манер показанного в коде листинга 5.13. Это регистрирование велось в постоянном режиме: поскольку разработчики хотели избавиться от прямой зависимости от проблемной библиотеки регистрирования log4net (https://logging.apache.org/log4net/), они воспользовались другой сторонней библиотекой, предоставляющей им абстракцию в виде надстройки над библиотеками регистрирования. Эта библиотека называлась Common.Logging (https://github.com/net-commons/common-logging). Но помощи не последовало, так как библиотека Common.Logging имитировала API библиотеки log4net, скрывая тот факт, что в их проектах зачастую случайно содержалась зависимость от обеих библиотек. Это привело к тому, что многие классы по-прежнему зависели от библиотеки log4net. Но гораздо важнее то, что, даже при сокрытии разработчиками приложения факта использования log4net за абстракцией, зависимость от сторонней библиотеки по-прежнему сохранилась и теперь это выразилось в том, что каждый класс стал зависеть от использования антипаттерна «Окружающий контекст», предоставленного библиотекой Common.Logging (подобно тому как это произошло в коде листинга 5.13).

Проблема начала проявляться при обнаружении ошибки в Common.Logging, при которой происходил сбой вызова статического метода `GetLogger` на определенных машинах разработчика при запуске внутри IIS. На таких машинах разработчика запуск приложения становился невозможным, поскольку первый вызов `LogManager.GetLogger` давал сбой. К моему несчастью, я как раз и был одним из тех двух разработчиков, столкнувшихся с данной проблемой.

Многие разработчики нашей организации помогали нам в попытках разобраться с произошедшим и потратили несметное количество часов, пытаясь выяснить, что

[1] Возможно, кому-то все это покажется надуманным, но в системах, над которыми мы работали на протяжении многих лет, нам встречалось довольно много ошибок, вызванных запросами, проходящими в полночь или при переходе на летнее время.

происходит, но никто из них не нашел решения или пути обхода возникшей проблемы. В конце концов я закомментировал все вызовы антипаттерна «Окружающий контекст» для тех путей кода, которые мне нужно было запускать локально для моей конкретной функции. На тот момент времени переделка под использование DI была, к сожалению, невозможна.

Не сочтите это за намерение придраться к Common.Logging или к log4net, но, позволяя коду приложения получить зависимость от сторонних библиотек, вы идете на вполне определенный риск. Этот риск возрастает, когда зависимость возникает от используемого в библиотеке антипаттерна «Окружающий контекст».

Мораль сей истории в том, что, используй разработчики не антипаттерн «Окружающий контекст», а правильные паттерны DI, я бы запросто смог локально поменять в корне композиции настроенный регистратор на его имитацию, не требующую загрузки Common.Logging. Несколько минут работы сэкономили бы организации потраченное впустую время.

Хотя использование антипаттерна «Окружающий контекст» не оказывает такого же разрушительного воздействия, как использование антипаттерна «Локатор сервисов», поскольку оно скрывает не произвольное число зависимостей, а всего лишь одну нестабильную зависимость, ему не место в качественно проработанной кодовой базе. Всегда найдутся более удачные варианты, речь о которых пойдет в следующем разделе.

Рефакторинг антипаттерна «Окружающий контекст» и переход к применению технологии DI

Не стоит удивляться, увидев применение антипаттерна «Окружающий контекст» даже в той кодовой базе, разработчики которой прекрасно разбираются в DI и во вреде, наносимом применением антипаттерна «Локатор сервисов». Убедить разработчиков, привыкших к использованию антипаттерна «Окружающий контекст», отказаться от него бывает нелегко. Кроме того, если переделка одного класса под использование DI не представляет особого труда, то избавление от укоренившихся проблем, подобных неэффективным и вредным стратегиям регистрирования, дается намного труднее. Обычно имеется солидный объем кода, причины регистрирования работы которого не всегда ясны. Если прежние разработчики давно покинули организацию, определение возможностей удаления подобных регистрирующих инструкций или их превращения в выдачу исключений может протекать весьма медленно. Тем не менее, если технология DI в кодовой базе уже применяется, переделка применения антипаттерна «Окружающий контекст» в применение DI не вызывает затруднений.

Класс-потребитель антипаттерна «Окружающий контекст» обычно содержит один или несколько его вызовов, возможно разбросанных по нескольким методам. Поскольку первым шагом переделки станет централизация вызова кода антипаттерна «Окружающий контекст», то лучшим местом для нее станет конструктор.

Создайте приватное поле, предназначенное только для чтения и содержащее ссылку на зависимость, и присвойте ему зависимость, содержавшуюся в антипаттерне «Окружающий контекст». Отныне этим новым приватным полем сможет воспользоваться весь остальной код класса. Теперь вызов кода антипаттерна «Окружающий контекст» можно будет заменить параметром конструктора, присваивающим полю значение, и граничным оператором, гарантирующим, что значением параметра конструктора не является `null`. Скорее всего, этот новый параметр конструктора нарушит работу потребителей. Но если технология DI уже была применена, изменения должны будут коснуться лишь корня композиции и тестов для проверки классов. Вполне ожидаемые результаты переделки применительно к `WelcomeMessageGenerator` показаны в листинге 5.14.

Листинг 5.14. Переделка, позволяющая уйти от использования антипаттерна «Окружающий контекст» и перейти к внедрению зависимости через конструктор

```
public class WelcomeMessageGenerator
{
    private readonly ITimeProvider timeProvider;

    public WelcomeMessageGenerator(ITimeProvider timeProvider)
    {
        if (timeProvider == null)
            throw new ArgumentNullException("timeProvider");

        this.timeProvider = timeProvider;
    }

    public string GetWelcomeMessage()
    {
        DateTime now = this.timeProvider.Now;
        ...
    }
}
```

Переделка антипаттерна «Окружающий контекст» относительно проста, поскольку по большей части она будет выполняться в приложении, где уже применена технология DI. Для приложений, где она еще не применяется, прежде чем заниматься переделкой антипаттерна «Окружающий контекст», сначала лучше решить проблемы, связанные с применением антипаттернов «Диктатор» и «Локатор сервисов».

Само понятие «окружающий контекст» выглядит как некий весьма удачный способ доступа к повсеместно используемой сквозной функциональности, но внешность обманчива. Будучи менее проблематичным, по сравнению с антипаттернами «Диктатор» и «Локатор сервисов», «Окружающий контекст» обычно является «прикрытием» для более крупных проблем проектирования, имеющихся в приложении. Паттерны, рассмотренные в главе 4, предоставляют более удачные решения, а в главе 10 будет показан порядок разработки приложений, позволяющий применять во всем приложении регистрирование и другие сквозные функциональности проще и прозрачнее.

Последним антипаттерном, рассматриваемым в этой главе, будет «Ограниченная конструкция». Зачастую результатом появления этого антипаттерна является желание добиться позднего связывания.

5.4. Антипаттерн «Ограниченная конструкция»

Самой большой сложностью правильной реализации DI является перемещение всех классов с зависимостями в корень композиции. Когда вы добьетесь этого, считайте, что вы уже практически прошли весь намеченный путь. Но даже после этого некоторые ловушки, требующие сосредоточенности, все же остаются.

Нередко допускается ошибка, при которой от зависимостей требуется наличие конструктора с определенной сигнатурой. Обычно она возникает из желания добиться позднего связывания, при котором зависимости могут быть определены во внешнем конфигурационном файле, благодаря чему могут подвергаться изменениям без перекомпиляции приложения.

ОПРЕДЕЛЕНИЕ

> Антипаттерн «Ограниченная конструкция» вынуждает все реализации заданной абстракции требовать от их конструкторов наличия одинаковой сигнатуры с целью разрешения позднего связывания.

Следует иметь в виду, что этот раздел относится только к сценариям, требующим позднего связывания. В тех случаях, когда ссылки на все зависимости делаются напрямую из корня приложения, подобная проблема не возникает. Но опять же возможности замены зависимостей без перекомпиляции стартового проекта у вас не будет. Практическое применение антипаттерна «Ограниченная конструкция» показано в листинге 5.15.

Листинг 5.15. Пример антипаттерна «Ограниченная конструкция»

```
public class SqlProductRepository : IProductRepository
{
    public SqlProductRepository(string connectionStr)
    {
    }
}

public class AzureProductRepository : IProductRepository
{
    public AzureProductRepository(string connectionStr)
    {
    }
}
```

Навязывание конкретной сигнатуры конструкторам в реализациях IProductRepository

Все реализации абстракции `IProductRepository` вынуждены иметь конструктор с одной и той же сигнатурой. В данном примере у конструктора должен быть только один аргумент типа `string`. Хотя само наличие у класса зависимости типа `string`

никаких возражений не вызывает, проблема для таких реализаций в том, что им навязывается наличие одинаковой сигнатуры конструктора. Эта проблема уже вкратце затрагивалась в подразделе 1.2.2. Сейчас предстоит более тщательное изучение ее особенностей.

5.4.1. Пример: позднее связывание ProductRepository

В учебном приложении электронной торговли некоторые классы зависели от интерфейса `IProductRepository`. Следовательно, прежде, чем создавать эти классы, нужно создать реализацию `IProductRepository`. На данный момент уже понятно, что самым правильным местом для выполнения этой задачи является корень композиции. В приложении среды ASP.NET Core обычно имеется в виду класс `Startup`. В листинге 5.16 показана соответствующая часть, в которой создается экземпляр `IProductRepository`.

Листинг 5.16. Неявное ограничение, накладываемое на конструктор ProductRepository

```
string connectionString = this.Configuration
    .GetConnectionString("CommerceConnectionString");

var settings =
    this.Configuration.GetSection("AppSettings");

string productRepositoryTypeName =
    settings.GetValue<string>("ProductRepositoryType");

var productRepositoryType =
    Type.GetType(
        typeName: productRepositoryTypeName,
        throwOnError: true);

var constructorArguments =
    new object[] { connectionString };

IProductRepository repository =
    (IProductRepository)Activator.CreateInstance(
        productRepositoryType, constructorArguments);
```

Считывание строки подключения из конфигурационного файла приложения

Считывание имени типа создаваемого хранилища из раздела AppSettings конфигурационного файла

Загрузка объекта Type, относящегося к типу хранилища

Создание экземпляра типа хранилища, ожидающего конкретной сигнатуры. Для тех компонентов, которые требуют другой сигнатуры конструктора, этот вызов даст сбой

Соответствующий конфигурационный файл показан в следующем примере кода:

```
{
    "ConnectionStrings": {
        "CommerceConnectionString":
            "Server=.;Database=MaryCommerce;Trusted_Connection=True;"
    },
    "AppSettings": {
        "ProductRepositoryType": "SqlProductRepository, Commerce.SqlDataAccess"
    },
}
```

Первое, что должно вызывать подозрение, — это считывание строки подключения из конфигурационного файла. Зачем нужна строка подключения, если `ProductRepository` планируется рассматривать в качестве абстракции?

Возможно, это маловероятно, но можно выбрать реализацию `ProductRepository` с использованием базы данных в памяти или XML-файла. Сервис-хранилище на основе REST, например Windows Azure Table Storage Service, предлагает более реалистичную альтернативу; на сегодняшний день наиболее популярным вариантом представляется реляционная база данных. Повсеместное распространение баз данных слишком легко позволяет забыть, что строка подключения неявно скрывает за собой выбор реализации.

Для позднего связывания `IProductRepository` вам также нужно определить, какой тип был выбран в качестве реализации. Это можно сделать путем чтения из конфигурации имени типа, указывающего на сборку, и создания экземпляра `Type` из этого имени. Это само по себе не проблематично. Сложность возникает, когда нужно создать экземпляр этого типа. Располагая классом `Type`, экземпляр можно создать с помощью класса `Activator`. Метод `CreateInstance` вызывает конструктор типа, и во избежание выдачи исключения нужно предоставить правильные параметры конструктора. В данном случае вы предоставляете строку подключения.

Если о приложении не известно ничего, кроме кода, показанного в листинге 5.16, то следует задаться вопросом, почему строка подключения передается в качестве аргумента конструктора неизвестному типу. Если бы реализация была построена в виде веб-сервиса на основе REST или в виде XML-файла, то это бы не имело никакого смысла.

Фактически в этом нет никакого смысла, поскольку мы имеем дело со случайным ограничением, накладываемым на конструктор зависимости. В данном случае имеется неявное требование, что у любой реализации `IProductRepository` должен быть конструктор, принимающий в качестве входных данных только одну строку. И это вдобавок к явному ограничению, предписывающему классу быть производным от `IProductRepository`.

ПРИМЕЧАНИЕ

> Неявное ограничение, предписывающее конструктору принимать только одну строку, все же оставляет вам весьма широкую лазейку для проявления гибкости, поскольку в строке можно закодировать различную информацию для ее последующего раскодирования. Вместо этого предположим, что ограничение выражается в конструкторе, принимающем TimeSpan и число, и вот теперь можно поразмышлять над тем, насколько сузятся возможности.

Можно согласиться с тем, что `IProductRepository` на основе XML-файла также потребует в качестве параметра конструктора строку, хотя она будет именем файла, а не строкой подключения. Но концептуально это все же будет выглядеть странно, поскольку имя этого файла нужно будет определить в элементе конфигурации `connectionStrings`. (В любом случае представляется, что такое гипотетическое хранилище товаров `XmlProductRepository` должно принимать в качестве параметра конструктора не имя файла, а `XmlReader`.)

СОВЕТ

Более подходящим и гибким вариантом является моделирование конструкции зависимости исключительно на явных ограничениях (интерфейсе или базовом классе).

5.4.2. Анализ антипаттерна «Ограниченная конструкция»

В предыдущем примере на реализатора накладывалось неявное ограничение по наличию у конструктора одного строкового параметра. Более распространенное ограничение заключается в том, чтобы у всех реализаций был конструктор без параметров, чтобы срабатывала простейшая форма `Activator.CreateInstance`:

```
IProductRepository repository =
    (IProductRepository)Activator.CreateInstance(productRepositoryType);
```

Хотя можно будет сказать, что это наименьший общий знаменатель, затраты на достижение гибкости будут весьма существенными. Независимо от того, каким образом ограничивается конструкция объекта, вы утрачиваете гибкость.

Негативные последствия применения антипаттерна «Ограниченная конструкция»

Объявление о том, что у всех реализаций зависимости должен быть конструктор без параметров, может звучать весьма заманчиво. Ведь инициализацию они могут выполнить в качестве своей внутренней операции; например, прочитать непосредственно из конфигурационного файла такие настроечные данные, как строка подключения. Но при этом возникнут другие ограничения, поскольку может появиться требование о компоновке приложения в виде уровней экземпляров, в которых заключены другие экземпляры. К примеру, в некоторых случаях, как показано на рис. 5.6, может понадобиться совместное использование экземпляра разными потребителями.

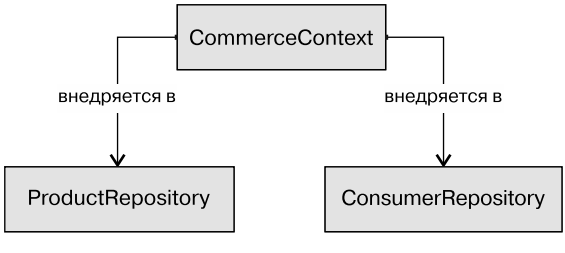


Рис. 5.6. Требуется создать единственный экземпляр класса CommerceContext и внедрить его в оба хранилища

Когда потребность в одной и той же зависимости испытывается не одним классом, может появиться желание о совместном использовании одного экземпляра

всеми этими классами. Такая возможность открывается только тогда, когда можно внедрить этот экземпляр извне. Хотя код для считывания информации о типе из конфигурационного файла и использования `Activator.CreateInstance` для создания надлежащего типа экземпляра можно записать внутри каждого такого класса, в реальности подобное совместное использование одного и того же экземпляра представляло бы немалую трудность. Вместо этого пришлось бы иметь дело с несколькими экземплярами одного и того же класса, допуская при этом повышенный расход оперативной памяти.

ПРИМЕЧАНИЕ

Тот факт, что технология DI позволяет совместно использовать один экземпляр многими потребителями, не означает, что это нужно делать всегда. Совместное использование экземпляра экономит память, но может вызвать проблемы взаимодействия, связанные, к примеру, с потоками. Вопрос о том, нужно ли совместно использовать экземпляр, тесно связан с понятием времени жизни объекта, которое рассматривается в главе 8.

Вместо накладывания неявных ограничений на способ создания объектов нужно реализовать корень композиции, чтобы он мог работать с любым закладываемым в него конструктором или фабричным методом. Теперь посмотрим, как можно перейти к применению технологии DI.

Рефакторинг антипаттерна «Ограниченная конструкция» и переход к применению технологии DI

Как можно справиться с задачей применения конструкторов компонентов, не имеющих никаких ограничений, когда требуется применить позднее связывание? Может возникнуть соблазн ввести абстрактную фабрику, способную создать экземпляры запрошенной абстракции, после чего запросить у реализации этой абстрактной фабрики наличия конкретной сигнатуры конструктора. Но это, скорее всего, приведет к усложнению самой этой фабрики. Рассмотрим такой подход.

Представим себе использование абстрактной фабрики для абстракции `IProductRepository`. Схема абстрактной фабрики требует также наличия интерфейса `IProductRepositoryFactory`. Эта структура показана на рис. 5.7.

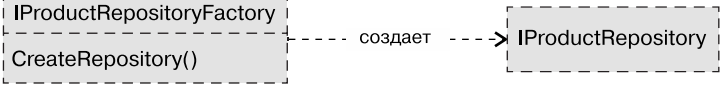


Рис. 5.7. Попытка использования структуры абстрактной фабрики для решения задачи позднего связывания

Реальная зависимость на этом рисунке представлена `IProductRepository`. Но, чтобы не накладывать на ее реализации предполагаемых ограничений, предпринимается попытка решения задачи позднего связывания путем ввода фабрики

IProductRepositoryFactory, которая будет использоваться для создания экземпляров IProductRepository. Еще одно требование заключается в наличии у любых фабрик конкретной сигнатуры конструктора.

Теперь предположим, что нужно воспользоваться показанной в листинге 5.17 реализацией IProductRepository, запрашивающей для работы экземпляр IUserContext.

Листинг 5.17. Класс SqlProductRepository, требующий IUserContext

```
public class SqlProductRepository : IProductRepository
{
    private readonly IUserContext userContext;
    private readonly CommerceContext dbContext;

    public SqlProductRepository(
        IUserContext userContext, CommerceContext dbContext)
    {
        if (userContext == null)
            throw new ArgumentNullException("userContext");
        if (dbContext == null)
            throw new ArgumentNullException("dbContext");

        this.userContext = userContext;
        this.dbContext = dbContext;
    }
}
```

Класс SqlProductRepository реализует интерфейс IProductRepository, но ему требуется экземпляр IUserContext. Поскольку единственный конструктор не является конструктором без параметров, IProductRepositoryFactory будет весьма кстати.

В данный момент нужно воспользоваться реализацией IUserContext, основанной на применении среды ASP.NET Core. Эта реализация (как уже выяснилось в листинге 3.12) называется AspNetUserContextAdapter. Поскольку реализация зависит от среды ASP.NET Core, она не определяется в той же сборке, что и SqlProductRepository. Поскольку перетаскивать ссылку в библиотеку, содержащую AspNetUserContextAdapter, а также SqlProductRepository, желания нет, единственным решением, показанным на рис. 5.8, остается реализация SqlProductRepositoryFactory в сборке, отличной от той, где находится SqlProductRepository.

Возможная реализация SqlProductRepositoryFactory показана в листинге 5.18.

Несмотря на то что IProductRepository и IProductRepositoryFactory выглядят как связанная пара, важно реализовать их в двух разных сборках. Дело в том, что фабрика, чтобы правильно связать друг с другом все зависимости, должна располагать ссылками на эти зависимости. По соглашению, чтобы у вас была возможность записать в конфигурационный файл полное имя типа, содержащее имя сборки, и воспользоваться для создания экземпляра Activator.CreateInstance, реализация IProductRepositoryFactory снова должна использовать антипаттерн «Ограниченная конструкция».

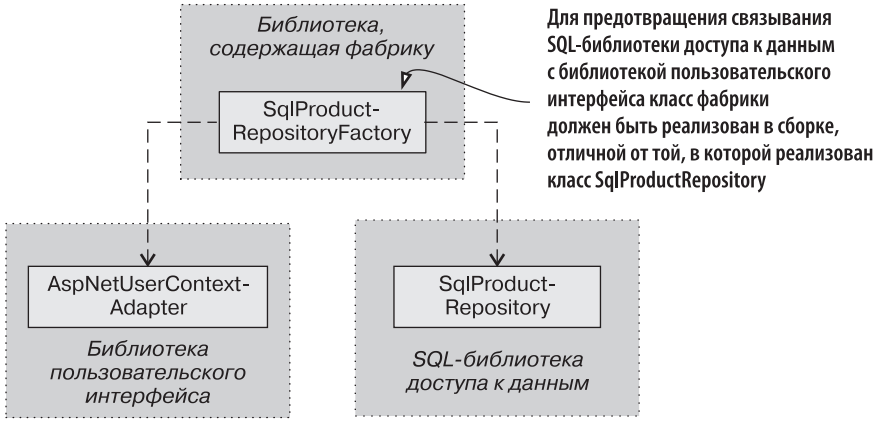


Рис. 5.8. Граф зависимостей с фабрикой SqlProductRepositoryFactory, реализованной в отдельной сборке

Листинг 5.18. Фабрика, создающая экземпляры SqlProductRepository

```
public class SqlProductRepositoryFactory
    : IProductRepositoryFactory
{
    private readonly string connectionString;

    public SqlProductRepositoryFactory(
        IConfigurationRoot configuration)
    {
        this.connectionString =
            configuration.GetConnectionString(
                "CommerceConnectionString");
    }

    public IProductRepository Create()
    {
        return new SqlProductRepository(
            new AspNetUserContextAdapter(),
            new CommerceContext(this.connectionString));
    }
}
```

Конструктор с конкретной сигнатурой, которая должна быть у всех реализаций IProductRepositoryFactory. Принимая разработанный Microsoft IConfigurationRoot, фабрика загружает нужные ей значения конфигурации. Она также выдает ошибку при создании, если такого значения нет

Загрузка строки подключения из конфигурационного файла и ее сохранение для дальнейшего использования

Создание нового экземпляра SqlProductRepository с использованием зависимостей, которые находятся в других сборках

Как только появляется потребность в связывании новой комбинации зависимостей, нужно реализовывать новую фабрику, которая занимается связыванием именно этой комбинации, а затем настраивать приложение на использование этой фабрики вместо предыдущей. Это означает, что вы не можете определить произвольную комбинацию зависимостей без написания и компилирования кода, но это можно сделать без перекомпиляции самого приложения. Такая абстрактная фабрика становится абстрактным корнем композиции, который определен в отдельной от основного приложения сборке. Несмотря на наличие такой возмож-

ности, при попытке ее применения будет чувствоваться присущее ей отсутствие гибкости.

Утрата гибкости связана с тем, что абстрактный корень композиции для удовлетворения потребностей создаваемых им графов объектов берет прямые зависимости конкретных типов в других библиотеках. В примере `SqlProductRepositoryFactory` фабрике необходимо создать экземпляр `AspNetUserContextAdapter` для передачи в `SqlProductRepository`. Что же делать, если основное приложение требует замены или перехвата реализации `IUserContext`? Приходится вносить изменения как в основное приложение, так и в проект `SqlProductRepositoryFactory`. Еще одна проблема заключается в том, что этой абстрактной фабрике становится крайне сложно управлять временем жизни объекта. Это та же самая проблема, которая была показана на рис. 5.5.

Единственным возможным решением победить утрату гибкости является применение DI-контейнера общего назначения. Поскольку DI-контейнеры, используя отображение, анализируют сигнатуру конструктора, абстрактный корень композиции не нуждается в информации о зависимостях, применяемых для построения его компонентов. Единственное, что нужно сделать абстрактному корню композиции, — это указать отображение между абстракцией и реализацией. Иными словами, корень композиции, использующий доступ к данным по технологии SQL, должен указать, что если приложению требуется `IProductRepository`, то должен быть создан экземпляр `SqlProductRepository`.

СОВЕТ

> Использование DI-контейнера может стать весьма действенным решением, предотвращающим возникновение антипаттерна «Ограниченная конструкция». Детальное изучение принципов работы и использования DI-контейнеров предстоит в части IV.

Потребность в абстрактном корне композиции возникает только при реальной необходимости в подключении новой сборки без обязательной перекомпиляции какой-либо из частей уже существующего приложения. Большинству приложений такая гибкость не нужна. Хотя может потребоваться замена уровня доступа к данным по технологии SQL уровнем доступа к данным по технологии Azure без обязательной перекомпиляции доменного уровня, обычно при этом вполне допустима по-прежнему подразумеваемая необходимость внесения изменений в проект запуска.

ПРИМЕЧАНИЕ

> Антипаттерн «Ограниченная конструкция» применяется только при использовании позднего связывания. Когда применяется раннее связывание, компилятор не позволяет вводить неявные ограничения на способы создания компонентов. Если есть возможность обойтись перекомпиляцией лишь проекта запуска, нужно сосредоточить в нем корень композиции. Позднее связывание усложняет приложение, удорожая тем самым сопровождение.

Поскольку технология DI представляет собой набор паттернов и приемов программирования, ни одно отдельно взятое средство не может чисто механически проверить правильность ее применения. В главе 4 рассматривались паттерны и приводилось описание надлежащего применения технологии DI, но это лишь одна сторона медали. Не менее важно изучить возможности неудач, даже при вынашивании самых благих намерений. Из ошибок можно извлечь весьма важные уроки, но постоянно учиться на своих собственных промахах совсем не обязательно — порой неплохо было бы учиться на ошибках других людей.

В этой главе в форме антипаттернов были даны описания наиболее часто встречающихся ошибок, допускаемых при намерении применить технологию DI. Нам все эти ошибки неоднократно попадались в реальной жизни, и мы чувствуем себя за них виноватыми. Но теперь вы уже должны понимать, чего следует избегать и что в идеале нужно делать. И все же еще остаются проблемы, решение которых представляется весьма непростым. Именно им и их устранению будет посвящена следующая глава.

Резюме

- Антипаттерны представляют собой описание часто принимаемого решения проблемы, порождающего явно негативные последствия.
- Антипаттерн «Диктатор» является доминирующим антипаттерном, представленным в этой главе. Он создает существенные препятствия на пути применения любой разновидности надлежащей технологии DI. Он возникает всякий раз, когда конструкция полагается на применение нестабильной зависимости везде, кроме корня композиции.
- Хотя применение ключевого слова `new` в отношении нестабильной зависимости является признаком проблемного кода, его использование в отношении стабильной зависимости не должно вызвать беспокойства. Кодовое слово `new` нельзя считать внезапно ставшим недопустимым, но нужно все же воздерживаться от его использования для получения экземпляров нестабильных зависимостей.
- Применение антипаттерна «Диктатор» является нарушением принципа инверсии зависимости.
- Антипаттерн «Диктатор» представляет способ создания экземпляров по умолчанию в большинстве языков программирования, поэтому он может встречаться даже в приложениях, для которых разработчики никогда не рассматривали применение технологии DI. Это настолько естественный и укоренившийся в сознании разработчиков способ создания новых объектов, что многим из них весьма нелегко от него отказаться.
- Внешняя реализация по умолчанию является противоположностью локальной реализации по умолчанию. Это реализация зависимости по умолчанию, когда

ее определение находится не в том модуле, в котором определен ее потребитель. Перетаскивание нежелательных модулей лишает многих преимуществ слабой связанности.

- Самым опасным антипаттерном, представленным в этой главе, является «Локатор сервисов», поскольку он выглядит так, будто решает проблему. Он предоставляет компонентам приложения, находящимся за пределами корня композиции, доступ к неограниченному набору нестабильных зависимостей.
- Антипаттерн «Локатор сервисов» негативно влияет на возможность многократного использования компонентов-потребителей. Он также скрывает от компонентов-потребителей имеющиеся у них зависимости, не позволяя составить истинное представление об уровне сложности таких компонентов и заставляя компоненты-потребители этого антипаттерна притягивать его к себе в качестве избыточной зависимости.
- Антипаттерн «Локатор сервисов» препятствует проверке конфигурации отношений между классами. Внедрение через конструктор в сочетании с чистой технологией DI позволяет проводить проверку во время выполнения программы, а внедрение через конструктор в сочетании с применением DI-контейнера позволяет проводить проверку при запуске приложения или же в качестве составной части простого автоматизированного теста.
- Статический антипаттерн «Локатор сервисов» приводит к появлению взаимозависимых тестов, поскольку он остается в памяти при выполнении следующего тестового задания.
- Антипаттерн «Локатор сервисов» определяется не механической структурой API, а той ролью, которую API играет в приложении. Поэтому DI-контейнер, заключенный в корне композиции, является не антипаттерном «Локатор сервисов», а компонентом инфраструктуры.
- Антипаттерн «Окружающий контекст» предоставляет коду приложения, находящемуся вне корня композиции, глобальный доступ к нестабильной зависимости или к ее поведению путем использования статических компонентов класса.
- Антипаттерн «Окружающий контекст» по своей структуре похож на паттерн «Одиночка», за исключением того, что он позволяет своей зависимости изменяться. Паттерн «Одиночка» гарантирует, что один созданный экземпляр никогда не изменится.
- Антипаттерн «Окружающий контекст» обычно встречается, когда у разработчиков в качестве зависимости есть повсеместно используемая сквозная функциональность, заставляющая полагать, что этим оправдывается уход от использования внедрения через конструктор.
- Антипаттерн «Окружающий контекст» придает нестабильной зависимости скрытность, усложняя тестирование и затрудняя изменение зависимости из-за его контекста.

- Антипаттерн «Ограниченная конструкция» вынуждает все реализации заданной абстракции требовать от их конструкторов наличия одинаковой сигнатуры с целью разрешения позднего связывания. Он ограничивает гибкость и может заставить проводить инициализацию реализаций внутри их самих.
- Появление антипаттерна «Ограниченная конструкция» можно предотвратить, используя DI-контейнер общего назначения, поскольку DI-контейнеры выполняют анализ сигнатуры конструктора, применяя отображение.
- Если можно обойтись перекомпиляцией лишь проекта запуска приложения, нужно сохранить централизацию вашего корня композиции в проекте запуска и воздержаться от использования позднего связывания. Позднее связывание усложняет приложение, что, в свою очередь, увеличивает затраты на обслуживание.

6 Проблемный код

В этой главе

- Как переделать проблемный код с избыточным внедрением через конструктор.
- Как обнаружить и предотвратить злоупотребление абстрактными фабриками.
- Как исправить проблемный код с зацикленностью зависимостей.

Вы не могли не заметить, что я (Марк) люблю готовить беарнский соус (или соус голландез). Во-первых, у него великолепный вкус, а во-вторых, его непросто готовить. Кроме готовки, есть еще совершенно иная проблема: он тут же должен быть подан к столу (по крайней мере, именно так мне представляется).

Самым неподходящим моментом его приготовления было прибытие гостей. Вместо их непринужденного приветствия и создания у них чувства своего искреннего гостеприимства, я лихорадочно взбивал соус на кухне, предоставляя им возможность развлекаться самостоятельно. После парочки таких представлений моя весьма общительная жена решила взять дело в свои руки. Мы жили напротив ресторана, и однажды она поболтала с поварами, чтобы выяснить, нет ли какого-то хитрого приема, позволяющего мне приготовить настоящий голландский соус заранее. Оказалось, что есть. Теперь я могу подавать своим гостям этот вкусный соус, не повергая их перед этим в атмосферу стресса и крайней неловкости.

В каждом ремесле есть свои тонкости. Справедливость этого утверждения распространяется и на разработку программных продуктов в целом, и на внедрение зависимостей в частности. Жизнь постоянно подбрасывает все новые испытания. Во многих случаях имеются хорошо известные способы, позволяющие справиться с возникающими проблемами. Многие годы мы наблюдали те трудности, с которыми сталкивались люди, изучавшие технологию DI, и многие из них были схожими по своей природе. В этой главе будут рассмотрены наиболее распространенные примеры проблемного кода, возникающего при применении технологии DI к кодовой базе, и способы избавления от проблем. Усвоив материал главы, вы сможете повысить свои навыки распознавания и исправления тех ситуаций, при которых появляется проблемный код.

По аналогии с двумя предыдущими главами этой части книги, данная глава имеет структуру каталога, и теперь это каталог проблем и решений (или, если так вам понятнее, переделок). Каждый раздел можно изучать по отдельности или читать последовательно — все это по вашему усмотрению. Целью каждого раздела ставится ваше ознакомление с решением часто возникающей проблемы, чтобы в случае ее появления вы могли встретить ее во всеоружии. Но сначала нужно дать определение проблемному коду.

ОПРЕДЕЛЕНИЕ

> Проблемный код — это не несомненный факт, а намек на то, что что-то в коде может быть не так. Вполне приличная идиома проектирования может рассматриваться в качестве проблемного кода по причине ее частого неправильного использования или при наличии более простых альтернатив, удачнее работающих в большинстве случаев. Если что-то называют проблемным кодом, то это не нападки на его создателей, а сигнал, призывающий пристальнее приглядеться к этому коду (http://wiki.c2.com/?CodeSmell).

Если антипаттерн является описанием решения часто встречающейся проблемы, порождающей явно негативные последствия, то проблемный код является конструкцией, способной осложнить работу. Проблемный код просто требует более пристального исследования.

6.1. Как справиться с проблемным кодом избыточного внедрения через конструктор

Если у вас нет каких-то особых требований, внедрение через конструктор (рассмотренное в главе 4) должно быть предпочтительным паттерном внедрения. Хотя реализация и использование внедрения через конструктор не требуют больших усилий, разработчики испытывают дискомфорт, когда их конструкторы приобретают следующий вид (листинг 6.1).

Листинг 6.1. Конструктор с большим количеством зависимостей

```
public OrderService(
    IOrderRepository orderRepository,
    IMessageService messageService,
    IBillingSystem billingSystem,
    ILocationService locationService,
    IInventoryManagement inventoryManagement)
{
    if (orderRepository == null)
        throw new ArgumentNullException("orderRepository");
    if (messageService == null)
        throw new ArgumentNullException("messageService");
    if (billingSystem == null)
        throw new ArgumentNullException("billingSystem");
    if (locationService == null)
        throw new ArgumentNullException("locationService");
    if (inventoryManagement == null)
        throw new ArgumentNullException("inventoryManagement");

    this.orderRepository = orderRepository;
    this.messageService = messageService;
    this.billingSystem = billingSystem;
    this.locationService = locationService;
    this.inventoryManagement = inventoryManagement;
}
```

Зависимости OrderService

Наличие большого количества зависимостей является признаком нарушения принципа единственной ответственности (Single Responsibility Principle, SRP). Это нарушение приводит к коду, который трудно сопровождать.

В этом разделе будет рассмотрена реальная проблема растущего числа параметров конструктора и выяснено, почему к внедрению через конструктор не стоит относиться плохо. При этом выяснится, что конструктору совсем не обязательно принимать длинные списки параметров и что можно сделать, чтобы этого не происходило. Переделывать код, избавляясь от чрезмерного внедрения зависимостей через конструктор, можно по-разному, поэтому, кроме всего прочего, будут рассмотрены два самых распространенных подхода: фасадные сервисы (Facade Services) и доменные события (domain events).

- Под фасадными сервисами понимаются абстрактные фасады[1], связанные с граничными объектами (перевод из книги Мартина Фаулера. — *Примеч. ред.*) (Parameter Objects)[2]. Но вместо объединения компонентов и предоставления их в качестве параметров фасадный сервис предоставляет только инкапсулированное поведение, скрывая компоненты.
- Доменные события позволяют захватывать действия, которые могут инициировать изменение состояния разрабатываемого приложения.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 221.

[2] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 297.

6.1.1. Распознавание избыточного внедрения через конструктор

Когда перечень параметров конструктора сильно разрастается, это называется *избыточным внедрением зависимостей через конструктор* и считается проблемным кодом[1]. Эта проблема общего порядка, не имеющая отношения к технологии DI, но усиливаемая ею. Хотя первой реакцией может стать отказ от технологии внедрения через конструктор из-за избыточности, мы должны быть ей благодарны за то, что она открыла нам общую проблему проектирования.

Мы не можем кого-то обвинять в антипатии к конструктору, показанному в коде листинга 6.1, но не вините в этой ситуации внедрение через конструктор. Можно согласиться с тем, что конструктор с пятью параметрами является проблемным кодом, но он свидетельствует о нарушении принципа единственной ответственности, а не о наличии проблем, связанных с самой технологией DI.

ПРИМЕЧАНИЕ

> Внедрение через конструктор облегчает обнаружение нарушения принципа единственной ответственности. Вместо чувства досады от избыточного внедрения через конструктор нужно все это воспринимать как полезный побочный эффект от данного способа внедрения. Это тревожный сигнал, свидетельствующий о том, что класс берет на себя слишком большую ответственность.

Мы установили пороговое значение в четыре аргумента конструктора. Добавляя третий аргумент, мы уже начинаем обдумывать возможность изменения конструкции, но для некоторых классов мы можем смириться и с четырьмя аргументами. Ваш лимит может отличаться от нашего, но его превышение свидетельствует о наступлении момента оценки сложившейся обстановки.

Как именно нужно переработать тот или иной разросшийся класс, зависит от конкретных обстоятельств: уже имеющейся модели объектов, предметной области, бизнес-логики и т. д. Неизменную пользу принесет движение в направлении разбиения постоянно разрастающегося «всемогущего класса» (God Class) на более мелкие, конкретизированные в соответствии с общеизвестными паттернами проектирования классы[2]. И все же бывают случаи, когда бизнес-требования обязывают к одновременному занятию сразу несколькими разнообразными делами. Такое часто происходит на границе приложения. Представьте себе работу крупного веб-сервиса, запускающего множество бизнес-событий.

---

[1] *Palermo J.* Constructor over-injection smell — follow up, 2010; https://mng.bz/jrzr.

[2] Всемогущим, или наивысшим, классом (God Class) называется объект, управляющий слишком большим количеством объектов в системе и вне всякой логики разросшийся до того, чтобы стать классом, выполняющим абсолютно все. См.: *Brown W. J. et al.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. — Wiley Computer Publishing, 1998. — P. 73.

ПРИМЕЧАНИЕ

Весьма заманчивой, но ошибочной попыткой решения проблемы избыточного внедрения через конструктор является применение внедрения через свойство, возможно, даже путем перемещения соответствующих свойств в базовый класс. Хотя путем устранения зависимостей конструктора с помощью свойств их число может быть сокращено, подобное изменение не способствует упрощению класса, и это должно вас беспокоить в первую очередь.

Можно разработать и реализовать объекты-сотрудники, позволяющие не нарушать принцип единственной ответственности. В главе 9 будет рассмотрено, как паттерн проектирования «Декоратор»[1] может помочь вам надстраивать сквозную функциональность вместо внедрения ее в потребителей в качестве сервисов. Тем самым можно будет сократить большое число аргументов конструктора. В некоторых сценариях для управления множеством зависимостей нужна одна точка входа. Одним из примеров может послужить работа веб-сервиса, запускающего сложные взаимодействия большого числа различных сервисов. Такая же проблема может быть у точки входа запланированного пакетного задания.

Учебное приложение электронной торговли, к которому мы время от времени возвращаемся, должно иметь возможность получать заказы. Зачастую это лучше всего делается с помощью отдельного приложения или подсистемы, поскольку на данном этапе меняется семантика транзакции. При просмотре корзины покупок могут динамически рассчитываться цены за единицы товара, обменные курсы и скидки. Но когда покупатель размещает заказ, все эти значения должны быть зафиксированы и заморожены в том виде, в котором они были представлены на момент приема заказа от покупателя. Обзор процесса размещения заказа показан в табл. 6.1.

Таблица 6.1. Когда подсистема заказов одобряет заказ, она должна выполнить ряд различных действий

Действие	Требуемые зависимости
Обновление заказа	IOrderRepository
Отправка покупателю квитанции по электронной почте	IMessageService
Уведомление системы бухгалтерского учета о сумме счета	IBillingSystem
Выбор наиболее подходящего склада для составления и отправки заказа на основе приобретенных товаров и близости к адресу доставки	ILocationService, IInventoryManagement
Выдача требования выбранному складу на составление и отправку всего заказа или его части	IInventoryManagement

Чтобы просто принять заказ, требуется пять различных зависимостей. Представьте себе другие зависимости, которые понадобятся для проведения иных операций, связанных с заказом!

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 209.

ПРИМЕЧАНИЕ

В большинстве примеров, показанных до этой страницы, имелись граничные операторы. Полагаем, что мы уже вполне достаточно подчеркнули их важность. Поэтому, начиная с листинга 6.2, мы станем опускать большинство граничных операторов.

Посмотрим, на что это будет похоже, если класс-потребитель `OrderService` импортирует все эти зависимости напрямую. В следующем листинге дан краткий обзор внутренних компонентов этого класса.

Листинг 6.2. Исходный класс OrderService со множеством зависимостей

```
public class OrderService : IOrderService
{
    private readonly IOrderRepository orderRepository;
    private readonly IMessageService messageService;
    private readonly IBillingSystem billingSystem;
    private readonly ILocationService locationService;
    private readonly IInventoryManagement inventoryManagement;

    public OrderService(
        IOrderRepository orderRepository,
        IMessageService messageService,
        IBillingSystem billingSystem,
        ILocationService locationService,
        IInventoryManagement inventoryManagement)
    {
        this.orderRepository = orderRepository;
        this.messageService = messageService;
        this.billingSystem = billingSystem;
        this.locationService = locationService;
        this.inventoryManagement = inventoryManagement;
    }

    public void ApproveOrder(Order order)
    {
        this.UpdateOrder(order);
        this.Notify(order);
    }

    private void UpdateOrder(Order order)
    {
        order.Approve();
        this.orderRepository.Save(order);
    }

    private void Notify(Order order)
    {
        this.messageService.SendReceipt(new OrderReceipt { ... });
        this.billingSystem.NotifyAccounting(...);
```

Запись нового статуса заказа в базу данных

Уведомление других систем о заказе

```
        this.Fulfill(order);
    }

    private void Fulfill(Order order)
    {
        this.locationService.FindWarehouses(...);
        this.inventoryManagement.NotifyWarehouses(...);
    }
}
```

Поиск ближайшего склада (ближайших складов)

Уведомление склада (складов) о заказе

Чтобы было проще разобраться в примере, мы опустили основные подробности конструкции класса. Но нетрудно представить, что такой класс был бы достаточно большим и сложным. Если допустить непосредственное потребление классом `OrderService` всех пяти зависимостей, то получится довольно большое число четко определенных зависимостей. Эта структура показана на рис. 6.1.

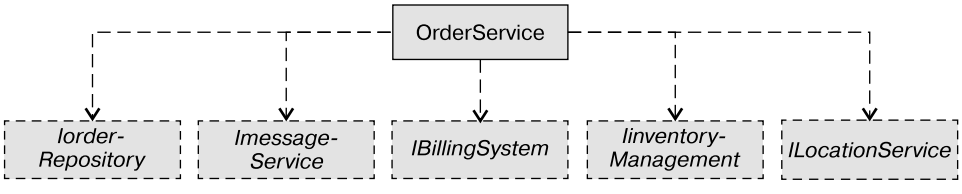


Рис. 6.1. У класса OrderService имеется пять непосредственных зависимостей, что говорит о нарушении принципа единственной ответственности

Если использовать для класса `OrderService` внедрение через конструктор (что и нужно сделать), получится конструктор с пятью параметрами. Это слишком много и свидетельствует о том, что у `OrderService` имеется чересчур много обязанностей. С другой стороны, без всех этих зависимостей просто не обойтись, поскольку при получении нового заказа класс `OrderService` должен стать реализацией всей требуемой функциональности. Решить эту проблему можно за счет изменения конструкции `OrderService` под использование фасадных сервисов. В следующем разделе мы покажем вам, как это делается.

6.1.2. Переделка избыточного внедрения через конструктор в фасадные сервисы

Первое, что нужно сделать при изменении конструкции `OrderService`, — это найти естественные группы взаимодействий. Сразу же ваше внимание должно привлечь взаимодействие между `ILocationService` и `IInventoryManagement`, поскольку они используются для поиска ближайших складов, способных выполнить заказ. Потенциально это может стать сложным алгоритмом.

После того как склады выбраны, их нужно уведомить о заказе. Если вдуматься, `ILocationService` является реализацией детали уведомления соответствующих

складов о заказе. Все взаимодействие может быть скрыто за интерфейсом `IOrderFulfillment`:

```
public interface IOrderFulfillment
{
    void Fulfill(Order order);
}
```

Реализация нового интерфейса `IOrderFulfillment` показана в листинге 6.3.

Листинг 6.3. Класс OrderFulfillment

```
public class OrderFulfillment : IOrderFulfillment
{
    private readonly ILocationService locationService;
    private readonly IInventoryManagement inventoryManagement;

    public OrderFulfillment(
        ILocationService locationService,
        IInventoryManagement inventoryManagement)
    {
        this.locationService = locationService;
        this.inventoryManagement = inventoryManagement;
    }

    public void Fulfill(Order order)
    {
        this.locationService.FindWarehouses(...);
        this.inventoryManagement.NotifyWarehouses(...);
    }
}
```

Интересно, что само по себе выполнение заказа во многом похоже на доменную концепцию. Видимо, обнаружилась неявная доменная концепция, которая перевоплотилась в явную.

Исходная реализация `IOrderFulfillment` потребляет две изначальные зависимости, то есть имеет конструктор с двумя параметрами, что нас вполне устраивает. В качестве дополнительного преимущества алгоритм поиска наиболее подходящего склада для конкретного заказа теперь заключен в компонент, пригодный для многократного использования. Новая абстракция `IOrderFulfillment` является фасадным сервисом, поскольку она скрывает две взаимодействующие зависимости вместе с их поведением.

ОПРЕДЕЛЕНИЕ

> Фасадный сервис скрывает естественную группу взаимодействующих зависимостей вместе с их поведением за одной-единственной абстракцией.

Как показано на рис. 6.2, эта переделка объединяет две зависимости в одну, но оставляет вас с четырьмя зависимостями в классе `OrderService`. Нужно поискать другие возможности объединения зависимостей в фасад.

У класса `OrderService` всего четыре зависимости, а у класса `OrderFulfillment` — две. Это неплохое начало, но `OrderService` можно сделать еще проще. Следующее, что можно подметить, касается наличия во всех требованиях уведомления о заказе других систем. Здесь напрашивается определение общей абстракции, моделирующей уведомления, возможно, чего-то вроде этого:

```
public interface INotificationService
{
    void OrderApproved(Order order);
}
```

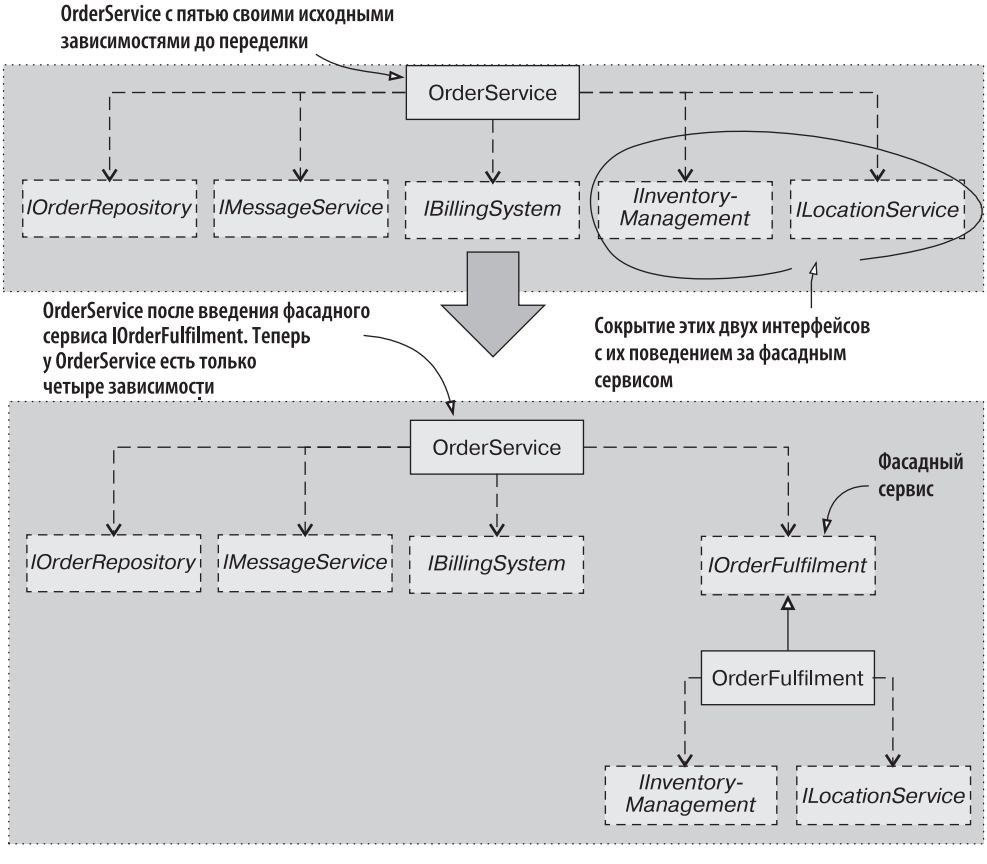


Рис. 6.2. Две зависимости OrderService, объединенные за фасадным сервисом

Каждое уведомление внешней системы можно реализовать с помощью этого интерфейса. Но поможет ли это, ведь в новый интерфейс будет заключена каждая зависимость. Количество зависимостей не уменьшится, так будет ли от этого какая-либо польза?

Да, будет. Поскольку во всех трех уведомлениях реализуется один и тот же интерфейс, их, как показано в листинге 6.4, можно заключить в один паттерн

«Компоновщик» (Composite)[1]. В этом листинге показана другая реализация `INotificationService`, являющаяся оболочкой для коллекции экземпляров `INotificationService` и вызывающая для всех них метод `OrderApproved`.

Листинг 6.4. Компоновщик, являющийся оболочкой для экземпляров INotificationService

```
public class CompositeNotificationService
    : INotificationService                          ← Реализация INotificationService
{
    IEnumerable<INotificationService> services;

    public CompositeNotificationService(
        IEnumerable<INotificationService> services)  ← Заключение в оболочку
    {                                                  последовательности
        this.services = services;                      экземпляров INotificationService
    }

    public void OrderApproved(Order order)
    {
        foreach (var service in this.services)
        {                                           ← Перенаправление входящих вызовов
            service.OrderApproved(order);             всем экземплярам, заключенным в оболочку
    }
}
```

В `CompositeNotificationService` реализуется `INotificationService`, а входящие вызовы перенаправляются его реализациям, заключенным в оболочку. Это избавляет потребителя от необходимости иметь дело с несколькими реализациями, что является особенностью его реализации. Получается, что классу `OrderService` можно позволить иметь зависимость от одного `INotificationService`, после чего, как показано далее, остаются только две зависимости (листинг 6.5).

Листинг 6.5. Переделка OrderService под использование двух зависимостей

```
public class OrderService : IOrderService
{
    private readonly IOrderRepository orderRepository;
    private readonly INotificationService notificationService;

    public OrderService(
        IOrderRepository orderRepository,
        INotificationService notificationService)
    {
        this.orderRepository = orderRepository;
        this.notificationService = notificationService;
    }

    public void ApproveOrder(Order order)
    {
        this.UpdateOrder(order);
```

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 196.

```
            this.notificationService.OrderApproved(order);
        }

        private void UpdateOrder(Order order)
        {
            order.Approve();
            this.orderRepository.Save(order);
        }
}
```

В этом есть смысл и с концептуальной точки зрения. По большому счету вам не нужно вникать в подробности того, как `OrderService` уведомляет другие системы, вы просто знаете, что это делается. Тем самым число зависимостей `OrderService` сокращается до двух, что является более разумным вариантом.

С точки зрения потребителя, функционально класс `OrderService` не изменился, то есть переделка удалась. С другой стороны, по заложенному в него замыслу класс `OrderService` изменился. Теперь его обязанностью является получение заказа, его сохранение и уведомление других систем. Подробности того, какие именно системы уведомляются и как это реализовано, отодвинуты на более детализированный уровень. Получившиеся в конечном итоге зависимости класса `OrderService` показаны на рис. 6.3.

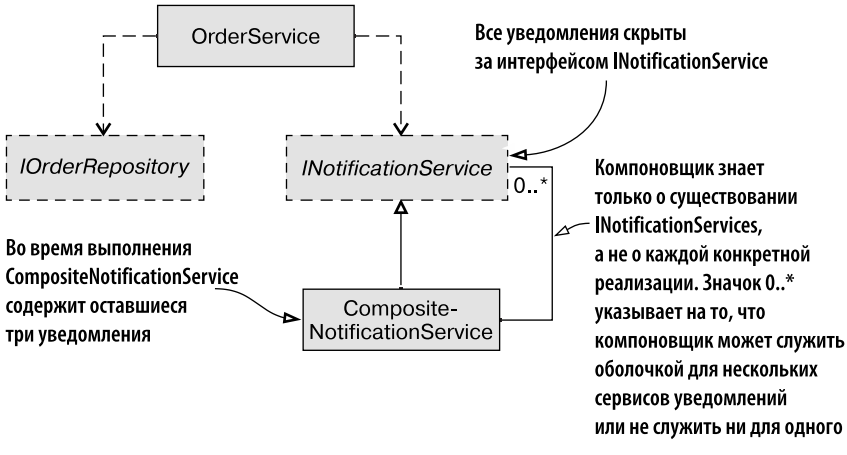


Рис. 6.3. Итоговое состояние OrderService с переделанными зависимостями

Теперь, воспользовавшись `CompositeNotificationService`, можно создать `OrderService` с его зависимостями (листинг 6.6).

Листинг 6.6. Корень композиции, переделанный под использование фасадных сервисов

```
var repository = new SqlOrderRepository(connectionString);

var notificationService = new CompositeNotificationService(
    new INotificationService[]
    {
        new OrderApprovedReceiptSender(messageService),
```

```
        new AccountingNotifier(billingSystem),
        new OrderFulfillment(locationService, inventoryManagement)
    });

var orderServive = new OrderService(repository, notificationService);
```

Несмотря на постоянное использование внедрения через конструктор, ни один конструктор класса не требует более двух параметров. `CompositeNotificationService` получает в качестве единственного аргумента `IEnumerable<INotificationService>`.

СОВЕТ

> Переделка под использование фасадных сервисов — это больше, чем просто эффектный трюк, позволяющий избавиться от внушительного количества зависимостей. Принципиальным моментом здесь является обнаружение естественных групп взаимодействия.

Полезным побочным эффектом является обнаружение этих естественных групп, вскрывающее ранее не обнаруженные доменные отношения и концепции. В ходе работы вы превращаете неявные концепции в явные. Каждая совокупность зависимостей становится сервисом, фиксирующим их взаимодействие на более высоком уровне, и единственной ответственностью потребителя становится организация этих высокоуровневых сервисов. При наличии сложного приложения, в котором потребитель получает слишком много зависимостей, переделку в фасадный сервис можно повторить. Вполне разумным шагом может стать создание фасадного сервиса, объединяющего фасадные сервисы.

Переделка под использование фасадных сервисов является хорошим способом, позволяющим справиться со сложностью, имеющейся в системе. Но в отношении примера с `OrderService` мы могли бы сделать еще один шаг, приводящий нас к доменным событиям.

6.1.3. Переделка избыточного внедрения через конструктор в доменные события

В листинге 6.5 показано, что все уведомления являются действиями, инициируемыми при подтверждении заказа. Соответствующая часть еще раз показана в следующем коде:

```
this.notificationService.OrderApproved(order);
```

Можно сказать, что подтверждение заказа имеет большое значение с точки зрения бизнеса. Подобные события называются доменными, и было бы ценно моделировать их в приложении предельно ясно.

ОПРЕДЕЛЕНИЕ

> Суть доменного события заключается в его использовании для захвата действий, способных вызвать изменение состояния разрабатываемого приложения (https://martinfowler.com/eaaDev/DomainEvent.html).

Хотя внедрение сервиса `INotificationService` является существенным улучшением `OrderService`, с его помощью решается только проблема на уровне класса `OrderService` и его непосредственных зависимостей. При использовании той же технологии переделки в отношении других классов системы нетрудно представить, как `INotificationService` превратится в нечто показанное в листинге 6.7.

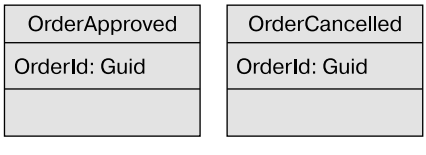
Листинг 6.7. INotificationService с возрастающим количеством методов

```
public interface INotificationService
{
    void OrderApproved(Order order);
    void OrderCancelled(Order order);
    void OrderShipped(Order order);
    void OrderDelivered(Order order);
    void CustomerCreated(Customer customer);
    void CustomerMadePreferred(Customer customer);
}
```

Каждый метод представляет собой доменное событие. Но абстракции с большим числом компонентов, как правило, нарушают принцип разделения интерфейсов, рассматриваемый в подразделе 6.2.1

В любой системе разумного размера и сложности вы бы легко получили десятки таких доменных событий, приводящих к постоянно меняющемуся интерфейсу `INotificationService`. С каждым изменением, вносимым в этот интерфейс, должны также обновляться и все его реализации. Кроме того, постоянно разрастающиеся интерфейсы также становятся причиной постоянно растущих в объеме реализаций. Но, если ввести отдельные типы для доменных событий и сделать их частью домена, как показано на рис. 6.4, появится интересная возможность еще большего обобщения.

Каждый класс дает описание конкретному изменению состояния системы. Оба класса являются простыми объектами данных



Даже притом, что эти классы кажутся одинаковыми, предоставление каждому событию своего класса позволяет событиям отображаться на компоненты, которые могут их обработать строго типизированным образом

Рис. 6.4. Доменные события, доведенные до фактических типов. Эти типы содержат только данные без поведения

Код доменных событий, фигурирующих на рис. 6.4, показан в листинге 6.8.

Листинг 6.8. Классы доменных событий OrderApproved и OrderCancelled

```
public class OrderApproved
{
    public readonly Guid OrderId;

    public OrderApproved(Guid orderId)
    {
        this.OrderId = orderId;
    }
}
```

```
public class OrderCancelled
{
    public readonly Guid OrderId;

    public OrderCancelled(Guid orderId)
    {
        this.OrderId = orderId;
    }
}
```

Хотя оба класса, и `OrderApproved`, и `OrderCancelled`, имеют одинаковую структуру и относятся к одной и той же сущности, моделирование этих событий в отдельных классах упрощает создание кода, отвечающего за конкретное событие. Когда каждое доменное событие в вашей системе получает свой собственный тип, у вас, как показано в листинге 6.9, появляется возможность переделки `INotificationService` в обобщенный интерфейс с одним-единственным методом.

Листинг 6.9. Обобщенный интерфейс IEventHandler<TEvent>, имеющий всего один метод

```
public interface IEventHandler<TEvent>
{
    void Handle(TEvent e);
}
```

Мы изменили имя INotificationService на IEventHandler, чтобы было понятнее, что у этого интерфейса более широкая сфера применения, нежели простое уведомление других систем

Обобщения

Обобщения вводят понятие параметров типа, позволяющих разрабатывать интерфейсы, классы и методы, откладывающие спецификацию своих типов до их объявления и создания клиентским кодом. Используя обобщения, такой интерфейс, класс или метод становится шаблоном.

В среду .NET Framework включено множество типов и методов, являющихся обобщениями, и многие из них, вероятнее всего, уже использовались вами. В книге уже попадались некоторые соответствующие примеры:

- интерфейс `IEnumerable<T>` в нескольких листингах (например, в листинге 2.3) в главах 2 и 3;
- класс `DbSet<T>` в листингах 2.2 и 3.11;
- метод `AddSingleton<T>()` в листинге 4.3;
- класс `Dictionary<TKey, TValue>` в листинге 5.7.

Интерфейс `IEventHandler<TEvent>` в листинге 6.9 ничем не отличается от этих обобщенных типов и методов среды. Если понятие обобщений вам незнакомо, советуем обратиться к соответствующей теме в руководстве по программированию на C# (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/).

ПРИМЕЧАНИЕ

Время от времени обобщенные типы и методы еще будут встречаться в этой книге.

В случае с `IEventHandler<TEvent>` класс, являющийся производным интерфейса, должен содержать определение типа `TEvent` — например, `OrderCancelled` — в объявлении класса. Затем этот тип будет использован в качестве типа параметра для имеющегося в классе метода `Handle`. Это позволит одному интерфейсу унифицировать сразу несколько классов, несмотря на различия в их типах. Кроме того, это позволит каждой из соответствующих реализаций быть строго типизированной, работая исключительно с тем типом, что был указан как `TEvent`.

Теперь на основе данного интерфейса можно создать классы, отвечающие за доменные события, такие как рассмотренный ранее класс `OrderFulfillment`. Основываясь на новом интерфейсе `IEventHandler<TEvent>`, в исходный класс `OrderFulfillment`, показанный в листинге 6.3, вносятся изменения, демонстрируемые в листинге 6.10.

Листинг 6.10. Класс OrderFulfillment, реализующий IEventHandler<TEvent>

```
public class OrderFulfillment
    : IEventHandler<OrderApproved>
{
    private readonly ILocationService locationService;
    private readonly IInventoryManagement inventoryManagement;

    public OrderFulfillment(
        ILocationService locationService,
        IInventoryManagement inventoryManagement)
    {
        this.locationService = locationService;
        this.inventoryManagement = inventoryManagement;
    }

    public void Handle(OrderApproved e)
    {
        this.locationService.FindWarehouses(...);
        this.inventoryManagement.NotifyWarehouses(...);
    }
}
```

Реализация IEventHandler<OrderApproved>

Заключенная в методе Handle логика идентична той, что была в листинге 6.3

Класс `OrderFulfillment` содержит реализацию `IEventHandler<OrderApproved>`, стало быть, он реагирует на события `OrderApproved`. Затем, как показано на рис. 6.5, в `OrderService` используется новый интерфейс `IEventHandler<TEvent>`.

В листинге 6.11 показан класс `OrderService`, зависящий от `IEventHandler<OrderApproved>`. По сравнению с листингом 6.5 логика `OrderService` остается практически неизменной.

Как и при использовании необобщенного `INotificationService`, вам по-прежнему нужен компоновщик, занимающийся целевой отправкой информации в список доступных обработчиков. Тем самым допускается добавление к приложению новых обработчиков, не требующее изменений `OrderService`.

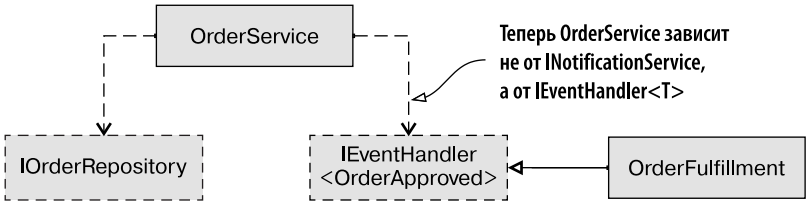


Рис. 6.5. Класс OrderService зависит не от интерфейса INotificationService, а от интерфейса IEventHandler<OrderApproved>

Листинг 6.11. Класс OrderService, зависящий от IEventHandler<OrderApproved>

```
public class OrderService : IOrderService
{
    private readonly IOrderRepository orderRepository;
    private readonly IEventHandler<OrderApproved> handler;

    public OrderService(
        IOrderRepository orderRepository,
        IEventHandler<OrderApproved> handler)
    {
        this.orderRepository = orderRepository;
        this.handler = handler;
    }

    public void ApproveOrder(Order order)
    {
        this.UpdateOrder(order);

        this.handler.Handle(
            new OrderApproved(order.Id));
    }
    ...
}
```

Теперь OrderService зависит не от INotificationService, а от IEventHandler<OrderApproved>

Подтверждение заказа означает создание доменного события OrderApproved и его отправку соответствующим обработчикам

Этот компоновщик показан в листинге 6.12. Как вы можете видеть, он похож на `CompositeNotificationService` из листинга 6.4.

Листинг 6.12. Компоновщик, который служит оболочкой для экземпляров IEventHandler<TEvent>

```
public class CompositeEventHandler<TEvent> : IEventHandler<TEvent>
{
    private readonly IEnumerable<IEventHandler<TEvent>> handlers;

    public CompositeEventHandler(
        IEnumerable<IEventHandler<TEvent>> handlers)
    {
        this.handlers = handlers;
    }

    public void Handle(TEvent e)
    {
```

Оборачивает коллекции экземпляров IEventHandler<TEvent>

```
        foreach (var handler in this.handlers)
        {
            handler.Handle(e);
        }
    }
}
```

Оборачивание коллекции экземпляров `IEventHandler<TEvent>`, как это сделано в `CompositeEventHandler<TEvent>`, позволяет добавлять к системе произвольные реализации обработчиков событий, обходясь без каких-либо изменений в потребителях `IEventHandler<TEvent>`. Используя новый класс `CompositeEventHandler<TEvent>`, можно создать `OrderService` с его зависимостями (листинг 6.13).

Листинг 6.13. Корень композиции для класса OrderService, переделанного под использование событий

```
var orderRepository = new SqlOrderRepository(connectionString);

var orderApprovedHandler = new CompositeEventHandler<OrderApproved>(
    new IEventHandler<OrderApproved>[]
    {
        new OrderApprovedReceiptSender(messageService),
        new AccountingNotifier(billingSystem),
        new OrderFulfillment(locationService, inventoryManagement)
    });

var orderService = new OrderService(orderRepository, orderApprovedHandler);
```

Аналогично корень композиции будет содержать конфигурацию для обработчиков других доменных событий. В следующем примере кода показано еще несколько обработчиков событий для `OrderCancelled` и `CustomerCreated`. Выводы из него мы оставляем на усмотрение читателей.

```
var orderCancelledHandler = new CompositeEventHandler<OrderCancelled>(
    new IEventHandler<OrderCancelled>[]
    {
        new AccountingNotifier(billingSystem),
        new RefundSender(orderRepository),
    });

var customerCreatedHandler = new CompositeEventHandler<CustomerCreated>(
    new IEventHandler<CustomerCreated>[]
    {
        new CrmNotifier(crmSystem),
        new TermsAndConditionsSender(messageService, termsRepository),
    });

var orderService = new OrderService(
    orderRepository, orderApprovedHandler, orderCancelledHandler);

var customerService = new CustomerService(
    customerRepository, customerCreatedHandler);
```

Достоинство обобщенного интерфейса вроде `IEventHandler<TEvent>` состоит в том, что добавление новых функций не повлечет за собой внесение каких-либо изменений ни в интерфейс, ни в любую из уже имеющихся реализаций. Если нужно выставить счет по подтвержденному заказу, нужно будет лишь добавить новую реализацию, исполняющую `IEventHandler<OrderApproved>`. Вносить изменения в `CompositeEventHandler<TEvent>` при создании нового доменного события не требуется.

В некотором смысле `IEventHandler<TEvent>` становится шаблоном для обычных функциональных блоков, на которые опирается приложение. Каждый стандартный блок отвечает за свое конкретное событие. Как уже было показано, может быть несколько функциональных блоков, отвечающих за одно и то же событие. Новые функциональные блоки могут подключаться, не требуя при этом внесения изменений в любую уже существующую бизнес-логику.

СОВЕТ

Присущая DI-контейнеру возможность автоматической регистрации является весьма действенным способом упрощения вашего корня композиции. Способ регистрации IEventHandler<TEvent> с использованием DI-контейнера показан в главах 13–15.

Хотя введение `IEventHandler<TEvent>` позволяет предотвратить проблему чрезмерного разрастания `INotificationService`, оно не избавляет от проблемы излишнего увеличения класса `OrderService`. Подробнее об этом мы поговорим в главе 10.

Надежная рассылка сообщений

Доведение доменных событий вашей системы до типов дает больше преимуществ, чем простое улучшение сопровождаемости приложения. Рассмотрим следующий сценарий.

В часы пик веб-службы складов могут выходить из строя. Но в этот момент уже была отправлена квитанция и уведомлена система выставления счетов. Выполнить откат обновления базы данных еще можно, но вернуть назад уведомление, увы, нельзя — электронное письмо покупателю уже было отправлено.

К сожалению, проблемы возникают не только с системой выставления счетов. Недавно произошел сбой одного из веб-серверов, на котором был запущен процесс подтверждения заказов. Письмо с подтверждением было отправлено покупателю непосредственно перед аварией, но ни система выставления счетов, ни склады уведомлены не были. Покупатель заказ так и не получил. Что нужно сделать для смягчения последствий подобных проблем?

Хотя есть несколько способов справиться с таким сценарием, на помощь могут прийти и доменные события: их можно выстроить в последовательность и поместить в надежную очередь сообщений, например в MSMQ, Azure Queue или таблицу базы

данных. Подобные действия позволят классу `OrderService` выполнять только следующие операции:

- приступить к транзакции;
- обновить заказ в базе данных в рамках транзакции;
- опубликовать событие `OrderAccepted` в надежной очереди в рамках транзакции[1];
- зафиксировать транзакцию.

Только после того, как событие `OrderAccepted` было зафиксировано в очереди, оно становится доступным для дальнейшей обработки. Тогда его можно будет передать каждому из доступных обработчиков данного конкретного события. Каждый обработчик может запускаться в своей собственной отдельной транзакции[2]. Если один из обработчиков даст сбой, можно будет повторить запуск этого отдельно взятого обработчика, не оказывая никакого влияния на другие обработчики. Можно даже запустить на параллельное выполнение сразу несколько обработчиков.

Обработка сообщений с использованием надежной очереди является формой надежной рассылки сообщений, предоставляющей определенные гарантии их успешной передачи. Это весьма действенное решение рассмотренного сценария, при котором допускается сбой серверов и недоступность внешних систем. Однако изучение способов реализации подобных надежных паттернов рассылки сообщений выходит за рамки данной книги[3].

Обнаружилось, что использование доменных событий является весьма эффективной моделью. Она позволяет определять код на более высоком концептуальном уровне и при этом допускает создание более надежных программных средств, особенно если приходится обмениваться данными с внешними системами, не являющимися частью транзакции вашей базы данных. Но независимо от того, какой подход к переделке будет выбран, будь то паттерны типа «Декоратор», фасадные сервисы, доменные события или что-нибудь другое, весьма важным моментом здесь является то, что избыточное внедрение через конструктор служит явным признаком проблемного кода. Его не стоит игнорировать, лучше предпринять соответствующие меры по переделке кода.

Избыточное внедрение через конструктор зачастую сопряжено с появлением проблемного кода, а в следующем разделе будет рассмотрена менее заметная проблема, которая на первый взгляд может быть похожа на вполне приемлемое решение множества хронических проблем. Но так ли это?

---

[1] Отличным способом предотвращения распределенных транзакций является паттерн «Исходящие сообщения» (Outbox). См.: http://gistlabs.com/2014/05/the-outbox/.

[2] Это приводит к согласованности в конечном счете. Подробности см. по адресам https://en.wikipedia.org/wiki/Eventual_consistency и https://ru.wikipedia.org/wiki/Согласованность_в_конечном_счёте.

[3] Отличная книга с основами теории сообщений, согласованности в конечном счете и публикации-подписки: *Boike D.* Learning NServiceBus, 2nd ed. — Packt Publishing, 2015.

6.2. Злоупотребление абстрактными фабриками

В самом начале освоения технологии DI одной из первых трудностей, с которой, скорее всего, придется столкнуться, станет абстракция, зависящая от динамически определяемых значений. Например, интерактивный картографический сайт может предлагать расчет маршрута между двумя географическими пунктами, предоставляя вам выбор желаемого построения маршрута. Вам нужен самый короткий маршрут? Или самый быстрый, исходя из известной загруженности трассы? Или же самый живописный маршрут?

В таком случае первой реакцией многих разработчиков станет использование абстрактной фабрики. Хотя, когда дело касается технологии DI, абстрактная фабрика по праву занимает свое место в разработке программных продуктов. Когда фабрики используются в компонентах приложений в качестве зависимостей, нередко происходит их чрезмерное применение. Зачастую имеются более удачные варианты.

В этом разделе будут рассмотрены два случая более удачных альтернатив абстрактным фабрикам. В первом случае будет дан ответ на вопрос, почему абстрактные фабрики не следует применять для создания непродолжительно существующих зависимостей с отслеживаемым состоянием. После этого будет дан ответ на вопрос, почему во многих случаях лучше будет обойтись без применения абстрактных фабрик для выбора зависимостей на основе динамически изменяемых данных.

6.2.1. Злоупотребление абстрактными фабриками для преодоления проблем с временем существования объектов

Если говорить о злоупотреблении абстрактными фабриками, то общим признаком проблемного кода являются методы фабрики, не имеющие параметров, но, как показано в листинге 6.14, имеющие зависимость в качестве возвращаемого типа.

Листинг 6.14. Абстрактная фабрика с не имеющим параметров методом Create

```
public interface IProductRepositoryFactory
{
    IProductRepository Create();
}
```

Не имеющий параметров фабричный метод, возвращающий новый экземпляр нестабильной зависимости

Абстрактные фабрики с не имеющим параметров методом `Create` часто используются, чтобы позволить потребителям контролировать время существования их зависимостей. В листинге 6.15 класс `HomeController` контролирует время существования `IProductRepository`, запрашивая этот интерфейс у фабрики и избавляясь от него по окончании использования.

Последовательность обмена данными между `HomeController` и его зависимостями показана на рис. 6.6.

Листинг 6.15. Класс HomeController, явно управляющий временем существования своей зависимости

```
public class HomeController : Controller
{
    private readonly IProductRepositoryFactory factory;

    public HomeController(
        IProductRepositoryFactory factory)
    {
        this.factory = factory;
    }

    public ViewResult Index()
    {
        using (IProductRepository repository =
            this.factory.Create())
        {
            var products =
                repository.GetFeaturedProducts();

            return this.View(products);
        }
    }
}
```

Внедрение абстрактной фабрики в потребитель

Абстрактная фабрика создает экземпляр хранилища, чье время существования должно четко контролироваться

Использование хранилища

Поскольку в IProductRepository реализуется IDisposable, созданный экземпляр должен ликвидироваться после завершения работы потребителем. Это превращает IProductRepository в абстракцию с протечкой, в чем мы вскоре убедимся

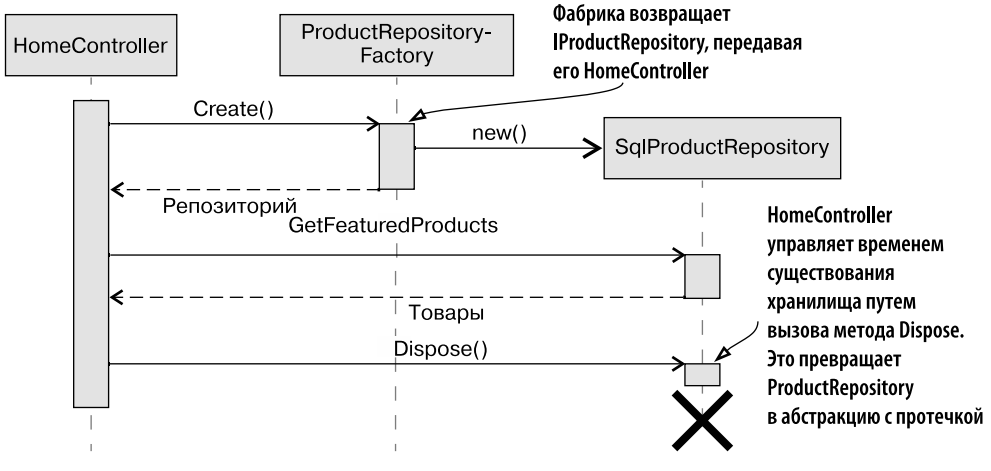


Рис. 6.6. Класс-потребитель HomeController управляет временем существования своей зависимости IProductRepository. Это делается путем запроса экземпляра хранилища из зависимости IProductRepositoryFactory и вызова метода Dispose в отношении экземпляра IProductRepository, когда его использование завершится

Вызов метода `Dispose` для хранилища требуется, когда используемая реализация удерживает такие ресурсы, как подключения к базе данных, которые должны быть закрыты определенным образом. Хотя реализация может потребовать той или иной очистки, это не означает, что ответственность за обеспечение надлежащей

очистки должна возлагаться на потребителя. Это подводит нас к понятию абстракции с протечкой.

Проблемный код, характеризующийся наличием абстракции с протечкой

Как раз потому, что разработка, основанная на тестировании (Test-Driven Development, TDD), обеспечивает тестируемость, безопаснее всего сначала определить интерфейсы, а затем последовательно заняться программированием на их основе. И все же бывают случаи, когда уже имеется конкретный тип, а теперь желательно выделить интерфейс. Совершая соответствующие действия, нужно позаботиться, чтобы детали базовой реализации не протекли сквозь швы. Одна из возможностей возникновения такой ситуации связана с тем, что интерфейс извлекается только из заданного конкретного типа, но какие-то параметры или возвращаемые типы все еще являются конкретными типами, определенными в библиотеке, от которой вы хотите абстрагироваться. Пример предлагается следующим интерфейсом:

```
public interface IRequestContext
{
    HttpContext Context { get; }
}
```

Интерфейс приложения пытается абстрагироваться от среды выполнения ASP.NET

Интерфейс все еще предоставляет HttpContext, являющийся частью ASP.NET. В результате получается абстракция с протечкой

Если требуется выделить интерфейс, это нужно делать рекурсивно, гарантируя то, что все типы, предоставляемые корневым интерфейсом, сами являются интерфейсами. Мы называем это глубоким выделением (Deep Extraction), а результат — глубокими интерфейсами (Deep Interfaces).

Это не означает, что интерфейсы не могут выставлять наружу какие-либо конкретные классы. Обычно достаточно выставить объекты данных, не имеющие поведения, такие как граничные объекты (Parameter Objects), модели представлений и объекты переноса данных (Data Transfer Objects, DTO). Они определены в той же библиотеке, что и интерфейс, а не в той библиотеке, от которой нужно абстрагироваться. Такие объекты данных являются частью абстракции.

Глубокое выделение требует осмотрительного подхода: оно не всегда приводит к наилучшему решению. Возьмем предыдущий пример. Рассмотрим следующую реализацию интерфейса `IHttpContext`, полученную за счет глубокого выделения и имеющую весьма подозрительный вид:

Определенный в приложении интерфейс для абстрагирования от имеющегося в ASP.NET типа HttpContext

```
public interface IHttpContext
{
    IHttpRequest Request { get; }
    IHttpResponse Response { get; }
    IHttpSession Session { get; }
    IPrincipal User { get; }
}
```

Компоненты интерфейса предоставляют другие определенные в приложении интерфейсы, абстрагирующие от компонентов, предоставляющих тип HttpContex. И это может углубляться на многие уровни

Хотя интерфейсы можно использовать на всю глубину, протечка модели HTTP все же еще вполне очевидна. Иными словами, `IHttpContext` по-прежнему является абстракцией с протечкой, как и все его субинтерфейсы.

Как тогда нужно вместо этого смоделировать интерфейс `IRequestContext`? Чтобы это понять, нужно посмотреть, чего хотят добиться его потребители. Например, если потребитель нуждается в определении роли пользователя, отправившего текущий веб-запрос, можно остановиться на интерфейсе `IUserContext`, рассмотренном в главе 3:

```
public interface IUserContext
{
    bool IsInRole(Role role);
}
```

Этот интерфейс `IUserContext` не показывает потребителю, что он работает как часть веб-приложения ASP.NET. Фактически эта абстракция позволяет запускать одного и того же потребителя как часть сервиса Windows или приложения для настольного компьютера. Для этого, вероятно, потребуется создание другой реализации `IUserContext`, но его потребители этого не замечают.

Следует всегда принимать во внимание, имеет ли рассматриваемая абстракция смысл для реализаций, отличных от той, которая подразумевается. Если нет, нужно пересмотреть архитектуру. Это возвращает нас к нашим фабричным методам, не имеющим параметров.

Фабричные методы, не имеющие параметров, являются абстракцией с протечкой

Насколько бы ни был полезен паттерн абстрактной фабрики, при его использовании следует проявлять осмотрительность. Зависимостям, созданным абстрактной фабрикой, концептуально должно требоваться значение среды выполнения, и должен быть вполне определенный смысл в переводе значения среды выполнения в абстракцию. Если возникает желание применить абстрактную фабрику, поскольку у вас на примете есть конкретная реализация, дело может закончиться появлением абстракции с протечкой.

Потребители, зависящие от `IProductRepository`, например `HomeController` из листинга 6.15, не должны беспокоиться о том, какой экземпляр они получают. Во время выполнения программы может понадобиться создание нескольких экземпляров, но для потребителя существует только один экземпляр.

ВНИМАНИЕ

> Концептуально существует только один экземпляр сервисной абстракции. В течение жизни потребителя его не должна интересовать возможность существования нескольких экземпляров зависимости. Иначе иной вариант мог бы вызвать для потребителей ненужные осложнения, а это означало бы, что абстракция не создана для их пользы.

Определив абстракцию `IProductRepositoryFactory` с не имеющим параметров методом `Create`, вы даете потребителю информацию о наличии и других экземпляров данного сервиса и что он должен иметь с этим дело. Поскольку другие реализации `IProductRepository` могут вообще не требовать нескольких экземпляров или детерминированной ликвидации (disposal), вы через не имеющий параметров метод `Create` допускаете протечку подробностей реализации через абстрактную фабрику. Иными словами, вами создана абстракция с протечкой.

Абстракции, реализующие IDisposable, являются абстракциями с протечкой

Код приложения не должен отвечать за управление временем жизни объектов. Внесение этой ответственности в код приложения означает усложнение того класса, куда она внесена, затруднение его тестирования и сопровождения. Часто приходится наблюдать, как логика управления временем существования дублируется по всему приложению, вместо того чтобы сконцентрироваться в корне композиции, к чему вы и стремитесь.

Применение технологии DI не может служить оправданием для создания приложений с утечкой памяти, поэтому нужно иметь возможность закрывать подключения и доступ к другим ресурсам явным образом и как можно раньше. С другой стороны, любая зависимость может представлять или не представлять внепроцессный ресурс, поэтому, если моделировать абстракцию с включением в нее метода `Dispose` или `Close`, получится абстракция с протечкой.

Абстракция вообще не должна быть ликвидируемой, поскольку способа предвидеть все возможные ее реализации просто не существует. Практически любой абстракции может в какой-то момент потребоваться ликвидируемая реализация, притом что другие реализации той же самой абстракции будут продолжать полагаться исключительно на управляемый код.

Это не означает, что в классах не должен реализовываться интерфейс `IDisposable`. Это говорит о том, что `IDisposable` не должен реализовываться в абстракциях. Поскольку клиент знает только об абстракции, он не может отвечать за управление временем существования конкретного экземпляра. Эту ответственность мы вернули обратно в корень композиции. Управление временем существования будет рассмотрено в главе 8.

Далее речь пойдет о том, как предотвратить применение проблемного кода, имеющего абстракции с протечкой.

Переделка в сторону лучшего решения

Потребление кода не должно быть связано с возможностью наличия более одного экземпляра `IProductRepository`. Поэтому нужно полностью избавиться от `IProductRepositoryFactory` и вместо этого позволить потребителям зависеть исключительно от `IProductRepository`, который у них должен быть внедрен через конструктор. Этот совет нашел свое отражение в листинге 6.16.

Как показано на рис. 6.7, этот код приводит к упрощенной последовательности взаимодействия `HomeController` и его единственной зависимости `IProductRepository`.

Листинг 6.16. Класс HomeController без управления временем существования его зависимости

```
public class HomeController : Controller
{
    private readonly IProductRepository repository;

    public HomeController(
        IProductRepository repository)
    {
        this.repository = repository;
    }

    public ViewResult Index()
    {
        var products =
            this.repository.GetFeaturedProducts();

        return this.View(products);
    }
}
```

Вместо внедрения абстрактной фабрики сам IProductRepository вводится непосредственно в потребляющий HomeController

Вместо управления временем существования IProductRepository путем его запроса из абстрактной фабрики и его ликвидации HomeController просто его использует. IDisposable в IProductRepository больше не реализуется

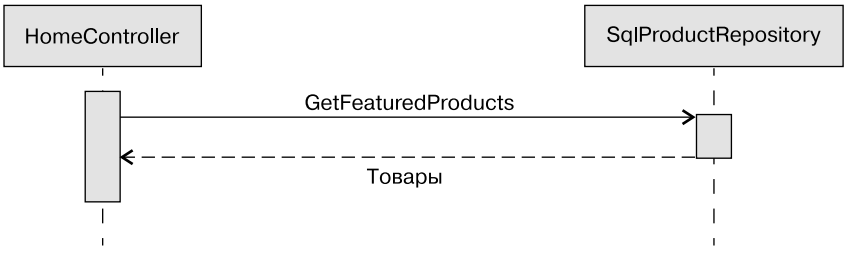


Рис. 6.7. По сравнению с рис. 6.6 удаление ответственности по управлению временем существования IProductRepository наряду с удалением зависимости IProductRepositoryFactory существенно упрощает взаимодействие с зависимостями HomeController

Хотя удаление управления временем существования упрощает `HomeController`, управлять временем жизни хранилища придется где-то в другом месте приложения. Зачастую для решения этой проблемы используется паттерн «Заместитель» (Proxy), пример которого дан в листинге 6.17[1].

Паттерн проектирования «Заместитель»

Паттерн проектирования «Заместитель» предоставляет заменитель или заместитель для другого объекта для управления доступом к нему[1]. Он позволяет отложить полные затраты на его создание и инициализацию до того момента, когда он понадобится. В заместителе реализуется такой же интерфейс, что и в замещаемом им объекте. Это заставляет потребители поверить, что они общаются с реальной реализацией.

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 246.

Листинг 6.17. Задержка создания SqlProductRepository с помощью «Заместитель»

```
public class SqlProductRepositoryProxy : IProductRepository
{
    private readonly string connectionString;

    public SqlProductRepositoryProxy(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public IEnumerable<Product> GetFeaturedProducts()
    {
        using (var repository = this.Create())
        {
            return repository.GetFeaturedProducts();
        }
    }

    private SqlProductRepository Create()
    {
        return new SqlProductRepository(
            this.connectionString);
    }
}
```

Заместительсоздает и вызывает внутри себя SqlProductRepository только тогда, когда вызывается его метод GetFeaturedProducts. Чтобы предотвратить возникновение антипаттерна «Диктатор», такой заместитель обычно должен быть частью корня композиции

Заместитель перенаправляет вызов настоящей реализации IProductRepository

Реализации SqlProductRepository по-прежнему реализует и Idisposable. Его временем существования на этот раз может управлять заместитель

Обратите внимание на то, что внутри `SqlProductRepositoryProxy` содержится поведение, похожее на имеющееся у фабрики со своим собственным приватным методом `Create`. Но это поведение заключено в заместитель и не допускает протечки по сравнению с абстрактной фабрикой `IProductRepositoryFactory`, предоставляющей из своего определения `IProductRepository`.

ПРИМЕЧАНИЕ

> Наличие поведения, похожего на имеющееся у фабрики (такого как метод Create из листинга 6.17), как правило, неизбежно. Но к фабричным абстракциям в масштабах всего приложения нужно относиться с недоверием.

`SqlProductRepositoryProxy` сильно связан с `SqlProductRepository`. Если `SqlProductRepositoryProxy` был определен в вашем доменном уровне, это станет реализацией антипаттерна «Диктатор» (см. раздел 5.1). Вместо этого нужно определить заместитель либо на вашем уровне доступа к данным, либо, скорее всего, в корне композиции.

Поскольку метод `Create` составляет часть графа объектов, корень композиции является вполне подходящим местом для класса-заместителя. В листинге 6.18 показана структура корня композиции с использованием `SqlProductRepositoryProxy`.

Листинг 6.18. Граф объектов с новым SqlProductRepositoryProxy

```
new HomeController(
    new SqlProductRepositoryProxy(
        connectionString));
```

Создание экземпляра класса-заместителя вместо SqlProductRepository

В том случае, если у абстракции много компонентов, создание реализаций заместителя становится слишком проблематичным. Но абстракции с большим числом компонентов часто нарушают принцип изоляции интерфейса (Interface Segregation Principle). Создание более конкретизированных абстракций решает множество проблем, таких как сложность создания заместителей, декораторов и тестовых дубликатов. Подробнее данный вопрос будет рассмотрен в разделе 6.3, а также к этой теме мы еще вернемся в главе 10.

Принцип изоляции интерфейса

Принцип изоляции интерфейса (ISP) гласит, что «ни один клиент не должен зависеть от методов, которые он не использует».

Это означает, что потребитель интерфейса должен задействовать все методы потребляемой зависимости. Если в абстракции есть методы, не используемые потребителем, интерфейс слишком раздут и в соответствии с ISP, он должен быть разбит на части. Тогда система останется несвязанной и более простой для переделки, изменения и повторного развертывания. Поэтому интерфейсы должны быть разработаны конкретизированно. Не следует возлагать на один интерфейс слишком много обязанностей, поскольку он станет громоздким для реализации.

ISP можно рассматривать как концептуальную основу принципа единственной ответственности (SRP). ISP утверждает, что интерфейсы должны моделировать только одно понятие, а SRP говорит о том, что у реализаций должна быть только одна ответственность.

На первый взгляд, ISP может показаться отдаленно связанным с DI. Это важно, поскольку интерфейс с раздутым моделированием тянет в направлении конкретной реализации. Зачастую это приводит к проблемному коду абстракции с протечкой и усложняет замену зависимостей. Причина в том, что некоторые компоненты интерфейса могут не иметь никакого смысла в контексте, отличающемся от того, на основе которого была выбрана исходная архитектура[1]. Но в главе 10 выяснится, что соблюдение ISP имеет решающее значение, когда дело доходит до эффективного применения технологии DI и аспектно-ориентированного программирования.

Но это еще не означает, что между реализацией и абстракцией должно всегда быть отношение «один к одному». Иногда нужно сделать число интерфейсов меньше числа их реализаций, то есть реализация может обеспечить работу большего числа интерфейсов[2].

---

1 *Seemann M.* Interfaces are not abstractions, 2010; https://mng.bz/8yvz.

2 Подробности см. в книге: *Мартин Р. К., Ньюкирк Дж. В., Косс Р. С.* Быстрая разработка программ. Принципы, примеры, практика. — М.: Вильямс, 2004. — Глава 12.

В следующем разделе будет рассмотрено злоупотребление абстрактными фабриками при выборе возвращаемой зависимости на основе предоставляемых данных среды выполнения.

6.2.2. Злоупотребление абстрактными фабриками при выборе возвращаемой зависимости на основе данных среды выполнения

В предыдущем разделе вы узнали, что абстрактные фабрики должны, как правило, принимать в качестве входных данных значения среды выполнения. Без них подробности реализации станут протекать к потребителю. Это не означает, что абстрактная фабрика, принимающая данные среды выполнения, является надлежащим решением для любой ситуации. Чаще всего это не так.

В этом разделе будут рассмотрены абстрактные фабрики, принимающие данные среды выполнения конкретно для того, чтобы решить, какую зависимость возвращать. Будет рассматриваться пример интерактивного картографического сайта, предлагающего расчет маршрута между двумя географическими пунктами, представленный в начале раздела 6.2.

Для вычисления маршрута приложению нужен алгоритм его построения, но ему все равно, какой именно. Каждый вариант предоставляет другой алгоритм, а приложение может управлять каждым алгоритмом построения маршрута как абстракцией, чтобы они все считались равнозначными. Вы должны сообщить приложению, какой из алгоритмов использовать, но вы не будете этого знать, пока приложение не будет запущено на выполнение, поскольку это сообщение основано на выборе пользователя.

В веб-приложении от браузера к серверу можно передавать только элементарные типы. Когда пользователь выбирает алгоритм построения маршрута из поля со списком, этот выбор нужно представить в виде числа или строки[1]. Перечисление `enum` относится к числам, поэтому на сервере можно представить выбор, воспользовавшись `RouteType`:

```
public enum RouteType { Shortest, Fastest, Scenic }
```

Вам нужен экземпляр `IRouteAlgorithm`, способный вычислить для вас маршрут:

```
public interface IRouteAlgorithm
{
    RouteResult CalculateRoute(RouteSpecification specification);
}
```

А вот теперь возникла проблема. Переменная `RouteType` — это данные среды выполнения, основанные на выборе пользователя. На сервер она отправляется вместе с запросом (листинг 6.19).

---

[1] Если проявить педантизм, мы можем отправлять только строки, но большинство веб-сред поддерживает преобразование элементарных типов.

Листинг 6.19. RouteController со своим методом GetRoute

```
public class RouteController : Controller
{
    public ViewResult GetRoute(
        RouteSpecification spec, RouteType routeType)
    {
        IRouteAlgorithm algorithm = ...

        var route = algorithm.CalculateRoute(spec);

        var vm = new RouteViewModel
        {
            ...
        };

        return this.View(vm);
    }
}
```

Получение IRouteAlgorithm для соответствующего RouteType. Но как?

Вызов выбранного IRouteAlgorithm

Отображение возвращенных данных маршрута модели RouteViewModel, которая может потребляться отображением

Заключение модели представления в MVC-объект ViewResult с помощью имеющегося в среде MVC вспомогательного метода View

Теперь вопрос: как получить соответствующий алгоритм? Если бы вы не читали эту главу, то первой реакцией стало бы, наверное, применение абстрактной фабрики:

```
public interface IRouteAlgorithmFactory
{
    IRouteAlgorithm CreateAlgorithm(RouteType routeType);
}
```

Это позволит реализовать метод `GetRoute` для `RouteController` путем внедрения `IRouteAlgorithmFactory` и использования этого интерфейса для перевода значения среды выполнения в нужную вам зависимость `IRouteAlgorithm`. Взаимодействие показано в листинге 6.20.

Класс `RouteController` отвечает за обработку веб-запросов. Метод `GetRoute` получает пользовательское определение исходного и конечного пунктов, а также выбранный тип маршрута `RouteType`. С помощью абстрактной фабрики значение среды выполнения `RouteType` отображается на экземпляр `IRouteAlgorithm`, то есть экземпляр `IRouteAlgorithmFactory` запрашивается с использованием внедрения через конструктор. Рассматриваемая последовательность отношений `RouteController` и его зависимостей показана на рис. 6.8.

В наиболее простой реализации `IRouteAlgorithmFactory` будет использоваться инструкция `switch` и на основе введенных данных будут возвращаться три различные реализации `IRouteAlgorithm`. Но мы предлагаем ее создание в качестве упражнения для читателя.

В этот самый момент вы можете задаться вопросом: «А в чем, собственно, подвох? Почему этот код называется проблемным?» Чтобы понять суть проблемы, нужно вернуться к принципу инверсии зависимости.

Листинг 6.20. Использование IRouteAlgorithmFactory в RouteController

```
public class RouteController : Controller
{
    private readonly IRouteAlgorithmFactory factory;

    public RouteController(IRouteAlgorithmFactory factory)
    {
        this.factory = factory;
    }

    public ViewResult GetRoute(
        RouteSpecification spec, RouteType routeType)
    {
        IRouteAlgorithm algorithm =
            this.factory.CreateAlgorithm(routeType);

        var route = algorithm.CalculateRoute(spec);

        var vm = new RouteViewModel
        {
            ...
        };

        return this.View(vm);
    }
}
```

Использование фабрики для отображения значения среды выполнения в параметре routeType на IRouteAlgorithm

Получив этот алгоритм, его можно использовать для вычисления маршрута и возвращения результата

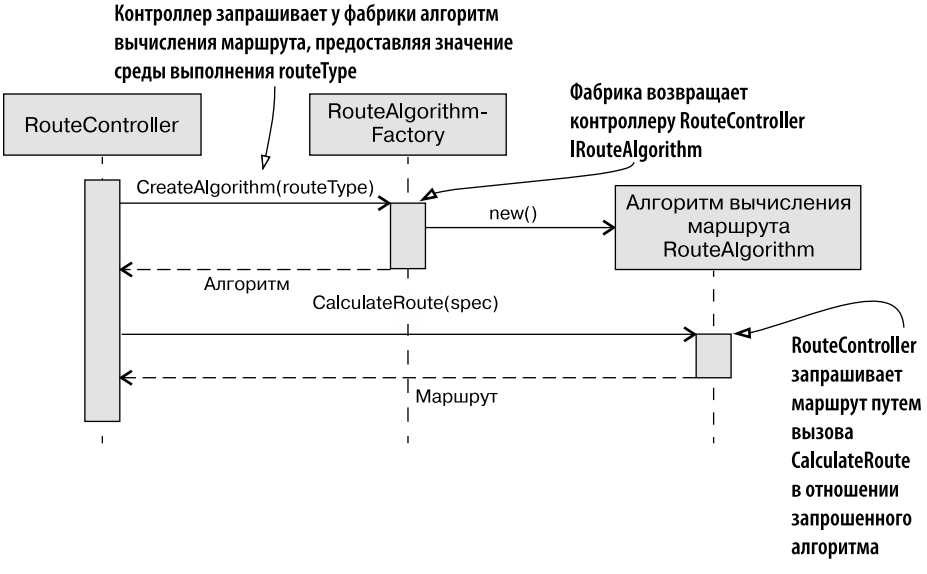


Рис. 6.8. RouteController предоставляет значение среды выполнения routeType фабрике IRouteAlgorithmFactory. Фабрика возвращает реализацию IRouteAlgorithm, а RouteController запрашивает маршрут, вызывая CalculateRoute. Взаимоотношения аналогичны показанным на рис. 6.6

Анализ проблемного кода

В подразделе 3.1.2 говорилось о принципе инверсии зависимости. Рассматривалось его утверждение, что абстракциями должен владеть использующий их уровень. Объяснялось, что форма абстракции и ее определение должны задаваться ее потребителем так, чтобы максимально соответствовать его потребностям. Когда речь снова зашла о нашем `RouteController` и возник вопрос соответствия рассматриваемой конструкции наилучшему варианту, мы согласились, что данная конструкция не подходит для `RouteController`.

Один из способов рассмотрения данного вопроса заключается в оценке числа зависимостей, имеющихся у `RouteController`, свидетельствующей о сложности класса. В разделе 6.1 уже было показано, что большое количество зависимостей является признаком проблемного кода, и обычно выходом из данной ситуации служит переделка с применением фасадных сервисов.

При введении абстрактной фабрики неизменно повышается число зависимостей, имеющихся у потребителя. Если взглянуть лишь на конструктор `RouteController`, можно поверить, что у контроллера только одна зависимость. Но `IRouteAlgorithm` также является зависимостью `RouteController`, даже если он не внедрен через конструктор этого контроллера.

На первый взгляд такое усложнение можно и не заметить, но оно сразу даст о себе знать, как только начнется модульное тестирование `RouteController`. Оно не только заставит протестировать взаимодействие, имеющееся у `RouteController` с `IRouteAlgorithm`, но и вынудит протестировать взаимодействие с `IRouteAlgorithmFactory`.

Переделка с целью улучшения

Число зависимостей можно уменьшить, объединив `IRouteAlgorithmFactory` и `IRouteAlgorithm`, что весьма похоже на переделку под использование фасадных сервисов, показанную в разделе 6.1. В идеале хотелось бы воспользоваться паттерном «Заместитель», применив способ, похожий на показанный в разделе 6.2.1. Но паттерн «Заместитель» применим только в случае предоставления абстракции со всеми данными, необходимыми для выбора подходящей зависимости. К сожалению, предварительное условие в отношении `IRouteAlgorithm` не выполняется, поскольку данная абстракция предоставляется только с `RouteSpecification`, но не с `RouteType`.

Прежде чем отказаться от паттерна-заместителя, важно проверить, имеет ли смысл на концептуальном уровне передать `RouteType` в `IRouteAlgorithm`. Наличие такого смысла означает, что реализация `CalculateRoute` содержит всю информацию, требуемую для выбора как подходящего алгоритма, так и значений времени выполнения, необходимых для вычисления маршрута. Но в таком случае передача `RouteType` в `IRouteAlgorithm` концептуально выглядит довольно странно. Реализации алгоритма никогда не понадобится использование `RouteType`. Вместо этого,

чтобы упростить конструкцию контроллера, определяется адаптер, выполняющий внутреннее направление к подходящему алгоритму вычисления:

```
public interface IRouteCalculator
{
    RouteResult Calculate(RouteSpecification spec, RouteType routeType);
}
```

В листинге 6.21 показано, как `RouteController` упрощается, когда зависит от `IRouteCalculator`, а не от `IRouteAlgorithmFactory`.

Листинг 6.21. Использование IRouteCalculator в RouteController

```
public class RouteController : Controller
{
    private readonly IRouteCalculator calculator;

    public RouteController(IRouteCalculator calculator)
    {
        this.calculator = calculator;
    }

    public ViewResult GetRoute(RouteSpecification spec, RouteType routeType)
    {
        var route = this.calculator.Calculate(spec, routeType);

        var vm = new RouteViewModel { ... };

        return this.View(vm);
    }
}
```

Использование IRouteCalculator сокращает число зависимостей. Теперь осталась всего одна зависимость

На рис. 6.9 показывается упрощенное взаимодействие `RouteController` и его единственной зависимости. Как следует из показанного на рис. 6.7, взаимодействие сводится только к вызову метода.

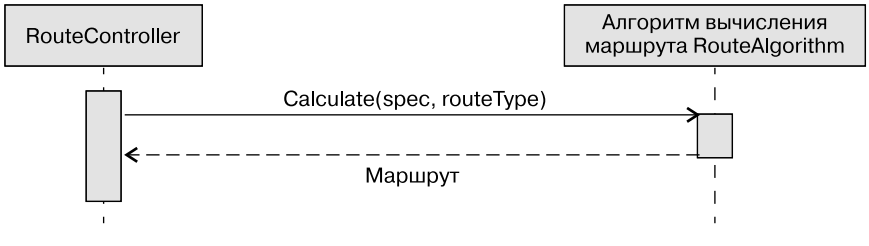


Рис. 6.9. По сравнению со схемой на рис. 6.8, упрощение взаимодействия RouteController с его (теперь уже единственной) зависимостью достигнуто за счет сокрытия IRouteAlgorithmFactory и IRouteAlgorithm за единственной абстракцией IRouteCalculator

Реализация `IRouteCalculator` может быть выполнена множеством способов. Один из них заключается во включении `IRouteAlgorithmFactory` в данный `RouteCalculator`. Но мы не отдаем ему предпочтение, поскольку `IRouteAlgorithmFactory` станет бес-

полезным дополнительным уровнем косвенности, без которого легко можно будет обойтись. Вместо этого реализации `IRouteAlgorithm` будут включены в конструктор `RouteCalculator` (листинг 6.22).

Листинг 6.22. IRouteCalculator, заключающий в себе словарь алгоритмов вычисления маршрута IRouteAlgorithm

```
public class RouteCalculator : IRouteCalculator
{
    private readonly IDictionary<RouteType, IRouteAlgorithm> algorithms;

    public RouteCalculator(
        IDictionary<RouteType, IRouteAlgorithm> algorithms)
    {
        this.algorithms = algorithms;
    }

    public RouteResult Calculate(RouteSpecification spec, RouteType type)
    {
        return this.algorithms[type].CalculateRoute(spec);
    }
}
```

При использовании только что определенного `RouteCalculator` конструкция `RouteController` может приобрести следующий вид:

```
var algorithms = new Dictionary<RouteType, IRouteAlgorithm>
{
    { RouteType.Shortest, new ShortestRouteAlgorithm() },
    { RouteType.Fastest, new FastestRouteAlgorithm() },
    { RouteType.Scenic, new ScenicRouteAlgorithm() }
};

new RouteController(
    new RouteCalculator(algorithms));
```

Переделка из абстрактной фабрики в адаптер позволяет существенно сократить число зависимостей между вашими компонентами. На рис. 6.10 показан граф зависимостей исходного решения с использованием фабрики, а на рис. 6.11 изображен граф объектов после переделки.

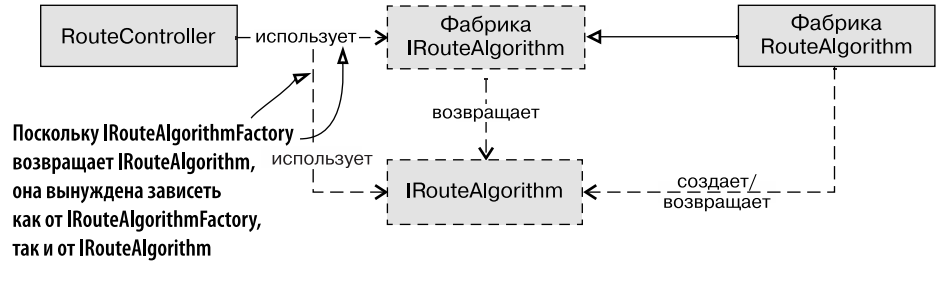


Рис. 6.10. Исходный граф зависимостей для RouteController с IRouteAlgorithmFactory

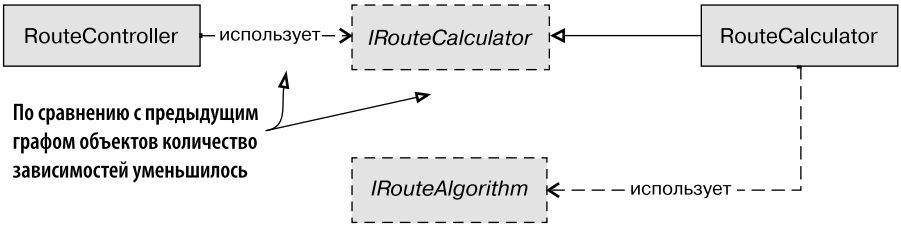


Рис. 6.11. Граф зависимостей для RouteController, когда вместо предыдущей конструкции имеется зависимость от IRouteCalculator

Когда для выбора зависимостей на основе предоставленных данных времени выполнения используются абстрактные фабрики, в большинстве случаев упростить конструкцию можно за счет переделки кода под использование адаптеров, не выставляющих напоказ, в отличие от абстрактных фабрик, исходную зависимость. Но это справедливо не только в отношении абстрактных фабрик. Данное обстоятельство хотелось бы обобщить.

Как правило, сервисные абстракции в своих определениях не должны раскрывать другие сервисные абстракции[1]. Это означает, что сервисная абстракция не должна принимать в качестве ввода другую сервисную абстракцию или же иметь сервисную абстракцию в качестве выходных параметров либо в качестве возвращаемого типа. Прикладные сервисы, зависящие от других прикладных сервисов, заставляют своих клиентов знать об обеих зависимостях.

ПРИМЕЧАНИЕ

Предыдущие выкладки — это скорее руководство к действию, чем строгое правило. Конечно же, бывают крайние ситуации, при которых наибольший смысл имеет применение возвращаемых абстракций, но проявляйте осторожность, когда дело доходит до редко возникающих ситуаций их использования при разрешении зависимостей. Именно из этих соображений данный вопрос рассматривается в качестве проблемного кода, а не в качестве антипаттерна.

Следующий проблемный код еще экзотичнее, поэтому встречается значительно реже. Ранее рассмотренные варианты проблемного кода могут оставаться незамеченными, а вот этот вариант трудно пропустить — ваш код просто прекратит компилироваться либо даст сбой во время выполнения.

6.3. Устранение зацикленности зависимостей

Временами реализации зависимостей приобретают зацикленность. Реализации требуется другая зависимость, чьей реализации требуется первая абстракция. Такой граф зависимостей нереализуем. Суть проблемы отображена на рис. 6.12.

[1] Это согласуется с принципом минимальной осведомленности (principle of least knowledge).

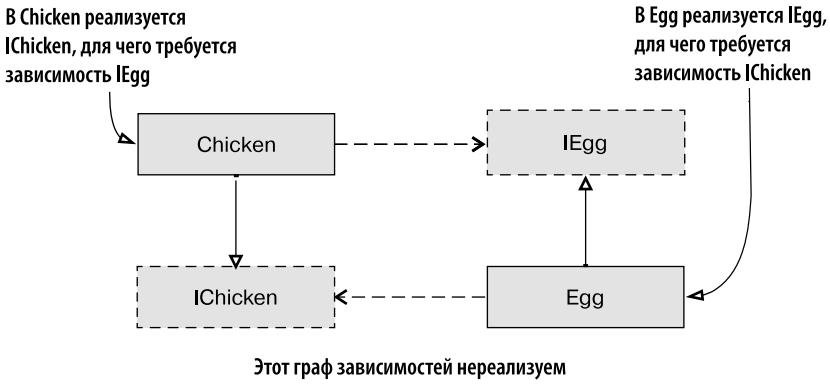


Рис. 6.12. Зацикленность зависимостей между Chicken и Egg

Вот как выглядит упрощенный пример, содержащий зацикленность зависимостей, показанную на рис. 6.12:

```
public class Chicken : IChicken
{
    public Chicken(IEgg egg) { ... }

    public void HatchEgg() { ... }
}

public class Egg : IEgg
{
    public Egg(IChicken chicken) { ... }
}
```

Chicken зависит от IEgg

Egg реализует IEgg

Egg зависит от Ichicken, который реализуется в Chicken

Можно ли на основе предыдущего примера составить граф объектов, содержащий эти классы?

```
new Chicken(
    new Egg(
        ???
    )
);
```

Для получения возможности создания нового экземпляра Chicken его конструктор должен быть предоставлен с уже существующим экземпляром Egg

Для получения возможности создания нового экземпляра Egg его конструктор должен быть предоставлен с уже существующим экземпляром Chicken, но предыдущий экземпляр Chicken еще не был создан, поскольку ему требуется уже существующий экземпляр Egg

У нас получается извечный спор о том, что появилось раньше: курица или яйцо? Подобный граф объектов создавать нельзя, поскольку обоим классам до создания их экземпляра требуется существование другого объекта. Пока зацикленность не устранена, возможность удовлетворения всех зависимостей отсутствует и приложение не сможет выполняться. Понятно, что что-то нужно сделать, но что именно?

В этом разделе будет рассмотрен вопрос зацикленных зависимостей, включая и их пример. Когда мы закончим, вашей первой реакцией должна стать попытка переделки ваших зависимостей, поскольку проблема обычно вызвана конструкцией

вашего приложения. Поэтому основной вывод из всего изложенного в разделе будет следующим: причиной зацикленных зависимостей обычно является нарушение принципа единственной ответственности.

Если переделка зависимостей не представляется возможной, зацикленность можно устранить за счет переделки внедрения через конструктор во внедрение через свойство. Это приведет к ослаблению инвариантности классов, поэтому на данный шаг следует решаться в последнюю очередь.

6.3.1. Пример: зацикленность зависимостей, вызванная нарушением принципа единственной ответственности

Мэри Роуэн (наш разработчик из главы 2) уже дошла в разработке своего приложения электронной торговли до его успешного практического применения. Но однажды у ее начальника возникло новое требование. Его недовольство вызвало то обстоятельство, что на практике в случае сбоев трудно определить, кто именно работал с определенным фрагментом данных в системе. Вариантом решения может стать сохранение изменений в таблице контрольного журнала, где будет вестись запись каждого изменения, вносимого каждым пользователем системы.

В конечном итоге Мэри додумалась до определения абстракции `IAuditTrailAppender`, показанной в листинге 6.23. (Следует заметить, что для демонстрации данного проблемного кода в реалистичной обстановке нам понадобится довольно сложный код. Поэтому следующий пример состоит из трех классов, и, прежде чем приступить к анализу кода, несколько страниц займет его объяснение.)

Листинг 6.23. Абстракция IAuditTrailAppender

```
public interface IAuditTrailAppender
{
    void Append(Entity changedEntity);
}
```

Позволяет добавлять записи в таблицу контрольного журнала путем передачи в нее изменяемой доменной сущности

Базовый класс, из которого выводятся все сущности

Для создания таблицы для хранения записей контрольного журнала `AuditEntries` Мэри воспользовалась SQL Server Management Studio. Ее определение показано в табл. 6.2.

Таблица 6.2. Таблица AuditEntries, созданная Мэри

Название столбца	Тип данных	Разрешает Null	Первичный ключ
Id	Uniqueidentifier	Нет	Да
UserId	Uniqueidentifier	Нет	Нет
TimeOfChange	DateTime	Нет	Нет
EntityId	Uniqueidentifier	Нет	Нет
EntityType	varchar(100)	Нет	Нет

После создания таблицы базы данных Мэри приступила к созданию реализации, показанной в листинге 6.24 абстракции `IAuditTrailAppender`.

Листинг 6.24. SqlAuditTrailAppender добавляет записи к таблице базы данных SQL

```
public class SqlAuditTrailAppender : IAuditTrailAppender
{
    private readonly IUserContext userContext;
    private readonly CommerceContext context;
    private readonly ITimeProvider timeProvider;

    public SqlAuditTrailAppender(
        IUserContext userContext,
        CommerceContext context,
        ITimeProvider timeProvider)
    {
        this.userContext = userContext;
        this.context = context;
        this.timeProvider = timeProvider;
    }

    public void Append(Entity changedEntity)
    {
        AuditEntry entry = new AuditEntry
        {
            UserId = this.userContext.CurrentUser.Id,
            TimeOfChange = this.timeProvider.Now,
            EntityId = entity.Id,
            EntityType = entity.GetType().Name
        };

        this.context.AuditEntries.Add(entry);
    }
}
```

Вспомним, что это интерфейс ITimeProvider из листинга 5.10

Создание нового объекта AuditEntry, который будет вставлен в таблицу AuditEntries. Эта запись составляется из текущего времени, информации, специфичной для предоставленной сущности, и идентификатора пользователя, выполняющего запрос

Важной частью ведения контрольного журнала является привязка изменения к пользователю. Для выполнения этой задачи `SqlAuditTrailAppender` требует зависимости от `IUserContext`. Она позволяет `SqlAuditTrailAppender` составлять запись, используя принадлежащее `IUserContext` свойство `CurrentUser`. Это то самое свойство, которое Мэри уже добавила в прошлом для реализации другой функции.

В листинге 6.25 показана текущая созданная Мэри версия `AspNetUserContextAdapter` (исходную версию можно увидеть в листинге 3.12).

Когда мы занимались изучением DI-паттернов и антипаттернов, Мэри тоже не сидела сложа руки. `IUserRepository` как раз и является одной из абстракций, добавленных ею в это время. В ближайшее время мы изучим реализацию `IUserRepository`.

Следующим шагом Мэри обновила классы, необходимые для заполнения контрольного журнала, одним из которых стал `SqlUserRepository`. В нем реализуется `IUserRepository`, и сейчас вполне подходящий момент, чтобы взглянуть на него. Код интересующей нас части этого класса показан в листинге 6.26.

Листинг 6.25. AspNetUserContextAdapter с добавленным свойством CurrentUser

```
public class AspNetUserContextAdapter : IUserContext
{

    private static HttpContextAccessor Accessor = new HttpContextAccessor();

    private readonly IUserRepository repository;

    public AspNetUserContextAdapter(
        IUserRepository repository)
    {
        this.repository = repository;
    }

    public User CurrentUser
    {
        get
        {
            var user = Accessor.HttpContext.User;
            string userName = user.Identity.Name;
            return this.repository.GetByName(userName);
        }
    }
    ...
}
```

Зависимость IUserRepository была добавлена Мэри, чтобы можно было извлекать информацию о пользователе из базы данных

Новое свойство

Получение имени вошедшего в систему пользователя из HttpContext и использование его для запроса экземпляра User из IUserRepository

Листинг 6.26. SqlUserRepository, требуемый для заполнения контрольного журнала

```
public class SqlUserRepository : IUserRepository
{
    public SqlUserRepository(
        CommerceContext context,
        IAuditTrailAppender appender)
    {
        this.appender = appender;
        this.context = context;
    }

    public void Update(User user)
    {
        this.appender.Append(user);
        ...
    }

    public User GetById(Guid id) { ... }

    public User GetByName(string name) { ... }
}
```

Для новой функции ведения контрольного журнала Мэри добавила зависимость от IAuditTrailAppender в конструктор SqlUserRepository

Метод Update изменен путем добавления вызова IAuditTrailAppender.Append. Это позволяет добавлять запись к контрольному журналу

Исходный код, не подвергшийся изменению

Этот метод используется свойством CurrentUser ранее рассмотренного класса AspNetUserContextAdapter

Мэри вплотную подошла к завершению своей задачи. Поскольку она добавила к методу `SqlUserRepository` аргумент конструктора, ей осталось обновить ко-

рень композиции. На данный момент та часть корня композиции, которая создает AspNetUserContextAdapter, выглядит следующим образом:

```
var userRepository = new SqlUserRepository(context);
IUserContext userContext = new AspNetUserContextAdapter(userRepository);
```

Поскольку IAuditTrailAppender был добавлен к конструктору SqlUserRepository в качестве зависимости, Мэри попыталась добавить его к корню композиции:

```
var appender = new SqlAuditTrailAppender(
    userContext,                          <-- Ой! А в этой строке Мэри
    context,                                  получила ошибку компиляции
    timeProvider);

var userRepository = new SqlUserRepository(context, appender);

IUserContext userContext = new AspNetUserContextAdapter(userRepository);
```

К сожалению, изменения, внесенные Мэри, не прошли компиляцию. Компилятор C# выразил недовольство: Cannot use local variable 'userContext' before it's declared (Не могу воспользоваться локальной переменной 'userContext' до ее объявления).

Поскольку SqlAuditTrailAppender зависит от IUserContext, Мэри пытается предоставить классу SqlAuditTrailAppender переменную userContext, которую она определила. Компилятор C# не принимает этот код, поскольку такая переменная должна быть определена до своего использования. Мэри пытается устранить проблему за счет перемещения определения и присваивания значения переменной userContext чуть выше, но это тут же заставляет компилятор C# выдать жалобу насчет переменной userRepository. Но когда она перемещает переменную userRepository вверх, компилятор начинает жаловаться на переменную appender, используемую до ее объявления.

Мэри начинает понимать, что у нее серьезные неприятности — в ее графе зависимостей возникла зацикленность. Разберемся, что пошло не так.

6.3.2. Анализ зацикленности зависимостей, с которой столкнулась Мэри

Зацикленность в созданном Мэри графе объектов появилась, как только она добавила к классу SqlUserRepository зависимость IAuditTrailAppender. Эта зацикленность зависимостей показана на рис. 6.13.

На рисунке показана зацикленность в графе объектов. Но этот граф является только частью истории. Вот как выглядит иное представление, выраженное в виде графа вызова методов, которым можно воспользоваться для анализа проблемы:

```
UserService.UpdateMailAddress(Guid userId, string newMailAddress)
    ➥ SqlUserRepository.Update(User user)
        ➥ SqlAuditTrailAppender.Append(Entity changedEntity)
            ➥ AspNetUserContextAdapter.CurrentUser
                ➥ SqlUserRepository.GetByName(string name)
```

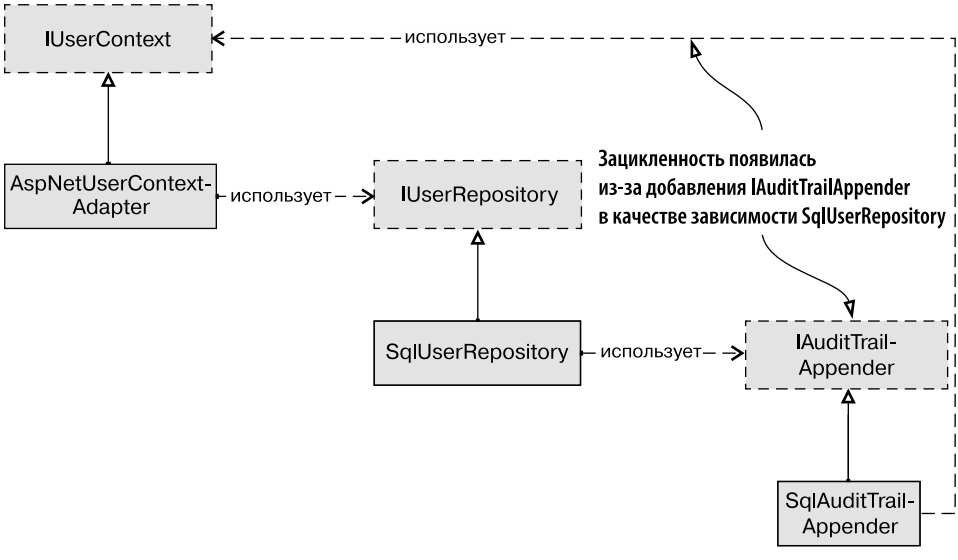


Рис. 6.13. Зацикленность зависимостей, в которую вовлечены AspNetUserContextAdapter, SqlUserRepository и SqlAuditTrailAppender

Этот граф вызовов показывает, как вызов начнется с метода `UpdateMailAddress`, принадлежащего `UserService`, что приведет к вызову метода `Update`, принадлежащего классу `SqlUserRepository`. Отсюда будет переход к `SqlAuditTrailAppender`, затем к `AspNetUserContextAdapter`, и все это завершится методом `GetByName`, принадлежащим `SqlUserRepository`.

ПРИМЕЧАНИЕ

Класс UserService не рассматривался, поскольку для развития нашей темы он не представляет особого интереса.

Из графа вызовов видно, что, несмотря на зацикленность графа объектов, рекурсия у графа вызовов методов отсутствует. Он станет рекурсивным, если в `GetByName`, к примеру, опять будет выполнен вызов `SqlAuditTrailAppender.Append`. Это приведет к бесконечным вызовам других методов, пока процесс не израсходует все пространство стека и не будет выдано исключение `StackOverflowException`. К счастью для Мэри, граф вызовов не является рекурсивным и не требует от нее переработки методов. Причина возникновения проблемы кроется в ином месте — в нарушении принципа единственной ответственности.

Когда рассматривались ранее объявленные классы `AspNetUserContextAdapter`, `SqlUserRepository` и `SqlAuditTrailAppender`, с обнаружением возможного нарушения принципа единственной ответственности могли возникнуть затруднения. Исходя из данных, представленных в табл. 6.3, все три класса были сконцентрированы на одной конкретной области.

Таблица 6.3. Абстракции и их роли в приложении

Абстракция	Роль	Методы
IAuditTrailAppender	Позволяет записывать важные изменения, внесенные пользователями	Один метод
IUserContext	Предоставляет потребителям информацию о пользователе, которому принадлежит текущий выполняемый запрос	Два метода
IUserRepository	Предоставляет операции, связанные с извлечением, запросом и сохранением данных о пользователях для заданной технологии доступа к данным	Три метода

Если присмотреться к `IUserRepository`, можно заметить, что функциональность в классе сгруппирована главным образом вокруг понятия пользователя, являющегося довольно обширным. Если придерживаться такого подхода к группировке пользовательских методов в одном классе, можно увидеть, что как `IUserRepository`, так и `SqlUserRepository` будут подвергаться изменениям довольно часто.

ПРИМЕЧАНИЕ

Постоянно меняющиеся абстракции являются красноречивым свидетельством нарушения принципа единственной ответственности. Здесь также затрагивается и принцип открытости/закрытости, рассмотренный в главе 4, который утверждает, что у вас должна быть возможность добавлять свойства, не испытывая при этом необходимости внесения изменений в уже существующие классы.

Если смотреть на принцип единственной ответственности с точки зрения связности, можно задаться вопросом, действительно ли методы в `IUserRepository` обладают сильной связью друг с другом. Насколько просто будет разбить класс на несколько более узкоспециализированных интерфейсов и классов?

6.3.3. Избавление от нарушений принципа единственной ответственности для решения проблемы зацикленности зависимостей

Избавиться от нарушений принципа единственной ответственности порой не так-то просто, поскольку это может повлечь за собой необходимость внесения изменений в потребителей абстракции. Но, как показано в листинге 6.27, в случае с нашим скромным приложением электронной торговли внести изменения сравнительно нетрудно.

В этом листинге метод `GetByName` извлекается из `IUserRepository` и `SqlUserRepository` и помещается в новую пару реализаций абстракции под названиями `IUserByNameRetriever` и `SqlUserByNameRetriever`. Новая реализация

SqlUserByNameRetriever не зависит от IAuditTrailAppender. В листинге 6.28 показана вся остальная часть SqlUserRepository.

Листинг 6.27. GetByName перемещается в IUserByNameRetriever

```
public interface IUserByNameRetriever
{
    User GetByName(string name);
}

public class SqlUserByNameRetriever : IUserByNameRetriever
{
    public SqlUserByNameRetriever(CommerceContext context)
    {
        this.context = context;
    }

    public User GetByName(string name) { ... }
}
```

Метод GetByName перемещается из IUserRepository в этот новый интерфейс IUserByNameRetriever

Листинг 6.28. Остальная часть IUserRepository и его реализация

```
public interface IUserRepository
{
    void Update(User user);
    User GetById(Guid id);
}

public class SqlUserRepository : IUserRepository
{
    public SqlUserRepository(
        CommerceContext context,
        IAuditTrailAppender appender
    {
        this.context = context;
        this.appender = appender;
    }

    public void Update(User user) { ... }
    public User GetById(Guid id) { ... }
}
```

Удаление метода GetByName

Удаление зависимости IUserContext

ПРИМЕЧАНИЕ

Чем больше у класса методов, тем выше шансы нарушения им принципа единственной ответственности. Здесь также прослеживается связь с принципом разделения интерфейсов, отдающим предпочтение более узкоспециализированным интерфейсам.

За счет разделения у Мэри изменились некоторые обстоятельства. Прежде всего, новые классы уменьшились и стали легче для понимания. Далее, снизилась

вероятность попадания в ситуацию, при которой Мэри придется постоянно обновлять существующий код. И последнее, но не менее значимое: разбиение класса `SqlUserRepository` привело к разрыву зацикленности зависимостей, поскольку новый класс `SqlUserByNameRetriever` не зависит от `IAuditTrailAppender`. Способ разрыва зацикленности зависимостей показан на рис. 6.14.

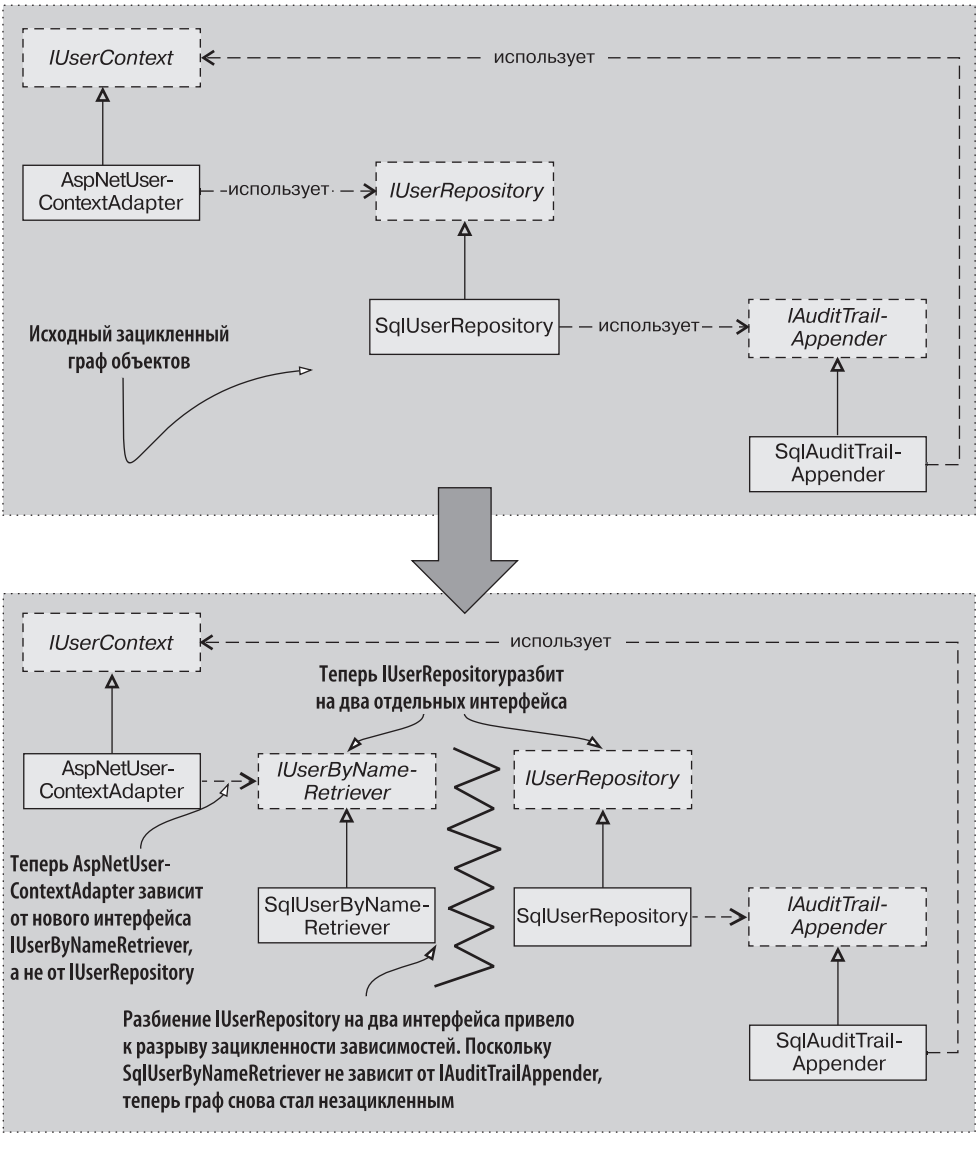


Рис. 6.14. Разрыв зацикленности зависимостей произошел благодаря разбиению IUserRepository на два интерфейса

Реализация контрольного журнала

Чтобы объяснить зацикленность зависимостей в этом разделе, выбрана особая реализация контрольного журнала. Но реализацией этой функции таким образом сами мы обычно не занимаемся. Контрольный журнал является сквозной функциональностью. Примененный Мэри способ реализации приводит к радикальным изменениям многих классов системы. Код получается громоздким и ненадежным, при этом нарушаются принципы открытости/закрытости и единственной ответственности (OCP и SRP).

Более простым решением было бы переопределение метода `DbContext.SaveChanges` внутри `CommerceContext`. Используя имеющееся у него свойство `ChangeTracker`, экземпляр `DbContext` позволяет запрашивать изменения. При этом устраняется необходимость внесения радикальных изменений и потребность в реализации соответствующего кода в каждом отдельно взятом классе и в написании соответствующих тестов. Но мы предпочитаем применение доменных событий в соответствии с объяснениями, изложенными в разделе 6.1. Предыдущий метод сохраняет изменение каждой сущности в виде записи в контрольном журнале, а доменные события предоставляют отслеживание на более высоком функциональном уровне.

Возьмем, к примеру, предыдущий пример с попыткой обновления почтового адреса пользователя. Когда выдается событие `UserMailAddressChanged`, его отслеживание можно добавить в журнал. С ним можно сохранить идентификатор пользователя, чей адрес изменился, время изменения и пользователя, который его внес. Получится контрольный журнал, предоставляющий отличный обзор происходящего в каждый момент времени. При визуализации доменных событий относительно заданного заказа `Order` в приложении электронной торговли может получиться представление, показанное в следующей таблице.

Хронология для заданного заказа Order

Дата	Пользователь	Описание
2018-11-21 15:21	Mary Rowan	Заказ создан
2018-11-21 15:26	Mary Rowan	Заказ подтвержден
2018-11-21 15:27	Mary Rowan	Заказ оплачен
2018-11-22 08:10	[system]	Заказ отправлен
2018-11-23 15:48	[system]	Заказ доставлен

Еще одно решение реализации контрольного журнала будет показано в главе 10.

Связывающий все воедино новый корень композиции показан в следующем примере кода:

```
var userContext = new AspNetUserContextAdapter(
    new SqlUserByNameRetriever(context));

var appender = new
    SqlAuditTrailAppender(
        userContext,
        context,
        timeProvider);

var repository = new SqlUserRepository(context, appender);
```

Теперь AspNetUserContextAdapter зависит от IUserByNameRetriever, поскольку он требует извлечения пользователей по их именам

Самой распространенной причиной зацикленности зависимостей является нарушение принципа единственной ответственности. Обычно неплохим решением по устранению нарушения является разбиение классов на более мелкие, узкоспециализированные классы, но существуют и другие стратегии разрыва зацикленности зависимостей.

6.3.4. Общие стратегии избавления от зацикленности зависимостей

Столкнувшись с зацикленностью зависимостей, мы сразу же интересуемся: «Где же я оступился?» Зацикленность зависимостей должна тут же подстегнуть к выявлению ее первоисточника. Любая зацикленность является просчетом проектирования, поэтому первой реакцией должна стать переделка вовлеченной в нее части кода, чтобы в первую очередь избавиться от возникновения зацикленности. Общие направления предпринимаемых при этом действий приведены в табл. 6.4.

Таблица 6.4. Некоторые стратегии перепроектирования для избавления от зацикленности зависимостей, изложенные в порядке предпочтительности

Стратегия	Описание
Разбиение классов	Как следует из примера с контрольным журналом, в большинстве случаев зацикленность можно разорвать, разбив классы, перегруженные методами, на более мелкие
Применение .NET-событий	Зачастую зацикленность можно разорвать, изменив одну из абстракций, заставив ее вместо явного вызова зависимости выдавать события, информирующие зависимость о том, что что-то произошло. Применение событий особенно пригодится в том случае, когда одна сторона вызывает в отношении своей зависимости только методы, не возвращающие значений (void)
Внедрение зависимости через свойство	Если ничем другим воспользоваться не удалось, разорвать зацикленность можно переделкой одного из классов с внедрения зависимости через конструктор под ее внедрение через свойство. Это должно стать крайним средством, поскольку им излечивается не сама болезнь, а ее симптомы

Не ошибитесь: зацикленность зависимостей является проблемным кодом. Сначала нужно проанализировать код, чтобы понять причину ее возникновения. И все же иногда изменить конструкцию не представляется возможным, даже если понятна основная причина зацикленности.

6.3.5. Крайнее средство: разрыв зацикленности путем использования внедрения через свойство

Бывает, что просчеты в конструкции вам неподконтрольны, но зацикленность все же следует разорвать. В таких случаях это можно сделать путем внедрения зависимости через свойство, даже если это решение будет носить временный характер.

ВНИМАНИЕ

К разрыву зацикленности путем использования внедрения через свойство нужно прибегать только в самом крайнем случае.

Чтобы разорвать зацикленность, нужно ее проанализировать с целью выявления места разрыва. Поскольку использование внедрения через свойство предлагается в качестве вспомогательного средства, важно тщательно изучить все зависимости, чтобы определить, где рассечение принесет меньше вреда.

В нашем примере с контрольным журналом избавиться от зацикленности можно путем замены внедрения зависимости от `SqlAuditTrailAppender` через конструктор на ее внедрение через свойство. Это означает, что сначала можно создать `SqlAuditTrailAppender`, внедрить эту зависимость в `SqlUserRepository`, после чего, как это показано в листинге, передать `SqlAuditTrailAppender` значение `AspNetUserContextAdapter` (листинг 6.29).

Листинг 6.29. Разрыв зацикленности с помощью внедрения через свойство

```
var appender =
    new SqlAuditTrailAppender(context, timeProvider);
var repository =
    new SqlUserRepository(context, appender);

var userContext = new
AspNetUserContextAdapter(
    new SqlUserByNameRetriever(context));

appender.UserContext = userContext;
```

Создание кода добавления записи без экземпляра IUserContext. Получается частично инициализированный экземпляр

Внедрение частично инициализированного кода добавления записи в хранилище

Использование внедрения через свойство для завершения инициализации кода добавления записи путем добавления IUserContext

Такое использование внедрения через свойство усложняет `SqlAuditTrailAppender`, поскольку теперь ему приходится иметь дело с зависимостью, которая еще недоступна. Это, как уже говорилось в разделе 4.3.2, приводит к временной связанности.

ПРИМЕЧАНИЕ

> Ранее, в подразделе 4.2.1, утверждалось, что классы никогда не должны выполнять работу с привлечением зависимостей в своих конструкторах. Кроме того, что это замедляет и снижает надежность создания объектов, использование внедренной зависимости может дать сбой, поскольку к тому времени ее полная инициализация может быть еще не проведена.

Если нет никакого желания подобным образом ослаблять любые исходные классы, можно воспользоваться тесно связанным с этой проблемой подходом, заключающимся во вводе виртуального заместителя, который оставляет `SqlAuditTrailAppender` нетронутым[1] (листинг 6.30).

Листинг 6.30. Разрыв зацикленности зависимостей с помощью виртуального заместителя

```
var lazyAppender = new LazyAuditTrailAppender();

var repository =
new SqlUserRepository(context, lazyAppender);

var userContext = new
AspNetUserContextAdapter(
    new SqlUserByNameRetriever(context));

lazyAppender.Appender =
    new SqlAuditTrailAppender(
        userContext, context, timeProvider);
```

Создание виртуального заместителя

Вставка реального кода добавления записей в свойство виртуального заместителя

В `LazyAuditTrailAppender` реализуется `IAuditTrailAppender` наподобие того, как это делается в `SqlAuditTrailAppender`. Но его зависимость `IAuditTrailAppender` получается путем внедрения через свойство, а не за счет внедрения через конструктор, что позволяет разорвать зацикленность, не нарушая инвариантов исходных классов. Виртуальный заместитель `LazyAuditTrailAppender` показан в листинге 6.31.

Листинг 6.31. Реализация виртуального заместителя LazyAuditTrailAppender

```
public class LazyAuditTrailAppender : IAuditTrailAppender
{
    public IAuditTrailAppender Appender { get; set; }

    public void Append(Entity changedEntity)
    {
        if (this.Appender == null)
        {
            throw new InvalidOperationException("Appender was not set.");
        }

        this.Appender.Append(changedEntity);
    }
}
```

Свойство, позволяющее разорвать зацикленность зависимостей

Контрольная инструкция

Перенаправление вызова

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 250.

Всегда имейте в виду, что лучшим способом избавления от зацикленности является переделка API, приводящая к исчезновению этой зацикленности. Но в редких случаях, когда сделать это невозможно или крайне нежелательно, в качестве крайнего средства зацикленность нужно разрывать путем использования внедрения через свойство. Это позволяет составить остальную часть графа объектов таким образом, чтобы она не подпала под зависимость, связанную со свойством. Когда вся остальная часть графа объектов уже заполнена, внедрение соответствующего экземпляра можно выполнить через свойство. Внедрение через свойство сигнализирует о необязательном характере зависимости, поэтому к изменениям нужно относиться осмотрительно.

Если усвоить несколько базовых принципов, внедрение зависимостей не составит особого труда. Но по мере его освоения вы гарантированно столкнетесь с проблемами, которые могут вас на время озадачить. Эта глава была посвящена ряду наиболее часто встречающихся проблем. Вместе с двумя предыдущими главами был сформирован каталог паттернов, антипаттернов и примеров проблемного кода. Этот каталог формирует часть II данной книги. В части III мы обратимся к трем измерениям DI: к компоновке объектов (Object Composition), управлению временем жизни объектов (Lifetime Management) и перехвату (Interception).

Резюме

- Постоянно меняющиеся зависимости — явный признак нарушения принципа единственной ответственности. Это также справедливо в отношении соблюдения принципа открытости/закрытости, утверждающего, что у вас должна быть возможность добавлять функции без необходимости внесения изменений в уже существующие классы.
- Чем больше у класса методов, тем выше шансы нарушения принципа единственной ответственности. Это также справедливо в отношении соблюдения принципа изоляции интерфейса, утверждающего, что клиентов не нужно заставлять зависеть от неиспользуемых ими методов.
- Если не раздувать абстракцию, можно решить множество проблем, связанных, к примеру, со сложностью создания заместителей, декораторов и тестовых дубликатов.
- Преимущество использования внедрений через конструктор состоит в более очевидном проявлении нарушений принципа единственной ответственности. Когда у одного класса слишком много зависимостей, это служит сигналом к необходимости изменения его конструкции.
- Когда список параметров конструктора становится слишком большим, это называется избыточным внедрением через конструктор, а соответствующий код считается проблемным. Это один из наиболее типичных примеров проблемного кода, не связанного с внедрением зависимостей, но еще более подчеркивающегося этим внедрением.
- Переделка кода с избыточным внедрением через конструктор может выполняться множеством способов, но неизменно беспроигрышным вариантом будет разбие-

ние большого класса на малые, более специализированные классы в соответствии с хорошо известными паттернами проектирования.

- Переделать код при избыточном внедрении через конструктор можно за счет использования фасадных сервисов. Фасадный сервис скрывает естественную группу взаимодействующих зависимостей с их поведением за одной-единственной абстракцией.
- Переделка с использованием фасадных сервисов позволяет обнаруживать подобные естественные группы и выявлять ранее нераскрытые связи и концепции предметной области. Фасадные сервисы родственны граничным объектам (Parameter Objects), но вместо объединения и предоставления компонентов они предоставляют только инкапсулированное поведение, скрывая при этом составляющие.
- Ликвидировать избыточное внедрение через конструктор можно за счет применения в приложении доменных событий. При их использовании осуществляется перехват действий, способных вызвать изменение состояния разрабатываемого вами приложения.
- Абстракцией с протечкой считается такая абстракция, например интерфейс, которая допускает утечку подробностей реализации, например характерных для уровня типов или характерного для реализации поведения.
- Абстракции, реализующие `IDisposable`, являются абстракциями с протечкой. `IDisposable` должны вводиться в действие не в абстракции, а в реализации.
- Концептуально существует только один экземпляр сервисной абстракции. Абстракции, допускающие утечку этой информации, делая ее доступной своим потребителям, не предназначены для этих потребителей.
- Как правило, сервисные абстракции не должны в своих определениях выставлять напоказ другие сервисные абстракции. Абстракции, зависящие от других абстракций, принуждают своих клиентов к осведомленности об обеих абстракциях.
- Когда дело доходит до применения DI, абстрактные фабрики зачастую находят чрезмерное применение. Во многих случаях для этого есть более подходящие альтернативы.
- Концептуально зависимости, созданные абстрактной фабрикой, должны требовать значение времени выполнения. Перевод значения времени выполнения в абстракцию должен иметь смысл на концептуальном уровне. Если необходимость введения абстрактной фабрики чувствуется в связи с приобретением возможности создания экземпляров конкретной реализации, может получиться абстракция с протечкой. Более удачным альтернативным решением может стать применение заместителя.
- Порой исключить фабричное поведение внутри некоторых классов невозможно. Но к фабричным абстракциям, применяемым в масштабах всего приложения, нужно относиться с недоверием.
- Применение абстрактной фабрики всегда приводит к росту числа зависимостей, имеющихся у потребителя, усложняя код.

- Когда абстрактные фабрики применяются для выбора зависимостей на основе предоставляемых данных среды выполнения, код чаще всего можно упростить, переделав его под применение фасадов, не выставляющих напоказ основную зависимость.
- Зацикливания зависимости обычно вызываются нарушением принципа единственной ответственности.
- Нужно отдавать предпочтение улучшению конструкции той части приложения, в которой содержится зацикленная зависимость. В большинстве случаев такое улучшение сводится к разбиению классов на более мелкие, специализированные.
- От зацикленных зависимостей можно избавляться за счет применения внедрения через свойство. Но разрывать зацикленность путем использования внедрения через свойство нужно только в самом крайнем случае. Это средство избавления от симптомов, а не от самой болезни.
- Классы никогда не должны выполнять работу, задействующую зависимости в своих конструкторах, поскольку внедряемая зависимость может быть еще не полностью проинициализирована.

Часть III
Чистое внедрение зависимостей

В главе 1 давалось краткое описание трех измерений, связанных с технологией DI: компоновки объектов (Object Composition), управления временем жизни (Lifetime Management) и перехвата (Interception). В этой части книги будет проведено более глубокое исследование данных трех измерений, и каждому из них будет посвящена отдельная глава. У многих DI-контейнеров имеются функциональные особенности, непосредственно связанные с этими измерениями. Часть из них предоставляет все три измерения, а часть поддерживает только некоторые из них.

Поскольку DI-контейнер является дополнительным инструментом, мы считаем, что полезнее будет объяснить суть основных принципов и приемов, обычно применяемых в контейнерах для реализации данных функциональных особенностей. Исходя из этого в части III рассматривается применение DI без использования DI-контейнера. Это практическое руководство по принципу «сделай сам», то есть то, что мы называем чистым внедрением зависимостей (Pure DI).

В главе 7 объясняется, как выполнять компоновку объектов в различных средах, таких как ASP.NET Core MVC, Console Applications и т. д. Внедрение зависимостей не во всех средах поддерживается одинаково хорошо, и даже среди тех, где поддержка имеет успех, ее детали сильно различаются. Для каждой среды может трудно даваться идентификация того шва (seam), который позволяет воспользоваться технологией DI. Но, как только такой шов будет найден, у вас появится решение для всех приложений, использующих данную конкретную среду. В главе 7 будет проделана соответствующая работа для большинства самых распространенных сред для .NET-приложений. Ее можно считать своеобразным каталогом швов, находящихся в средах выполнения приложений.

Хотя компоновка объектов при использовании чистой технологии DI не представляет особых сложностей, после изучения вопросов управления продолжительностью их жизни в главе 8 вы должны начать видеть преимущества применения настоящего DI-контейнера. Организовать надлежащее управление временем жизни различных объектов из их графа вполне возможно, но для этого понадобится больше пользовательского кода, чем для компоновки объектов. И ничто в объеме этого кода не добавит приложению какой-либо конкретной ценности с точки зрения бизнеса. Кроме объяснения основ управления временем жизни, в главе 8 также содержится каталог самых распространенных жизненных циклов (lifestyle). Этот каталог служит в качестве словаря для рассмотрения жизненных циклов во всей части IV книги. Хотя самостоятельно реализовывать жизненные циклы совсем не обязательно, весьма полезно знать, как они работают.

В остальных главах части III объясняется последнее измерение технологии DI: перехват. В главе 9 будет рассмотрена часто возникающая проблема реализации сквозных задач на основе компонентов. Это будет сделано с помощью паттерна проектирования «Декоратор». Глава 9 послужит также в качестве введения в две последующие главы.

В главе 10 будет рассмотрена парадигма аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP) и будет показано, как тщательно проработанный проект приложения, основанный на принципах SOLID, позволяет создавать легко сопровождаемый код, не требующий использования каких-либо

специальных инструментов. Эта глава считается кульминацией книги — именно здесь многие читатели, использующие программу раннего доступа, сказали, что они начали просматривать очертания чрезвычайно эффективного способа моделирования программных средств.

Кроме применения SOLID-принципов проектирования, существуют и другие способы, позволяющие практиковать аспектно-ориентированное программирование. Вместо использования паттернов и принципов можно воспользоваться специализированным инструментарием, например связыванием аспектов в ходе компиляции (compile-time weaving) и инструментами динамического перехвата. Соответствующие вопросы рассматриваются в главе 11.

7 Компоновка приложений

В этой главе

- Компоновка консольных приложений.
- Компоновка приложений универсальной платформы Windows (UWP-приложений).
- Компоновка приложений для среды ASP.NET Core MVC.

Приготовление нескольких изысканных блюд — задача непростая, особенно если вы сами хотите принять участие в застолье. Невозможно есть и готовить одновременно, и для многих блюд полная готовность к подаче на стол наступает в последнюю минуту приготовления. Профессиональные повара знают, как справиться с многими такими трудностями. Помимо множества секретов своего мастерства, они руководствуются главным принципом *мизанпласа*, что в вольном переводе означает *«все на своих местах»*. Все, что должно быть заранее приготовлено, готовится заблаговременно. Овощи чистятся и шинкуются, мясо режется на куски, варятся бульоны, прогревается духовка, раскладываются приборы и т. д.

Если на десерт подается мороженое, оно может быть приготовлено накануне. Если в первом блюде содержатся мидии, их можно почистить за несколько часов до подачи на стол. Даже такой капризный компонент, как беарнский соус, может быть приготовлен за час до его употребления. Когда гости готовы сесть за стол, останутся только последние приготовления: разогрев соуса во время жарки мяса и т. д. Во многих случаях окончательно стол накрывается не более чем за 5–10 минут. Весь этот процесс показан на рис. 7.1.

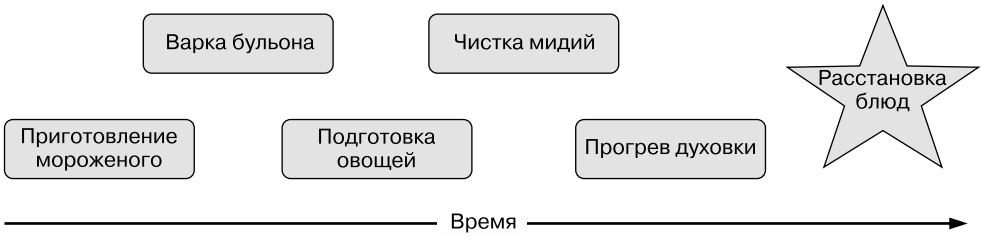


Рис. 7.1. Мизанплас подразумевает предварительное приготовление всех компонентов блюд, чтобы можно было как можно быстрее и без особых усилий накрыть на стол

Принцип мезанпласа подобен разработке приложений со слабой связанностью, достигаемой за счет применения технологии DI. Можно заранее написать все требуемые компоненты и заняться их компоновкой при абсолютной необходимости.

Как и все аналогии, эту мы можем развивать только до определенного момента. В кулинарном искусстве приготовление блюд и их подача на стол разделены по времени, а в разработке приложений разделение происходит по модулям и уровням. Порядок компоновки компонентов корня композиции показан на рис. 7.2.

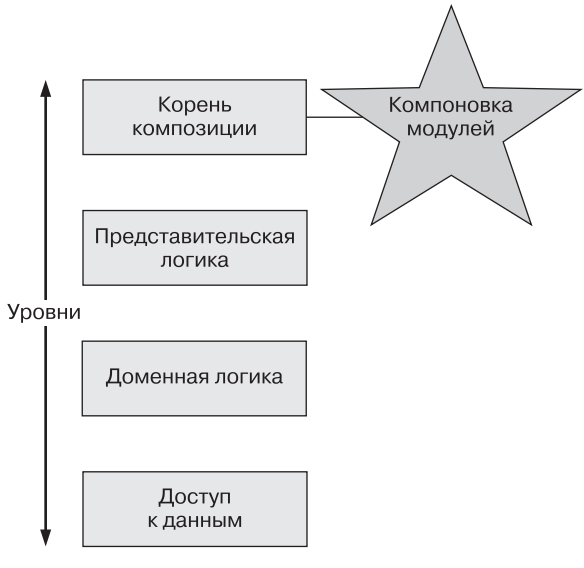


Рис. 7.2. В корне композиции выполняется компоновка всех независимых модулей приложения

Первым событием при выполнении приложения станет компоновка объектов. Как только будет собран граф объектов, компоновка объектов завершится и начнут работу составленные компоненты. В этой главе мы сосредоточимся на корнях композиции нескольких сред выполнения приложений. В отличие от мизанпласа компоновка объектов не происходит как можно позже, а выполняется, когда требуется интеграция различных модулей.

ОПРЕДЕЛЕНИЕ

Компоновка объектов представляет собой выстраивание иерархии взаимосвязанных компонентов. Эта компоновка происходит внутри корня композиции.

Компоновка объектов является основой технологии DI и самой простой частью для понимания. Вы уже знаете, как она выполняется, поскольку занимались компоновкой объектов при каждом создании объектов, содержащих другие объекты.

В разделе 4.1 приводились основы порядка и способов компоновки приложений. В этой главе повторений не будет. Вместо этого хочется оказать помощь в преодолении некоторых сложностей, возникающих при компоновке объектов. Эти сложности обусловливаются не самой компоновкой объектов, а теми средами выполнения приложений, в которых вы работаете. Проблемы носят особый характер применительно к каждой среде и требуют соответствующих решений. Исходя из нашего опыта, эти проблемы представляют собой одно из главных препятствий на пути успешного применения технологии DI, поэтому на них и будет сконцентрировано наше внимание. Такой подход придаст главе скорее практический, чем теоретических характер, тем самым отличая ее от предыдущих глав.

ПРИМЕЧАНИЕ

Если есть желание усвоить лишь применение DI в избранной вами среде, можно сразу перейти к соответствующему разделу данной главы. Каждый раздел предназначался для автономного изучения.

Совсем не трудно составить полную иерархию зависимостей приложения, полностью контролируя время жизни приложения (как в случае с приложениями командной строки). Но некоторые среды из семейства .NET (например, ASP.NET Core) используют инверсию управления, способную порой затруднить применение DI. Ключом к использованию DI для этой конкретной среды является понимание каждого ее шва. В этой главе будет рассмотрен способ реализации корня композиции в самых распространенных средах .NET Core.

Каждый раздел будет начинаться с общего введения в применение DI в конкретной среде, а затем последует подробный пример на базе повсеместно используемого в этой книге учебного приложения электронной торговли. Начнем с самой простой среды, в которой применяется DI, а затем постепенно проработаем вопросы использования этой технологии в более сложных средах. Пока самой простой средой применения DI является консольное приложение, рассматриваемое в следующем разделе.

ПРИМЕЧАНИЕ

Некоторые старые среды семейства .NET (такие как PowerShell и ранние версии ASP.NET Web Forms) являются совершенно неблагоприятными для применения DI. А вот новые среды .NET Core имеют по отношению к DI более дружественный характер. В этой книге основное внимание будет сконцентрировано на этих самых новых средах .NET Core.

7.1. Компоновка консольных приложений

Бесспорно, консольное приложение проще всего поддается компоновке. В отличие от большинства других сред выполнения приложений .NET BCL консольное приложение практически не включает в себя инверсии управления. Когда выполнение доходит до точки входа в приложение (обычно это метод `Main` в классе `Program`), вы предоставлены самим себе. Никаких специальных событий, на которые можно подписаться, никаких интерфейсов для реализации и крайне малое число сервисов, доступных для использования.

Вполне подходящим корнем композиции является класс `Program`. Модули приложения компонуются в методе `Main`, позволяющем им начинать работу. В этом нет ничего особенного, но перейдем к примеру.

7.1.1. Пример: обновление курсов валют с помощью программы UpdateCurrency

В главе 4 рассматривались способы предоставления функции конвертации валют для учебного приложения электронной торговли. В подразделе 4.2.4 была представлена абстракция `ICurrencyConverter`, применяющая обменные курсы одной валютной единицы на другие валюты. Поскольку `ICurrencyConverter` представляет собой интерфейс, можно было создавать множество разных реализаций, но в примере использовалась база данных. Код примера в главе 4 предназначался для демонстрации порядка извлечения данных и выполнения конвертации валют, поэтому порядок обновления в базе данных обменных курсов валют не приводился.

ПРИМЕЧАНИЕ

Полностью исходный код примера доступен в сопровождающем книгу сборнике исходного кода.

Чтобы продолжить пример, узнаем, как создается простое консольное приложение .NET Core, позволяющее администратору или привилегированному пользователю обновлять обменные курсы валют без необходимости непосредственного взаимодействия с базой данных. Консольное приложение обращается к базе данных и обрабатывает входящие аргументы командной строки. Поскольку целью этой программы является обновление обменных курсов валют в базе данных, мы назовем ее `UpdateCurrency`. Она принимает два аргумента командной строки:

- код валюты;
- обменный курс этой валюты на основную валюту (USD).

Доллар США (USD) является в нашей системе основной валютой, и все курсы обмена на другие валюты хранятся относительно этой валюты. Например, обменный курс USD на EUR выражается как 1 USD стоимостью 0,88 EUR (по курсу

декабря 2018 года). Когда нужно обновить обменный курс, используется следующая командная строка:

```
d:\> dotnet commerce\UpdateCurrency.dll EUR "0.88"
Updated: 0.88 EUR = 1 USD.
```

ПРИМЕЧАНИЕ

Файл консольного приложения в среде .NET Core имеет расширение .dll (а не .exe) и может быть запущен с помощью команды dotnet, в которой имя DLL-файла используется в качестве первого аргумента.

Выполнение программы приведет к обновлению базы данных и выводу новых значений обратно на консоль. Посмотрим на процесс создания такого консольного приложения.

7.1.2. Построение корня композиции для программы UpdateCurrency

В `UpdateCurrency` используется точка входа по умолчанию для консольной программы: метод `Main` в классе `Program`. Она выполняет роль корня композиции приложения (листинг 7.1).

Листинг 7.1. Корень композиции консольного приложения

```
class Program
{
    static void Main(string[] args)
    {
        string connectionString = LoadConnectionString();

        CurrencyParser parser =
            CreateCurrencyParser(connectionString);

        ICommand command = parser.Parse(args);
        command.Execute();
    }

    static string LoadConnectionString()
    {
        var configuration = new ConfigurationBuilder()
            .SetBasePath(AppContext.BaseDirectory)
            .AddJsonFile("appsettings.json", optional: false)
            .Build();

        return configuration.GetConnectionString(
            "CommerceConnectionString");
    }

    static CurrencyParser CreateCurrencyParser(string connectionString) ...
}
```

Загрузка значений конфигурации

Построение графа объектов

Вызов нужных функций

Единственной ответственностью класса `Program` является загрузка конфигурационных значений, компоновка всех соответствующих модулей и предоставление возможности скомпонованному графу объектов приступить к функционированию. В этом примере компоновка модулей приложения перенесена в метод `CreateCurrencyParser`, а метод `Main` отвечает за вызов методов в скомпонованном графе объектов. Метод `CreateCurrencyParser` компонует свой граф объектов, используя жестко задаваемые зависимости. К изучению его реализации мы вскоре вернемся.

Любой корень композиции должен проделать только четыре действия: загрузить конфигурационные значения, построить граф объектов, вызвать желаемые функции и освободить граф объектов (поговорим об этом в следующей главе). Как только все это будет выполнено, он должен уйти в сторону и предоставить все остальное вызванному экземпляру.

ПРИМЕЧАНИЕ

Как уже указывалось в подразделе 4.1.3, нужно отделить загрузку конфигурационных значений от методов, занимающихся компоновкой объектов, как показано в листинге 7.1. Тем самым компоновка объектов будет отделена от конфигурации используемой системы, позволяя провести тестирование независимо от наличия или отсутствия надлежащего конфигурационного файла.

Теперь при наличии этой инфраструктуры можно потребовать от `CreateCurrencyParser` создания `CurrencyParser`, проводящего анализ входящих аргументов и выполняющего в конечном итоге соответствующую команду. В этом примере используется чистая технология DI, но ее ничего не стоит заменить DI-контейнером, подобным тому, что рассматривается в части IV этой книги.

7.1.3. Компоновка графа объектов в CreateCurrencyParser

Существование метода `CreateCurrencyParser` обусловлено явно выраженной целью подключения всех зависимостей для программы `UpdateCurrency`. Его реализация показана в листинге 7.2.

Листинг 7.2. Метод CreateCurrencyParser, выполняющий компоновку графа объектов

```
static CurrencyParser CreateCurrencyParser(string connectionString)
{
    IExchangeRateProvider provider =
        new SqlExchangeRateProvider(
            new CommerceContext(connectionString));

    return new CurrencyParser(provider);
}
```

Компоновка графа объектов

Граф объектов, показанный в этом листинге, не отличается особой глубиной. Классу `CurrencyParser` требуется экземпляр интерфейса `IExchangeRateProvider`, а для обмена информацией с базой данных в методе `CreateCurrencyParser` сконструирован `SqlExchangeRateProvider`.

Класс `CurrencyParser` использует внедрение через конструктор, поэтому ему передается только что созданный экземпляр `SqlExchangeRateProvider`. Затем из метода возвращается сам новоиспеченный класс `CurrencyParser`. Если вам интересно, то сигнатура конструктора класса `CurrencyParser` имеет следующий вид:

```
public CurrencyParser(IExchangeRateProvider exchangeRateProvider)
```

Напомним, что `IExchangeRateProvider` является интерфейсом, реализуемым `SqlExchangeRateProvider`. В качестве части корня композиции `CreateCurrencyParser` содержит жестко заданное отображение `IExchangeRateProvider` на `SqlExchangeRateProvider`. Но остальная часть кода остается со слабой связанностью, поскольку является только потребителем абстракции.

Этот пример может показаться простым, но в нем скомпонованы типы из трех разных уровней приложения. Уделим немного внимания порядку взаимодействия этих уровней в данном примере.

7.1.4. Более пристальный взгляд на разбиение UpdateCurrency на уровни

Корень композиции — это место сбора компонентов всех уровней. Точка входа и корень композиции составляют единственный код исполняемого файла. Как показано на рис. 7.3, вся реализация делегирована нижестоящим уровням.

Схема на рис. 7.3 может показаться сложной, но на ней представлена практически вся кодовая база консольного приложения. Основная часть логики приложения состоит из анализа входящих аргументов и выбора надлежащей команды на основе входных данных. Все это происходит на уровне сервисов приложения, чьим единственным средством прямого обмена информацией с доменным уровнем является интерфейс `IExchangeRateProvider` и класс `Currency`.

`IExchangeRateProvider` вставляется в `CurrencyParser` корнем композиции и применяется впоследствии в качестве абстрактной фабрики для создания экземпляра `Currency`, используемого `UpdateCurrencyCommand`. Уровень доступа к данным предоставляет реализации доменных абстракций на основе SQL Server. Хотя напрямую с этими реализациями ни один из других классов приложения не взаимодействует, `CreateCurrencyParser` отображает абстракции на конкретные классы.

ПРИМЕЧАНИЕ

Если вспомнить материал раздела 6.2, к использованию абстрактных фабрик нужно относиться осмотрительно. Но в данном случае применение абстрактной фабрики вполне оправданно, поскольку ею может воспользоваться только корень композиции.

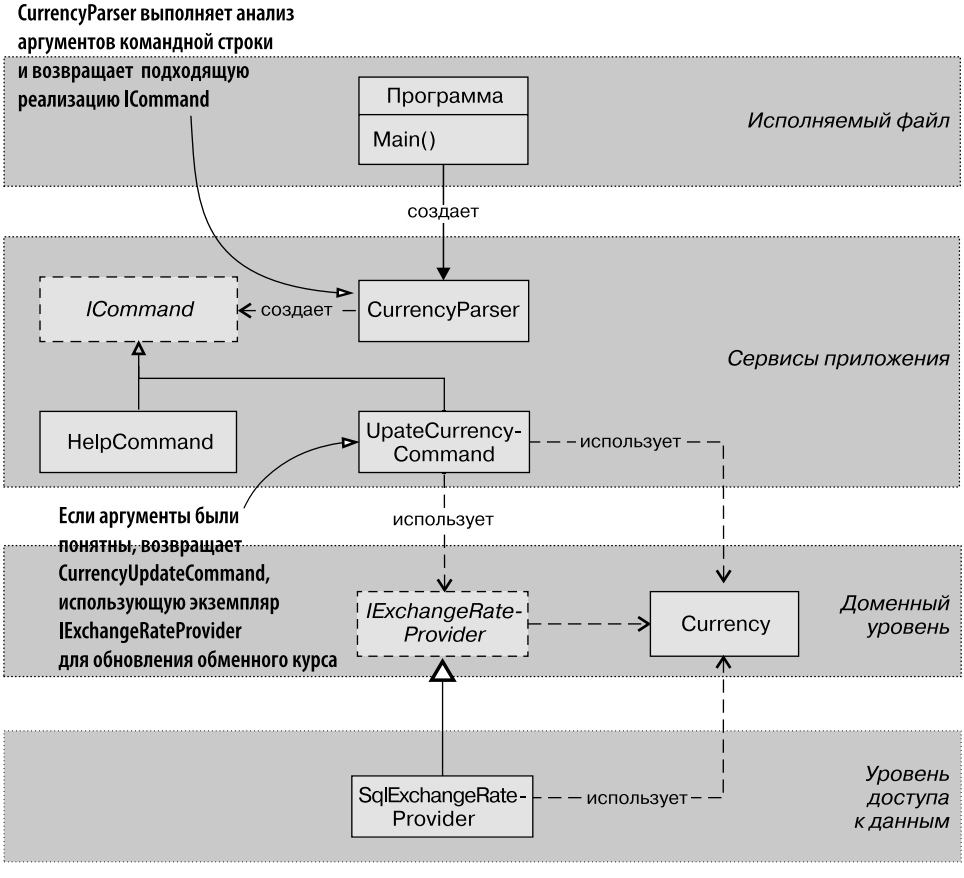


Рис. 7.3. Компонентный состав приложения UpdateCurrency

Использование технологии DI с консольным приложением не составляет труда, поскольку внешняя инверсия управления практически отсутствует. Среда .NET Framework запускает процесс и передает управление методу `Main`. Это аналогично работе с универсальной платформой Windows (UWP), позволяющей проводить компоновку объектов без каких-либо швов.

7.2. Компоновка UWP-приложений

Компоновка UWP-приложения дается почти так же легко, как и компоновка консольного приложения. В этом разделе будет создано небольшое UWP-приложение для управления товарами приложения электронной торговли, для чего мы воспользуемся паттерном «Модель — представление — модель представления» (Model — View — ViewModel, MVVM). Будут рассмотрены вопросы размещения корня композиции, порядка создания и инициализации моделей представлений, порядок

привязки представлений к соответствующим им моделям представлений и порядок обеспечения перехода от одной страницы к другой.

Точка входа в UWP-приложение довольно проста, и, хотя ею не предоставляются швы, явно предназначенные для применения технологии DI, можно без особого труда скомпоновать приложение любым удобным для вас способом.

Что представляет собой UWP-приложение

Компания Microsoft определила UWP следующим образом: «Универсальная платформа Windows (Universal Windows Platform, UWP) является специальной платформой для создания приложений под Windows 10. Приложения для этой платформы могутразрабатываться с применением единого набора API, единого прикладного пакета. и единого хранилища, доступного для всех устройств, работающих под Windows 10 (персонального компьютера, планшета, смартфона, Xbox, HoloLens, Surface Hub и многого другого). С ней легче поддерживаются несколько размеров экрана и различные модели взаимодействия с пользователем, будь то сенсорная панель, мышь или клавиатура, игровое устройство управления или перьевой ввод. В основу UWP-приложений положена идея того, что у пользователей должно быть одинаковое восприятие при работе на всех имеющихся у них устройствах и что им захочется воспользоваться самым удобным или самым продуктивным из устройств применительно к решаемой задаче»[1].

В этом разделе мы не собираемся изучать UWP как таковую. Предполагается, что базовые знания о создании UWP-приложений у вас уже есть[2].

7.2.1. Компоновка UWP

Точка входа в UWP-приложения определяется в его классе `App`. Как и в случае с большинством остальных классов в UWP, этот класс разбит на два файла: `App.xaml` и `App.xaml.cs`. Разберемся, что происходит при запуске приложения в `App.xaml.cs`.

ПРИМЕЧАНИЕ

Код этого примера доступен в прилагаемом к книге сборнике исходного кода.

Когда в Visual Studio создается новый UWP-проект, в файле `App.xaml.cs` определяется метод `OnLaunched`, в котором задается страница, выводимая на экран при запуске приложения; в данном случае это `MainPage` (листинг 7.3).

---

[1] Источник: https://mng.bz/DVVg.

[2] Почерпнуть знания о UWP можно в издании: *Chatterjee A.* Building Apps for the Universal Windows Platform. — Apress, 2017.

Листинг 7.3. Метод OnLaunched файла App.xaml.cs

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    ...

    rootFrame.Navigate(typeof(MainPage), e.Arguments);

    ...
}
```

Кроме всего прочего, UWP-проект по умолчанию в Visual Studio приводит пользователя при запуске на основную страницу MainPage, для чего вызывает метод Frame.Navigate

Метод `OnLaunched` похож на метод консольного приложения `Main` — он является для вашего приложения точкой входа. Класс `App` становится корнем композиции приложения. Для компоновки страницы можно воспользоваться как DI-контейнером, так и чистой технологией DI, которая и будет использована в следующем примере.

7.2.2. Пример: подключение полнофункционального клиента управления товарами

В примере предыдущего раздела создавалось консольное торговое приложение для установки обменных курсов. В этом примере будет создаваться UWP-приложение, позволяющее управлять товарами. Снимки экрана показаны на рис. 7.4 и 7.5.

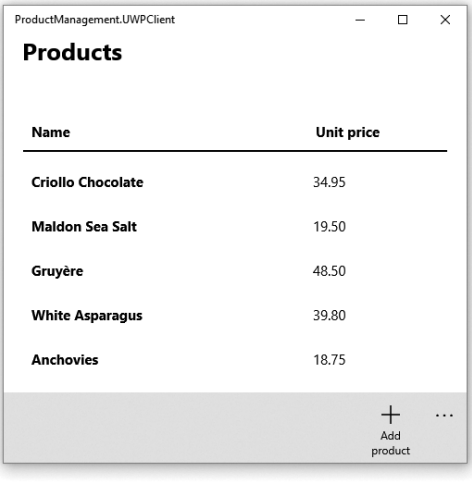


Рис. 7.4. Главная страница приложения управления товарами представляет собой перечень товаров. Товары можно редактировать или удалять, указав на соответствующую строку, или же можно добавить новый товар, указав на элемент управления Add Product

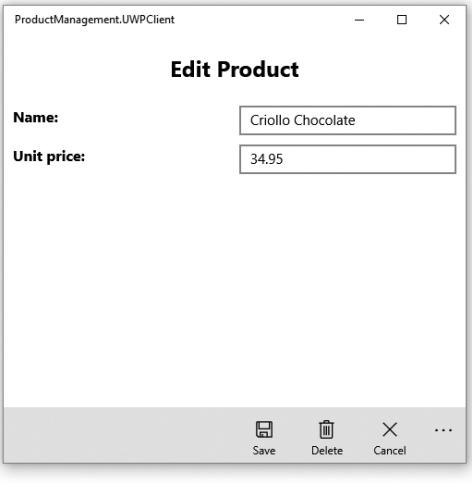


Рис. 7.5. Страница редактирования данных о товаре приложения управления товарами позволяет изменять наименование товара и его цену за единицу, выраженную в долларах. В этом приложении используется панель команд по умолчанию для UWP-платформы

Все приложение реализовано с применением MVVM-подхода и содержит четыре уровня, показанные на рис. 7.6. Часть с самой развитой логикой изолирована от других модулей; речь в данном случае идет о логике представления. Весьма тонкий уровень UWP-клиента занимается практически только определением пользовательского интерфейса и делегированием реализаций другим модулям.

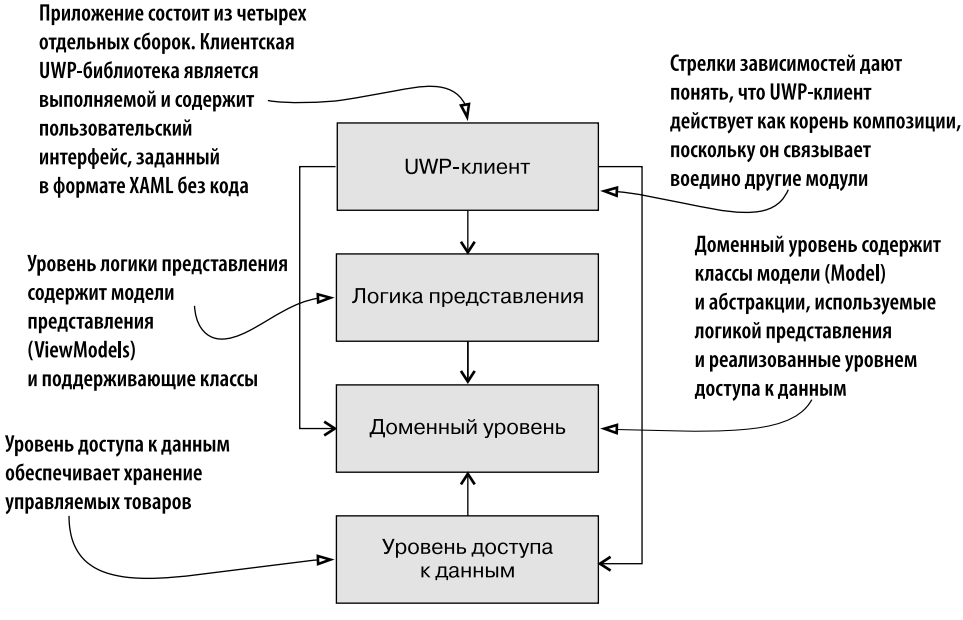


Рис. 7.6. Четыре отдельные сборки клиентского приложения для управления товарами

Схема, показанная на рис. 7.6, похожа на ту, что вы видели в предыдущих главах, но к ней еще добавлен уровень логики представления. Уровень доступа к данным может напрямую подключаться к базе данных, как это уже делалось в веб-приложении электронной торговли, или же он может подключаться к веб-сервису управления товарами. Порядок хранения информации не имеет к логике представления никакого отношения, поэтому вдаваться в подробности в этой главе мы не будем.

При использовании паттерна MVVM модель представления, ViewModel, присваивается принадлежащему странице свойству `DataContext`, а обязанности правильного представления данных при запуске новых моделей представлений или внесении изменений в существующие модели представлений берут на себя меха-

[1] Более подробные сведения о MVVM можно найти в издании: *Smith J.* Patterns: WPF Apps With the Model-View-ViewModel Design Pattern, 2009, https://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

низмы привязки данных и их шаблонной обработки. Но прежде, чем вы сможете создать первую модель представления, нужно будет определить ряд конструкций, позволяющих моделям представлений переходить к другим моделям представлений. Кроме того, чтобы модель представления была проинициализирована данными времени выполнения, требуемыми при демонстрации страницы пользователю, моделям представлений следует позволить выполнить реализацию специализированного интерфейса. Прежде чем перейти к сути приложения, `MainViewModel`, следующий раздел нужно будет посвятить решению именно этих задач.

MVVM

«Модель — представление — модель представления» (Model — View — View Model, MVVM) является паттерном проектирования, для которого особенно хорошо подходит UWP-платформа[1]. Он делит код пользовательского интерфейса на три различные ответственности.

- *Модель, Model, составляет основу для модели приложения*. Зачастую (но не всегда) это доменная модель. Она нередко состоит из так называемых обычных старых объектов CLR (Plain Old CLR Objects, POCO). Обратите внимание, что модель обычно выражается нейтральным по отношению к пользовательскому интерфейсу способом; ее непосредственное представление пользовательским интерфейсом не предполагается, поэтому не выдает никакую характерную для UWP-платформы функциональность.
- *Представление, View, является тем самым пользовательским интерфейсом, на который мы смотрим*. Применительно к среде UWP представление может быть декларативно выражено в формате XAML и использовать для представления данных привязку данных или их шаблонную обработку. Представления могут быть выражены без применения обслуживающего их кода (code-behind), и фактически зачастую предпочтительнее именно этот вариант, поскольку он помогает сконцентрировать представления исключительно на пользовательском интерфейсе.
- *Модель представления, View Model, является своеобразным мостом между представлением и моделью*. Каждая модель представления является классом, выполняющим преобразование и предоставление модели специфичным для технологии способом. В среде UWP это означает, что могут предоставляться списки в виде `System.Collections.ObjectModel.ObservableCollection`, пользовательские действия в виде `System.Windows.Input.ICommand` и т. д.

Роль модели представления в MVVM отличается от роли модели представления в MVC-приложении. В MVC модель представления является объектом данных, не содержащим какого-либо поведения, и он обновляется в коде вашего приложения. А в MVVM модель представления — это компоненты с зависимостями. В вашем UWP-приложении модели представлений будут скомпонованы с использованием технологии DI.

Внедрение зависимостей в MainViewModel

Главная страница, `MainPage`, содержит только XAML-разметку, без какого-либо специализированного обслуживающего кода (custom code-behind). Для отображения данных и обработки команд пользователя используется привязка данных. Чтобы это стало возможным, необходимо присвоить свойству страницы `DataContext` значение `MainViewModel`. Но это является формой внедрения через свойство. А нам хотелось бы вместо него воспользоваться внедрением через конструктор. Чтобы решить эту задачу, мы заменяем конструктор `MainPage` по умолчанию переопределенным конструктором, принимающим в качестве аргумента `MainViewModel`, где внутри конструктора выполняется присваивание этого аргумента свойству `DataContext`:

```
public sealed partial class MainPage : Page
{
    public MainPage(MainViewModel vm)
    {
        this.InitializeComponent();

        this.DataContext = vm;
    }
}
```

`MainViewModel` предоставляет данные, такие как список товаров, а также команды для создания, обновления или удаления товара. Проявление этих функциональных возможностей зависит от сервиса, предоставляющего доступ к каталогу товаров: от абстракции `IProductRepository`. Помимо `IProductRepository`, `MainViewModel` также нуждается в сервисе, которым можно было бы воспользоваться для управления ее оконной среды, подразумевая под этим переход на другие страницы. Еще одна соответствующая зависимость называется `INavigationService`:

```
public interface INavigationService
{
    void NavigateTo<TViewModel>(Action whenDone = null, object model = null)
        where TViewModel : IViewModel;
}
```

ПРИМЕЧАНИЕ

> В C# 4 введены необязательные аргументы метода, позволяющие опускать аргументы для некоторых параметров. В таком случае компилятор C# предоставляет вызову объявленное значение по умолчанию. В предыдущем листинге необязательными являются оба параметра метода. В листинге 7.4 при вызове NavigateTo аргументы иногда опускаются.

Метод `NavigateTo` является обобщенным, поэтому тип модели представления (ViewModel), который нужен ему для перехода, должен быть предоставлен в качестве его аргумента обобщенного типа. Аргументы метода передаются сервисом

переходов в созданную модель представления. Чтобы это заработало, в модели представления должен быть реализован интерфейс `IViewModel`. Поэтому в методе `NavigateTo` задается ограничение обобщенного типа `where TViewModel : IViewModel`[1]. Интерфейс `IViewModel` показан в следующем фрагменте кода:

```
public interface IViewModel
{
    void Initialize(Action whenDone, object model);
}
```

Инициализация ViewModel

Метод `Initialize` содержит точно такие же аргументы, что и метод `INavigationService.NavigateTo`. Сервис переходов вызовет `Initialize` в отношении созданной модели представления. В `model` представлены данные, такие как `Product`, которые должны быть проинициализированы моделью представления. Действие `whenDone`, которое вскоре будет рассмотрено, позволяет создаваемой модели представления получить уведомление, когда пользователь из нее выходит.

Теперь, используя предыдущие определения интерфейса, можно создать модель представления для главной страницы `MainPage`. Класс `MainViewModel` показан в листинге 7.4.

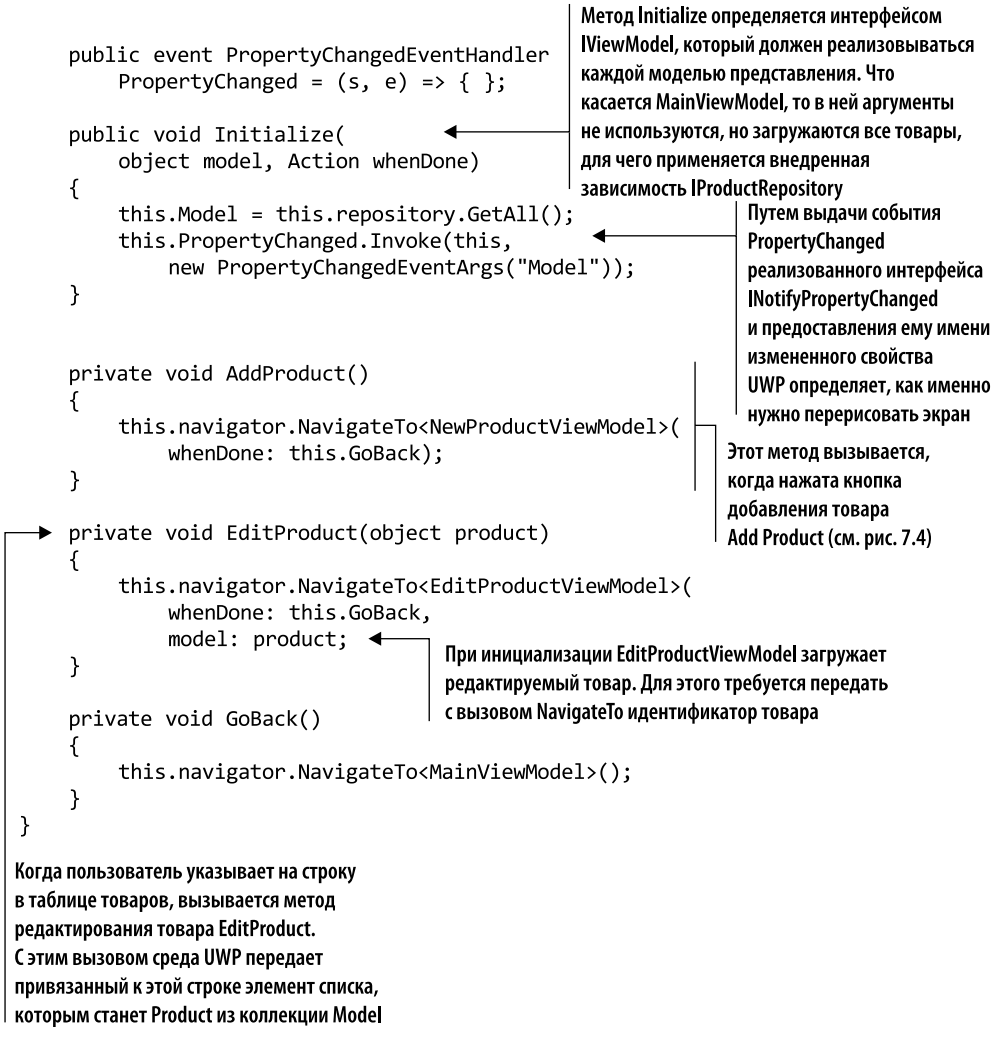
Листинг 7.4. Класс MainViewModel

```
public class MainViewModel : IViewModel,
    INotifyPropertyChanged
{
    private readonly INavigationService navigator;
    private readonly IProductRepository repository;

    public MainViewModel(
        INavigationService navigator,
        IProductRepository repository)
    {
        this.navigator = navigator;
        this.repository = repository;

        this.AddProductCommand =
            new RelayCommand(this.AddProduct);
        this.EditProductCommand =
            new RelayCommand(this.EditProduct);
    }

    public IEnumerable<Product> Model { get; set; }
    public ICommand AddProductCommand { get; }
    public ICommand EditProductCommand { get; }
```

Чтобы получить возможность проинформировать представление о том, что оно должно быть обновлено, в модели представления должен быть реализован интерфейс INotifyPropertyChanged

В модели представления содержатся несколько свойств, к которым привязана XAML-разметка страницы MainPage. Модель представляет собой список товаров, показанный в табличном представлении; свойства ICommand — действия, выполняемые при нажатии соответствующих кнопок

[1] Использование ограничителей обобщенного типа позволяет сузить перечень типов, которыми можно воспользоваться в качестве аргумента обобщенного типа. Проверкой для вас занимается компилятор C#.



Оба командных метода, `AddProduct` и `EditProduct`, предписывают `INavigationService` переход на страницу для соответствующей модели представления. В случае `AddProduct` это соответствует `NewProductViewModel`. Метод `NavigateTo` предоставляется с делегатом, который будет вызываться `NewProductViewModel`, когда пользователь завершит работу с этой страницей. Это приведет к вызову принадлежащего `MainViewModel` метода `GoBack`, который выполнит переход приложения обратно на `MainViewModel`. Чтобы нарисовать завершенную картину, в листинге 7.5 показывается упрощенная версия XAML-определения `MainPage` и то, как XAML-разметка привязана к свойствам `Model`, `EditProductCommand` и `AddProductCommand`, принадлежащим `MainViewModel`.

Листинг 7.5. XAML-разметка MainPage

```
<Page x:Class="Ploeh.Samples.ProductManagement.UWPClient.MainPage"
    xmlns:commands="using:ProductManagement.PresentationLogic.UICommands"
    ...>
    <Grid>
        <Grid.RowDefinitions>
            ...
        </Grid.RowDefinitions>

        <GridView ItemsSource="{Binding Model}"
            commands:ItemClickCommand.Command="{Binding EditProductCommand}"
            IsItemClickEnabled="True">
            <GridView.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="2*"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <StackPanel Grid.Column="0">
                            <TextBlock Text="{Binding Name}" />
                        </StackPanel>
                        <StackPanel Grid.Column="1">
                            <TextBlock Text="{Binding UnitPrice}" />
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </GridView.ItemTemplate>
        </GridView>

        <CommandBar Grid.Row="5" Grid.ColumnSpan="3" Grid.Column="0">
            <AppBarToggleButton Icon="Add" Label="Add product"
                Command="{Binding AddProductCommand}" />
        </CommandBar>
    </Grid>
</Page>
```

ПРИМЕЧАНИЕ

В имеющемся в XAML табличном представлении GridView используется специализированная команда пользовательского интерфейса ItemClickCommand, позволяющая привязывать указания на строки GridView или щелчки на них кнопкой мыши к принадлежащей модели представления команде EditProductCommand. Рассмотрение данной команды пользовательского интерфейса выходит за рамки книги, но специализированная команда доступна в сопровождающем книгу исходном коде, где можно увидеть полную XAML-версию.

Хотя в предыдущей версии XAML использовалось старое расширение разметки Binding, вы в качестве UWP-разработчика можете применить более новое расширение разметки x:Bind, обладающее поддержкой во время компиляции, но требующее в это же время исправления тех типов, которые обычно определены в классе кода, обслуживающего представление. Поскольку имеет место привязка к модели представления, хранящейся в нетипизированном свойстве `DataContext`, поддержка во время компиляции утрачивается, и поэтому возникает потребность в возвращении к расширению разметки Binding[1].

Двумя основными элементами в `MainPage` XAML являются `GridView` и `CommandBar`. Табличное представление `GridView` используется для отображения доступных товаров и их привязки к обоим свойствам: `Model` и `EditProductCommand`; его шаблон данных `DataTemplate` привязывается к свойствам `Name` и `UnitPrice` элементов `Product`, принадлежащих `Model`. Панель команд `CommandBar` отображает общую ленту с операциями, доступными для вызова пользователем. `CommandBar` привязывается к свойству `AddProductCommand`. Теперь, имея в своем распоряжении определения `MainViewModel` и `MainPage`, можно приступать к сборке приложения.

Сборка MainViewModel

Перед сборкой `MainViewModel` посмотрим на все классы, задействованные в графе зависимостей. На рис. 7.7 показан граф для приложения, начинающийся с `MainPage`. Теперь, когда определены все строительные блоки приложения, можно заняться их компоновкой. Для этого нужно создать как `MainViewModel`, так и `MainPage`, а затем внедрить `ViewModel` в конструктор `MainPage`. Для сборки `MainViewModel` нужно скомпоновать ее с ее зависимостями:

```
IViewModel vm = new MainViewModel(navigationService, productRepository);
Page view = new MainPage(vm);
```

Как уже было показано в листинге 7.3, шаблон Visual Studio по умолчанию вызывает `Frame.Navigate(Type)`. Метод `Navigate` создает от вашего имени новый экземпляр страницы `Page` и показывает эту страницу пользователю. Способа предоставления экземпляра `Page` методу `Navigate` не существует, но это обстоятельство можно обойти, самостоятельно назначив страницу, созданную для свойства `Content` основного кадра приложения `Frame`:

```
var frame = (Frame)Window.Current.Content;
frame.Content = view;
```

Поскольку это важные части для сборки приложения воедино, это именно то, что будет делаться вами в корне композиции.

---

[1] См. MSDN-статью Data Binding in Depth, https://mng.bz/mz9P.

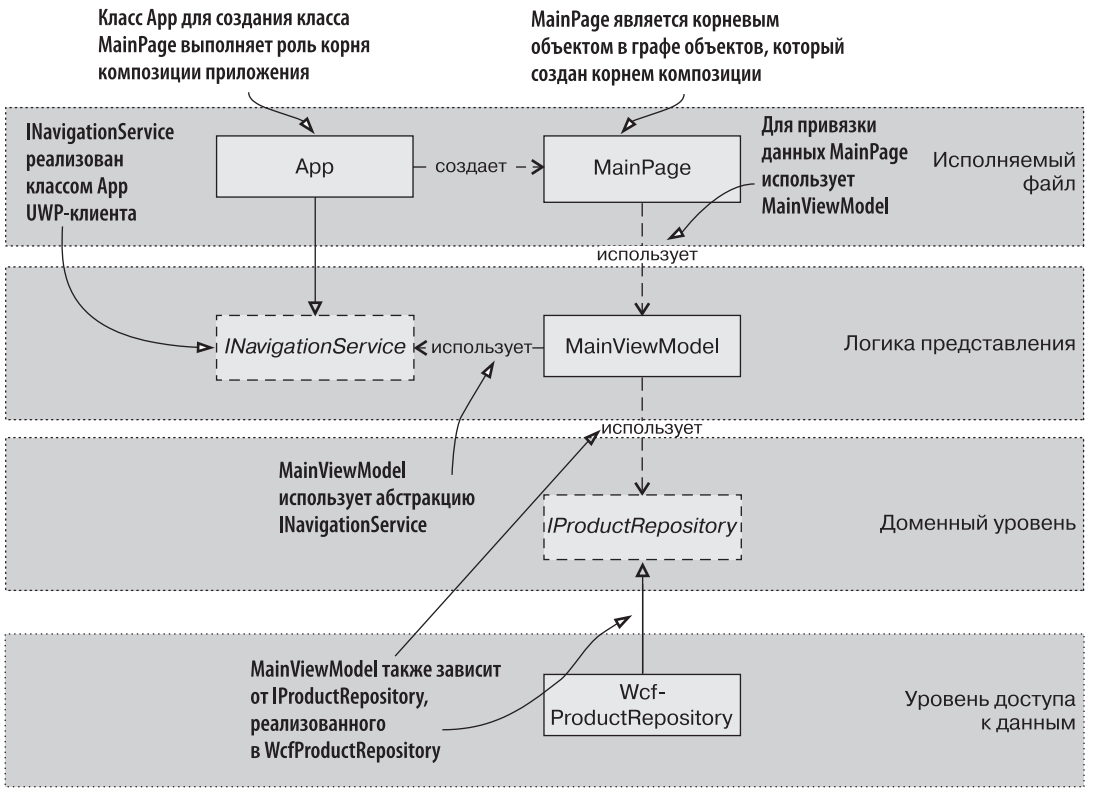


Рис. 7.7. Граф зависимостей полнофункционального клиента управления товарами

7.2.3. Реализация корня композиции UWP-приложения

Для создания корня композиции существует множество способов. Для рассматриваемого примера, чтобы он был относительно компактным, был выбран способ помещения как логики переходов, так и конструкции пары View/ViewModel в файл `App.xaml.cs`. Корень композиции приложения показан на рис. 7.8.

ПРИМЕЧАНИЕ

Важной частью корня композиции является компоновщик (Composer). Это унифицированное понятие, позволяющее сослаться на любой объект или метод, выполняющий компоновку зависимостей. Его более подробное описание вы найдете в следующей главе.

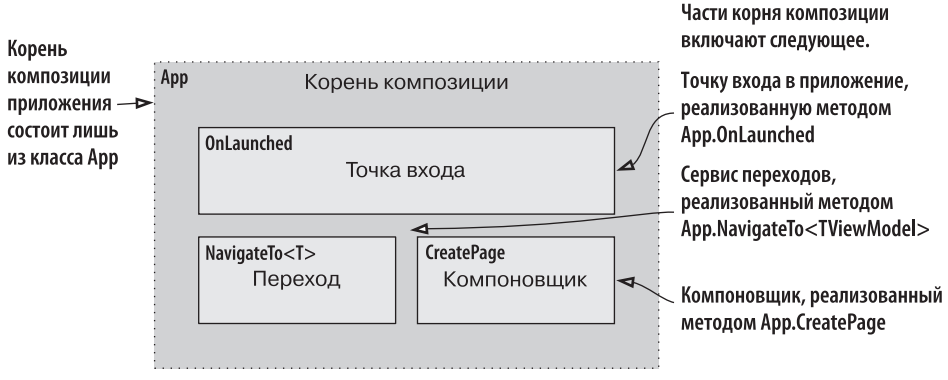


Рис. 7.8. Корень композиции полнофункционального клиента управления товарами

Корень композиции в действии показан в листинге 7.6.

Листинг 7.6. Класс App приложения управления товарами, содержащий корень композиции

```
public sealed partial class App : Application, INavigationService
{
    protected override void OnLaunched(
        LaunchActivatedEventArgs e)
    {
        if (Window.Current.Content == null)
        {
            Window.Current.Content = new Frame();
            Window.Current.Activate();
            this.NavigateTo<MainViewModel>(null, null);
        }
    }

    public void NavigateTo<TViewModel>(
        Action whenDone, object model)
        where TViewModel : IViewModel
    {
```

Точка входа в приложение

Создает при запуске новый кадр Frame и активирует его

Создает новую пару MainPage и MainViewModel и показывает MainPage пользователю

```
        var page = this.CreatePage(typeof(TViewModel));
        var viewModel = (IViewModel)page.DataContext;

        viewModel.Initialize(whenDone, model);

        var frame = (Frame)Window.Current.Content;
        frame.Content = page;
    }

    private Page CreatePage(Type vmType)
    {
        var repository = new WcfProductRepository();

        if (vmType == typeof(MainViewModel))
        {
            return new MainPage(
                new MainViewModel(this, repository));
        }
        else if (vmType == typeof(EditProductViewModel))
        {
            return new EditProductPage(
                new EditProductViewModel(repository));
        }
        else if (vmType == typeof(NewProductViewModel))
        {
            return new NewProductPage(
                new NewProductViewModel(repository));
        }
        else
        {
            throw new Exception("Unknown view model.");
        }
    ...
}
```

NavigateTo запускает компоновку и инициализацию, обеспечивая вывод на экран созданной страницы

CreatePage создает требуемую пару View/ViewModel путем компоновки модели представления и ее внедрения в конструктор представления. Следует заметить, что MainViewModel зависит от интерфейса Inavigation-Service, но, поскольку он реализуется классом App, он может быть внедрен в MainViewModel напрямую с помощью ключевого слова this

Фабричный метод `CreatePage` похож на примеры корня композиции, рассмотренные в разделе 4.1. Для создания надлежащей пары он, соответственно, состоит из большого списка инструкций `else if`.

ПРИМЕЧАНИЕ

> Чтобы не усложнять пример, метод CreatePage, показанный в листинге 7.6, создает новый экземпляр Page при каждом вызове. Особой потребности в этом нет, но так его проще реализовать.

Среда UWP предлагает для корня композиции весьма простое место. Все, что нужно сделать, — это удалить вызов `Frame.Navigate(Type)` из `OnLaunched` и установить в качестве значения `Frame.Content` самостоятельно созданный объект класса `Page`, составляемый с использованием модели представления и ее зависимостей.

В большинстве других сред присутствует более высокий уровень инверсии управления, что означает, что прежде надо суметь идентифицировать надлежащие точки расширения для сборки нужного графа объектов. Одной из таких сред является ASP.NET Core MVC.

7.3. Компоновка приложений среды ASP.NET Core MVC

Среда ASP.NET Core MVC была сконструирована для поддержки DI. Она поставляется со своим собственным механизмом компоновки, которым можно воспользоваться для создания своих собственных компонентов; хотя, как вы увидите, она не принуждает к использованию DI-контейнера для компонентов вашего приложения. Можно воспользоваться чистой технологией DI или же любым DI-контейнером, который вам нравится[1].

Из этого раздела вы узнаете о порядке использования основной точки расширения среды ASP.NET Core MVC, что позволит вам подключать свою логику для компоновки классов контроллеров с их зависимостями. Здесь среда ASP.NET Core MVC будет рассматриваться с позиции компоновки объектов в соответствии с технологией DI. Но в одну главу не вместить весь оставшийся неохваченным большой объем информации, касающейся разработки приложений в среде ASP.NET Core. Если есть желание получить дополнительные сведения по этой теме, обратите внимание на издание: *Lock A.* ASP.NET Core in Action. — Manning, 2018. После этого будет рассмотрен способ подключения прослоек, требующих зависимостей.

ПРИМЕЧАНИЕ

> В «классической» ASP.NET компания Microsoft разработала отдельные среды для MVC и Web API. В случае с ASP.NET Core компания Microsoft создала единую унифицированную среду, справляющуюся как с MVC, так и с Web API под эгидой ASP.NET Core MVC. С позиции технологии DI подключение Web API идентично MVC-приложению в среде ASP.NET Core. Это означает, что данный раздел также применим для создания в среде .NET Core интерфейсов Web API.

Как это всегда и случается с практикой использования технологии DI в прикладной среде, ключом к ее применению является нахождение надлежащей точки расширения. В среде ASP.NET Core MVC такой точкой является интерфейс под названием `IControllerActivator`. То, как он вписывается в среду, показано на рис. 7.9.

Контроллеры являются центральным звеном среды ASP.NET Core MVC. Ими обрабатываются запросы и определяется порядок ответа. Если нужно запросить базу

---

[1] Разработчики среды ASP.NET Core определили абстракцию над DI-контейнерами с намерениями разрешить полную замену встроенной реализации DI-контейнерами сторонних производителей. По нашему мнению, это стало серьезной ошибкой, вызывающей разочарование у тех, кто поддерживает DI-контейнеры. Из-за дефицита времени и книжного пространства размышления на эту тему выходят за рамки данной книги. Но мы советуем все-таки сохранить встроенный DI-контейнер, даже если при создании компонентов вашего приложения применяется DI-контейнер стороннего производителя. Более подробную информацию по этому вопросу можно найти по адресу https://simpleinjector.org/blog/2016/06/.

данных, проверить и сохранить входящие данные, вызвать доменную логику и т. д., то все соответствующие действия инициализируются из контроллера. Контроллер не может все это делать самостоятельно, он просто делегирует работу соответствующим зависимостям. Здесь на первый план выходит применение технологии DI.

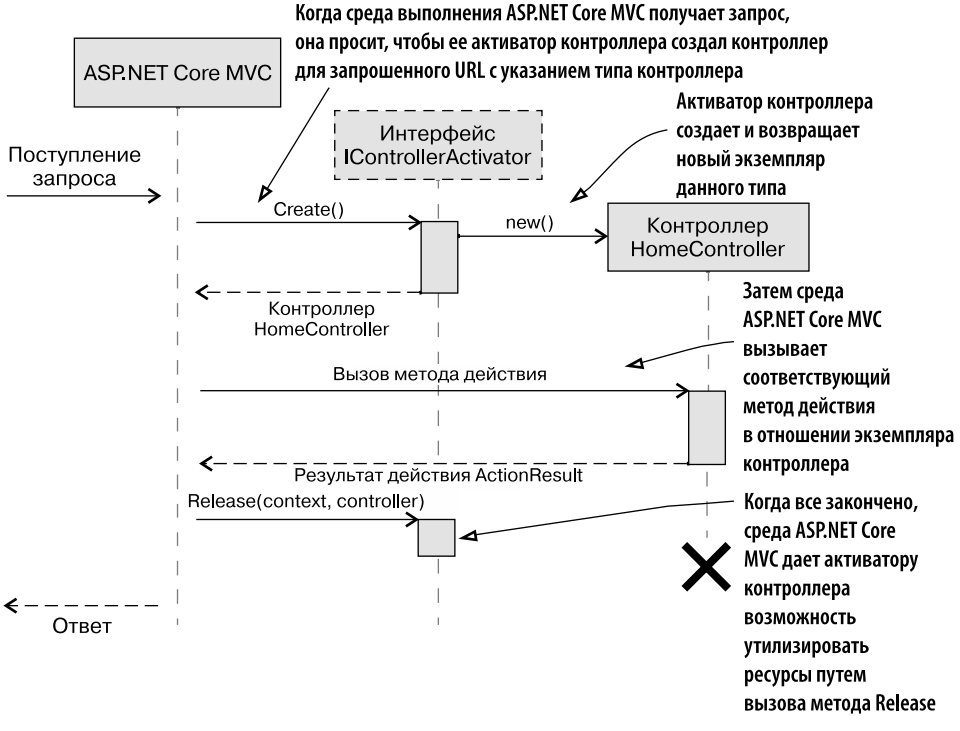


Рис. 7.9. Среда ASP.NET Core MVC требует конвейер

Вам нужна возможность предоставления зависимостей заданному классу контроллера, в идеале — путем внедрения через конструктор. Такая возможность предоставляется специализированным интерфейсом `IControllerActivator`.

7.3.1. Создание специализированного активатора контроллера

Создание специализированного активатора контроллера не представляет особого труда. Оно требует от вас реализации интерфейса `IControllerActivator`:

```
public interface IControllerActivator
{
    object Create(ControllerContext context);
    void Release(ControllerContext context, object controller);
}
```

Метод `Create` предоставляет `ControllerContext`, содержащий такую информацию, как `HttpContext` и тип контроллера. Этот метод предоставляет возможность сборки всех требуемых зависимостей и их предоставления контроллеру до возвращения экземпляра. Соответствующий пример вскоре будет показан.

Если созданы какие-либо ресурсы, требующие последующей явной утилизации, этого можно добиться при вызове метода `Release`. Подробнее высвобождение компонентов будет рассмотрено в следующей главе. Более практичным способом обеспечения утилизации зависимостей является их добавление в список одноразовых объектов запроса путем использования метода `HttpContext.Response.RegisterForDispose`. Хотя реализация специализированного активатора контроллера — задача не из простых, он не будет использоваться, пока мы не сообщим о нем среде ASP.NET Core MVC.

Использование специализированного активатора контроллера в среде ASP.NET Core

Специализированный активатор контроллера может быть добавлен в качестве части последовательности запуска приложения — обычно это делается в классе `Startup`. Для этого используется вызов `AddSingleton<IControllerActivator>` в отношении экземпляра `IServiceCollection`. В листинге 7.7 показан класс `Startup` из учебного приложения электронной торговли.

Листинг 7.7. Класс Startup из приложения электронной торговли

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        this.Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        var controllerActivator = new CommerceControllerActivator(
            Configuration.GetConnectionString("CommerceConnectionString"));

        services.AddSingleton<IControllerActivator>(controllerActivator);
}

    public void Configure(ApplicationBuilder app, IHostingEnvironment env)
    {
        ...
    }
}
```

Код, показанный в листинге, создает новый экземпляр специализированного активатора контроллера `CommerceControllerActivator`. Его добавление в список известных сервисов с использованием `AddSingleton` гарантирует, что создание контроллеров перехватывается вашим собственным активатором контроллеров. Не исключено, что этот код покажется смутно знакомым, ведь дело в том, что нечто подобное вы уже видели в подразделе 4.1.3. Тогда мы обещали показать вам в главе 7, как реализуется собственный активатор контроллера, и вот мы сюда и пришли.

Пример: реализация CommerceControllerActivator

Можно вспомнить, что рассматриваемое в главах 2 и 3 учебное приложение электронной торговли предоставляло посетителю веб-сайта список товаров и цен на них. В разделе 6.2 была добавлена функция, позволяющая пользователям вычислять маршрут между двумя точками на местности. Было показано несколько фрагментов кода корня композиции, но полностью его пример так и не был продемонстрирован. Вместе с классом `Startup` из листинга 7.7 класс `CommerceControllerActivator` из листинга 7.8 показывает завершенный корень композиции.

Чтобы собрать контроллеры с их зависимостями, учебному приложению электронной торговли нужен собственный активатор контроллера. Хотя весь граф объектов выглядит значительно глубже, с позиции самих контроллеров объединение всех непосредственных зависимостей составляет не более двух элементов (рис. 7.10).

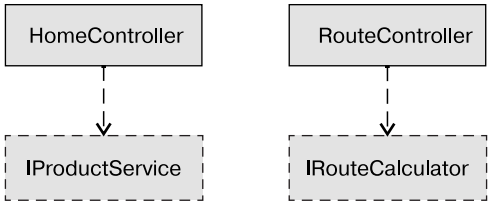


Рис. 7.10. Два контроллера в учебном приложении с их зависимостями

В листинге 7.8 показан `CommerceControllerActivator`, выполняющий компоновку как `HomeController`, так и `RouteController` с их зависимостями.

Листинг 7.8. Создание контроллеров с использованием собственного активатора контроллера

```
public class CommerceControllerActivator : IControllerActivator
{
    private readonly string connectionString;

    public CommerceControllerActivator(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public object Create(ControllerContext context)
    {
        Type type = context.ActionDescriptor
            .ControllerTypeInfo.AsType();
```

Получение типа создаваемого контроллера из ControllerContext

```
        if (type == typeof(HomeController))
        {
            return this.CreateHomeController();
        }
        else if (type == typeof(RouteController))
        {
            return this.CreateRouteController();
        }
        else
        {
            throw new Exception("Unknown controller " + type.Name);
        }
    }

    private HomeController CreateHomeController()
    {
        return new HomeController(
            new ProductService(
                new SqlProductRepository(
                    new CommerceContext(
                        this.connectionString)),
                new AspNetUserContextAdapter()));
    }

    private RouteController CreateRouteController()
    {
        var routeAlgorithms = ...;
        return new RouteController(
            new RouteCalculator(routeAlgorithms));
    }

    public void Release(
        ControllerContext context, object controller)
    {
    }
}
```

Возвращение соответствующего контроллера на основе заданного типа при условии, что запрошенным типом является либо HomeController, либо RouteController

Явная сборка контроллеров с требуемыми им зависимостями и их возвращение. В обоих типах используется внедрение через конструктор, поэтому зависимости предоставляются через их конструкторы

Метод Release мы пока оставляем пустым, так как еще вернемся к нему в разделе 8.2

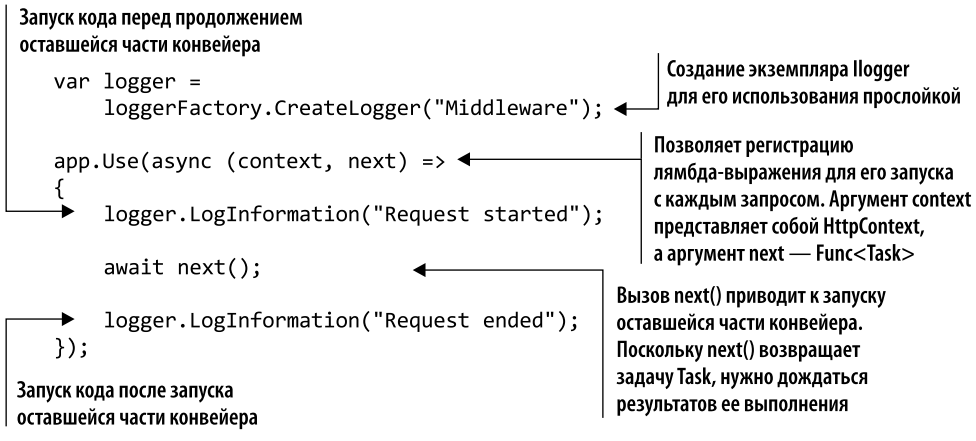
ПРИМЕЧАНИЕ

Как уже ранее утверждалось, в среде ASP.NET Core содержится ее собственный встроенный DI-контейнер (который будет рассматриваться в главе 15). Для регистрации ваших зависимостей можно воспользоваться и этим встроенным DI-контейнером. В главе 12 будет рассмотрен способ принятия решения о том, что следует задействовать: чистую технологию DI или DI-контейнер. В этой части книги наш выбор был сделан в пользу чистой технологии DI.

Когда экземпляр `CommerceControllerActivator` зарегистрирован в `Startup`, он, как и положено, создает все необходимые контроллеры с нужными им зависимостями. Все не относящиеся к контроллерам другие компоненты, которым нередко требуется использование технологии DI, называются в среде ASP.NET Core компонентами-*прослойками*, или *промежуточным ПО* (middleware).

7.3.2. Создание собственных компонентов-прослоек с использованием чистой технологии DI

Среда ASP.NET Core делает относительно простым подключение дополнительного поведения в конвейере запросов. Подобное поведение может повлиять на запрос и ответ. Такие расширения конвейера запросов называются в среде In ASP.NET Core прослойками. Обычным использованием подключаемой прослойки к конвейеру запросов является применение метода расширения `Use`:



ПРИМЕЧАНИЕ

> Здесь впервые в книге показан пример кода, в котором используются ключевые слова C# 5.0 async и await. Вероятно, разработчики, программирующие на C#, уже сталкивались с примерами асинхронного программирования, поскольку вся среда ASP.NET Core выстроена вокруг модели асинхронного программирования. Но обсуждение вопросов асинхронного программирования выходит за рамки данной книги[1]. К счастью, когда дело касается применения технологии DI, никаких вопросов насчет асинхронного программирования не возникает, так как построение графа объектов неизменно является скоротечным процессом и никогда не зависит от какого-либо ввода-вывода, в силу чего всегда должно выполняться в синхронном режиме.

Но часто бывает так, что до и после запуска основной логики обработки запросов нужно выполнить какую-либо дополнительную работу. Поэтому может появиться желание выделить логику прослойки в ее собственный класс. Подобное желание предотвратит засорение вашего класса `Startup` и предоставит возможность применить по отношению к этой логике модульное тестирование. Тело предыдущего

[1] Руководство по приемам использования асинхронного программирования и по применению ключевых слов async и await можно найти по адресу https://docs.microsoft.com/en-us/dotnet/csharp/async.

лямбда-выражения `Use` можно извлечь и поместить в метод `Invoke` вновь создаваемого класса `LoggingMiddleware`:

```
public class LoggingMiddleware
{
    private readonly ILogger logger;

    public LoggingMiddleware(ILogger logger)
    {
        this.logger = logger;
    }

    public async Task Invoke(
        HttpContext context, Func<Task> next)
    {
        this.logger.LogInformation("Request started");
        await next();
        this.logger.LogInformation("Request ended");
    }
}
```

Конструктор принимает требуемую зависимость

Метод Invoke содержит логику, которая ранее предоставлялась встроенной в строку кода

После того как логика прослойки переместилась в класс `LoggingMiddleware`, конфигурация `Startup` может быть минимизирована и сведена к следующему коду:

```
var logger = loggerFactory.CreateLogger("Middleware");

app.Use(async (context, next) =>
{
    var middleware = new LoggingMiddleware(logger);

    await middleware.Invoke(context, next);
});
```

Создание нового компонента middleware с его зависимостями

Вызов middleware путем передачи аргументов context и next

ПРИМЕЧАНИЕ

Когда граф объектов созданного компонента middleware становится сложнее, может возникнуть потребность в перемещении создания компонента в такое место, где выполняется компоновка других компонентов. В нашем предыдущем примере таким местом может стать CommerceControllerActivator. Но создание соответствующего кода мы в качестве упражнения возложим на наших читателей.

Отличительной особенностью среды ASP.NET Core MVC является ее разработка с прицелом на использование технологии DI, поэтому вам для применения DI в приложении нужно по большей части знать об использовании только одной точки расширения. Компоновка объектов является одним из трех важных измерений DI (другие — управление временем жизни и перехват).

В этой главе были показаны приемы компоновки приложений из модулей со слабой связанностью в нескольких различных средах. Некоторые среды действи-

тельно облегчают проведение такой компоновки. При написании консольных приложений и Windows-клиентов (таких как UWP) все происходящее в точке входа в приложение в той или иной степени находится под вашим контролем. Это дает вам четко выраженный и легко внедряемый корень композиции. В других средах, таких как ASP.NET Core, вам приходится прикладывать больше усилий, но они все же предоставляют швы, которыми можно воспользоваться для определения порядка компоновки приложения. Среда ASP.NET Core была разработана с прицелом на применение технологии DI, поэтому компоновка приложения сведена к реализации собственного интерфейса `IControllerActivator` и к добавлению его к среде.

Без компоновки объектов не бывает внедрения зависимостей, но, возможно, вы еще не полностью осознали значение времени жизни объекта, когда создание объектов перемещается из классов-потребителей. Вам может показаться вполне нормальным то, что внешний вызывающий объект (зачастую это DI-контейнер) создает новый экземпляр зависимостей — но когда внедренные экземпляры высвобождаются? А что, если внешний вызывающий объект не создает всякий раз новые экземпляры, вручая вместо них уже существующий экземпляр? Ответы на эти вопросы и станут темой следующей главы.

Резюме

- Компоновка объектов является действием по выстраиванию иерархии взаимосвязанных компонентов и производится в корне композиции.
- Корень композиции должен выполнять всего четыре действия: загрузку конфигурационных значений, построение графа объектов, вызов нужной функции и высвобождение графа объектов.
- Зависимость от конфигурационных файлов должна быть только у корня композиции, потому что для библиотек более гибким решением будет возможность их императивной настройки вызывающими их объектами.
- Нужно отделять загрузку конфигурационных значений от методов, выполняющих компоновку объектов. Это позволит протестировать компоновку объектов при отсутствии конфигурационного файла.
- Паттерн «Модель — представление — модель представления» (Model — View — View Model, MVVM) является конструкцией, в которой модель представления является своеобразным мостом между представлением и моделью. Каждая модель представления является классом, выполняющим преобразование и предоставление модели специфичным для технологии способом. В MVVM модели представлений являются компонентами приложения, компоновка которых будет выполняться с использованием технологии DI.
- В консольном приложении в качестве корня композиции вполне подходит класс `Program`.
- В UWP-приложении в качестве корня композиции вполне подходит класс `App`, а его метод `OnLaunched` является основной точкой входа.

- В приложении среды ASP.NET Core MVC правильной точкой расширения, подходящей для подключения компоновки объектов, является интерфейс `IControllerActivator`.
- Практическим способом обеспечения ликвидации зависимости в среде ASP.NET Core является использование метода `HttpContext.Response.RegisterForDispose` с целью добавления зависимостей в список запрошенных объектов, подлежащих ликвидации.
- Прослойки могут добавляться к среде ASP.NET Core путем регистрации функции в конвейере, реализующем небольшую часть корня композиции. Это приведет к включению прослойки в композицию и к ее запуску.

8 Время жизни объектов

В этой главе

- Управление временем жизни зависимости.
- Работа с ликвидируемыми (disposable) зависимостями.
- Использование жизненных циклов Singleton, Transient и Scoped.
- Предотвращение неудачного выбора жизненного цикла или его исправление.

Со временем качество основной массы продуктов и напитков изменяется, но происходит это по-разному. Кому-то сыр грюйер, созревавший 12 месяцев, нравится больше, чем шестимесячный, а вот Марк любит, чтобы его спаржа была посвежее обоих этих сыров[1]. Зачастую получить продукт нужной выдержки нетрудно, но бывает и так, что с этим возникают сложности. Это особенно ощутимо, когда дело касается вина (рис. 8.1).

Вино со временем становится лучше, пока в какой-то момент не перестоит и не станет почти безвкусным. Это зависит от многих факторов, включая место происхождения и сезон сбора винограда. Винная тема нас интересует, но вряд ли мы

[1] Стивен спаржу не любит, какой бы свежести она ни была, — он предпочитает виски самой большой выдержки.

сможем предсказать, когда вино достигнет самого пика своего вкуса. Поэтому мы полагаемся на мнение авторитетов: дома читаем соответствующие книги, а в ресторане ориентируемся на то, что скажет сомелье. Он разбирается в винах лучше нас, поэтому мы с легкостью соглашаемся с его выбором.

Если вы изучили предыдущие главы, то уже в курсе, что избавление от контроля — это ключевая концепция DI. Исходным здесь является принцип инверсии управления, предусматривающий делегирование полномочий по контролю над зависимостями третьей стороне, но, кроме того, подразумевает нечто большее, чем простое разрешение кому-то другому выбирать реализацию необходимой абстракции. Когда предоставление зависимости доверено компоновщику (Composer), следует допустить, что управлять временем жизни этой зависимости вы не можете.

Рис. 8.1. Вино, сыр и спаржа. Возможно, сочетание этих продуктов покажется несколько странным, но их выдержка сильно влияет на общие качества

ОПРЕДЕЛЕНИЕ

> Компоновщик — обобщенное понятие, отсылающее к любому объекту или методу, выполняющему компоновку зависимостей. Он является важной частью корня композиции. В качестве компоновщика часто применяется DI-контейнер, но также им может быть любой метод, самостоятельно выстраивающий граф объектов (при использовании чистой технологии DI).

Сомелье хорошо знает содержимое винного погреба ресторана и способен принять куда более обоснованное решение, чем мы. Точно так же, как мы доверяем ему выбор напитков, мы должны верить тому, что компоновщик справится с управлением временем жизни зависимостей эффективнее, чем их потребитель. Компоновка составляющих и управление ими — это все, за что он отвечает.

В этой главе нам предстоит изучить управление временем жизни объектов. В этом важно разбираться, поскольку точно так же, как потребление вина не в том возрасте (как вашем собственном, так и вина) может иметь крайне негативные последствия, неправильная настройка времени жизни зависимостей может повлечь за собой снижение производительности. Хуже того, в области управления временем жизни можно получить некий эквивалент испорченных продуктов — утечку

ресурсов. Усвоение принципов правильного управления жизненными циклами компонентов позволит принимать обоснованные решения и правильно настраивать ваши приложения.

Сначала вашему вниманию будет представлено введение в управление временем жизни зависимостей, а затем мы рассмотрим ликвидируемые зависимости. Первая часть главы призвана дать вам исходную информацию и познакомить с основополагающими принципами, необходимыми для принятия обоснованных решений о жизненных циклах ваших приложений, их области действия и конфигурации.

Затем будут рассмотрены различные стратегии выбора времени жизни. Эта часть главы — своеобразный каталог доступных жизненных циклов. В большинстве случаев один из этих базовых паттернов жизненного цикла весьма удачно впишется в решение конкретной задачи, поэтому предварительное ознакомление с ними поможет справиться со многими затруднительными ситуациями.

ОПРЕДЕЛЕНИЕ

Под жизненным циклом (lifestyle) понимается формализованный способ описания предполагаемой продолжительности жизни зависимости.

В конце главы будут названы вредные привычки, или антипаттерны, относящиеся к управлению временем жизни. К этому моменту у вас должно сложиться внятное представление о том, чего можно, а чего нельзя добиться за счет управления временем жизни и использования обычного жизненного цикла. Для начала посмотрим на время жизни объектов и на то, какое отношение оно имеет к технологии DI.

8.1. Управление временем жизни зависимостей

До сих пор разговор шел в основном о том, как технология DI позволяет выполнять компоновку зависимостей. В предыдущей главе этот вопрос был исследован весьма подробно, но, как упоминалось в разделе 1.4, компоновка объектов является всего лишь одним из аспектов DI. Еще одним является управление временем жизни объектов.

Когда в этой книге впервые было сказано, что в сферу влияния технологии DI входит управление временем жизни, мы не удосужились объяснить всю глубину связи между компоновкой объектов и временем их жизни. И вот наконец-то добрались до этой темы, не представляющей особой сложности, так что приступим к ней!

В этом разделе рассмотрим управление временем жизни объектов применительно к зависимостям. На основе общего случая компоновки объектов изучим ее влияние на продолжительность жизни зависимостей. Но сначала нам предстоит выяснить, почему компоновка объектов влияет на управление временем их жизни.

8.1.1. Введение в управление временем жизни

Нужно избавляться от стремления контролировать зависимости, а вместо этого запрашивать их с помощью конструктора или других паттернов DI. Соглашаясь с этим, мы должны полностью забыть о контроле. Чтобы разобраться в причинах такого положения вещей, рассмотрим данный вопрос поэтапно. Начнем с того, что означает стандартный жизненный цикл .NET-объектов для зависимостей. Скорее всего, вы уже знаете это, но все же потерпите — через полстраницы мы изложим все обстоятельства.

Простой жизненный цикл зависимостей

Известно, что в технологию DI заложено разрешение поручить организацию обслуживания необходимых вам зависимостей стороннему модулю (обычно корню композиции). Это также означает необходимость разрешить ему управлять временем жизни зависимостей. Проще всего разобраться в этом, создав объект. Вот как выглядит слегка перестроенный фрагмент кода корня композиции из учебного приложения электронной торговли (полный пример см. в листинге 7.8):

```
var productRepository =
    new SqlProductRepository(
        new CommerceContext(connectionString));

var productService =
    new ProductService(
        productRepository,
        userContext);
```

Надеемся, вполне очевидно, что класс `ProductService` не управляет созданием `productRepository`. В данном случае `SqlProductRepository`, вероятно, будет создан в ту же миллисекунду, но в качестве мысленного эксперимента мы можем вставить между этими двумя строками кода вызов `Thread.Sleep`, чтобы продемонстрировать возможность их произвольного разделения по времени. Казалось бы, это довольно странный шаг, но дело в том, что не все объекты графа зависимостей должны создаваться одновременно.

Потребители не контролируют создание своих зависимостей, а как обстоят дела с их уничтожением? Вообще-то момент уничтожения объектов в среде .NET контролю не поддается. Незадействованные объекты удаляет сборщик мусора, но если используемые объекты не являются ликвидируемыми, явно уничтожить объект невозможно.

ПРИМЕЧАНИЕ

Термин «ликвидируемый» (disposable) применяется в качестве краткого именования экземпляров объектов, относящихся к типам, в которых реализован интерфейс IDisposable.

Объекты могут быть собраны как мусор, когда они выходят из области видимости. И наоборот, они существуют до тех пор, пока у какого-нибудь другого объекта есть ссылка на них. Хотя потребитель не способен уничтожить объект явно — оставляя это на откуп сборщику мусора, — он может поддерживать его существование, сохраняя ссылку на него. Именно так и происходит при внедрении через конструктор, поскольку зависимость сохраняется в приватном поле:

```
public class HomeController
{
    private readonly IProductService service;

    public HomeController(IProductService service)
    {
        this.service = service;
    }
}
```

Вставка зависимости в конструктор класса

Сохранение ссылки на зависимость в приватном поле по крайней мере до тех пор, пока существует потребляющий ее экземпляр HomeController

Следовательно, как только потребитель покинет область видимости, то же самое произойдет и с зависимостью. Но, даже когда потребитель выходит из области видимости, зависимость может существовать, если ссылка на нее имеется у других объектов. В противном случае она будет удалена как мусор. Для опытных .NET-разработчиков это, скорее всего, не новость, но вот то, о чем станем говорить в дальнейшем, может вызвать у них определенный интерес.

Усложнение жизненного цикла зависимостей

До сих пор в анализе жизненного цикла не было ничего необычного, но теперь его можно немного усложнить. Что получится, если одна и та же зависимость понадобится сразу нескольким потребителям? Один из вариантов — предоставление каждому потребителю его собственного экземпляра (рис. 8.2).

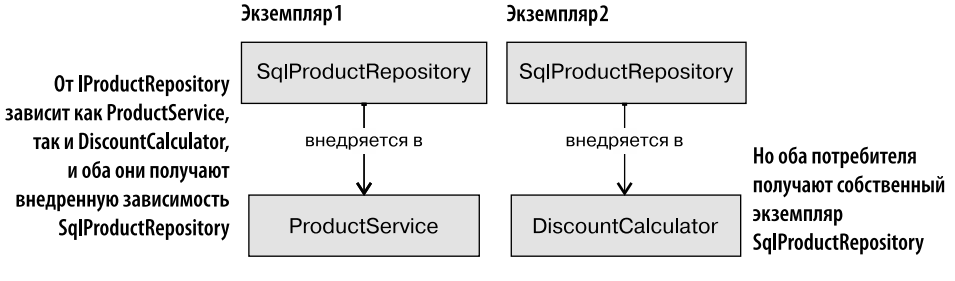


Рис. 8.2. Компоновка нескольких индивидуальных экземпляров зависимости

В листинге 8.1 составлена композиция из нескольких потребителей нескольких экземпляров зависимости, что соответствует показанному на рис. 8.2.

Листинг 8.1. Создание композиции из нескольких экземпляров зависимости

```
var repository1 = new SqlProductRepository(connString);
var repository2 = new SqlProductRepository(connString);

var productService = new ProductService(repository1);

var calculator = new DiscountCalculator(repository2);
```

Экземпляр IProductRepository требуется сразу двум потребителям, но с помощью одинаковых строк кода подключаются два отдельных экземпляра

Теперь repository1 можно передать новому экземпляру ProductService

А repository2 передается новому экземпляру DiscountCalculator

Если разбираться с жизненными циклами каждого из хранилищ `Repository` из листинга 8.1, то выяснится, что ничего, по сравнению с ранее рассмотренным корнем композиции учебного приложения электронной торговли, не изменилось. Каждая из зависимостей покидает область видимости и подпадает под сборку мусора, как только из области видимости уйдут ее потребители. Это может происходить в разное время, но ситуация практически ничем не отличается от ранее рассмотренной. Она сложилась бы немного иначе, если бы одна и та же зависимость была у обоих потребителей (рис. 8.3).

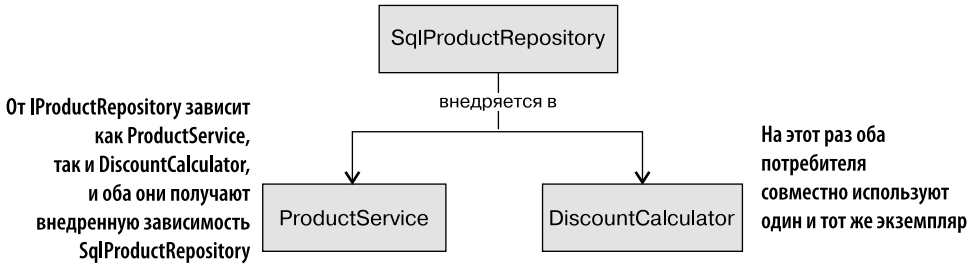


Рис. 8.3. Многократное использование одного и того же экземпляра зависимости путем внедрения его в несколько потребителей

Если это применить к коду листинга 8.1, получится код, показанный в листинге 8.2.

Листинг 8.2. Создание композиции из одного экземпляра зависимости

```
var repository = new SqlProductRepository(connString);

var productService = new ProductService(repository);

var calculator = new DiscountCalculator(repository);
```

Вместо создания двух разных экземпляров SqlProductRepository создается единственный экземпляр, внедряемый в оба потребителя. И оба они сохраняют ссылку для дальнейшего применения

Определить при сравнении листингов 8.1 и 8.2, какой из них лучше, невозможно. При изучении раздела 8.3, станет понятно, что, принимая решение о месте и способе многократного использования зависимости, нужно рассмотреть несколько факторов.

ПРИМЕЧАНИЕ

Потребители находятся в беспечном неведении о совместном использовании зависимости. Поскольку они оба принимают любую предоставляемую им версию зависимости, то приспособиться к этому изменению в конфигурации зависимости можно без каких-либо изменений в исходном коде. Это является результатом следования принципу подстановки Лисков.

Принцип подстановки Лисков

В исходном виде принцип подстановки является теоретической и абстрактной концепцией, сформулированной Барбарой Лисков в 1987 году. Но в объектно-ориентированном проектировании его можно раскрыть следующим образом: «Методы, потребляющие абстракции, должны иметь возможность воспользоваться любым классом, полученным из этой абстракции, не замечая разницы».

У нас должна быть возможность выполнить подстановку произвольной реализации, не вносящей разлада в систему, вместо абстракции. Если не придерживаться принципа подстановки Лисков, система станет нестабильной, поскольку нельзя будет производить замену зависимостей, а это может привести к сбоям в работе потребителя.

По сравнению с предыдущим примером жизненный цикл зависимости `Repository` претерпел существенные изменения. Прежде чем переменная `repository` станет подходящим объектом для сборки мусора, из области видимости должны уйти оба ее потребителя, и сделать это они могут в разное время. Предсказать, когда именно окончится время существования зависимости, становится труднее. С увеличением числа потребителей ситуация только усугубляется.

При довольно большом числе потребителей, вероятнее всего, всегда найдется потребитель, поддерживающий существование зависимости. Но вряд ли это станет проблемой: вместо множества одинаковых экземпляров будет только один, что позволит сэкономить память. Высокая востребованность этой особенности привела к ее формализации в паттерн жизненного цикла под названием Singleton («Одиночка»). Несмотря на сходство названий, не следует путать его с паттерном проектирования «Одиночка»[1]. Подробнее рассмотрим эту тему в подразделе 8.3.1.

Ключевым моментом для оценки является то, что компоновщик (Composer) намного сильнее влияет на время жизни зависимостей, чем любой отдельно взятый потребитель. Компоновщик решает, когда создавать экземпляры, и самостоятельно

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 157.

выбирает, стоит ли задавать их совместное использование. Он определяет, вышла ли зависимость с одним потребителем из области видимости, или же до высвобождения зависимости из области видимости должны выйти все ее потребители.

ОПРЕДЕЛЕНИЕ

> Высвобождение — это процесс определения того, какие зависимости могут быть разыменованы и, возможно, ликвидированы. Корень композиции запрашивает из компоновщика граф объектов. Завершив работу с этим обработанным графом, корень композиции информирует об этом компоновщик. После этого компоновщик может решить, какие из зависимостей конкретно этого графа могут быть высвобождены.

Это сравнимо с походом в ресторан с хорошим сомелье, который основную часть своего рабочего времени проводит в винном погребе: закупает новые вина, снимает пробы с имеющихся бутылок, проверяя состояние вина, и работает с шеф-поварами, подбирая оптимальное сочетание подаваемых блюд и вин. В результате нам подают винную карту, в которую включены только те напитки, которые подходят к сегодняшнему меню. Мы вольны выбирать вино по вкусу, но заранее знаем, что сомелье лучше разбирается в подборе вин в ресторане и в их сочетаемости с блюдами. Зачастую по решению сомелье в хранилище годами лежит множество бутылок. В следующем разделе будет показано, что и компоновщик может принять решение о поддержании жизнеспособности экземпляров путем сохранения ссылок на них.

8.1.2. Управление временем жизни с применением чистой технологии DI

В предыдущем разделе рассматривались способы варьирования композиции зависимостей с целью повлиять на продолжительность их жизни. В этом разделе нам предстоит рассмотреть способы реализации этих возможностей с помощью чистой технологии DI и двух самых распространенных жизненных циклов: Transient и Singleton.

В главе 7 для компоновки приложений создавались специализированные классы. Одним из них для среды ASP.NET Core MVC был `CommerceControllerActivator`, который и был нашим компоновщиком. В листинге 7.8 показана реализация принадлежащего ему метода `Create`.

Следует напомнить, что метод `Create` при каждом вызове оперативно создает полный граф объектов. Каждая зависимость оказывается принадлежащей исключительно выдаваемому контроллеру и не подлежит совместному использованию с другими потребителями. Когда экземпляр контроллера выходит из области видимости (что случается при каждом ответе сервера на запрос), все зависимости также выходят из области видимости. Такой жизненный цикл часто называют Transient («Кратковременный»), подробнее он будет рассмотрен в подразделе 8.3.2.

Проанализируем граф объектов, созданный `CommerceControllerActivator` (рис. 8.4), чтобы понять, можно ли его улучшить. Классы `AspNetUserContextAdapter`

и RouteCalculator являются сервисами без сохранения состояния, поэтому нет смысла создавать новый экземпляр при каждом обслуживании запроса. Строка подключения также вряд ли будет изменяться, поэтому ее можно многократно использовать во всех запросах. А вот класс SqlProductRepository зависит от CommerceContext — реализации контекста базы данных DbContext из Entity Framework, который должен применяться при обработке различных запросов[1].

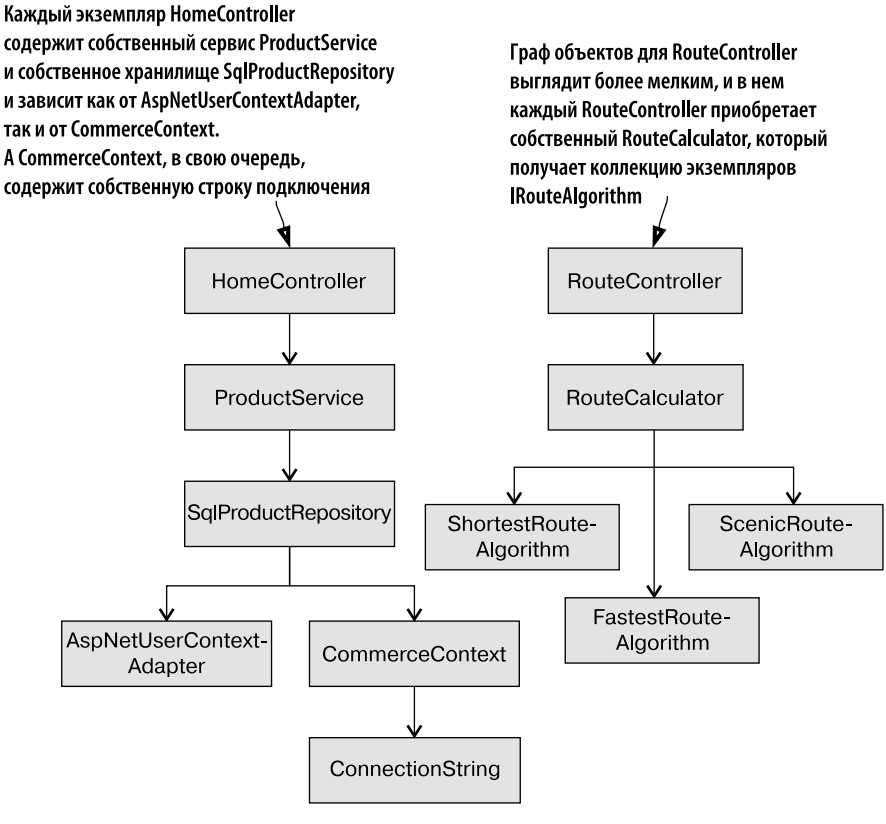


Рис. 8.4. Граф объектов, созданный CommerceControllerActivator, который создает экземпляры HomeController и RouteController с их зависимостями

С учетом данной конфигурации в более удачной реализации CommerceControllerActivator одни и те же экземпляры как AspNetUserContextAdapter, так и RouteCalculator будут при создании новых экземпляров ProductService и SqlProductRepository задействоваться повторно. Короче говоря, вам следует настраивать AspNetUserContextAdapter и RouteCalculator на использование жизненного цикла Singleton, а ProductService и SqlProductRepository — жизненного цикла Transient. Как реализуется это изменение, показано в листинге 8.3.

[1] Экземпляры DbContext из Entity Framework не являются потокобезопасными и не могут использоваться несколькими запросами в параллельном режиме.

Листинг 8.3. Управление временем жизни в CommerceControllerActivator

```
public class CommerceControllerActivator : IControllerActivator
{
    private readonly string connectionString;
    private readonly IUserContext userContext;
    private readonly RouteCalculator calculator;

    public CommerceControllerActivator(string connectionString)
    {
        this.connectionString = connectionString;

        this.userContext =
            new AspNetUserContextAdapter();

        this.calculator =
            new RouteCalculator(
                this.CreateRouteAlgorithms());
    }

    public object Create(ControllerContext context)
    {
        Type type = context.ActionDescriptor
            .ControllerTypeInfo.AsType();

        switch (type.Name)
        {
            case "HomeController":
                return this.CreateHomeController();

            case "RouteController":
                return this.CreateRouteController();

            default:
                throw new Exception("Unknown controller " + type.Name);
        }
    }

    private HomeController CreateHomeController()
    {
        return new HomeController(
            new ProductService(
                new SqlProductRepository(
                    new CommerceContext(
                        this.connectionString)),
                this.userContext));
    }

    private RouteController CreateRouteController()
    {
        return new RouteController(this.calculator);
    }

    public void Release(ControllerContext context,
        object controller) { ... }
}
```

Поля только для чтения, предназначенные для хранения зависимостей-одиночек

Создание зависимостей-одиночек и сохранение их в приватных полях. Таким образом, их могут повторно использовать все запросы на протяжении всей жизни приложения

Создание кратковременных экземпляров всякий раз, когда от CommerceControllerActivator требуется создание нового экземпляра. В эти экземпляры внедряются ранее созданные зависимости-одиночки

Метод Release пока оставлен пустым, мы вернемся к нему в разделе 8.2

ПРИМЕЧАНИЕ

Ключевое слово readonly в листинге 8.3 предоставляет дополнительные гарантии того, что после его назначения Singleton-экземпляры станут постоянными и их невозможно будет заменить. Больше ни для чего ключевое слово readonly при реализации жизненного цикла Singleton не требуется.

В MVC-приложении целесообразно загрузить конфигурационные значения в классе `Startup`. Поэтому строка подключения в листинге 8.3 предоставляется конструктору активатора `CommerceControllerActivator`.

Функционально код листинга 8.3 эквивалентен коду листинга 7.8 — он лишь немного эффективнее, поскольку некоторые зависимости используются совместно. Сохранением ссылок на созданные вами зависимости можно поддерживать их существование сколь угодно долго. В данном примере в `CommerceControllerActivator` сразу же после инициализации создаются обе Singleton-зависимости, но можно было воспользоваться и ленивой инициализацией.

Возможность тонкой настройки жизненного цикла каждой зависимости может оказаться немаловажной не только для поддержания высокой производительности, но и для того, чтобы приложение вело себя корректно. Например, паттерн проектирования «Посредник» (Mediator) зависит от совместно используемого директора (director), через которого обмениваются данными несколько компонентов[1]. Работоспособность определяется совместным применением «Посредника» всеми задействованными сотрудничающими компонентами (collaborators).

До сих пор рассматривалось влияние инверсии управления, не позволяющее потребителям управлять временем жизни зависимостей, поскольку созданием объектов они не управляют. А так как в среде .NET используется сборка мусора, потребители не могут также непосредственно уничтожать объекты. При этом без ответа остается вопрос: а как обстоят дела с ликвидируемыми зависимостями? Заострим внимание на этом деликатном вопросе.

8.2. Работа с ликвидируемыми зависимостями

Хотя среда .NET является управляемой платформой со сборщиком мусора, она не утратила возможности взаимодействия с неуправляемым кодом. В случае такого взаимодействия среда .NET имеет дело с неуправляемой областью памяти, не подвергаемой сборке мусора. Чтобы исключить утечку памяти, нужен вполне определенный механизм высвобождения неуправляемой области памяти. Для этого в основном и предназначен интерфейс `IDisposable`.

Вполне вероятно, что некоторые реализации зависимостей будут содержать неуправляемые ресурсы. К примеру, ликвидируемыми (disposable) являются подключения ADO.NET, поскольку у них есть склонность к использованию неуправляемой памяти. В результате реализации, связанные с базами данных, например хранилища на их основе, будут, скорее всего, также ликвидируемыми. Как же тогда

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 319.

следует моделировать ликвидируемые зависимости? Нужно ли при этом позволять абстракциям быть ликвидируемыми? Этот может выглядеть следующим образом.

```
public interface IMyDependency : IDisposable
```

ВНИМАНИЕ

> Технически это возможно, но сама идея далека от совершенства. Это проблемная конструкция, поскольку, как объяснялось в подразделе 6.2.1, служит признаком абстракции с протечкой.

Если есть желание добавить `IDisposable` к своему интерфейсу, то, вероятнее всего, вы представили себе вполне конкретную реализацию. Но допускать утечку сведений через конструкцию интерфейса недопустимо. Это затруднит реализацию интерфейса в других классах и привнесет в абстракцию неопределенность.

Кто будет отвечать за уничтожение ликвидируемой зависимости? Может ли это быть потребитель?

8.2.1. Потребление ликвидируемых зависимостей

Представим ради интереса, что у нас имеется ликвидируемая абстракция вроде следующего интерфейса `IOrderRepository` (листинг 8.4).

Листинг 8.4. IDisposable реализуется в IOrderRepository

```
public interface IOrderRepository : IDisposable
```

Как классу `OrderService` следует распорядиться такой зависимостью? Во многих руководствах по проектированию, включая встроенную в Visual Studio систему анализа кода Code Analysis, говорится, что если класс содержит ликвидируемый ресурс в качестве компонента, то он сам должен реализовать `IDisposable` и ликвидацию ресурса. Как это делается, показано в листинге 8.5.

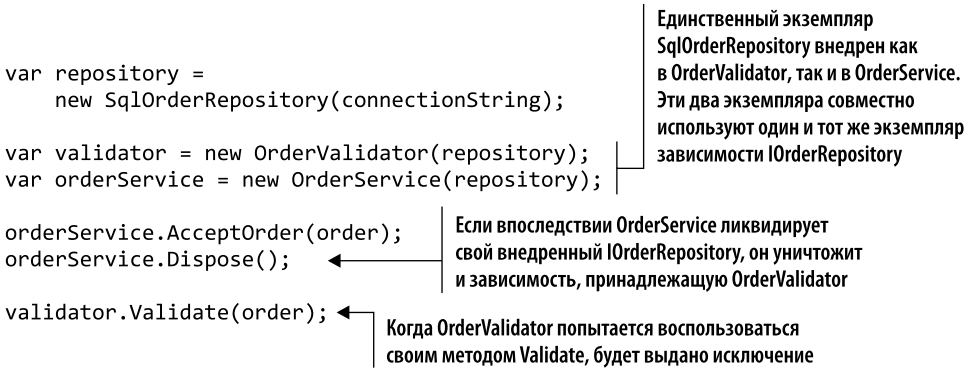
Листинг 8.5. Использование классом OrderService ликвидируемой зависимости

```
public sealed class OrderService : IDisposable
{
    private readonly IOrderRepository repository;

    public OrderService(IOrderRepository repository)
    {
        this.repository = repository;
    }

    public void Dispose()
    {
        this.repository.Dispose();
    }
}
```

OrderService также реализует IDisposable

Реализация Dispose и ликвидация зависимости IOrderRepository

Но получается, что это не самая удачная идея, поскольку компонент `repository` был изначально внедрен и может использоваться другими потребителями совместно:



Было бы безопаснее не ликвидировать внедренное хранилище, но тогда будет проигнорирован тот факт, что абстракция является ликвидируемой. К тому же в таком случае число компонентов, выставляемых наружу абстракцией, превышает число компонентов, используемых клиентом, а это нарушение принципа изоляции интерфейса (см. подраздел 6.2.1). Объявление абстракции в качестве производной от `IDisposable` не дает никакой выгоды.

Впрочем, могут встречаться сценарии, где требуется просигнализировать о начале и конце краткосрочной области, и для этой цели иногда применяется `IDisposable`. Прежде чем приступить к изучению способов, с помощью которых компоновщик может управлять временем жизни ликвидируемой зависимости, нужно понять, как следует распоряжаться недолговечными ликвидируемыми объектами.

ОПРЕДЕЛЕНИЕ

Недолговечным и ликвидируемым считается объект с вполне понятным и коротким временем жизни, не превышающим одного вызова метода.

Создание недолговечных ликвидируемых объектов

Многие API, имеющиеся в .NET BCL, используют `IDisposable`, чтобы просигнализировать об окончании конкретной области видимости. Один из наиболее ярких примеров — WCF-прокси.

Нужно помнить, что использование `IDisposable` для этих целей не должно служить признаком абстракции с протечкой, поскольку рассматриваемые типы не всегда являются абстракциями. Однако некоторые из них все же являются ими, и что с этим делать?

WCF-прокси и IDisposable

`IDisposable` реализуется во всех автоматически создаваемых WCF-прокси, и не следует забывать, что метод `Dispose` в отношении прокси нужно вызывать при первой же возможности[1]. Многие подключения при отправке первого запроса автоматически создают на сервисе сессию, которая сохраняется на нем, пока не истечет время ожидания или она не будет уничтожена явным образом.

Если забыть ликвидировать WCF-прокси после их использования, число сессий станет множиться, пока не будет исчерпан лимит параллельных подключений из одного и того же источника. Когда лимит выбран, выдается исключение. Слишком большое количество сессий также накладывает неподъемную нагрузку на сервис, поэтому очень важно ликвидировать WCF-прокси как можно раньше.

После ликвидации объект повторно применить невозможно. Если нужно снова вызвать тот же API, придется создавать новый экземпляр. В качестве примера: этот прием хорошо сочетается со способом использования WCF-прокси или команд ADO.NET. Прокси создается, вызываются его операции, и по завершении их работы прокси ликвидируется. Как это согласуется с применением технологии DI, если ликвидируемые абстракции считаются абстракциями с протечкой?

Как всегда, поможет маскировка ненужных подробностей за интерфейсом. Если вернуться к UWP-приложению из раздела 7.2, то там для сокрытия деталей обмена информацией с хранилищем данных от уровня логики представления использовалась абстракция `IProductRepository`. Тогда мы проигнорировали подробности реализации, поскольку на тот момент это было не так уж и важно. Но представим, что UWP-приложение должно обмениваться данными с вебслужбой WCF. Вот как выглядит удаление товара с позиции модели представления `EditProductViewModel`:

Путем предоставления идентификатора товара внедренному хранилищу Repository у него запрашивается удаление этого товара. В EditProductViewModel ссылка на Repository может содержаться, не вызывая каких-то опасений, так как интерфейс IProductRepository не является производным IDisposable

```
private void DeleteProduct()
{
    this.productRepository.Delete(this.Model.Id);
    this.whenDone();
}
```

Но при рассмотрении реализации WCF вырисовывается совсем иная картина. Реализация `WcfProductRepository` с методом `Delete` имеет следующий вид (листинг 8.6).

[1] Хотя сервисы невозможно создавать с применением среды .NET Core, приложение, выполняемое в среде .NET Core, все же может быть их потребителем.

Класс `WcfProductRepository` не имеет изменяемого состояния, поэтому в него внедряется фабрика `ChannelFactory<TChannel>`, которой можно воспользоваться для создания канала. Слово «канал» служит просто другим названием для WCF-прокси, и он в автоматическом режиме создает интерфейс, получаемый вами безо всяких усилий в тот самый момент, когда в Visual Studio или с помощью `svcutil.exe` вы создаете ссылку на сервис.

Листинг 8.6. Использование WCF-канала в качестве недолговечного ликвидируемого объекта

```
public class WcfProductRepository : IProductRepository
{
    private readonly ChannelFactory<IProductManagementService> factory;

    public WcfProductRepository(
        ChannelFactory<IProductManagementService> factory)
    {
        this.factory = factory;
    }

    public void Delete(Guid productId)
    {
        using (var channel =
            this.factory.CreateChannel())
        {
            channel.DeleteProduct(productId);
        }
    }
    ...
}
```

Метод Delete создает WCF-канал, являющийся недолговечным ликвидируемым объектом. Этот канал создается и ликвидируется в рамках одного и того же вызова метода

Поскольку этот интерфейс является производным интерфейса `IDisposable`, его можно заключить в инструкцию `using`. Затем канал используется для удаления товара. При выходе из области видимости `using` канал ликвидируется.

ВНИМАНИЕ

Хотя применение инструкции using в работе с недолговечными зависимостями признается вполне оптимальным приемом, по отношению WCF это не совсем так. Вопреки всем методическим рекомендациям классы WCF-прокси могут при вызове Dispose выдавать исключения. Выдача исключения из блока using приводит к потере исходной информации об исключении. Вместо того чтобы полагаться на инструкцию using, нужно реализовать блок finally и проигнорировать любые исключения, выдаваемые Dispose. Мы ссылаемся здесь на инструкцию using, чтобы продемонстрировать общую концепцию реализации недолговечных ликвидируемых объектов[1].

При каждом вызове в отношении класса `WcfProductRepository` метод быстро открывает новый канал и ликвидирует его после использования. Он отличается

[1] Дополнительные сведения можно найти по адресу https://mng.bz/5Y6z.

чрезвычайной кратковременностью существования, поэтому-то мы и называем подобную ликвидируемую абстракцию недолговечным ликвидируемым объектом.

Но постойте! Разве мы не утверждали, что ликвидируемая абстракция является абстракцией с протечкой? Конечно, утверждали, но прагматические цели должны быть сбалансированы с принципами. По крайней мере, в данном случае `WcfProductRepository` и `IProductManagementService` определены в той же специализированной библиотеке WCF. Тем самым гарантируется, что абстракция с протечкой может быть ограничена кодом, от которого вполне резонно ожидать осведомленности об этой сложности и способности с ней справиться.

Следует заметить, что недолговечные ликвидируемые объекты никогда не внедряются в потребителя. Вместо этого применяется фабрика, которую вы используете для управления временем жизни недолговечного ликвидируемого объекта.

Фабрика `ChannelFactory<TChannel>` является потокобезопасной и может внедряться как Singleton-зависимость. В таком случае то, что мы выбрали внедрение `ChannelFactory<TChannel>` в конструктор `WcfProductRepository`, может вызвать удивление: создать зависимость можно в самом коде, сохранив ее в статическом поле. Но такой прием приведет к вполне естественной зависимости `WcfProductRepository` от конфигурационного файла, который на момент создания нового экземпляра `WcfProductRepository` уже должен существовать. Как отмечалось в подразделе 2.2.3, на конфигурационные файлы должны полагаться только готовые приложения.

Таким образом, ликвидируемые абстракции следует рассматривать как абстракции с протечкой. Иногда нужно просто смириться с протечкой, чтобы избежать ошибок (таких как отклоненные WCF-подключения), но, соглашаясь на этот шаг, предпринять все возможное, чтобы обуздать протечку и не дать ей распространиться на все приложение. Теперь исследуем потребление ликвидируемых зависимостей, обращая внимание на способы их подачи и управления ими в угоду потребителю.

8.2.2. Управление ликвидируемыми зависимостями

Поскольку мы так упорно настаивали на том, что ликвидируемые абстракции считаются абстракциями с протечкой, из этого можно сделать вывод, что абстракции не должны быть ликвидируемыми. Но в то же время иногда ликвидируемыми являются реализации: если ликвидацию провести неправильно, в приложении произойдет утечка ресурсов. Кому-то или чему-то следует заняться их ликвидацией.

СОВЕТ

Старайтесь реализовать сервисы, чтобы в них были не ссылки на ликвидируемые объекты, а происходили, как показано на рис. 8.6, их создание при необходимости и ликвидация. Это упрощает управление, поскольку сервис, как и другие объекты, может попасть под сборку мусора.

Как всегда, ответственность за это возлагается на компоновщик. Он лучше знает, когда создавать ликвидируемый экземпляр и когда его ликвидировать. Компоновщику проще хранить ссылку на ликвидируемый экземпляр и в нужное время вызывать принадлежащий ему метод `Dispose`. Задача заключается в определении этого времени. Как узнать, что все потребители покинули область видимости?

Без соответствующей информации узнать об этом невозможно. Но зачастую ваш код находится в окружении какого-либо контекста с четко определенным временем жизни, а также событиями, сообщающими о завершении пребывания в определенной области видимости. Например, в среде ASP.NET Core такой областью для внедряемых экземпляров может являться отдельно взятый веб-запрос. В конце веб-запроса среда сообщает `IControllerActivator`, который обычно и является нашим компоновщиком, что он должен высвобождать все зависимости для заданного объекта. Затем на компоновщик возлагается отслеживание этих зависимостей и принятие решения о том, должно ли что-либо ликвидироваться на основе жизненных циклов этих зависимостей.

Высвобождение зависимостей

Высвобождение графа объектов не нужно отождествлять с их ликвидацией. Как утверждалось во введении, под высвобождением понимается процесс определения того, какие зависимости могут быть разыменованы и, возможно, ликвидированы, а какие следует сохранить для последующего применения. Именно компоновщик решает, должен ли высвобожденный объект ликвидироваться или использоваться повторно.

Высвобождение графа объектов является сигналом компоновщику о том, что корень графа вышел из области видимости, поэтому, если в самом корне реализован интерфейс `IDisposable`, он должен быть ликвидирован. Но зависимости, принадлежащие корню, могут одновременно использоваться и другими корнями, поэтому компоновщик может принять решение сохранить некоторые из них, поскольку у него есть сведения, что от них еще зависят какие-то объекты. Последовательность событий показана на рис. 8.5.

Чтобы высвободить зависимости, компоновщик должен отслеживать все ликвидируемые зависимости, которые когда-либо обслуживал, и то, для каких потребителей он их обслуживал, получая таким образом возможность ликвидировать их, как только будет высвобожден последний потребитель. Компоновщик должен заботиться также о ликвидации объектов в надлежащем порядке.

ВНИМАНИЕ

Объекту может потребоваться вызов его зависимостей в ходе их ликвидации, что может вызвать проблемы, когда зависимости уже ликвидированы. Поэтому ликвидация должна проводиться в порядке, обратном созданию, то есть снаружи вовнутрь.

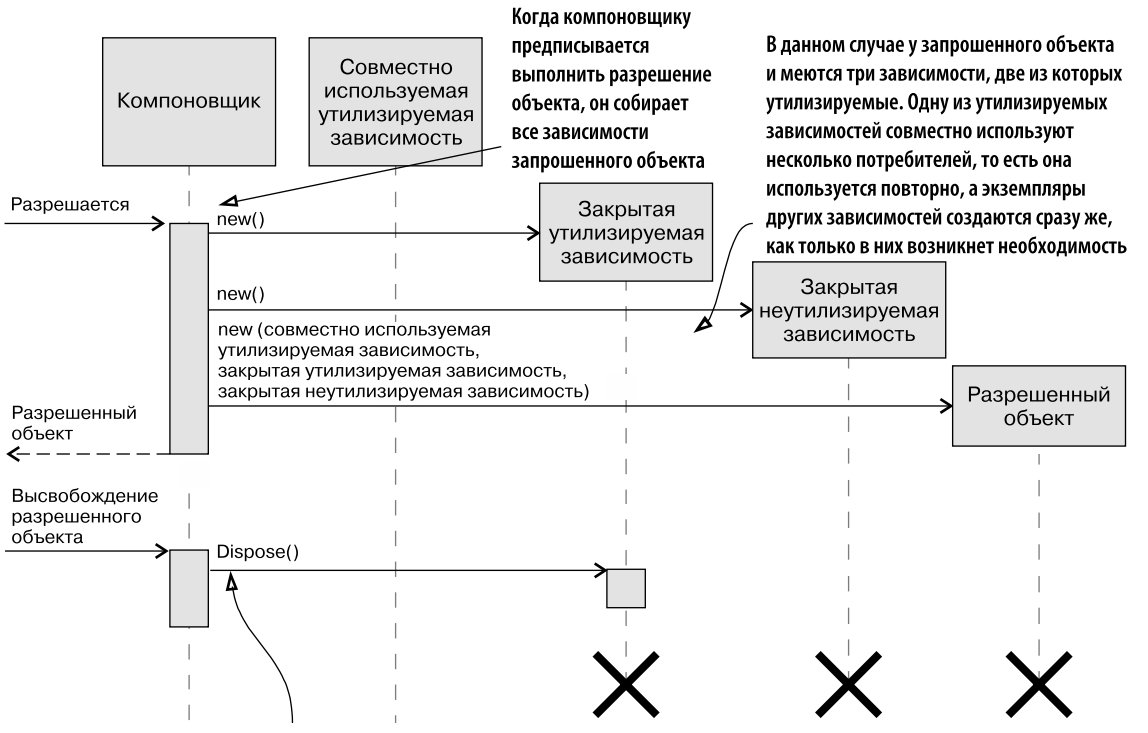


Рис. 8.5. Последовательность событий для высвобождения зависимостей

Нужно ли ликвидировать DbContext?

`CommerceContext` — это специфическая для нашего проекта версия `DbContext`, принадлежащая среде Entity Framework Core, в которой реализуется `IDisposable`. В прошлом мы часто спорили с коллегами и разработчиками на онлайн-форумах о необходимости ликвидации экземпляров `DbContext`. Обычно эти дискуссии вытекали из того, что в `DbContext` в качестве недолговечных ликвидируемых объектов использовались подключения к базам данных: подключения открывались и закрывались в рамках одного и того же вызова метода. Например, вызов метода `SaveChanges` в отношении `DbContext` приводил к созданию и открытию подключения к базе данных с последующей ликвидацией этого подключения после сохранения всех изменений.

Но в среде Entity Framework Core 2.0 положение вещей изменилось. С выходом версии 2 в этой среде поддерживается организация пула `DbContext`, подобного организации пула подключений в среде ADO.NET. При этом разрешается повторное использование одного и того же экземпляра `DbContext`, что при определенных условиях может повысить производительность приложения. Но при вызове `Dispose` экземпляры `DbContext` возвращаются обратно в свой пул, поэтому отказ от вызова `Dispose` может привести к истощению пула.

Из этого следует, что всегда нужно гарантировать правильную ликвидацию приспособленных для нее объектов. Даже если окажется, что именно в вашем случае вызов `Dispose` можно опустить, такой внешний компонент, как Entity Framework Core, может это поведение в любой момент изменить по своему усмотрению.

ПРИМЕЧАНИЕ

Подробные сведения о среде Entity Framework Core можно найти в книге: *Smith J.* Entity Framework Core in Action. — Manning, 2018.

Вернемся к примеру `CommerceControllerActivator` из листинга 8.3. Как оказалось, в код листинга вкралась ошибка, потому что `IDisposable` реализован в `CommerceContext`. Код листинга 8.3 создает новые экземпляры `CommerceContext`, но они в этом коде никогда не ликвидируются. В связи с этим может произойти утечка ресурсов, поэтому исправим эту ошибку в новой версии компоновщика.

Во-первых, нужно принять в расчет, что у компоновщика для веб-приложения должна иметься возможность обслуживать множество одновременно поступающих запросов, поэтому он должен связывать каждый экземпляр `CommerceContext` либо с создаваемым им корневым объектом, либо со связанным с ним запросом. В следующем примере запрос будет использоваться для отслеживания ликвидируемых объектов, поскольку тогда не придется определять статический словарь. Правильно применять статическое изменяемое состояние намного труднее, поскольку реализовывать соответствующий код нужно в потокобезопасном стиле. В листинге 8.7 показано, как в `CommerceControllerActivator` выполняется разрешение запросов для экземпляров `HomeController`.

Листинг 8.7. Привязка ликвидируемых зависимостей к веб-запросу

```
private HomeController CreateHomeController(ControllerContext context)
{
    var dbContext =
        new CommerceContext(this.connectionString);

    TrackDisposable(context, dbContext);

    return new HomeController(
        new ProductService(
            new SqlProductRepository(dbContext),
            this.userContext));
}

private static void TrackDisposable(
    ControllerContext context, IDisposable disposable)
{
    IDictionary<object, object> items =
        context.HttpContext.Items;

    object list;

    if (!items.TryGetValue("Disposables", out list))
    {
        list = new List<IDisposable>();
        items["Disposables"] = list;
    }

    ((List<IDisposable>)list).Add(disposable);
}
```

Создание экземпляра, требующего ликвидации

Отслеживание этого экземпляра путем его привязки к текущему запросу

Метод TrackDisposable сохраняет утилизируемые экземпляры в списке, связанном с запросом, помещая его в словарь HttpContext.Items. Если списка не существует, он будет создан. Утилизируемый экземпляр добавляется к списку

Метод `TrackDisposable` сохраняет ликвидируемые экземпляры в списке, связанном с запросом, помещая его в словарь `HttpContext.Items`. Если списка не существует, он будет создан. Ликвидируемый экземпляр добавляется к списку.

Метод `CreateHomeController` начинает свою работу с разрешения всех зависимостей. Это похоже на реализацию, показанную в листинге 8.3, но перед возвращением разрешенного сервиса метод должен сохранить зависимость с запросом таким образом, чтобы ее можно было ликвидировать, как только будет высвобожден контроллер. Блок-схема приложения из листинга 8.7 показана на рис. 8.6.

При реализации `CommerceControllerActivator` в листинге метод `Release` был оставлен пустым. Он пока не реализован, поскольку расчет делался на выполнение своих обязанностей сборщиком мусора. Но когда дело касается ликвидируемых зависимостей, важно воспользоваться для очистки возможностью, предоставляемой этим методом. Его реализация показана в листинге 8.8.

В методе `Release` слегка упрощен, что предотвращает ликвидацию некоторых ликвидируемых экземпляров в случае выдачи исключения. Особо щепетильным нужно убедиться, что ликвидация экземпляров продолжается, даже если при ликвидации одного из них выдано исключение. Сделать это предпочтительнее, применив

инструкции `try` и `finally`. Пусть создание соответствующего кода станет упражнением для читателей.

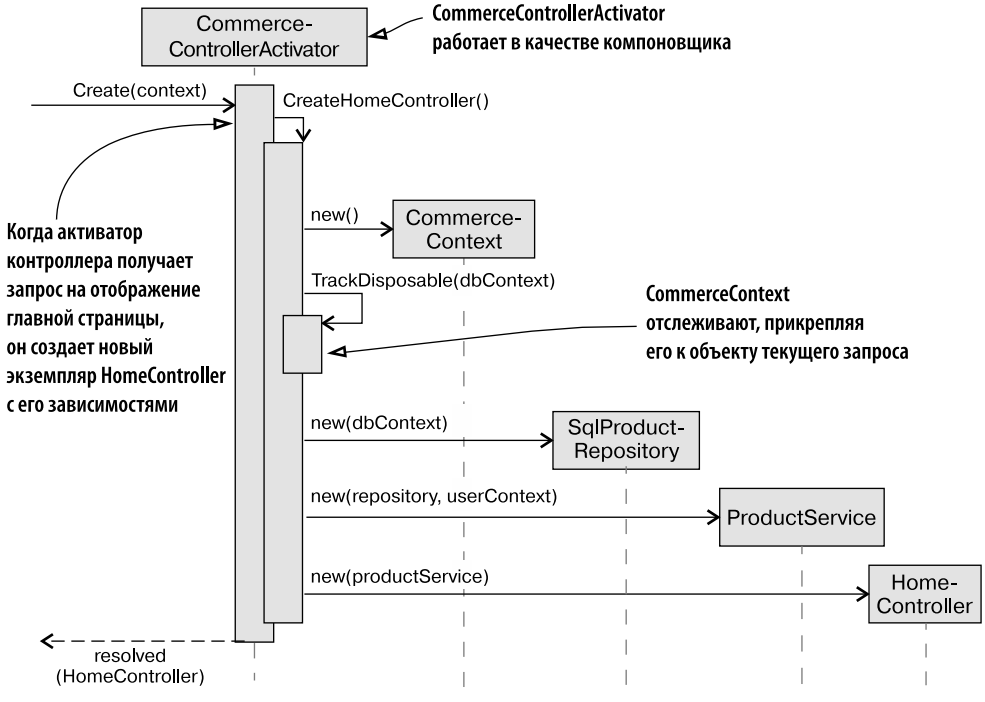


Рис. 8.6. Отслеживание ликвидируемых зависимостей

Листинг 8.8. Высвобождение ликвидируемых зависимостей

```
public void Release(ControllerContext context, object controller)
{
    var disposables =
        (List<IDisposable>)context.HttpContext
            .Items["Disposables"];

    if (disposables != null)
    {
        disposables.Reverse();

        foreach (IDisposable disposable in disposables)
        {
            disposable.Dispose();
        }
    }
}
```

Получение списка отслеживаемых зависимостей из словаря элементов

Замена порядка следования записей в списке ликвидируемых зависимостей на обратный, чтобы экземпляры могли ликвидироваться в порядке, обратном последовательности их создания

Последовательный перебор коллекции и поочередная утилизация всех экземпляров

В контексте среды ASP.NET Core MVC решение с использованием `TrackDisposable` и `Release` может быть сведено к простому вызову `HttpContext.Response.RegisterForDispose`, поскольку это позволит сделать, по сути, то же самое. В обоих вариантах реализуются ликвидация в обратном порядке и продолжение ликвидации объектов в случае сбоя. Поскольку данная глава не посвящена конкретно среде ASP.NET Core MVC, нам захотелось ознакомить вас с более универсальным решением, иллюстрирующим основную идею.

СОВЕТ

Для управления временем жизни как нельзя лучше подходят DI-контейнеры. Они могут справиться со сложными комбинациями жизненных циклов и предлагают, например, метод Release для явного высвобождения компонентов, когда они станут не нужны. В ситуации, когда поддержка корня композиции с использованием чистой технологии DI затрудняется, нужно рассмотреть вариант с переключением на применение вместо нее DI-контейнера. (Вникать в подробности всего этого мы будем при рассмотрении DI-контейнеров в главе 12.)

Где должно происходить высвобождение зависимостей?

После прочтения всего изложенного остаются два вопроса: где нужно высвобождать графы объектов и кто за это несет ответственность? Важно отметить, что код, запрашивающий граф объектов, отвечает также за запрос на его высвобождение. Поскольку запрос графа объектов обычно является частью корня композиции, этот же корень инициирует высвобождение графа.

ПРИМЕЧАНИЕ

Высвобождение потребуется после того, как корень композиции завершит работу с разрешенным корневым объектом.

В листинге 8.9 еще раз показан код метода `Main` консольного приложения из раздела 7.1, но теперь с дополнительным методом `Release`.

Листинг 8.9. Корень композиции, высвобождающий разрешенный граф объектов

```
static void Main(string[] args)
{
    string connStr = LoadConnectionString();

    CurrencyParser parser =
        CreateCurrencyParser(connStr);

    ICommand command = parser.Parse(args);
    command.Execute();

    Release(parser);
}
```

Запрос корневого объекта CurrencyParser

Использование этого корневого объекта

Требование его высвобождения после завершения операции

Создавая консольное приложение, вы его полностью контролируете. Как говорилось в разделе 7.1, инверсии управления в нем нет. А при использовании некой среды зачастую получается, что именно она контролирует как запрос графа объектов, так и требование высвободить его. Наглядным примером может послужить среда ASP.NET Core MVC. Работая с MVC, мы имеем дело со средой, вызывающей принадлежащие `CommerceControllerActivator` методы `Create` и `Release`. Между такими вызовами среда задействует разрешенные экземпляры контроллера.

Подведем краткие итоги рассмотрения некоторых подробностей управления временем жизни зависимостей. Потребитель не в состоянии управлять временем жизни внедренных зависимостей, это обязанность компоновщика, способного принимать решение, нужно ли организовывать совместное использование одного экземпляра несколькими потребителями или же следует каждому потребителю дать собственный, личный экземпляр. Такие жизненные циклы, называемые Singleton и Transient, — всего лишь наиболее распространенные составляющие большого набора жизненных циклов. Следующий раздел будет посвящен проработке каталога самых распространенных стратегий жизненного цикла.

8.3. Каталог жизненных циклов зависимостей

Рассмотрев принципы, на которых базируется управление временем жизни зависимостей, уделим немного времени изучению наиболее распространенных жизненных циклов. Как говорилось во введении, под жизненным циклом понимается формализованный способ описания предполагаемого времени жизни зависимости. Это позволяет иметь общий словарь, как и для паттернов проектирования, что помогает быстрее понять, когда и как происходит ожидаемый выход зависимости из области видимости или, быть может, предполагается использовать ее повторно.

В этом разделе рассматриваются три наиболее распространенных жизненных цикла (табл. 8.1). Поскольку циклы Singleton и Transient вам уже встречались, с них и начнем.

Таблица 8.1. Жизненные циклы, рассматриваемые в данном разделе

Название	Описание
Singleton	Постоянное повторное использование единственного экземпляра
Transient	Постоянное предоставление новых экземпляров
Scoped	Предоставление в каждой подразумеваемой или явно определенной области видимости не более одного экземпляра каждого типа

ПРИМЕЧАНИЕ

В этом разделе мы рассматриваем аналогичные друг другу примеры. Но, чтобы можно было сосредоточиться на главном, создаются неглубокие иерархии и порой во избежание лишних сложностей игнорируется проблема с ликвидируемыми зависимостями.

Жизненный цикл Scoped получил довольно широкое распространение: большинство самых экзотических жизненных циклов являются его вариациями. По сравнению с более сложными жизненными циклами Singleton может показаться обыденным, однако он представляет собой часто встречающуюся и вполне подходящую стратегию жизненного цикла.

8.3.1. Жизненный цикл Singleton

В отдельных случаях жизненный цикл Singleton будет использоваться в приводимых в книге примерах без явного на то указания. Его название вроде бы вполне понятно, но в то же время чем-то все-таки смущает. И все же в нем есть вполне определенный смысл, поскольку задаваемое им поведение похоже на поведение паттерна Singleton («Одиночка»), но вот структуры у них разные.

ПРИМЕЧАНИЕ

> В области ответственности отдельно взятого компоновщика будет только один экземпляр компонента с жизненным циклом Singleton. При каждом запросе компонента потребителю предоставляется один и тот же экземпляр.

Как в случае с жизненным циклом Singleton, так и при использовании паттерна проектирования Singleton фигурирует только один экземпляр зависимости, но на этом их сходство заканчивается. Паттерн проектирования Singleton предоставляет глобальную точку доступа к своему экземпляру, что похоже на рассмотренный в разделе 5.3 антипаттерн «Окружающий контекст». А вот получить через статический компонент доступ к зависимости с областью видимости Singleton потребитель не может. Если запросить предоставление экземпляра у двух различных компоновщиков, будут получены два разных экземпляра. Поэтому важно, чтобы не было путаницы между жизненным циклом Singleton и одноименным паттерном проектирования.

Поскольку в работе используется только один экземпляр, при жизненном цикле Singleton, как правило, потребляется минимальный объем памяти, что делает программу более эффективной. Единственным исключением будет случай довольно редкого применения экземпляра, потребляющего значительный объем памяти. В таких случаях экземпляр может быть заключен в виртуальный заместитель, который будет рассмотрен в подразделе 8.4.2.

Когда следует использовать жизненный цикл Singleton

Жизненный цикл Singleton нужно задействовать везде, где только можно. Есть только две основные проблемы, которые могут воспрепятствовать этому.

- Когда у компонента отсутствует потокобезопасность. Поскольку экземпляр Singleton потенциально совместно используется сразу несколькими потребителями, он должен быть способен обрабатывать параллельный доступ.

- Когда время жизни одной из зависимостей компонента ожидаемо короче времени жизни остальных зависимостей, возможно, из-за того, что она не обладает потокобезопасностью. Задание компоненту жизненного цикла Singleton сделает слишком длинной жизнь его зависимостей. В этом случае такая зависимость оказывается захваченной (Captive Dependency). Более подробно захваченные зависимости будут рассмотрены в разделе 8.4.1.

По определению потокобезопасными являются все сервисы, не сохраняющие состояние, равно как и неизменяемые типы, и, вполне очевидно, классы, специально разработанные с прицелом на потокобезопасность. В таких случаях настраивать их под жизненный цикл, отличный от Singleton, нет никакого смысла.

Некоторые зависимости могут эффективно работать в соответствии со своим предназначением только при совместном использовании. Например, к таковым относятся реализации паттерна проектирования «Предохранитель» (Circuit Breaker)[1], рассматриваемого в главе 9, а также кэш-области, реализованные в оперативной памяти. В этих случаях важно то, что реализации потокобезопасны.

Теперь в качестве примера для исследования пристальнее рассмотрим хранилище, реализуемое в оперативной памяти.

Пример: использование потокобезопасного хранилища в оперативной памяти

Еще раз поговорим о реализации активатора `CommerceControllerActivator`, подобного тем, что рассматривались в подразделах 7.3.1 и 8.1.2. Вместо `IProductRepository` на основе SQL Server можно воспользоваться потокобезопасным хранилищем, реализованным в оперативной памяти. Чтобы от применения хранилища данных в оперативной памяти был определенный смысл, оно должно совместно использоваться всеми запросами, а следовательно, быть потокобезопасным. Именно это проиллюстрировано на рис. 8.7.

Вместо явной реализации хранилища с помощью паттерна проектирования Singleton можно, воспользовавшись жизненным циклом Singleton, задействовать конкретный класс, применив к нему соответствующую область видимости. В листинге 8.10 показано, как компоновщик может возвращать новые экземпляры при каждом запросе к нему на разрешение `HomeController`, при этом все экземпляры совместно используют `IProductRepository`.

Следует отметить, что и хранилище, и `userContext` придерживаются жизненного цикла Singleton. Но жизненные циклы можно смешивать по своему усмотрению. На рис. 8.8 показано, что происходит с `CommerceControllerActivator` в ходе выполнения программы.

Жизненный цикл Singleton один из самых простых в реализации. Он требует всего лишь сохранения ссылки на объект и предоставления этого же объекта при

---

[1] *Nygard Michael T.* Release It! Design and Deploy Production-Ready Software. — Pragmatic Bookshelf, 2007. — P. 104.

каждом его запросе. Экземпляр не выходит из области видимости до тех пор, пока из нее не выйдет компоновщик. Когда это произойдет, компоновщик должен ликвидировать объект, если он относится к ликвидируемому типу.

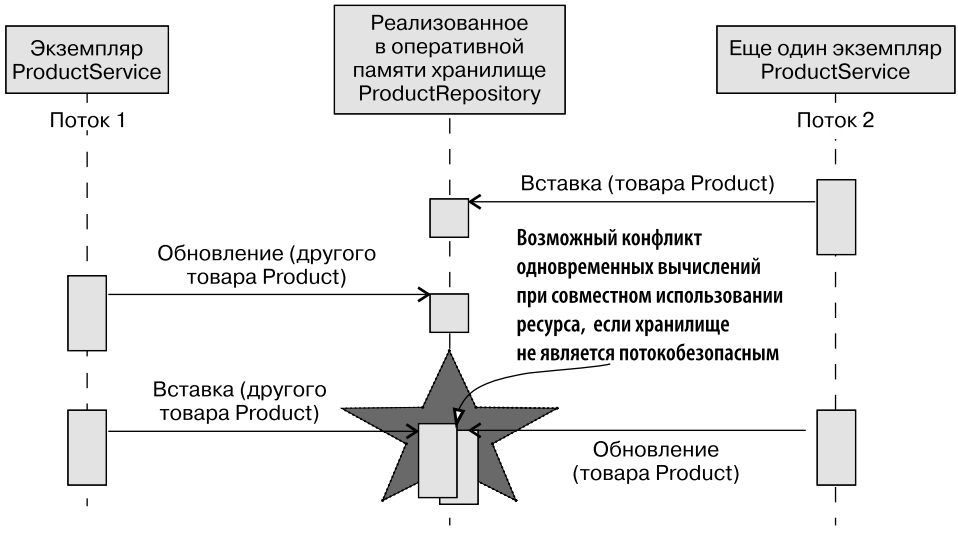


Рис. 8.7. Когда к совместно используемому ресурсу, такому как реализованный в оперативной памяти IProductRepository, обращаются сразу несколько ProductService, каждый из которых запущен в отдельном потоке, от вас требуется обеспечить потокобезопасность такого ресурса

Листинг 8.10. Управление жизненным циклом Singleton

```
public class CommerceControllerActivator : IControllerActivator
{
    private readonly IUserContext userContext;
    private readonly IProductRepository repository;

    public CommerceControllerActivator()
    {
        this.userContext = new FakeUserContext();
        this.repository = new InMemoryProductRepository();
    }
    ...

    private HomeController CreateHomeController()
    {
        return new HomeController(
            new ProductService(
                this.repository,
                this.userContext));
    }
}
```

В местах хранения экземпляров-одиночек ссылки на зависимости с жизненным циклом Singleton сохраняются в течение всего времени жизни компоновщика

Создание объектов-одиночек в конструкторе компоновщика

При каждом запросе на разрешение экземпляра HomeController компоновщик создает кратковременный ProductService с двумя внедренными в него Singleton-объектами

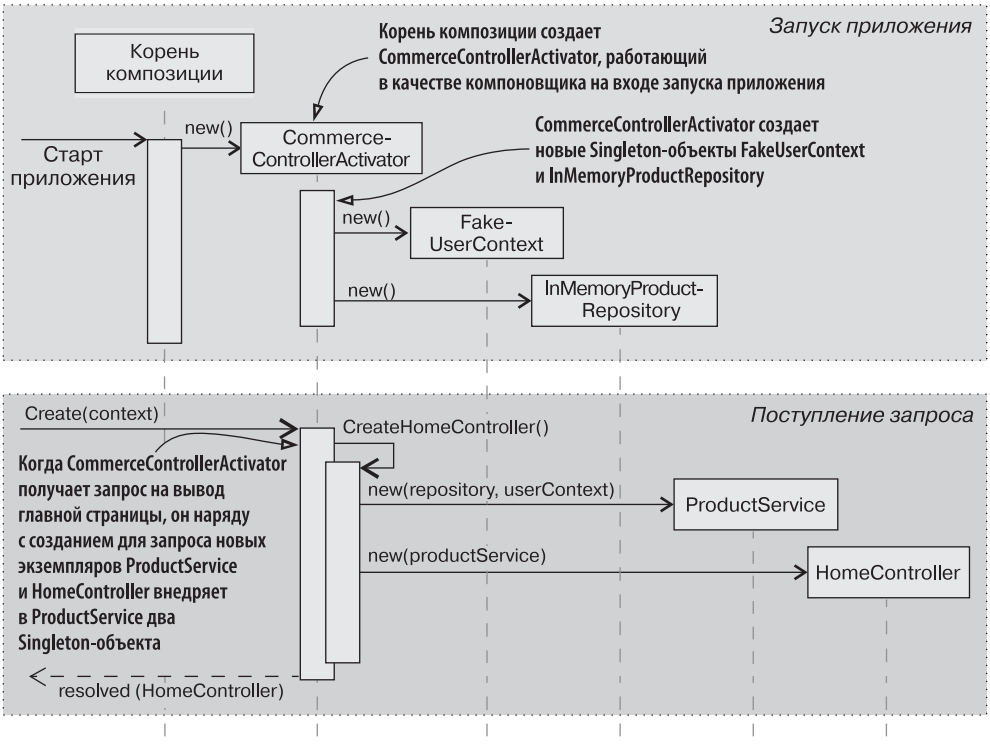


Рис. 8.8. Компоновка Singleton-объектов с использованием CommerceControllerActivator

Еще одним простым в реализации жизненным циклом является Transient. Его мы сейчас и рассмотрим.

8.3.2. Жизненный цикл Transient

Жизненным циклом Transient предусматривается возвращение нового экземпляра при каждом запросе. Если в возвращаемом экземпляре не реализуется интерфейс `IDisposable`, то отслеживать нечего. И наоборот, когда в экземпляре реализуется `IDisposable`, компоновщик должен отслеживать и явным образом ликвидировать этот экземпляр, как только поступит запрос на высвобождение соответствующего графа объектов. В основной массе приводимых в этой книге примеров, связанных с построением графов объектов, подразумевается использование жизненного цикла Transient.

ВНИМАНИЕ

Когда дело доходит до применения жизненного цикла Transient, следует знать, что DI-контейнеры могут вести себя по-разному. Некоторые из них отслеживают Transient-компоненты и стремятся ликвидировать их, как только их потребитель выходит из области видимости, есть и такие, которые этого не делают, поэтому Transient-компоненты не ликвидируются.

Стоит отметить, что автономные и подобные им приложения склонны разрешать применять всю иерархию объектов только один раз — при запуске приложения. Это означает, что может быть создано всего несколько экземпляров даже Transient-компонентов и они могут существовать довольно долго. В том случае, когда у зависимости имеется только один потребитель, итоговое разрешение графа чистых Transient-компонентов будет эквивалентно разрешению графа чистых Singleton-компонентов или любого их сочетания. Дело в том, что граф разрешается только один раз, поэтому разница в поведении никогда не чувствуется.

Когда следует использовать жизненный цикл Transient

Жизненный цикл Transient — самый безопасный из циклов, но также и один из наименее эффективных. Он может стать причиной создания и последующего сбора в качестве мусора множества экземпляров, даже если вполне достаточно было бы обойтись всего одним.

Но если есть сомнения по поводу потокобезопасности компонента, жизненный цикл Transient будет безопасен, поскольку у каждого потребителя есть собственный экземпляр зависимости. Во многих случаях жизненный цикл Transient можно смело заменять жизненным циклом Scoped, также гарантирующим последовательный доступ к зависимости.

Пример: выделение нескольких хранилищ

Ранее в этой главе уже встречались примеры использования жизненного цикла Transient. В листинге 8.3 хранилище создается и внедряется на месте в методе разрешения и компоновщик не сохраняет ссылки на него. Затем в листингах 8.8 и 8.9 было показано, как обращаться с ликвидируемым Transient-компонентом.

В этих примерах можно было заметить, что `userContext` так и остается Singleton-компонентом. Это сервис без сохранения состояния в чистом виде, поэтому нет никакого смысла создавать новый экземпляр для каждого созданного `ProductService`. Заслуживает внимания тот факт, что можно смешивать зависимости с разными жизненными циклами.

ВНИМАНИЕ

> Хотя зависимости с разными жизненными циклами и можно смешивать, следует убедиться, что у потребителя есть только те зависимости, срок жизни которых равен его собственному сроку жизни или превышает его, потому что потребитель будет поддерживать существование своих зависимостей, сохраняя их в собственных приватных полях. Если этого не сделать, появятся захваченные зависимости, речь о которых пойдет в подразделе 8.4.1.

Если одна и та же зависимость требуется сразу нескольким компонентам, каждому из них предоставляется отдельный экземпляр. В листинге 8.11 показан метод, выполняющий разрешение контроллера в среде ASP.NET Core MVC.

Листинг 8.11. Разрешение Transient-экземпляров AspNetUserContextAdapter

```
private HomeController CreateHomeController()
{
    return new HomeController(
        new ProductService(
            new SqlProductRepository(this.connStr),
            new AspNetUserContextAdapter(),
            new SqlUserRepository(
                this.connStr,
                new AspNetUserContextAdapter())));
}
```

Зависимость IUserContext требуется обоим классам, ProductService и SqlUserRepository. Когда для AspNetUserContextAdapter выбирается жизненный цикл Transient, каждый потребитель получает собственный, личный экземпляр. Соответственно, ProductService получает один экземпляр, а SqlUserRepository — другой

Жизненный цикл Transient подразумевает, что каждый потребитель получает личный экземпляр зависимости, даже если несколько потребителей в одном и том же графе объектов имеют одинаковую зависимость (как в листинге 8.11). Если у многих потребителей имеется одна и та же зависимость, такой подход может быть неэффективным, но если реализация не является потокобезопасной, то более эффективный жизненный цикл Singleton не подойдет. В таких случаях может оказаться предпочтительным жизненный цикл Scoped.

8.3.3. Жизненный цикл Scoped

Пользователи веб-приложения хотят получать отклик от него как можно быстрее, даже когда в это же время к системе обращаются и другие пользователи. Не хотелось бы, чтобы наш запрос помещался в очередь наряду с запросами других пользователей. Если перед нашим запросом будет слишком много запросов других пользователей, то, возможно, дожидаться ответа придется очень долго. Для решения этой проблемы запросы в веб-приложениях обрабатываются в параллельном режиме. Инфраструктура среды ASP.NET Core скрывает это от нас, позволяя каждому запросу выполняться в собственном контексте и с собственными экземплярами контроллеров (если задействован ASP.NET Core MVC).

Поскольку обработка выполняется в параллельном режиме, зависимости, не обладающие потокобезопасностью, не могут использоваться в качестве одиночек с жизненным циклом Singleton. В то же время, если нужно организовать совместную работу с зависимостью различных потребителей в рамках одного и того же запроса, применение их с жизненным циклом Transient может стать неэффективным или даже проблематичным.

Хотя ядро ASP.NET Core выполняет отдельно взятый запрос в асинхронном режиме и в этот процесс обычно вовлекаются сразу несколько потоков, им гарантируется последовательное выполнение этого кода — по крайней мере при должном ожидании завершения асинхронных операций[1]. Это означает, что, если есть возмож-

[1] Потоки по-прежнему используются последовательно, один после другого, а не параллельно.

ность совместно использовать зависимости в рамках отдельно взятого запроса, потокобезопасность не станет проблемой. Более подробно порядок работы асинхронного многопоточного подхода в среде ASP.NET Core рассмотрен в подразделе 8.4.3.

Хотя концепция веб-запроса ограничивается веб-приложениями и веб-сервисами, концепция запроса имеет куда более широкое толкование. Запросы используются для выполнения отдельных операций во многих приложениях с длительным временем работы. Например, при создании сервисного приложения, выполняющего поэлементную обработку из очереди, каждый обрабатываемый элемент может быть представлен как отдельно взятый запрос, состоящий из собственного набора зависимостей.

То же самое можно сказать и о приложениях для настольных систем и смартфонов. Хотя верхние корневые типы (представления или `ViewModels`) потенциально могут жить довольно долго, щелчок на кнопке может рассматриваться в качестве запроса и область видимости этой операции можно ограничить, задав для нее изолированную сферу с собственным набором зависимостей. Таким образом и формируется понятие жизненного цикла Scoped, в рамках которого принимается решение о повторном использовании экземпляров в пределах заданной области видимости. Порядок работы жизненного цикла Scoped показан на рис. 8.9.

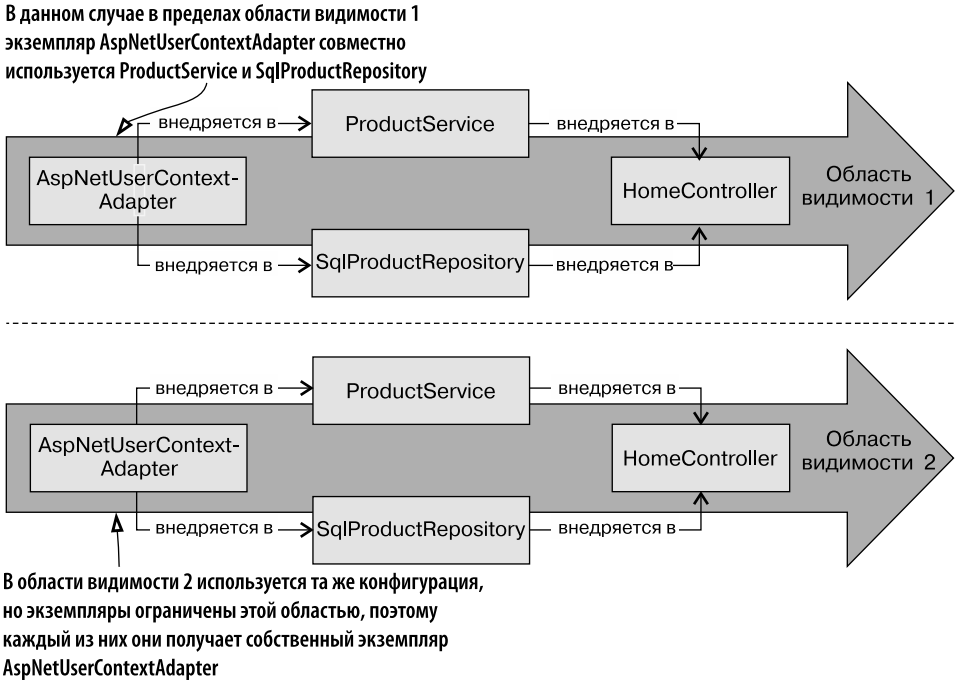


Рис. 8.9. Жизненный цикл Scoped показывает, что для каждой определенной области видимости создается как минимум один экземпляр зависимости

ОПРЕДЕЛЕНИЕ

> Зависимости с жизненным циклом Scoped ведут себя подобно Singleton-зависимостям в пределах одной четко определенной области видимости или одного запроса, но не используются совместно в пределах сразу нескольких областей. У каждой области видимости имеется собственный кэш связанных с ней зависимостей.

Следует отметить, что у DI-контейнеров могут быть специализированные версии поддержки жизненного цикла Scoped, ориентированные на конкретную технологию. Кроме того, любые ликвидируемые компоненты должны быть ликвидированы, как только область видимости окончится.

Когда следует использовать жизненный цикл Scoped

Применять жизненный цикл Scoped имеет смысл для приложений с длительным временем работы, предназначенных для операций обработки и нуждающихся в запуске в условиях некоторой изолированности. Изоляция требуется при выполнении операций в параллельном режиме или когда каждая операция содержит собственное состояние. Хорошим примером удачного применения жизненного цикла Scoped могут послужить веб-приложения, поскольку они обычно обрабатывают запросы в параллельном режиме и эти запросы, как правило, содержат какое-то изменяемое состояние, специфичное для них. Но, даже если веб-приложение запускает некие фоновые операции, не связанные с веб-запросом, ценность применения жизненного цикла Scoped не исчезает. Даже эти фоновые операции могут в большинстве своем быть отображены на концепцию запроса.

СОВЕТ

> Если когда-либо в веб-запросе понадобится сформировать объект DbContext среды Entity Framework Core, жизненный цикл Scoped подойдет для этого как нельзя лучше. Экземпляры DbContext не являются потокобезопасными, но обычно каждому веб-запросу требуется всего один экземпляр DbContext.

Все жизненные циклы допускают смешивание, и Scoped не исключение: например, некоторые зависимости настраиваются с жизненным циклом Singleton, а другие используются совместно несколькими потребителями в рамках одного запроса.

Пример: компоновка приложения с длительным временем выполнения, использующего зависимость dbContext, существующую в конкретной области видимости

Далее будет показано, как скомпоновать консольное приложение с длительным временем выполнения, имеющим зависимость `DbContext`, существование которой ограничено конкретной областью видимости. Это консольное приложение является версией программы UpdateCurrency, рассмотренной в разделе 7.1.

Как и программа UpdateCurrency, новое консольное приложение считывает обменные курсы валют. Но цель создания данной версии — выводить курсы обмена конкретной валюты раз в минуту и делать это до тех пор, пока пользователь не остановит приложение. Основные классы приложения показаны на рис. 8.10.

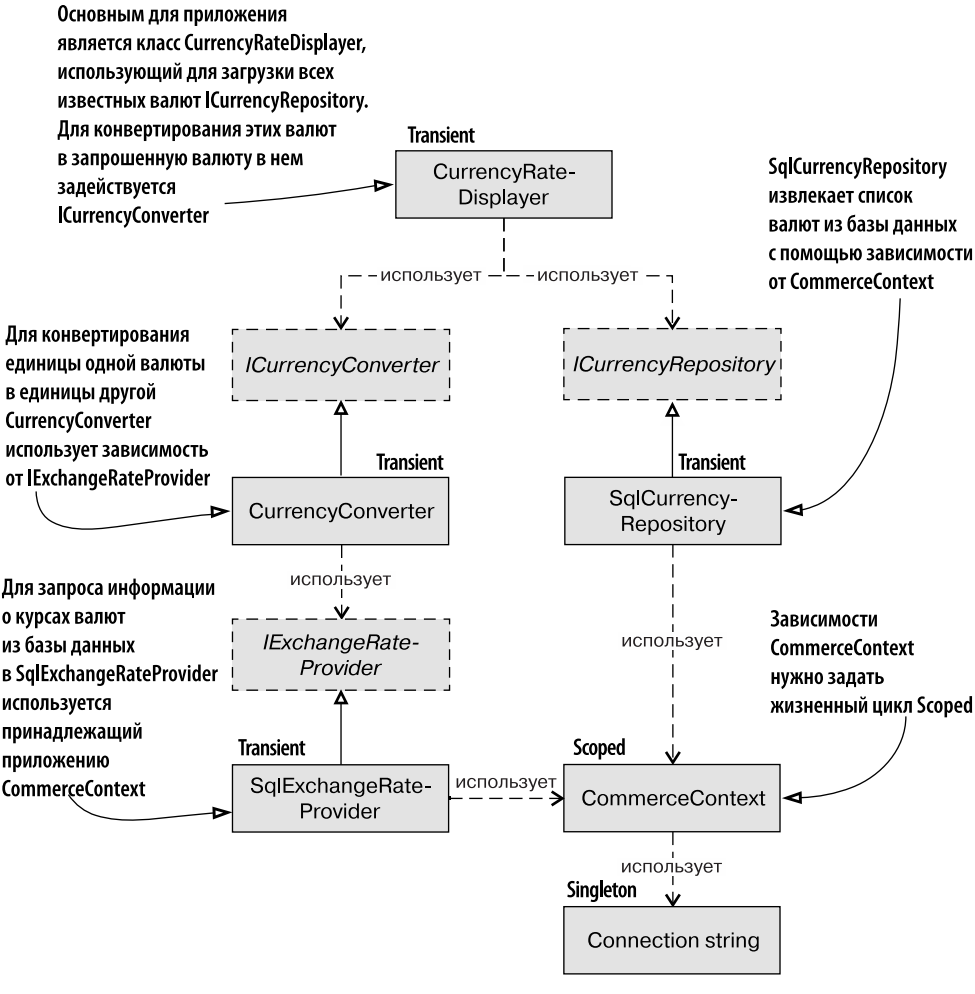


Рис. 8.10. Схема классов программы CurrencyMonitoring

В программе CurrencyMonitoring многократно используются зависимости `SqlExchangeRateProvider` и `CommerceContext` из программы UpdateCurrency, рассмотренной в главе 7, и абстракция `ICurrencyConverter`, о которой шла речь в главе 4. Новыми компонентами являются абстракция `ICurrencyRepository` и сопутствующая ей реализация. Кроме того, новый и специфичный для данной программы компонент — `CurrencyRateDisplayer`, его код показан в листинге 8.12.

Листинг 8.12. Класс CurrencyRateDisplayer

```
public class CurrencyRateDisplayer
{
    private readonly ICurrencyRepository repository;
    private readonly ICurrencyConverter converter;

    public CurrencyRateDisplayer(
        ICurrencyRepository repository,
        ICurrencyConverter converter)
    {
        this.repository = repository;
        this.converter = converter;
    }

    public void DisplayRatesFor(Money amount)
    {
        Console.WriteLine(
            "Exchange rates for {0} at {1}:",
            amount,
            DateTime.Now);

        IEnumerable<Currency> currencies =
            this.repository.GetAllCurrencies();

        foreach (Currency target in currencies)
        {
            Money rate = this.converter.Exchange(
                amount,
                target);

            Console.WriteLine(rate);
        }
    }
}
```

Загрузка всех известных валют

Вычисление обменных курсов для заданной суммы в целевой валюте

Вывод запрошенных обменных курсов на консоль

Приложение можно запустить из командной строки, взяв в качестве аргумента строковое значение `"EUR 1.00"`. В результате будет выведен следующий текст:

```
Exchange rates for EUR 1.00000 at 12/10/2018 22:55:00.
CAD 1.48864
USD 1.13636
DKK 7.46591
EUR 1.00000
GBP 0.89773
```

Для сборки приложения нужно будет создать корень композиции. В данном случае он состоит из двух классов (рис. 8.11).

Класс `Program` использует класс `Composer` для разрешения нового графа объектов приложения. В листинге 8.13 показан класс `Composer` со своим методом `CreateRateDisplayer`. Он обеспечивает создание при каждом разрешении всего лишь одного экземпляра зависимости `CommerceContext`, существующей в конкретной области видимости.

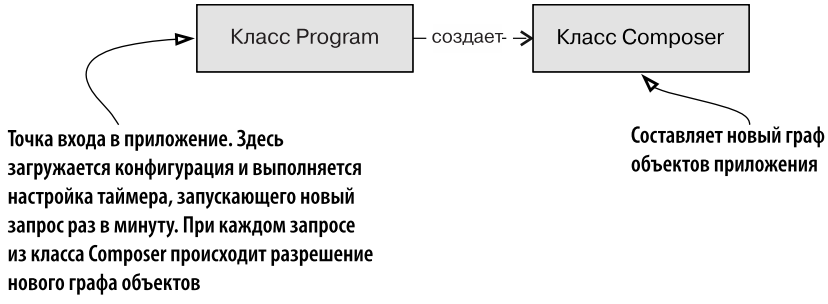


Рис. 8.11. Инфраструктура приложения содержит два класса, Program и Composer

Листинг 8.13. Класс Composer, ответственный за составление графа объектов

```
public class Composer
{
    private readonly string connectionString;

    public Composer(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public CurrencyRateDisplayer CreateRateDisplayer()
    {
        var context =
            new CommerceContext(this.connectionString);

        return new CurrencyRateDisplayer(
            new SqlCurrencyRepository(
                context),
            new CurrencyConverter(
                new SqlExchangeRateProvider(
                    context)));
    }
}
```

Сохранение полей для одиночек (Singleton-объектов). В данном случае это всего лишь строка подключения, но в типовом приложении экземпляров-одиночек будет больше

Публичный метод, позволяющий составлять корневой Transient-тип CurrencyRateDisplayer

Создание новых зависимостей с жизненным циклом Scoped

Внедрение Scoped-зависимостей в граф объектов с жизненным циклом Transient

Остальная часть корня композиции является точкой входа в приложение и представлена классом `Program`. Он отвечает за считывание входных аргументов и конфигурационного файла, а также настройку таймера, запускающего вывод обменных курсов раз в минуту. Полная картина представлена в листинге 8.14.

В классе `Program` происходит настройка таймера `Timer`, который по истечении определенного времени вызывает метод `DisplayRates`. Несмотря на то что в данном примере метод `DisplayRates` вызывается только раз в минуту, он может запросто вызываться в параллельном режиме сразу несколькими потоками и его даже можно заставить работать в асинхронном режиме. Код не утратит своей работоспособности, поскольку собственный набор экземпляров, существующий в конкретной области видимости, создается и управляется при каждом вызове, что позволяет каждой операции запускаться изолированно от других.

Листинг 8.14. Точка входа в приложение, управляющая областями видимости[1]

```
public static class Program
{
    private static Composer composer;

    public static void Main(string[] args)
    {
        var money = new Money(
            currency: new Currency(code: args[0]),
            amount: decimal.Parse(args[1]));

        composer = new Composer(LoadConnectionString());

        var timer = new Timer(interval: 60000);

        timer.Elapsed += (s, e) => DisplayRates(money);
        timer.Start();

        Console.WriteLine("Press any key to exit.");
        Console.ReadLine();
    }

    private static void DisplayRates(Money money)
    {
        CurrencyRateDisplayer displayer =
            composer.CreateRateDisplayer();

        displayer.DisplayRatesFor(money);
    }

    private static string LoadConnectionString() { ... }
}
```

Создание нового объекта Money на основе поступивших аргументов командной строки. Это сумма, для которой будут показаны обменные курсы

Создает объект System.Timers.Timer[1]. Запускает таймер раз в минуту. По истечении интервала вызывает метод DisplayRates

После запуска таймера программа ожидает пользовательского ввода и завершает работу, как только это произойдет

Запрос компоновщика для разрешения CurrencyRateDisplayer в рамках текущего запроса

ПРИМЕЧАНИЕ

В целях упрощения в предыдущем примере не показано высвобождение графа объектов. Данная концепция в контексте приложений среды ASP.NET Core MVC показана в листингах 8.7 и 8.8. Решение, работающее с консольными приложениями, будет выглядеть так же, поэтому мы предложим вам реализовать его в качестве упражнения.

Жизненный цикл Transient подразумевает, что каждый потребитель получает личный экземпляр зависимости, а жизненный цикл Scoped гарантирует получение одного и того же экземпляра всеми потребителями всех разрешенных графов конкретной области. Помимо обычных моделей жизненного цикла, таких как Singleton, Transient и Scoped, существуют также модели, которые можно отнести к проблемному коду или даже антипаттернам. Некоторые из этих весьма неудачных вариантов жизненного цикла рассматриваются в следующем разделе.

[1] Класс System.Timers.Timer является частью .NET Standard 2.0 и .NET Core 2.0.

8.4. Неудачный выбор жизненного цикла

Всем нам известно, что некоторые стили жизни вредят здоровью, один из них — курение. То же самое справедливо и для применения жизненных циклов в технологии DI. Здесь можно наделать кучу ошибок. В этом разделе будут рассмотрены варианты неудачного выбора, представленные в табл. 8.2.

Таблица 8.2. Неудачно выбранные жизненные циклы

Наименование	Тип	Описание
Захваченные зависимости (Captive Dependencies)	Ошибка	Сохраняет ссылки на зависимости дольше их ожидаемого времени жизни
Абстракции с протечкой (Leaky Abstractions)	Проблема проектирования	Использует абстракции с протечкой, допуская при этом утечку выбора жизненного цикла и делая его доступным потребителям
Жизненный цикл, привязанный к конкретному потоку (Per-thread Lifestyle)	Ошибка	Приводит к ошибкам при одновременно выполняемых вычислениях, привязывая экземпляры к времени жизни потока

Из табл. 8.2 следует, что захваченные зависимости (Captive Dependencies) и жизненный цикл, привязанный к конкретному потоку (Per-thread Lifestyle), могут стать источником ошибок в вашем приложении. Зачастую эти ошибки проявляются только после развертывания приложения для практического применения, поскольку они могут навредить одновременно выполняемыми вычислениями. Запущенное разработчиками приложение выполняется сравнительно недолго, обрабатывая один запрос за другим. Тот же самый режим используется и для тестирования, где происходит последовательный прогон приложения. Из-за этого подобные проблемы можно упустить из виду, и проявятся они только при одновременном обращении к приложению сразу нескольких пользователей.

Когда допускается утечка сведений о выбранном нами жизненном цикле и они становятся известны потребителям, ошибки обычно не появляются (по крайней мере, это случается не сразу). Но все же это усложняет как сам код потребителей зависимости, так и код его тестов и может повлечь за собой радикальные изменения всей кодовой базы. В результате возрастает вероятность возникновения ошибок.

8.4.1. Захваченные зависимости

При управлении временем жизни можно попасть в весьма распространенную западню, которая называется захваченной зависимостью. Она возникает, когда зависимость сохраняется потребителем дольше предполагаемого срока. Может даже получиться так, что такая зависимость будет одновременно использоваться несколькими потоками или запросами, даже если она не будет потокобезопасной.

ОПРЕДЕЛЕНИЕ

Захваченной считается зависимость, которая непреднамеренно слишком долго остается жизнеспособной из-за того, что ее потребителю задано более продолжительное время жизни по сравнению с ожидаемым временем жизни зависимости.

Широко распространенным примером захваченной зависимости может послужить внедрение кратковременной зависимости в потребителя-одиночку. Этот Singleton-потребитель существует на протяжении всего времени жизни компоновщика, что справедливо и для его зависимости. Проблема показана в листинге 8.15.

Листинг 8.15. Пример захваченной зависимости

```
public class Composer
{
    private readonly IProductRepository repository;

    public Composer(string connectionString)
    {
        this.repository = new SqlProductRepository(
            new CommerceContext(connectionString));
    }
    ...
}
```

Создание SqlProductRepository в качестве объекта-одиночки и его сохранение для повторного использования

Внедрение CommerceContext в объект-одиночку. Теперь объект CommerceContext становится захваченной зависимостью: он не является потокобезопасным и не предназначен для совместного применения сразу несколькими потоками

Так как для всего приложения имеется только один экземпляр `SqlProductRepository`, а ссылка на `CommerceContext` в `SqlProductRepository` хранится в его приватном поле, фактически `CommerceContext` будет также фигурировать только в одном экземпляре `SqlProductRepository`. В результате возникает проблема, связанная с тем, что объект `CommerceContext` не является потокобезопасным и не предназначен для того, чтобы пережить какой-то (по времени) отдельно взятый запрос. Поскольку захват `CommerceContext` объектом `SqlProductRepository` сохраняется и после ожидаемого времени его высвобождения, эта зависимость называется захваченной.

ВНИМАНИЕ

Компонент должен ссылаться только на те зависимости, ожидаемое время жизни которых равно времени жизни самого компонента или превышает его.

Проблемы захваченных зависимостей зачастую проявляются во время работы с DI-контейнером. Вызвано это динамичной природой DI-контейнеров, из-за чего можно легко упустить из виду форму выстраиваемого графа объектов. Но, как показывает предыдущий пример, проблема может возникнуть и в ходе работы с чистой технологией DI. Уменьшить ее вероятность можно за счет тщательного структурирования кода в корне композиции чистого внедрения зависимостей. Пример такого подхода показан в листинге 8.16.

Листинг 8.16. Смягчение проблемы захваченных зависимостей при использовании чистой технологии DI

```
public class CommerceControllerActivator : IControllerActivator
{
    private readonly string connStr;                    | Поля для хранения
    private readonly IUserContext userContext;          | объектов-одиночек

    public CommerceControllerActivator(string connectionString)
    {
        this.connStr = connectionString;                |
        this.userContext =                              |-- Создание объектов-одиночек
            new AspNetUserContextAdapter();             |
    }

    public object Create(ControllerContext ctx)
    {
        var context = new CommerceContext(this.connStr);    | Создание
        var provider = new SqlExchangeRateProvider(context); | Scoped-зависимостей


        Type type = ctx.ActionDescriptor
            .ControllerTypeInfo.AsType();

        if (type == typeof(HomeController))
        {
            return this.CreateHomeController(context);  <-- Предоставление
        }                                                   фабричным методам
        else if (type == typeof(ExchangeController))        созданных
        {                                                   Scoped-зависимостей
            return this.CreateExchangeController(
                context, provider);                     <--
        }
        else
        {
            throw new Exception("Unknown controller " + type.Name);
        }
    }

    private HomeController CreateHomeController(
        CommerceContext context)
    {
        return new HomeController(                  |
            new ProductService(                     |
                new SqlProductRepository(           |
                    context),                       |-- Построение графа объектов, содержащего
                this.userContext));                 |   Transient-, Scoped- и Singleton-экземпляры
    }

    private RouteController CreateExchangeController(
        CommerceContext context,
        IExchangeRateProvider provider) { ... }
}
```

ПРИМЕЧАНИЕ

Проблема захваченных зависимостей настолько часто возникает в ходе работы с DI-контейнером, что для обнаружения таких зависимостей в некоторых DI-контейнерах проводится анализ созданных графов объектов[1].

В коде листинга 8.16 создание всех зависимостей разбито на три фазы. При этом обнаружение и устранение захваченных зависимостей существенно упрощаются. Вот эти фазы:

- создание объектов-одиночек в ходе запуска приложения;
- создание Scoped-экземпляров в начале запроса;
- создание на основе запроса конкретного графа объектов, состоящих из Transient-, Scoped- и Singleton-экземпляров.

При использовании данной модели все имеющиеся в приложении Scoped-зависимости создаются для каждого запроса, даже если и не задействуются. При кажущейся неэффективности такого подхода следует помнить, что, как говорилось в подразделе 4.2.2, конструкторы компонентов должны быть свободны от всяческой логики, за исключением контрольных проверок и логики сохранения поступающих зависимостей. Тем самым конструкция становится быстродействующей и защищенной от большинства проблем снижения производительности, а создание небольшого количества неиспользуемых зависимостей — несерьезная проблема.

С точки зрения неверной конфигурации захваченные зависимости являются одной из наиболее распространенных и труднообнаруживаемых настроечных или программных ошибок, связанных с неудачным выбором жизненным циклом. Более часто, чем хотелось бы, мы тратили много времени на поиск ошибок, вызванных захваченными зависимостями. Именно поэтому очень высоко оцениваем инструментальную поддержку обнаружения захваченных зависимостей при использовании DI-контейнеров. Несмотря на то что захваченные зависимости обычно порождаются ошибками конфигурирования или программирования, есть и другие неудачно выбранные жизненные циклы, вызванные ошибками проектирования, например навязыванием вариантов жизненного цикла потребителю.

8.4.2. Применение абстракций с протечкой, допускающей доступность для потребителей сведений о выбранном жизненном цикле

Неудачный выбор жизненного цикла может быть связан и с необходимостью отложить создание зависимости. Если имеется редко востребуемая высокозатратная зависимость, ее экземпляр лучше создать в ходе выполнения приложения уже после того, как будет составлен граф объектов. В этом есть вполне определенный смысл.

---

[1] Все DI-контейнеры, рассматриваемые в этой книге, содержат функции обнаружения захваченных зависимостей.

Но тогда проблема перекладывается на плечи потребителей зависимости. Подобное поведение приводит к протечке к потребителю сведений о реализации корня композиции и выбранных в нем реализациях. Зависимость превращается в абстракцию с протечкой, нарушающую принцип инверсии зависимостей.

В этом разделе будут рассмотрены два широко распространенных примера того, как можно допустить протечку к потребителю зависимости сведений о выборе ее жизненного цикла. В обоих примерах реализовано одно и то же решение: вместо абстракции с протечкой создан класс-оболочка, скрывающий выбор жизненного цикла и реализующий исходную абстракцию.

Lazy<T> в качестве абстракции с протечкой

Вернемся еще раз к неоднократно использовавшемуся примеру `ProductService`, впервые приведенному в листинге 3.9. Представим, что создание одной из его зависимостей нам дорого обходится и не все варианты выполнения кода приложения требуют ее наличия.

При выборе решения можно склоняться к применению имеющегося в среде .NET класса `System.Lazy<T>`. Он позволяет обращаться к базовому значению через его свойство `Value`. Но это значение будет создано только при первом запросе. После этого `Lazy<T>` кэширует значение на весь период существования своего экземпляра.

Польза данного приема заключается в возможности отложить создание зависимостей. Но вскоре выяснится, что внедрение `Lazy<T>` непосредственно в конструктор потребителя — это ошибка. Пример ошибочного использования `Lazy<T>`, показан в листинге 8.17.

Листинг 8.17. Lazy<T> в качестве абстракции с протечкой

```
public class ProductService : IProductService
{
    private readonly IProductRepository repository;
    private readonly Lazy<IUserContext> userContext;

    public ProductService(
        IProductRepository repository,
        Lazy<IUserContext> userContext)
    {
        this.repository = repository;
        this.userContext = userContext;
    }

    public IEnumerable<DiscountedProduct> GetFeaturedProducts()
    {
        return
            from product in this.repository
                .GetFeaturedProducts()
            select product.ApplyDiscountFor(
                this.userContext.Value);
    }
}
```

Вместо того чтобы зависеть от IUserContext, теперь ProductService зависит от Lazy<IUserContext>. Таким образом, экземпляр IUserContext создается только по мере надобности. В этом нет ничего хорошего, поскольку Lazy<IUserContext> является абстракцией с утечкой

Свойство Value в Lazy<IUserContext> гарантирует, что зависимость IUserContext создается только один раз. Когда метод GetFeaturedProducts в IProductRepository возвращает пустой список, инструкция select не выполняется и свойство Value не будет вызываться, препятствуя созданию IUserContext

Структура корня композиции для `ProductService` из листинга 8.17, показана в листинге 8.18.

Листинг 8.18. Компоновка ProductService, имеющего зависимость от Lazy<IUserContext>

```
Lazy<IUserContext> lazyUserContext =
    new Lazy<IUserContext>(
        () => new AspNetUserContextAdapter())

new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        lazyUserContext));
```

Откладывает создание реальной зависимости AspNetUserContextAdapter путем заключения ее создания в Lazy<IUserContext>

Поскольку теперь ProductService зависит от Lazy<IUserContext>, а не от IUserContext, Lazy<IUserContext> внедряется непосредственно в его конструктор

После просмотра этого кода может возникнуть вопрос: а что, собственно, в нем плохо? Далее перечисляется ряд проблем, вызываемых такой конструкцией, но важно понимать, что в использовании `Lazy<T>` в корне композиции нет ничего плохого, однако внедрение `Lazy<T>` в компонент приложения приводит к возникновению абстракции с протечкой. Теперь вернемся к проблемам.

Во-первых, то, что потребитель может зависеть от `Lazy<IUserContext>`, усложняет код как самого потребителя, так и его модульных тестов. Можно подумать, что необходимость вызова `userContext.Value` — это всего лишь незначительная плата за возможность ленивой загрузки высокозатратной зависимости, но это не так. При создании модульных тестов приходится не только создавать экземпляры `Lazy<T>`, в которые заключается исходная зависимость, но и писать дополнительные тесты, чтобы проверить, не будет ли свойство `Value` вызываться в неподходящий момент.

Поскольку создание ленивой зависимости представляется довольно важным фактором оптимизации производительности, было бы странно не проверить правильность ее реализации. Для этого понадобится создать как минимум один дополнительный тест для каждого потребителя зависимости. А ведь у такой зависимости могут быть десятки потребителей, и всем им для проверки правильности понадобятся дополнительные тесты.

Во-вторых, замена существующей зависимости ленивой зависимостью позже в процессе разработки вызывает радикальные изменения во всем приложении. При наличии десятков потребителей этой зависимости такая замена может превратиться в серьезную проблему, поскольку, как уже говорилось, в изменении будет нуждаться не только код самих потребителей, но и код всех их тестов. Внесение подобных расходящихся как круги по воде изменений станет очень трудоемким и рискованным делом.

Чтобы избежать этого, можно изначально сделать ленивыми все зависимости, поскольку теоретически каждая зависимость может в будущем стать высокозатратной. Это способно застраховать от необходимости внесения любых каскадных изменений в будущем. Но это было бы чистым безумием, и мы надеемся, вы согласны с тем, что

это не самый лучший способ решения проблемы. Сказанное особенно справедливо, если учесть, что каждая зависимость, как вскоре выяснится, потенциально может превратиться в целый список реализаций. Тогда может появиться мысль изначально сделать все зависимости в формате `IEnumerable<Lazy<T>>`, что стало бы еще большим безрассудством.

И наконец, из-за большого объема необходимых изменений и значительного числа тестов, которые нужно добавить, становится очень трудно не наделать при программировании ошибок, которые сведут пользу от этих изменений к нулю. Например, если вновь созданный компонент случайно оказывается зависимым не от `Lazy<IUserContext>`, а от `IUserContext`, то каждый граф, содержащий его, всегда будет получать заранее загруженную реализацию `IUserContext`.

Это не означает, что создание ленивых зависимостей запрещено. Но хотелось бы повторить утверждение из подраздела 4.2.1: не стоит нагружать конструкторы компонентов какой-либо логикой, кроме граничных операторов и хранения входящих зависимостей. Тогда конструкция классов станет быстродействующей и надежной, к тому же такие компоненты не потребуют высоких затрат при создании экземпляров.

Однако в некоторых случаях у вас просто не будет выбора, например, когда дело коснется компонентов стороннего производителя, контроль над которыми у вас минимальный. В таком случае `Lazy<T>` пригодится как нельзя лучше. Но вместо того, чтобы допустить зависимость всех потребителей от `Lazy<T>`, эту абстракцию нужно спрятать за виртуальным заместителем (Virtual Proxy), поместив его в корень композиции[1]. Пример такого подхода показан в листинге 8.19.

Листинг 8.19. Виртуальный заместитель, используемый в качестве оболочки для Lazy<T>

```
public class LazyUserContextProxy : IUserContext
{
    private readonly Lazy<IuserContext> userContext;

    public LazyUserContextProxy(
        Lazy<IuserContext> userContext)
    {
        this.userContext = userContext;
    }

    public bool IsInRole(Role role)
    {
        IuserContext real = this.userContext.Value;
        return real.IsInRole(role);
    }
}
```

Реализация IUserContext

Зависит от Lazy<IUserContext>, что позволяет создавать IUserContext в ленивом режиме

Создание и вызов настоящей реализации IUserContext произойдут только при вызове принадлежащего заместителю метода IsInRole

[1] Дополнительные сведения о виртуальных заместителях есть в издании: *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 250.

Этот новый `LazyUserContextProxy` позволяет `ProductService` зависеть от `IUserContext`, а не от `Lazy<IUserContext>`. Теперь новый конструктор `ProductService` выглядит так:

```
public ProductService(
    IProductRepository repository,
    IUserContext userContext)
```

Порядок построения графа объектов для `HomeController` при внедрении `LazyUserContextProxy` в `ProductService` показан в листинге 8.20.

Листинг 8.20. Компоновка ProductService путем внедрения виртуального заместителя

```
IUserContext lazyProxy =
    new LazyUserContextProxy(
        new Lazy<IUserContext>(
            () => new AspNetUserContextAdapter()));

new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        lazyProxy));
```

Создание виртуального заместителя, который послужит оболочкой для Lazy<T>, позволяя создавать реальную зависимость ленивым способом

Поскольку в LazyUserContextProxy реализуется IUserContext, теперь имеется возможность позволить ProductService зависеть от IUserContext при внедрении LazyUserContextProxy в его конструктор

Как показывает код листинга 8.19, само по себе иметь класс, зависящий от `Lazy<T>`, не так уж и плохо, но все это нужно было бы сконцентрировать внутри корня композиции, чтобы был только один класс, получающий эту зависимость от `Lazy<IUserContext>`. Зависимость от `Func<T>` имеет практически такой же эффект, что и зависимость от `Lazy<T>`, и решение будет таким же. Это позволяет избежать усложнения кода, добавления модульных тестов, внесения радикальных изменений и появления досадных ошибок. Далее будет показано, что то же самое справедливо и для внедрения `IEnumerable<T>`.

IEnumerable<T> в качестве абстракции с протечкой

По аналогии с использованием `Lazy<T>` с целью отложенного создания зависимостей зачастую возникает необходимость работать с коллекцией зависимостей конкретной абстракции. Для этого можно воспользоваться одной из абстракций из коллекции BCL, например `IEnumerable<T>`. Хотя в применении `IEnumerable<T>` как таковом в качестве абстракции для представления коллекции зависимостей нет ничего плохого, ее использование в неподобающем месте может вызвать появление абстракции с протечкой. Пример неправильного применения `IEnumerable<T>` показан в листинге 8.21.

Листинг 8.21. IEnumerable<T> в качестве абстракции с протечкой

```
public class Component
{
    private readonly IEnumerable<ILogger> loggers;

    public Component(IEnumerable<ILogger> loggers)
    {
        this.loggers = loggers;
    }

    public void DoSomething()
    {
        foreach (var logger in this.loggers)
        {
            logger.Log("DoSomething called");
        }

        ...
    }
}
```

Внедрение коллекции зависимостей ILogger

Проход по коллекции и работа с зависимостями

ПРИМЕЧАНИЕ

Временно проигнорируем совет из подраздела 5.3.2, где утверждалось, что кодовую базу приложения не следует засорять регистрацией событий. Способы разработки приложений с учетом сквозной функциональности подробно рассматриваются в главе 10.

Хотелось бы, чтобы потребители не сталкивались с наличием нескольких экземпляров конкретной зависимости. Данная подробность реализации становится доступной благодаря протечке через зависимость `IEnumerable<ILogger>`. Как объяснялось ранее, у каждой зависимости может быть несколько реализаций, но потребители не должны знать об этом. Как и в предыдущем примере применения `Lazy<T>`, когда потребителей несколько, эта протечка усложняет систему и повышает затраты на ее сопровождение, поскольку перебор коллекции приходится выполнять каждому потребителю. То же самое касается и кода тестов.

Хотя опытные разработчики выдают конструкции `foreach`, подобные следующей, за считаные секунды, но все усложняется, когда изменяется порядок обработки коллекции зависимостей, например, при условии, что регистрация должна продолжаться даже при сбое одного из регистраторов:

```
foreach (var logger in this.loggers)
{
    try
    {
        logger.Log("DoSomething called");
    }
    catch
    {
    }
}
```

Пустая инструкция catch позволяет продолжить регистрацию в случае сбоя

Или, возможно, нужно не только продолжить обработку, но и зарегистрировать сбой в следующем регистраторе. В таком случае в качестве резервного для регистратора, допустившего сбой, послужит следующий регистратор:

```
for (int index = 0; index < this.loggers.Count; index++)
{
    try
    {
        this.loggers[index].Log("DoSomething called");
    }
    catch (Exception ex)
    {
        if (loggers.Count > index + 1)
        {
            loggers[index + 1].Log(ex);
        }
    }
}
```

Перенаправление вызова в базовую реализацию регистратора

Перенаправление исключения в резервный регистратор, которым является следующий регистратор, если таковой имеется в списке

Или, может быть... думаем, саму идею вы уяснили. Досадно было бы иметь подобные конструкции кода повсеместно. Тогда при желании изменить стратегию регистрации пришлось бы вносить каскадные изменения по всему приложению. В идеале хотелось бы сконцентрировать соответствующую информацию в одном месте.

Решить эту конструктивную проблему позволяет паттерн проектирования «Компоновщик» (Composite). Он должен быть вам уже знаком, поскольку рассматривался в главах 1 и 6 (см. рис. 1.8 и листинги 6.4 и 6.12). Компоновщик для `ILogger` показан в листинге 8.22.

Листинг 8.22. Упаковка IEnumerable<T> компоновщиком[1]

```
public class CompositeLogger : ILogger
{
    private readonly IList<ILogger> loggers;

    public CompositeLogger(IList<ILogger> loggers)
    {
        this.loggers = loggers;
    }

    public void Log(LogEntry entry)
    {
        for (int index = 0; index < this.loggers.Count; index++)
        {
            try
            {
                this.loggers[index].Log(entry);
            }
```

ILogger реализуется в CompositeLogger

CompositeLogger зависит от IList<ILogger>, чтобы позволить пересылку запросов на регистрацию всем доступным компонентам ILogger

Реализация принадлежащего ILogger метода Log. В данном случае предполагается, что ILogger содержит единственный метод, принимающий метод LogEntry

[1] Чтобы уяснить замысел определения ILogger с использованием простого метода Log, взгляните на следующий вопрос со StackOverflow: https://mng.bz/QgdG.

```
            catch (Exception ex)
            {
                if (loggers.Count > index + 1)
                {
                    var logger = loggers[index + 1];
                    logger.Log(new LogEntry(ex));
                }
            }
        }
    }
}
```

Упаковка исключения, выданного сбойным регистратором в объект LogEntry, чтобы его можно было передать резервному регистратору

В следующем фрагменте кода показан способ составления графа объектов для `Component` с использованием нового `CompositeLogger`, сохраняющего зависимость `Component` от единственного `ILogger` вместо `IEnumerable<ILogger>`:

```
Ilogger composite =
    new CompositeLogger(new ILogger[]
    {
        new SqlLogger(connectionString),
        new WindowsEventLogLogger(source: "MyApp"),
        new FileLogger(directory: "c:\\logs")
    });

new Component(composite);
```

Создает композицию с несколькими реализациями ILogger

Создает новый объект Component, используя Composite

Как было показано уже много раз, в качественной конструкции приложения соблюдается принцип инверсии зависимостей и не допускается применение абстракций с протечкой. В результате получается более понятный и легко сопровождаемый код с повышенной устойчивостью к ошибкам программирования. Рассмотрим еще один пример проблемного кода, который не влияет на конструкцию приложения как таковую, но может вызвать трудноустранимые проблемы одновременно выполняемых вычислений.

8.4.3. Ошибки одновременно выполняемых вычислений, вызываемые привязкой экземпляров к времени жизни потока

Временами приходится иметь дело с зависимостями, которые не обладают потокобезопасностью, но не испытывают необходимости быть привязанными к времени жизни запроса. Возникает соблазн синхронизировать время жизни такой зависимости с временем жизни потока. При всей привлекательности подобная практика приводит к ошибкам.

ВНИМАНИЕ

В некоторых DI-контейнерах этот метод называется жизненным циклом в рамках одного потока (Per-thread Lifestyle), и в них для него имеется встроенная поддержка. Но его использования нужно избегать!

Порядок применения зависимости `SqlExchangeRateProvider` в рассмотренном в листинге 7.2 методе `CreateCurrencyParser` показан в листинге 8.23. Для каждого потока, выполняющего приложение, эта зависимость создается однократно.

Листинг 8.23. Время жизни зависимости, привязанное к времени жизни потока

```
[ThreadStatic]
private static CommerceContext context;

static CurrencyParser CreateCurrencyParser(
    string connectionString)
{
    if (context == null)
    {
        context = new CommerceContext(
            connectionString);
    }

    return new CurrencyParser(
        new SqlExchangeRateProvider(context),
        context);
}
```

Статическое поле помечено атрибутом [ThreadStatic]. Среда CLR гарантирует, что такое поле не будет совместно использоваться сразу несколькими потоками. Вместо этого она предоставит каждому выполняемому потоку отдельный экземпляр поля. Если к полю обращаются из другого потока, в нем будет содержаться другое значение

Если поток, в котором выполняется текущий код, пока не имеет проинициализированного экземпляра CommerceContext, создается новый экземпляр. Он сохраняется в соответствующем статическом по отношению к потоку поле

Внедрение зависимости в рамках одного потока в граф объектов с жизненным циклом Transient

Кажущаяся безобидность всего этого далека от истины как ничто другое. Далее будут рассмотрены две проблемы, связанные с кодом этого листинга.

Время жизни потока крайне редко поддается определению

Предсказание времени жизни потока может стать очень трудной задачей. Когда поток создается и запускается с использованием кода `new Thread().Start()`, приобретается свежий блок принадлежащей потоку статической памяти. Это означает, что при вызове в таком потоке `CreateCurrencyParser` принадлежащие потоку статические поля не будут установлены, из-за чего будут созданы новые экземпляры.

Но при запуске потоков из пула потоков с помощью `ThreadPool.QueueUserWorkItem`, возможно, будет получен уже существующий или вновь созданный поток в зависимости от того, что в этом пуле имеется. Даже если вы сами не создаете потоки, этим может заняться среда, как уже говорилось по поводу среды ASP.NET Core. Это означает, что при всей кратковременности жизни некоторых потоков есть и такие, продолжительность жизни которых равна времени выполнения всего приложения. Все еще больше усложняется при отсутствии гарантии выполнения операций в одном потоке.

Асинхронные модели приложений, вызывающие проблемы многопоточности

Современные среды выполнения приложений по своей природе асинхронны. Даже если в вашем коде не реализуются новые асинхронные паттерны программирования, применяющие ключевые слова `async` и `await`, используемая среда может все же

решить, что запрос должен завершиться в потоке, отличном от того, в котором он был запущен. Например, среда ASP.NET всецело построена вокруг этой асинхронной модели программирования. Но даже среды с более солидным возрастом, такие как ASP.NET Web API и ASP.NET Web Forms, позволяют запросам запускаться в асинхронном режиме.

Для зависимостей, привязанных к конкретному потоку, это создает большую проблему. Когда запрос продолжается в другом потоке, он по-прежнему ссылается на те же самые зависимости, даже если некоторые из них привязаны к исходному потоку. Подобная ситуация проиллюстрирована на рис. 8.12.

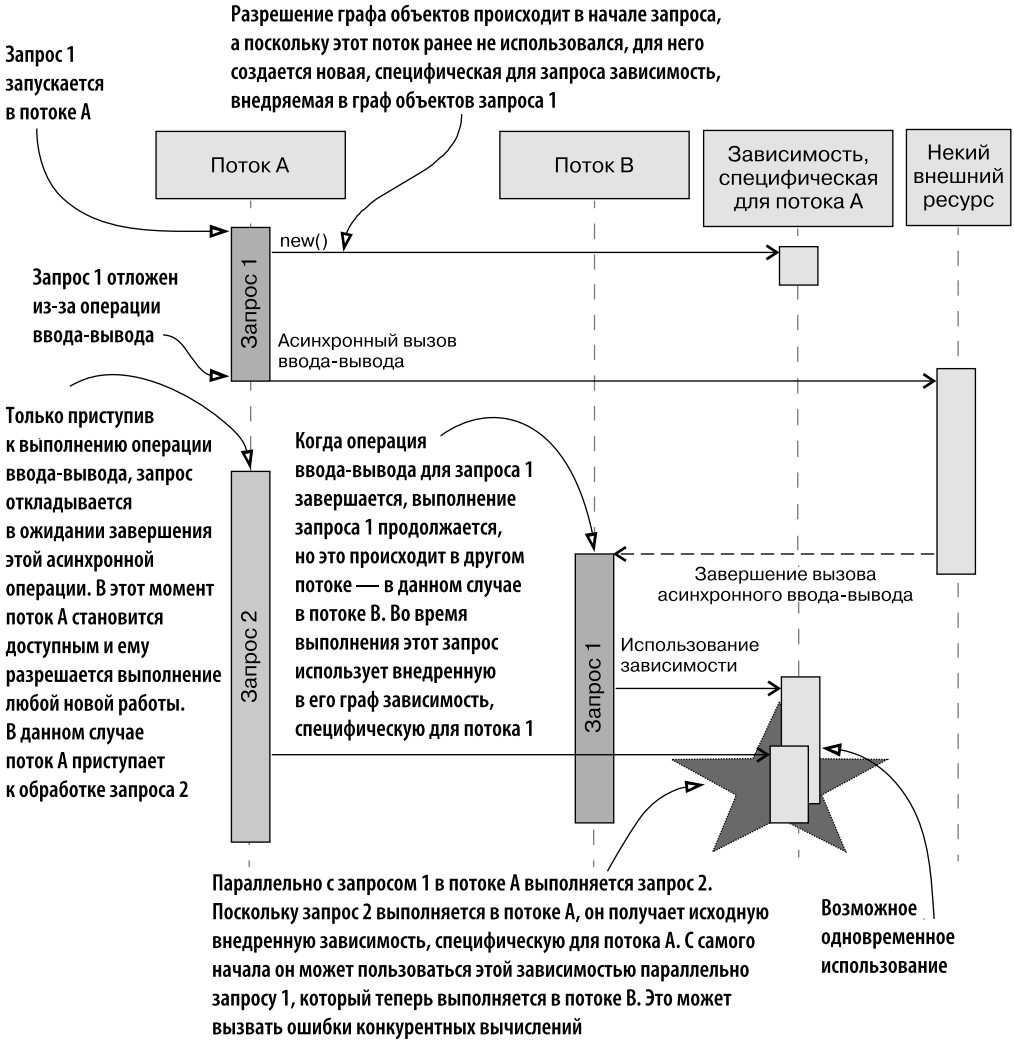


Рис. 8.12. Привязанные к конкретному потоку зависимости могут вызвать в асинхронной среде ошибки одновременно выполняемых вычислений

ПРИМЕЧАНИЕ

Граф объектов запроса 1, показанный на рис. 8.12, перемещается из одного потока в другой, а вот зависимость остается специфической для потока. Как только граф перемещается в другой поток, зависимость фактически становится захваченной.

Использование зависимостей, специфических для потока, при запуске в асинхронном контексте — крайне неудачная затея, поскольку может вызвать проблемы одновременно выполняемых вычислений, которые обычно трудно обнаруживаются и воспроизводятся. Такие проблемы могут возникать, только если специфическая для потока зависимость не является потокобезопасной, как чаще всего и бывает. В противном случае отлично сработал бы жизненный цикл Singleton.

Решением этой проблемы является ограничение области видимости рамками запроса или операции, и есть несколько способов достичь этой цели. Вместо привязки времени жизни зависимости к времени жизни потока нужно, как говорилось в разделе 8.3.3, привязать ее время жизни к периоду существования области видимости запроса. Это еще раз продемонстрировано в листинге 8.24.

Листинг 8.24. Сохранение зависимостей с определенной областью видимости в локальных переменных

```
static CurrencyParser CreateCurrencyParser(
    string connectionString)
{
    var context = new CommerceContext(
        connectionString);

    return new CurrencyParser(
        new SqlExchangeRateProvider(context),
        context);
}
```

Создание зависимости с жизненным циклом Scoped

Внедрение зависимости с жизненным циклом Scoped в граф объектов с жизненным циклом Transient

ПРИМЕЧАНИЕ

Данное решение может существенно сократить время жизни зависимости. Обычно при этом не возникает никаких проблем, но если такое все же случается, следует рассмотреть возможность объединения зависимостей в пул или же упаковать доступ к статической по отношению к потоку зависимости в заместитель, который станет обращаться к ней только из своего собственного метода. Это позволит предотвратить случайное перемещение зависимости из одного потока в другой. Мы предлагаем читателям в качестве упражнения создать соответствующий код.

Жизненные циклы, рассмотренные в этой главе, относятся к наиболее распространенным, но у вас могут возникнуть особые потребности, которые с их помощью удовлетворить невозможно. В подобной ситуации нужно сразу понять, что выбран неверный подход. Незначительные изменения конструкции позволят четко вписаться в стандартные паттерны.

Осознавать просчеты неприятно, но в результате принятых мер качество и сопровождаемость кода возрастут. Суть в том, что, если ощущается необходимость в реализации собственного жизненного цикла или создании абстракции с протечкой, сначала нужно серьезно пересмотреть созданную вами конструкцию. Поэтому мы решили не включать в эту книгу жизненные циклы, приспосабливаемые к конкретным условиям. Из материалов следующей главы станет ясно, что зачастую справляться с подобными ситуациями стоит перепроектированием или использованием перехватов.

Резюме

- Компоновщик — обобщенное понятие, относящееся к любому объекту или методу, формирующему зависимости. Он является важной частью корня композиции.
- Компоновщик может быть DI-контейнером, а также любым методом, составляющим графы объектов самостоятельно с использованием чистой технологии DI.
- Компоновщик сильнее влияет на продолжительность жизни зависимостей, чем любой отдельный потребитель. Компоновщик решает, когда создаются экземпляры. Выбирая, следует ли совместно использовать экземпляры, он определяет, выходит ли зависимость из области видимости с одним потребителем, или, прежде чем она может быть высвобождена, оттуда должны выйти все потребители.
- Жизненный цикл — формализованный способ описания предполагаемого времени жизни зависимости.
- Возможность четкой настройки жизненного цикла каждой зависимости играет важную роль в повышении производительности и может быть важна для обеспечения правильного поведения. Чтобы система работала должным образом, некоторые зависимости должны использоваться совместно сразу несколькими потребителями.
- Принцип подстановки Лисков утверждает, что у вас должна быть возможность подстановки абстракции для произвольной реализации без изменения надлежащей работы системы.
- Несоблюдение принципа подстановки Лисков приводит к нестабильности приложений из-за невозможности замены зависимостей, что может вызвать сбои в работе потребителя.
- Недолговечным и ликвидируемым является объект с четко определенным коротким временем жизни, обычно не превышающим продолжительности одного вызова метода.
- Реализация сервисов требует тщательной проработки и не допускает наличия ссылок на ликвидируемые зависимости, предполагая, что они создаются и ликвидируются по мере надобности. Это упрощает управление памятью, так как сервис может подвергаться сборке мусора подобно всем остальным объектам.

- Ответственность за ликвидацию зависимостей возлагается на компоновщик. Он лучше разбирается в моментах создания ликвидируемых экземпляров, а также знает, что экземпляр необходимо ликвидировать.
- Высвобождение — это процесс определения того, какие зависимости могут быть разыменованы и, возможно, ликвидированы. Сигнал на высвобождение разрешенной зависимости компоновщику подает корень композиции.
- Ответственность за правильный порядок ликвидации объектов возлагается на компоновщик. Объект может потребовать вызова своих зависимостей в ходе ликвидации, но если они уже были ликвидированы, возникают проблемы. Поэтому ликвидация должна происходить в порядке, обратном порядку создания объектов.
- Жизненный цикл Transient подразумевает возвращение нового экземпляра при каждом запросе зависимости. Каждый потребитель получает собственный экземпляр зависимости.
- При использовании жизненного цикла Singleton в области видимости отдельно взятого компоновщика будет только один экземпляр компонента. При каждом запросе этого компонента потребителю предоставляется один и тот же экземпляр.
- Зависимости с жизненным циклом Scoped в пределах одной четко определенной области видимости или одного запроса ведут себя подобно зависимостям-одиночкам, при этом их совместное применение в нескольких областях видимости полностью исключено. У каждой области видимости имеется собственный набор связанных с ней зависимостей.
- Жизненный цикл Scoped имеет смысл задействовать в приложениях с длительным временем работы, выполняющим операции обработки, которые требуется запускать в условиях изоляции. Эти условия нужны при параллельном выполнении таких операций или при наличии у каждой операции собственного состояния.
- Жизненный цикл Scoped как нельзя лучше подойдет и при условии, что вам когда-либо придется составлять в веб-запросе контекст базы данных `DbContext` из Entity Framework Core. Экземпляры `DbContext` не являются потокобезопасными, но обычно каждому веб-запросу требуется всего один экземпляр `DbContext`.
- Графы объектов могут состоять из зависимостей с различными жизненными циклами, но нужно гарантировать, что у потребителя будут только те зависимости, время жизни которых равно времени жизни самого потребителя или превышает его, поскольку потребитель будет держать свои зависимости в жизнеспособном состоянии. Невыполнение этого условия приводит к появлению захваченных зависимостей.
- Захваченной считается зависимость, которая слишком долго и непреднамеренно остается жизнеспособной по причине того, что для ее потребителя задано более продолжительное время жизни по сравнению с ожидаемым временем жизни зависимости.

- Зачастую захваченные зависимости вызывают ошибки в ходе работы с DI-контейнером, хотя проблема может возникать и во время работы с чистой технологией DI.
- Вероятность возникновения проблем при работе с чистой технологией DI может быть снижена за счет тщательно проработанной структуры корня композиции.
- Проблема захваченных зависимостей так часто возникает в ходе работы с DI-контейнером, что для обнаружения таких зависимостей в некоторых DI-контейнерах проводится анализ созданных графов объектов.
- Порой создание зависимости нужно отложить. Но внедрение зависимости в виде абстракции `Lazy<T>`, `Func<T>` или `IEnumerable<T>` для этого вряд ли подойдет, поскольку превратит зависимость в абстракцию с протечкой. Вместо этого информацию о зависимости нужно скрыть, применив заместитель или паттерн проектирования «Компоновщик».
- Время жизни зависимости не следует привязывать к времени жизни потока. Зачастую время жизни потока точному определению не поддается, и использование данного приема в асинхронной среде может нарушить работу приложения в многопоточной среде. Вместо этого стоит воспользоваться жизненным циклом Scoped или скрыть доступ к статическому по отношению к потоку значению, применив заместитель.

9 Перехват

В этой главе

- Перехват вызовов между двумя взаимодействующими объектами.
- Освоение паттерна проектирования «Декоратор».
- Применение сквозной функциональности с использованием декораторов.

Одной из самых привлекательных сторон кулинарии является возможность соединять множество ингредиентов, часть которых сами по себе не дают приятных вкусовых ощущений, в единое целое, имеющее более высокую вкусовую ценность, чем сумма его составляющих. Зачастую все начинается с простого ингредиента — основы блюда, — который дополняется и украшается, превращаясь в итоге в истинный деликатес.

Возьмем, к примеру, телячью отбивную. В исключительной ситуации ее можно съесть и в сыром виде, но в большинстве случаев предпочтительнее пожарить. Но если ее просто бросить на горячую сковороду, ничего хорошего не получится. Исправить ситуацию можно несколькими поэтапными действиями.

- Обжаривание котлеты на сливочном масле не даст мясу подгореть, но вкуса ей, скорее всего, не прибавит.
- Добавление соли усилит вкус мяса.
- Добавление других специй, например перца, сделает вкус более насыщенным.

- Панировочные сухари с добавлением соли и перца, в которых вы обваляете мясо, не только улучшат вкус, но и придадут исходному ингредиенту новую текстуру. На этом этапе вы приблизитесь к тому, что по-итальянски называется cotoletta[1].
- А то, что вы, прежде чем панировать мясо, поместите в сделанный в нем разрез ветчину, сыр и чеснок, вознесет результат на вершину кулинарного мастерства. У вас получится одно из самых изысканных блюд — телячий кордон блю.

Между жареной телячьей котлетой и телячьим кордон блю весьма существенная разница, но базовые ингредиенты одни и те же. Изменения вызваны используемыми добавками. Для создания нового блюда имеющуюся телячью котлету можно дополнить, не изменяя основные продукты.

Слабая связанность позволяет выполнять аналогичные действия и при разработке программных средств. Если программирование ведется на основе использования интерфейса, то базовая реализация может быть преобразована или улучшена путем ее заключения в другие реализации этого интерфейса. Общее представление о практическом применении данной технологии уже можно было получить, изучая код листинга 8.19, где с ее помощью модифицировалось время жизни высокозатратной зависимости путем ее заключения в виртуальный заместитель[2].

Этот подход можно обобщить, обеспечив возможность перехвата вызова сервиса потребителем. Именно этот вопрос и будет рассмотрен в данной главе.

Как в аналогии с телячьей котлетой, начнем с базового ингредиента и станем добавлять ингредиенты, улучшая исходный, но не изменяя ядро того, что было в самом начале. Перехват является одной из самых эффективных возможностей, извлекаемой из слабого связывания. Он позволяет без особого труда применять принцип единственной ответственности и разделения обязанностей.

В предыдущих главах на перевод кода в положение истинной слабой связанности затрачивалась масса усилий. В данной главе мы начнем извлекать выгоду из этих затрат. Общая структура главы весьма заурядна. Начнем с общих сведений о перехвате, включая пример. Затем перейдем к разговору о сквозной функциональности. В этой главе мало теории и много примеров, поэтому, если тема вам уже знакома, можете смело переходить к изучению главы 10, посвященной аспектно-ориентированному программированию.

ОПРЕДЕЛЕНИЕ

Сквозная функциональность — это аспекты программы, затрагивающие основную часть приложения. Зачастую они являются нефункциональными требованиями. Они не имеют прямого отношения к какой-либо конкретной функции и обычно применяются к уже имеющимся функциональным возможностям.

---

[1] Cotoletta — итальянская отбивная котлета в панировке.

[2] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 250.

Освоив главу, вы сможете воспользоваться перехватом для разработки кода со слабой связанностью с помощью паттерна проектирования «Декоратор». У вас должна появиться возможность успешно соблюдать принцип разделения обязанностей и применять сквозную функциональность, сохраняя при этом хорошее состояние кода.

Глава начинается с основного вводного примера и развивается в направлении усвоения все более сложных понятий и примеров. Заключительное и наиболее развитое понятие может быть кратко объяснено и без примеров. Но, поскольку его смысл можно полностью раскрыть только на конкретном примере, глава завершается всесторонней многостраничной демонстрацией того, как оно работает. Однако, чтобы добраться до него, нужно пройти все с самого начала, то есть усвоить введение в перехват.

9.1. Введение в перехват

В понятии перехвата нет ничего сложного: нам нужна возможность перехвата вызова, проходящего от потребителя к сервису, и выполнения какого-нибудь кода до или после вызова сервиса. Это нужно сделать таким образом, чтобы не принуждать к изменениям ни потребителя, ни сервис.

ОПРЕДЕЛЕНИЕ

> Перехват — это возможность перехвата вызовов между двумя взаимодействующими компонентами, позволяющая обогатить или изменить поведение зависимости, не внося изменений в сами взаимодействующие компоненты.

Представим, к примеру, что к классу `SqlProductRepository` нужно добавить проверки, связанные с соблюдением мер безопасности. Можно, конечно, изменить для этого сам класс `SqlProductRepository` или код потребителя, но с применением перехвата проверки приобретут характер перехвата вызовов `SqlProductRepository`, для чего будет использоваться некий промежуточный фрагмент кода. На рис. 9.1 показан обычный вызов сервиса потребителем, перехватываемый посредником, способным выполнить свой собственный код до или после передачи вызова реальному сервису.

ВНИМАНИЕ

> Инструментом перехвата является набор принципов и паттернов проектирования программных продуктов, связанных с технологией DI (сюда входят слабое связывание, принцип подстановки Лисков, и это далеко не полный перечень). Без этих принципов и паттернов перехват невозможен.

В этом разделе состоится знакомство с перехватом. Читателям требуется усвоить, что, по сути, он является применением паттерна проектирования «Декоратор». Ничего страшного, если представление об этом паттерне слегка стерлось из памяти, поскольку разговор начнется с его описания. Когда он завершится, у вас должно

сложиться довольно полное понимание порядка работы декораторов. Для начала будет приведен простой пример с демонстрацией паттерна, а затем рассмотрена связь перехвата с паттерном «Декоратор».

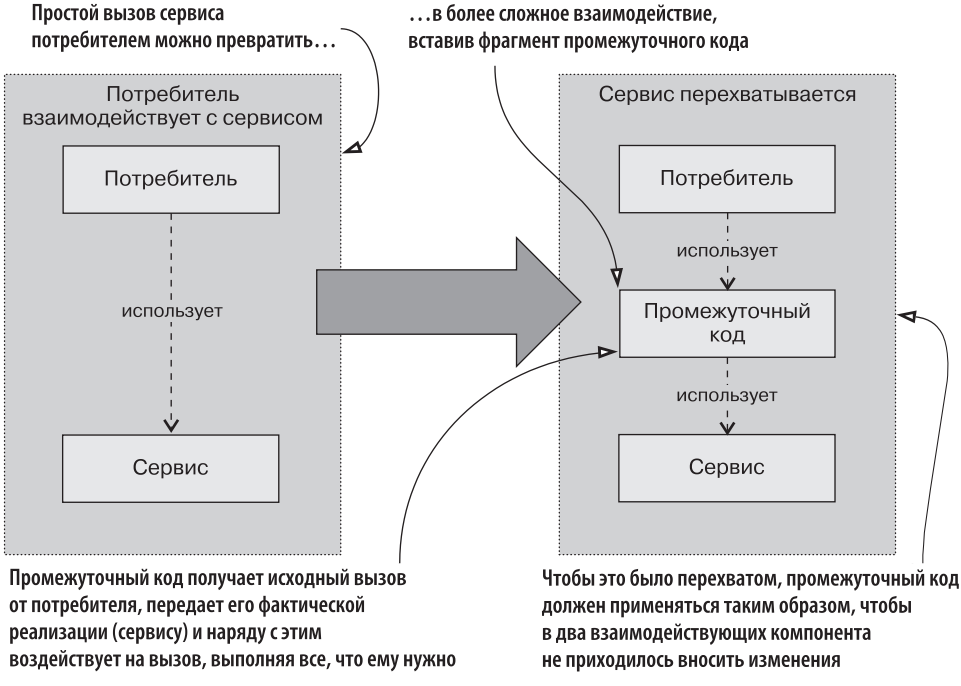


Рис. 9.1. Суть перехвата

9.1.1. Паттерн проектирования «Декоратор»

Как и многие другие его соратники, «Декоратор» является весьма почтенным и подробно рассмотренным паттерном проектирования, опередившим появление технологии DI на десять лет. Поскольку его роль в технологии перехвата ведущая, требуется освежить наши знания.

Описание паттерна «Декоратор» появилось в книге Design Patterns: Elements of Reusable Object-Oriented Software, созданной коллективом авторов под началом Эриха Гаммы (Erich Gamma), которая вышла в издательстве Addison-Wesley в 1994 году: «Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности»[1].

На рис. 9.2 показано, что «Декоратор» работает за счет заключения одной реализации абстракции в другую реализацию той же самой абстракции. Созданная таким образом оболочка делегирует операции содержащейся в ней реализации,

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 210.

добавляя определенное поведение до и/или после вызова заключенного в оболочку объекта.

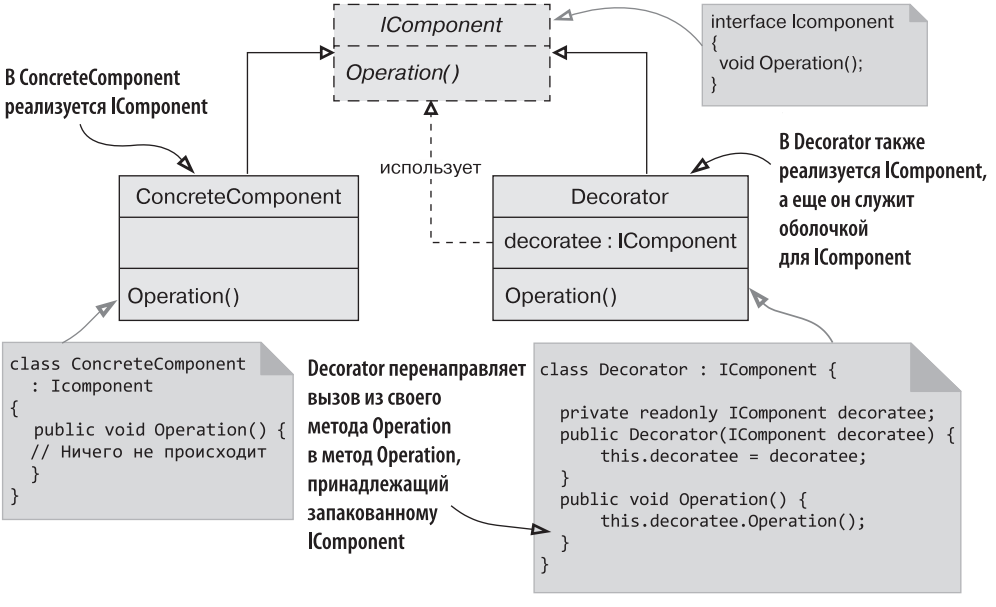


Рис. 9.2. Общая структура паттерна «Декоратор»

Способность динамически прикреплять обязанности означает, что можно принять решение о применении декоратора в ходе работы программы, не закладывая эту функциональную зависимость в программу при компиляции, как делалось бы при использовании подклассов.

В декоратор можно заключить еще один декоратор, который может послужить оболочкой для еще одного декоратора и т. д., создавая тем самым конвейер перехватов (рис. 9.3). В основе должна быть автономная реализация, выполняющая желаемую работу.

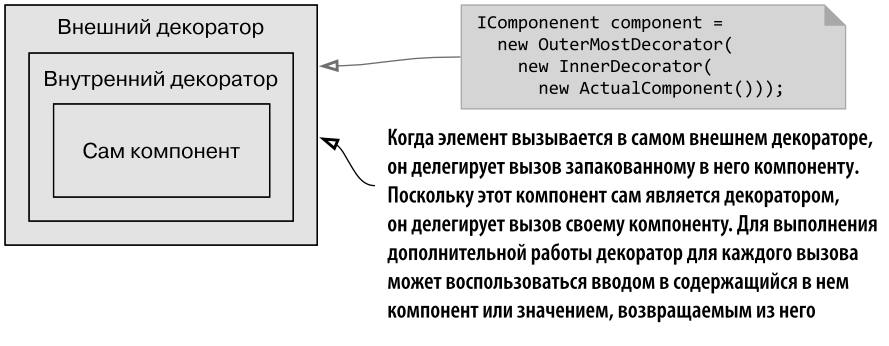


Рис. 9.3. Декоратор, подобно матрешке, включает в себя другой декоратор, в котором хранится автономный компонент

Предположим, к примеру, что имеется абстракция под названием `IGreeter`, содержащая метод `Greet`:

```
public interface IGreeter
{
    string Greet(string name);
}
```

Для нее можно создать простую реализацию, выдающую формальное приветствие:

```
public class FormalGreeter : IGreeter
{
    public string Greet(string name)
    {
        return "Hello, " + name + ".";
    }
}
```

Самая простая реализация декоратора делегирует вызов декорируемому объекту без каких-либо действий:

```
public class SimpleDecorator : IGreeter
{
    private readonly IGreeter decoratee;

    public SimpleDecorator(IGreeter decoratee)
    {
        this.decoratee = decoratee;
    }

    public string Greet(string name)
    {
        return this.decoratee.Greet(name);
    }
}
```

Декоратор заключает в себя компонент той же абстракции, которую он реализует

Вызов метода Greet передается декорируемому компоненту без каких-либо изменений; его возвращаемое значение также возвращается напрямую

На рис. 9.4 показаны отношения между `IGreeter`, `FormalGreeter` и `SimpleDecorator`. Поскольку `SimpleDecorator` не делает ничего, кроме перенаправления вызова, толку от него никакого. А по-настоящему декоратор мог бы использоваться для изменения входных данных до делегирования вызова.

ПРИМЕЧАНИЕ

В приводимых далее примерах кода основное внимание будет уделяться методу Greet, поскольку остальной код декоратора изменяться не будет.

Рассмотрим метод `Greet` класса `TitledGreeterDecorator`:

```
public string Greet(string name)
{
    string titledName = "Mr. " + name;
    return this.decoratee.Greet(titledName);
}
```

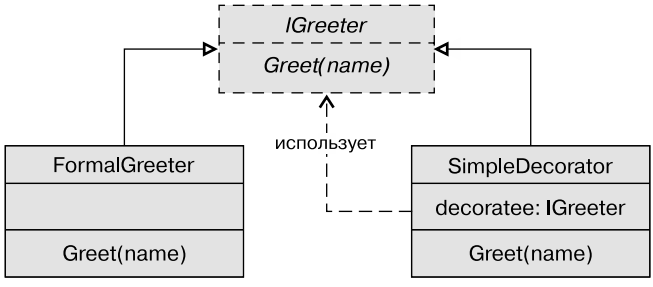


Рис. 9.4. IGreeter реализуется как в SimpleDecorator, так и в FormalGreeter, но SimpleDecorator служит оболочкой для IGreeter и перенаправляет все вызовы своего метода Greet в задекорированный метод Greet

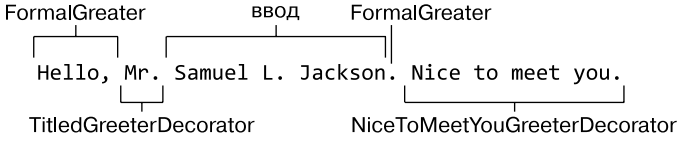
В аналогичной ситуации в декораторе при создании `NiceToMeetYouGreeterDecorator` может быть принято решение об изменении возвращаемого значения до его возвращения:

```
public string Greet(string name)
{
    string greet = this.decoratee.Greet(name);
    return greet + " Nice to meet you.";
}
```

При наличии двух предыдущих примеров можно первый пример заключить во второй и составить комбинацию, изменяющую как ввод, так и вывод:

```
IGreeter greeter =
    new NiceToMeetYouGreeterDecorator(
        new TitledGreeterDecorator(
            new FormalGreeter()));
string greet = greeter.Greet("Samuel L. Jackson");
Console.WriteLine(greet);
```

На выходе получается следующий результат:



В декораторе также может быть принято решение не вызывать основную реализацию:

```
public string Greet(string name)
{
    if (name == null)
    {
        return "Hello world!";
    }

    return this.decoratee.Greet(name);
}
```

Контрольная инструкция обеспечивает при нулевом вводе поведение по умолчанию, при этом запакованный компонент не вызывается

Отказ от вызова основной реализации влечет более заметные последствия, нежели делегирование вызова. Хотя в игнорировании декорируемой функции нет ничего предосудительного, теперь декоратор заменяет, а не расширяет исходное поведение[1]. Более распространенным является сценарий остановки выполнения путем выдачи исключения, рассматриваемый в подразделе 9.2.3.

От других классов, содержащих зависимости, декоратор отличает реализация заключенным в него объектом той же самой абстракции, которая реализована в самом паттерне. Это позволяет компоновщику заменять исходный компонент декоратором без изменения потребителя. Запакованный объект в декораторе часто объявляется как абстрактный тип — декоратор оборачивает интерфейс, а не специализированную конкретную реализацию. В этом случае декоратор должен придерживаться принципа подстановки Лисков и одинаково относиться ко всем задекорированным объектам.

Вот, собственно, и все. Больше о паттерне «Декоратор» сказать практически нечего. Вам уже приходилось видеть его в действии в некоторых местах этой книги. Так, код примера `SecureMessageWriter` в подразделе 1.2.2 является декоратором. Сейчас рассмотрим конкретный пример применения декоратора для реализации сквозной функциональности.

9.1.2. Пример: реализация ведения контрольного журнала с помощью декоратора

В этом примере будет еще раз реализовано ведение контрольного журнала для `IUserRepository`. Если помните, эта тема уже рассматривалась в разделе 6.3, где ведение контрольного журнала использовалось в качестве примера при объяснении способа устранения зацикленности зависимостей. При ведении этого журнала записываются все важные действия, выполняемые в системе пользователями, чтобы потом их проанализировать.

Ведение контрольного журнала является типичным примером сквозной функциональности: оно может потребоваться, но не должно влиять на основную функциональность для чтения и редактирования данных о пользователях. Именно так и делалось в разделе 6.3. Поскольку интерфейс `IAuditTrailAppender` был внедрен в сам класс `SqlUserRepository`, мы принудительно поставили его в известность о контроле и заставили реализовать его. Тем самым был нарушен принцип единственной ответственности, который предписывает не разрешать реализацию контроля в `SqlUserRepository`, из чего можно сделать вывод, что декоратор послужит более подходящей альтернативой.

---

[1] Можно возразить, что это уже применение не паттерна «Декоратор», а другого паттерна — «Цепочки обязанностей» (Chain of Responsibility) (*Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 263). Дело в том, что эти два паттерна тесно связаны. Чтобы не усложнять повествование, мы используем название «декоратор».

Реализация контроля для хранилища пользователей с применением декоратора

Контроль можно реализовать с помощью декоратора путем ввода нового класса `AuditingUserRepositoryDecorator`, заключающего в себе еще один `IUserRepository` и реализующего ведение контрольного журнала. Взаимоотношения типов показаны на рис. 9.5.

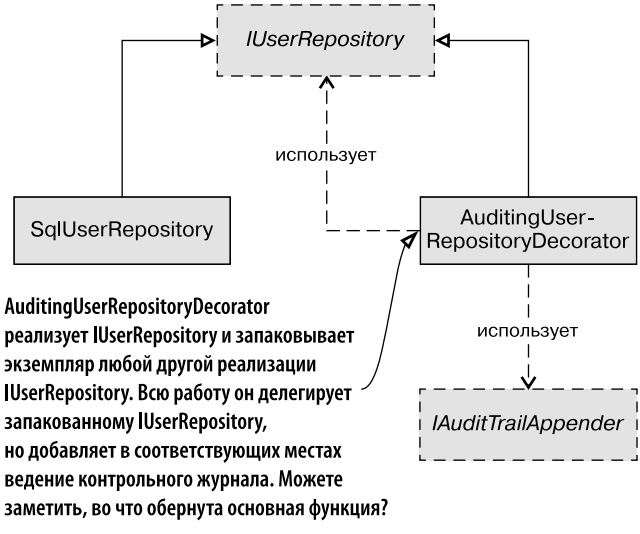


Рис. 9.5. AuditingUserRepositoryDecorator добавляет контроль к любой реализации IUserRepository

Кроме декорированного `IUserRepository` класс `AuditingUserRepositoryDecorator` нуждается в сервисе, ведущем контрольный журнал. В качестве его можно взять `IAuditTrailAppender`, рассмотренный в разделе 6.3. Реализация сервиса показана в листинге 9.1.

Листинг 9.1. Объявление AuditingUserRepositoryDecorator

```
public class AuditingUserRepositoryDecorator
    : IUserRepository
{
    private readonly IAuditTrailAppender appender;
    private readonly IUserRepository decoratee;

    public AuditingProductRepository(
        IAuditTrailAppender appender,
        IUserRepository decoratee)
    {
        this.appender = appender;
        this.decoratee = decoratee;
    }

    ...
}
```

Реализует и декорирует IUserRepository

В AuditingUserRepositoryDecorator реализуется та же абстракция, которая им декорируется. Для запроса интерфейса IUserRepository, который может быть запакован этим классом и которому он может делегировать свою основную реализацию, применяется стандартное внедрение через конструктор. Кроме задекорированного хранилища класс также запрашивает интерфейс IAuditTrailAppender, которым может воспользоваться для ведения контрольного журнала операций, реализованных задекорированным хранилищем. Примеры реализации двух методов в AuditingUserRepositoryDecorator показаны в листинге 9.2.

Листинг 9.2. Реализация AuditingUserRepositoryDecorator

```
public User GetById(Guid id)
{
    return this.decoratee.GetById(id);
}

public void Update(User user)
{
    this.decoratee.Update(user);
    this.appender.Append(user);
}
```

Пропуск контроля для операций чтения

Операция записи, задекорированная ведением контрольного журнала

В контроле нуждаются не все операции. Обычно требуется контролировать все операции создания, обновления и удаления, а операции чтения можно проигнорировать. Поскольку метод GetById является чистой операцией чтения, вызов делегируется задекорированному хранилищу и результат тут же возвращается. В то же время метод Update должен попасть под контроль. Реализация по-прежнему делегируется задекорированному хранилищу, но после того, как метод, которому делегирован вызов, успешно возвратит результат, будет задействован IAuditTrailAppender, выполняющий контрольную запись операции в журнал.

Такой декоратор, как AuditingUserRepositoryDecorator, подобен обвалке телячьей котлеты в панировке: она дополняет исходный ингредиент, не модифицируя его. Сама по себе панировка — это не просто пустая оболочка, она имеет собственный список ингредиентов. Как настоящая панировка получается из панировочных сухарей и специй, так и AuditingUserRepositoryDecorator содержит IAuditTrailAppender.

Следует отметить, что внедренный IAuditTrailAppender сам является абстракцией, следовательно, реализации могут варьироваться независимо от класса AuditingUserRepositoryDecorator, который всего лишь координирует действия задекорированных IUserRepository и IAuditTrailAppender. Можно создать любую подходящую вам реализацию IAuditTrailAppender, но в листинге 6.24 мы выбрали реализацию, основанную на использовании Entity Framework. Теперь посмотрим, как можно подключить все нужные зависимости, чтобы сделать код работоспособным.

Компоновка AuditingUserRepositoryDecorator

В главе 8 были приведены несколько примеров компоновки экземпляра HomeController. В листинге 8.11 представлена простая реализация экземпляров с жизненным циклом Transient. Листинг 9.3 показывает, как скомпоновать этот HomeController, используя задекорированное хранилище SqlUserRepository.

Листинг 9.3. Компоновка декоратора

```
private HomeController CreateHomeController()
{
    var context = new CommerceContext();

    IAuditTrailAppender appender =
        new SqlAuditTrailAppender(
            this.userContext,
            context);

    IUserRepository userRepository =
        new AuditingUserRepositoryDecorator(
            appender,
            new SqlUserRepository(context));

    IProductService productService =
        new ProductService(
            new SqlProductRepository(context),
            this.userContext,
            userRepository);

    return new HomeController(productService);
}
```

Создание нового экземпляра SqlUserRepository. Внедрение в экземпляр декоратора как SqlUserRepository, так и основанной на использовании SQL Server реализации IAuditTrailAppender. Как SqlUserRepository, так и AuditingUserRepositoryDecorator являются экземплярами IUserRepository

Вместо непосредственного внедрения SqlUserRepository в экземпляр ProductService внедрение декоратора, который запаковывает SqlUserRepository. ProductService видит только интерфейс IUserRepository и ничего не знает ни о SqlUserRepository, ни о декораторе

ВНИМАНИЕ

В листинге 9.3 показан упрощенный пример, игнорирующий все, что связано с временем жизни. Поскольку CommerceContext является ликвидируемым типом, этот код может вызвать утечку ресурсов. Более корректная реализация была бы интерполяцией листинга 9.3 с моделью, рассмотренной в разделе 8.3.3, но мы уверены: вы поймете, что на данном этапе код за счет этого начнет усложняться.

Следует отметить, что здесь получилось добавить поведение в `IUserRepository`, не изменяя исходный код существующих классов. Для добавления функции ведения контрольного журнала изменять код `SqlUserRepository` не приходится. Вспомним, что в подразделе 4.4.2 эту положительную особенность мы назвали принципом открытости-закрытости.

Теперь, после изучения примера перехвата обращения к конкретному хранилищу `SqlUserRepository` с помощью его декорирования с применением `AuditingUserRepositoryDecorator`, обратим внимание на создание понятного и легко сопровождаемого кода, не теряющего этих свойств в условиях противоречивых или изменяющихся требований и предназначенного для реализации сквозной функциональности.

9.2. Реализация сквозной функциональности

Большинство приложений должны позаботиться об аспектах, не имеющих прямого отношения к какой-либо конкретной функции, — они решают более широкий круг задач. Как правило, этим занимается множество разобщенных областей кода, которые могут находиться даже в разных модулях или на разных уровнях. Поскольку

речь пойдет о пересечении довольно широкой области кодовой базы, назовем это сквозной функциональностью (Cross-Cutting Concerns). Список соответствующих примеров приведен в табл. 9.1 (он неисчерпывающий и дан просто для наглядности).

Таблица 9.1. Типовые примеры сквозной функциональности

Аспект	Описание
Ведение контрольного журнала	Любая операция, вносящая изменения в данные, должна заноситься в контрольный журнал. В записи должны присутствовать метка времени, идентификационные данные пользователя, который внес изменения, и информация о том, что именно изменилось. Пример приведен в подразделе 9.1.2
Регистрация событий	Регистрация событий немного отличается от ведения контрольного журнала и концентрируется на записях событий, отражающих состояние приложения. Это могут быть события, интересующие IT-персонал, но могут быть и связанные с ведением бизнеса
Мониторинг производительности	Немного отличается от регистрации событий, поскольку имеет дело в основном с записью данных о производительности, а не о конкретных событиях. Если имеются соглашения об уровне обслуживания (Service Level Agreements, SLAs), которые не могут отслеживаться с помощью стандартной инфраструктуры, нужно реализовать собственный мониторинг производительности. Для него можно выбрать настраиваемые счетчики производительности Windows, но все равно потребуется добавить код, захватывающий данные
Проверка допустимости	Обычно операции должны вызываться с допустимыми данными. Это может быть либо простая проверка допустимости данных, введенных пользователем, либо проверка более сложного бизнес-правила. Хотя сама проверка всегда зависит от контекста, ее вызов и обработка результатов зачастую от контекста не зависят и могут рассматриваться как сквозная функциональность
Обеспечение безопасности	Проведение некоторых операций может разрешаться только конкретным пользователям. Зачастую это основывается на том, что они занимают конкретные должности или принадлежат к определенным группам и на вас возлагается обеспечение безопасности
Кэширование	Реализация кэширования может повысить производительность, но кэшировать конкретные компоненты доступа к данным не имеет никакого смысла. Поэтому для различных реализаций доступа к данным может потребоваться включение или отключение кэширования
Обработка ошибок	Приложению может понадобиться обработка конкретных исключений и регистрация их выдачи, а также их преобразование или вывод пользователю сообщения. Для работы с ошибками в нужном ключе можно воспользоваться декоратором, занимающимся обработкой ошибок
Обеспечение устойчивости к сбоям	Временами внешние ресурсы становятся недоступными. Реляционным базам данных, чтобы не допустить повреждения данных, необходимо обрабатывать транзакционные операции, что может привести к взаимным блокировкам. Чтобы справиться с возникающей проблемой, можно воспользоваться декоратором, который позволит реализовывать паттерны обеспечения устойчивости к сбоям, например паттерн «Предохранитель»

При отрисовке архитектуры многоуровневого приложения сквозную функциональность обычно изображают в виде вертикальных прямоугольников, расположенных рядом с уровнями. Пример такой схемы показан на рис. 9.6.

В этом разделе будет рассмотрен ряд примеров, иллюстрирующих порядок использования перехвата в виде декоратора для организации сквозной функциональности. Выберем из табл. 9.1 аспекты обеспечения устойчивости к сбоям, обработки ошибок и обеспечения безопасности. Как и в случае с многими другими понятиями, проще будет разобраться в абстрактном представлении перехвата, но, как известно, дьявол кроется в деталях. Чтобы вы могли хорошо освоить технику перехвата, в этом разделе рассматривается три примера. По завершении их разбора у вас должно сложиться более четкое представление о том, что такое перехват и как с ним нужно работать. Поскольку в подразделе 9.1.2 мы уже рассмотрели вводный пример, обратимся к более сложным случаям, показывающим порядок применения перехвата с логикой произвольной сложности.

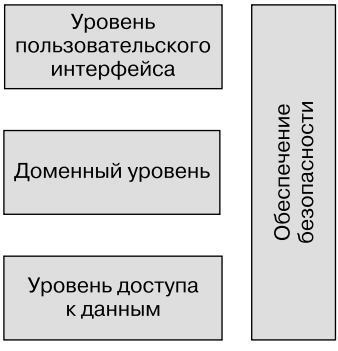


Рис. 9.6. На схеме архитектуры приложения сквозная функциональность представлена в виде вертикальных прямоугольников, охватывающих все уровни. В данном случае сквозной функциональностью является обеспечение безопасности

9.2.1. Перехват с использованием паттерна «Предохранитель»

Любое приложение, обменивающееся данными с внешним ресурсом, может иногда сталкиваться с его недоступностью. Упала сеть, база данных перешла в автономный режим, веб-сервисы подверглись DDOS-атакам... В таких случаях вызывающее приложение должно суметь восстановить свою работу и справиться с проблемой.

У большинства .NET API есть срок ожидания по умолчанию, гарантирующий, что вызов внешнего ресурса не заблокирует навечно потребляющий поток. И все же как при выдаче исключения по истечении срока ожидания направить сбойному ресурсу следующий вызов? Нужно ли пытаться вызвать его еще раз? Поскольку срок ожидания зачастую показывает, что адресат либо отключен, либо перегружен запросами, новый блокирующий вызов может только навредить. Предпочтительнее предположить худшее и тут же выдать исключение. Именно это соображение и положено в основу паттерна «Предохранитель».

Это стабильный паттерн, который увеличивает надежность приложения, так как быстро выдает сбой, что избавляет от зависания и сопутствующего ему напрасного потребления ресурсов. Он может послужить неплохим примером нефункционального требования и настоящей сквозной функциональности, поскольку имеет мало общего с функцией, реализованной в вызове внешнего ресурса.

Сам паттерн «Предохранитель» устроен непросто и может вызвать затруднения при реализации, но потрудиться над его созданием придется лишь единожды. Его,

если захочется, можно реализовать в многократно используемой библиотеке, что позволит применять его к многим компонентам с помощью паттерна «Декоратор».

Паттерн «Предохранитель»

Паттерн проектирования «Предохранитель» получил свое название от одноименного электрического выключателя[1]. Он сконструирован, чтобы обрывать подключение при возникновении сбоя, препятствуя тем самым распространению отказов.

В программных приложениях, в которых истек срок ожидания или возникли другие ошибки связи, при попытке достучаться до системы неприятная ситуация может только усугубиться. Если удаленная система перегружена, множественные попытки подключиться к ней могут поставить ее на грань фола, а пауза может дать ей шанс восстановиться. На вызывающей стороне потоки, заблокированные в ожидании тайм-аута, могут лишить потребляющее приложение возможности реагировать на внешние события, заставляя пользователя ждать появления сообщения об ошибке. Лучше обнаружить отсутствие связи и быстро в течение определенного времени выдать сбой.

Конструкция предохранителя решает эту задачу, разрывая подключение при возникновении ошибки. Обычно предусматривается срок ожидания, позволяющий впоследствии повторить попытку подключения. Таким образом, он может автоматически восстановить работу, как только восстановится удаленная система. Упрощенно смена состояний в предохранителе показана на рис. 9.7.

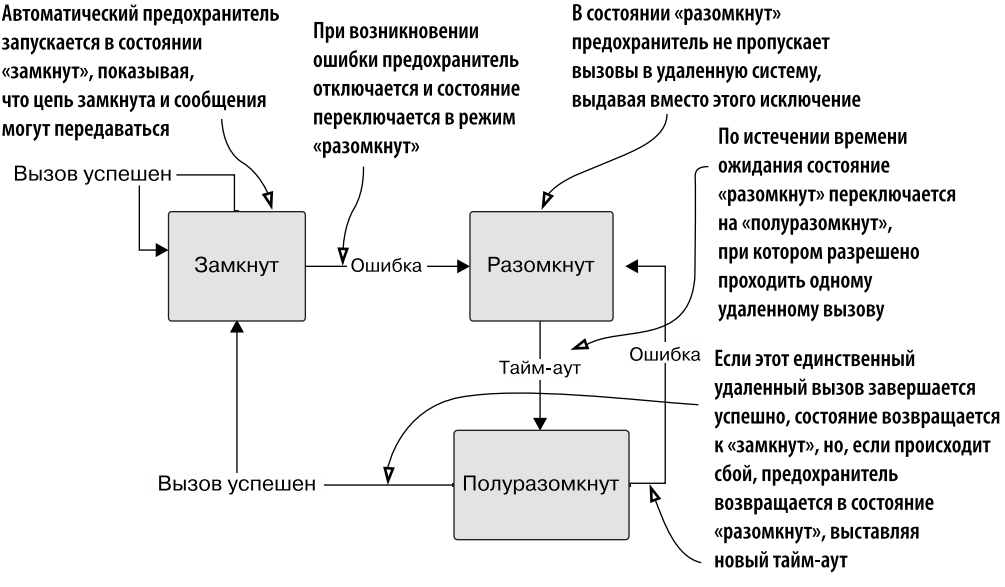


Рис. 9.7. Упрощенная схема смены состояний в паттерне «Предохранитель»

[1] *Nygard M. T.* Release It! Design and Deploy Production-Ready Software. — Pragmatic Bookshelf, 2007. — P. 104.

Конструкцию предохранителя можно сделать сложнее, чем на рис. 9.7. Во-первых, можно не размыкать цепь при каждой спорадически возникающей ошибке, а использовать некое пороговое число ошибок. Во-вторых, размыкать цепь только при возникновении ошибок определенного типа. Применение тайм-аутов и исключений связи нас вполне устраивает, а вот исключение `NullReferenceException`, вероятнее всего, будет указывать на просчет в программе, а не на случайную ошибку.

Рассмотрим пример, показывающий, как паттерн «Декоратор» может использоваться для добавления поведения предохранителя к существующему внешнему компоненту. В этом примере акцентируем внимание на многократном применении предохранителя, а не на том, как именно он реализован.

Пример: создание предохранителя для IProductRepository

В разделе 7.2 рассматривалось создание UWP-приложения, обменивающегося информацией с внутренним источником данных, таким как WCF или сервис Web API, с использованием интерфейса `IProductRepository`. В листинге 8.6 применялось хранилище `WcfProductRepository`, в котором за счет вызова операций WCF-сервиса реализуется интерфейс `IProductRepository`. Поскольку эта реализация не имеет явного механизма обработки ошибок, любая ошибка обмена данными всплывет наверх до уровня вызывающего кода.

Лучший сценарий для применения предохранителя трудно придумать. Хотелось бы получить быстрый отказ от вызова в самом начале выдачи исключения, тогда вызывающий поток не заблокируется и сервис не будет перегружен. В листинге 9.4 показано, что сначала объявляется декоратор для `IProductRepository` и путем внедрения через конструктор запрашиваются необходимые зависимости.

Листинг 9.4. Декорирование с использованием предохранителя

```
public class CircuitBreakerProductRepositoryDecorator
    : IProductRepository
{
    private readonly ICircuitBreaker breaker;
    private readonly IProductRepository decoratee;

    public CircuitBreakerProductRepositoryDecorator(
        ICircuitBreaker breaker,
        IProductRepository decoratee)
    {
        this.breaker = breaker;
        this.decoratee = decoratee;
    }
    ...
}
```

Декоратор IProductRepository, смысл применения которого в том, что он как реализует, так и заключает в себя реализацию IProductRepository

Еще одной зависимостью является интерфейс ICircuitBreaker, которым можно воспользоваться для реализации паттерна «Предохранитель»

Теперь любой вызов можно заключить в задекорированный `IProductRepository` (листинг 9.5).

Прежде чем вызвать задекорированное хранилище, нужно проверить состояние предохранителя. Метод `Guard` можно пройти, когда предохранитель находится либо в состоянии «замкнут», либо в состоянии «полуразомкнут», а вот когда он

в состоянии «разомкнут», метод выдает исключение. Тем самым гарантируется быстрый отказ, когда есть все основания полагать, что вызов не будет успешным. Пройдя метод `Guard`, можно попытаться обратиться к задекорированному хранилищу. Если вызов окажется неудачным, сработает предохранитель. В этом примере мы не стали ничего усложнять, но в настоящей реализации перехватывать исключения и задействовать предохранитель нужно только для избранных типов исключений.

Листинг 9.5. Применение предохранителя к методу Insert

```
public void Insert(Product product)
{
    this.breaker.Guard();

    try
    {
        this.decoratee.Insert(product);
        this.breaker.Succeed();
    }
    catch (Exception ex)
    {
        this.breaker.Trip(ex);
        throw;
    }
}
```

Проверка состояния предохранителя

Обращение к задекорированному хранилищу и вызов метода Succeed при успешном вызове

Если вызов метода Insert дает сбой, срабатывает предохранитель

Срабатывание предохранителя вернет его из состояний «замкнут» и «полуразомкнут» в состояние «разомкнут». Когда предохранитель вернется из состояния «разомкнут» в состояние «полуразомкнут», определяется продолжительностью тайм-аута.

И наоборот, если вызов завершается успешно, предохранитель получает сигнал. Если он уже находился в состоянии «замкнут», то в нем и останется. Если же был в состоянии «полуразомкнут», то возвратится в состояние «замкнут». Послать сигнал об успешном вызове, когда предохранитель находится состоянии «разомкнут», невозможно, поскольку метод `Guard` гарантирует, что до этого дело не дойдет.

Все остальные методы выглядят почти так же, единственное отличие в том, что метод вызывается в отношении задекорированного объекта `decoratee`, и в наличии дополнительной строки кода для методов, возвращающих значение. Этот вариант можно увидеть внутри блока `try` для метода `GetAll`:

```
var products = this.decoratee.GetAll();
this.breaker.Succeed();
return products;
```

Поскольку предохранителю нужно сообщать об успешности вызова, приходится сохранять возвращаемое значение от задекорированного хранилища, прежде чем возвращать его. Это единственное различие методов, возвращающих значение, и методов, не делающих этого.

На данный момент реализация `ICircuitBreaker` осталась открытой, но настоящая реализация — это повторно используемый комплекс классов, задействующий

паттерн проектирования «Состояние» (State)[1]. Здесь мы не станем углубляться в реализацию предохранителя `CircuitBreaker`, но отметим, что важной является возможность перехвата с помощью кода произвольной сложности.

ПРИМЕЧАНИЕ

Если вас заинтересовала реализация класса CircuitBreaker, вы найдете ее в коде, сопровождающем книгу.

Компоновка приложения с помощью реализации предохранителя

Чтобы скомпоновать `IProductRepository` с функциональностью, добавленной предохранителем, настоящую реализацию можно заключить в декоратор:

```
var channelFactory = new ChannelFactory<IProductManagementService>("*");

var timeout = TimeSpan.FromMinutes(1);

ICircuitBreaker breaker = new CircuitBreaker(timeout);

IProductRepository repository =
    new CircuitBreakerProductRepositoryDecorator(
        breaker,
        new WcfProductRepository(channelFactory));
```

Декорирование WcfProductRepository

В листинге 7.6 UWP-приложение было скомпоновано из нескольких зависимостей, включая экземпляр класса `WcfProductRepository`, код которого показан в листинге 8.6. Этот экземпляр можно задекорировать, внедрив его в экземпляр `CircuitBreakerProductRepositoryDecorator`, поскольку в нем реализуется тот же интерфейс. В данном примере новый экземпляр класса `CircuitBreaker` создается при каждом разрешении зависимостей. Это соответствует жизненному циклу Transient.

В UWP-приложении, где разрешение зависимостей выполняется только один раз, применение предохранителя с жизненным циклом Transient не вызывает проблем, но в большинстве случаев для подобной функциональности этот жизненный цикл неоптимален. На другом конце будет только один веб-сервис. Если он становится недоступным, предохранитель должен прерывать все попытки подключения к нему. Если используются несколько экземпляров `CircuitBreakerProductRepositoryDecorator`, это должно выполняться для них всех.

Есть все основания установить для `CircuitBreaker` жизненный цикл Singleton, но это также означает, что данный класс должен быть потокобезопасным. По своей природе `CircuitBreaker` поддерживает состояние, поэтому потокобезопасность должна быть реализована в явном виде. Это еще сильнее усложняет реализацию.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 352.

Более компактная версия ICircuitBreaker

Представленный здесь интерфейс `ICircuitBreaker` содержит три компонента: `Guard`, `Succeed` и `Trip`. В альтернативном определении интерфейса его объем за счет применения делегирования можно свести к одному методу:

```
public interface ICircuitBreaker
{
        T Execute<T>(Func<T> action);
}
```

Это позволит выбрать более лаконичный вариант использования `ICircuitBreaker` в каждом методе:

```
public IEnumerable<Product> GetAll()
{
        this.breaker.Execute(() => this.decoratee.GetAll());
}
```

Мы выбрали более понятную старомодную версию `ICircuitBreaker`, поскольку хотели получить возможность сосредоточиться на главной теме перехвата. Хотя нам лично нравится передача продолжений, мы посчитали, что в данном контексте она станет скорее отвлекать, чем помогать. Если в итоге предпочтение будет отдано какому-нибудь одному из этих определений, это не изменит выводы по текущей главе.

Несмотря на сложность предохранителя, с его помощью легко выполнить перехват `IProductRepository`. Хотя первый пример перехвата в подразделе 9.1.2 был сравнительно простым, пример использования предохранителя демонстрирует, что класс может быть перехвачен с применением сквозной функциональности, которую без особого труда можно сделать намного сложнее исходной реализации.

Паттерн «Предохранитель» гарантирует быстрый отказ приложения от вызова вместо связывания ценных ресурсов. В идеале приложение вообще не должно зависать. Для решения этой задачи можно с помощью перехвата реализовать несколько вариантов обработки ошибок.

9.2.2. Оповещение о выдаче исключений, выполняемое с помощью паттерна «Декоратор»

Временами зависимости склонны к выдаче исключений. Даже хорошо продуманный код будет (и должен) выдавать исключения, если возникает ситуация, с которой он не в состоянии справиться. К этой категории относятся и клиенты, потребляющие внешние ресурсы. Примером может послужить класс `WcfProductRepository` из нашего учебного UWP-приложения. Когда веб-сервис недоступен, хранилище начнет выдавать исключения. Предохранитель не сможет переломить ситуацию. Несмотря на перехват WCF-клиента, тот все же будет выдавать исключения, поскольку он делает это быстрее.

Перехват можно использоваться для добавления механизма обработки ошибок. Не хотелось бы обременять таким механизмом саму зависимость. Поскольку за-

висимость должна рассматриваться как многократно задействуемый компонент, потребляемый при реализации множества различных сценариев, добавить к ней стратегию обработки ошибок, пригодную для всех сценариев, невозможно. Это нарушило бы принцип единственной ответственности.

Использование перехвата для работы с исключениями позволяет соблюсти принцип открытости/закрытости. Появляется возможность реализовать наилучшую стратегию обработки ошибок для любой конкретной ситуации. Рассмотрим пример.

В предыдущем примере экземпляр `WcfProductRepository` для работы с клиентским приложением управления товарами, изначально представленный в подразделе 7.2.2, был заключен в предохранитель. Этот предохранитель работал с ошибками, только чтобы заставить клиента как можно быстрее отказаться от вызова, но тот все равно выдавал исключения. Если оставить их необработанными, они сделают состояние приложения аварийным, поэтому нужно реализовать декоратор, разбирающийся в обработке некоторых из возникающих в данном процессе ошибок.

Вместо введения приложения в аварийное состояние можно выбрать вариант вывода на экран окна с сообщением о неудачном завершении операции и необходимости чуть позже сделать еще одну попытку. В этом примере при выдаче исключения должно появляться сообщение, показанное на рис. 9.8.

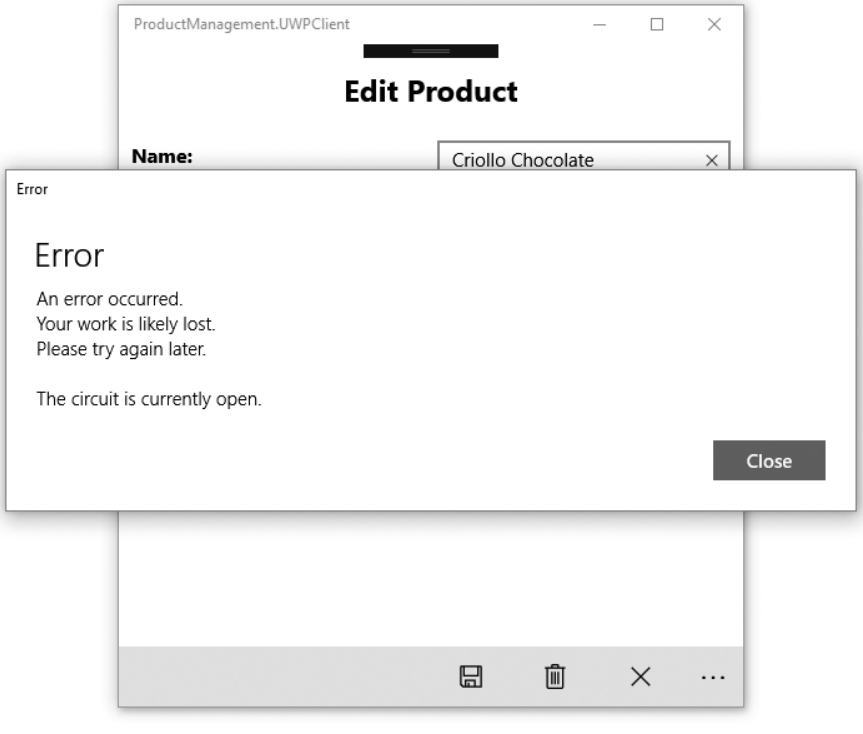


Рис. 9.8. Приложение управления товарами обрабатывает исключения, выданные при обмене данными, демонстрируя пользователю сообщение. Обратите внимание на то, что в данном случае сообщение об ошибке поступает от предохранителя, а не от исходного сбоя подключения

Реализовать такое поведение не составляет особого труда. По аналогии с действиями, рассмотренными в подразделе 9.2.1, добавляется новый класс `ErrorHandlingProductRepositoryDecorator`, декорирующий интерфейс `IProductRepository`. В листинге 9.6 показан пример одного из методов этого интерфейса, но все его методы похожи друг на друга.

Листинг 9.6. Обработка исключений с применением ErrorHandlingProductRepositoryDecorator

```
public void Insert(Product product)
{
    try
    {
        this.decoratee.Insert(product);
    }
    catch (CommunicationException ex)
    {
        this.AlertUser(ex.Message);
    }
    catch (InvalidOperationException ex)
    {
        this.AlertUser(ex.Message);
    }
}
```

Делегирование декоратору decoratee

Предупреждение пользователя

Метод `Insert` — это представитель реализации класса `ErrorHandlingProductRepositoryDecorator`. При выдаче исключения предпринимается попытка вызвать задекорированный компонент и предупредить пользователя, сообщив ему об ошибке. Следует заметить, что обрабатывается только конкретный набор известных исключений, поскольку подавление всех исключений может стать опасным. В процессе предупреждения пользователя строка форматируется и выводится на экран пользователя с помощью метода `MessageBox.Show`. Все это происходит внутри метода `AlertUser`.

Повторим еще раз: функциональность добавляется к исходной реализации (`WcfProductRepository`) путем реализации паттерна «Декоратор». При этом за счет постоянного добавления новых типов вместо изменения уже существующего кода соблюдаются оба принципа: единственной ответственности и открытости/закрытости. К данному моменту у вас уже должно сложиться представление о паттерне, предлагающем более общую схему по сравнению с «Декоратором». Кратко рассмотрим последний пример, связанный с обеспечением безопасности.

9.2.3. Использование декоратора для предотвращения несанкционированного доступа к функциям, работающим с конфиденциальной информацией

Обеспечение безопасности — еще один довольно распространенный пример реализации сквозной функциональности. Нам нужно максимально обезопасить свои приложения от несанкционированного доступа к конфиденциальным данным и функциям, работающим с ними.

ПРИМЕЧАНИЕ

> Безопасность — обширная тема, охватывающая множество областей, включая рассекречивание конфиденциальной информации и взлом сетей[1]. В этом подразделе кратко рассмотрим тему авторизации: предоставления гарантий доступности конкретных действий только авторизированным пользователям (или системам).

Хотелось бы так же, как и при использовании предохранителя, перехватить вызов метода и проверить наличие прав на его выполнение. Если их нет, то вместо того, чтобы разрешать выполнение вызова, нужно выдать исключение. Принцип тот же, различаются только критерии определения правомерности вызова.

Обычный подход к реализации логики авторизации заключается в применении мер безопасности на основе ролей, где роль (или роли) пользователя сопоставляется с жестко заданным для намеченной операции значением. Если придерживаться нашего интерфейса `IProductRepository`, то можно было бы начать с безопасного декоратора хранилища товаров `SecureProductRepositoryDecorator`. Поскольку, исходя из увиденного в предыдущих разделах, можно прийти к выводу, что все методы выглядят одинаково, в листинге 9.7 показана реализация только двух методов.

Листинг 9.7. Явная проверка авторизации с помощью декоратора

```
public class SecureProductRepositoryDecorator
    : IProductRepository
{
    private readonly IUserContext userContext;
    private readonly IProductRepository decoratee;

    public SecureProductRepositoryDecorator(
        IUserContext userContext,
        IProductRepository decoratee)
    {
        this.userContext = userContext;
        this.decoratee = decoratee;
    }

    public void Delete(Guid id)
    {
        this.CheckAuthorization();
        this.decoratee.Delete(id);
    }

    public IEnumerable<Product> GetAll()
    {
        return this.decoratee.GetAll();
    }
    ...
```

Декоратор зависит от IUserContext, который позволяет ему проверять роль текущего пользователя

Метод Delete начинается с контрольной инструкции, проводящей явную проверку прав пользователя на выполнение данной операции. Если таких прав у него нет, тут же выдается исключение. Пройти контрольную инструкцию для обращения к задекорированному хранилищу пользователь может только при наличии у него требуемой роли

Проверка разрешения требуется не всем методам. В данном приложении в ней нуждаются только CUD-методы: create, update и delete. Запрашивать все товары разрешается каждому пользователю, вошедшему в эту систему

[1] Чтобы получить более полное представления о вопросах обеспечения безопасности, можно обратиться к изданию: *Howard M., LeBlanc D.* Writing Secure Code. 2nd Ed. — Microsoft Press, 2003.

```
    private void CheckAuthorization()
    {
        if (!this.userContext.IsInRole(
            Role.Administrator))
        {
            throw new SecurityException(
                "Access denied.");
        }
    }
}
```

Роль, требующаяся для выполнения CUD-операций, жестко запрограммирована в виде роли Administrator. Если у пользователя этой роли нет, выдается исключение

ПРИМЕЧАНИЕ

Примеры декоратора в листингах 9.2, 9.5–9.7 показывают только часть его кода, поскольку все его методы выглядят одинаково.

Нашему проекту сквозной функциональности, реализованной на основе декоратора, свойственна повторяемость кода. В реализации предохранителя для всех методов интерфейса `IProductRepository` задействуется один и тот же шаблон кода. Если бы нужно было добавить предохранитель к другой абстракции, пришлось бы использовать один и тот же код в еще большем количестве методов.

Применительно к декоратору безопасности ситуация только ухудшилась бы, поскольку потребовалось бы расширение некоторых методов, а другие методы так и оставались бы связанными со сквозными операциями. Но в целом проблема идентична.

Если сквозную функциональность нужно применить к различным абстракциям, то код придется делать повторяемым, что по мере разрастания системы станет создавать серьезные проблемы при сопровождении приложений. Вполне резонное предположение о существовании способов предотвращения повторяемости кода приводит нас к рассмотрению очень важной темы — аспектно-ориентированного программирования, которому будет посвящена следующая глава.

Резюме

- Перехват — это способность перехватить вызовы между двумя взаимодействующими компонентами и сделать это так, чтобы можно было обогатить или изменить поведение зависимости, не испытывая при этом необходимости внесения изменений в сами взаимодействующие компоненты.
- Необходимым условием перехвата является слабое связывание. Если программирование ведется на основе интерфейсов, основную реализацию можно преобразовать или улучшить, запаковав ее в другие реализации этого интерфейса.
- По своей сути перехват является применением паттерна проектирования «Декоратор».
- Паттерн проектирования «Декоратор» обеспечивает гибкую альтернативу использованию подклассов за счет динамического прикрепления к объекту до-

полнительных ответственностей. Он работает, запаковывая одну реализацию абстракции в другую реализацию той же самой абстракции. Это позволяет декораторам вкладываться друг в друга наподобие матрешки.

- Сквозная функциональность представляет собой нефункциональные аспекты кода, пересекающие широкую область кодовой базы. Типовыми примерами сквозной функциональности являются ведение контрольного журнала, регистрация событий, проверка допустимости, обеспечение безопасности и кэширование.
- «Предохранитель» — это паттерн проектирования, который улучшает живучесть приложения и повышает надежность системы, обрывая подключение при возникновении сбоя, чтобы остановить распространение отказов.

10 Аспектно-ориентированное проектирование программного обеспечения

В этой главе

- Повторение SOLID-принципов.
- Использование аспектно-ориентированного программирования для предотвращения повторяемости кода.
- Применение SOLID-принципов для аспектно-ориентированного программирования.

Между приготовлением домашней еды и работой на профессиональной кухне существует большая разница. Дома на приготовление блюда можно потратить сколько угодно времени, а на профессиональной кухне ключевым фактором служит эффективность. Важным аспектом здесь выступает окружение. Это не только предварительная подготовка ингредиентов, но и наличие необходимого оборудования, включая кастрюли, сковородки, разделочные доски, дегустационные ложки — и все то, что играет значимую роль в вашем рабочем пространстве.

Важными факторами эффективности работы на кухне являются эргономика и планировка кухни. При неудачной планировке персонал будет мешать друг дру-

гу, бестолково суетиться и хвататься то за одно, то за другое. Свести к минимуму перемещение персонала во избежание ненужной суеты и сосредоточиться на выполняемой задаче поможет выделение рабочих зон с соответствующим специализированным оборудованием. При правильном исполнении это позволит увеличить общую эффективность работы на кухне.

В разработке программных средств такой кухней можно считать кодовую базу. Команды годами совместно трудятся на одной кухне, и для обеспечения эффективности и последовательности их работы нужна правильная архитектура, сводящая повторяемость кода к минимуму. Насколько довольными останутся ваши «гости», зависит от успешности кухонной стратегии.

Одной из ключевых архитектурных стратегий, которой можно воспользоваться для повышения эргономики при разработке программ, является аспектно-ориентированное программирование (Aspect-Oriented Programming, AOP). Оно может существовать в виде оборудования (инструментария) или жестко заданной планировки (программного проекта). AOP тесно связано с перехватом. Чтобы полностью оценить потенциал перехвата, нужно изучить концепцию AOP и такие принципы проектирования программных средств, как SOLID.

Эта глава начинается с введения в AOP. Поскольку одним из наиболее эффективных способов применения AOP является использование широко известных паттернов проектирования и принципов объектно-ориентированного программирования, в этой главе мы напомним о пяти SOLID-принципах, которые рассматривались в предыдущих главах.

Существует распространенное заблуждение, что для AOP требуется определенный инструментарий. В этой главе будет продемонстрировано, что это не так: мы покажем, как можно воспользоваться проектированием программных средств, придерживаясь в качестве основы для AOP SOLID-принципов, позволяющих создать эффективную, последовательную и легко сопровождаемую кодовую базу. В следующей главе будут рассмотрены две широко известные формы AOP, требующие применения специального инструментария. Но обе они имеют существенные недостатки по сравнению с чистой, ориентированной на проектирование формой AOP, рассматриваемой в этой главе.

Если вы уже знакомы с SOLID-принципами и основами AOP, можете сразу перейти к изучению раздела 10.3, в котором содержится основной посыл данной главы. В противном случае начните читать введение в аспектно-ориентированное программирование.

10.1. Введение в AOP

AOP было изобретено в Xerox Palo Alto Research Center (PARC) в 1997 году, когда инженеры компании спроектировали AspectJ — AOP-расширение для языка Java. AOP — это парадигма, сфокусированная на концепции применения эффективной и удобной в сопровождении сквозной функциональности. Это абстрактная

концепция с собственным набором терминов, большинство которых не имеет никакого отношения к рассматриваемым нами вопросам.

ОПРЕДЕЛЕНИЕ

> Аспектно-ориентированное программирование нацелено на сокращение объема рутинного кода, требующегося для реализации сквозной функциональности и других паттернов создания программного кода. Эта цель достигается реализацией таких паттернов в одном месте и их применением к кодовой базе либо декларативно, либо на основе соглашений без внесения изменений в сам код.

В примерах ведения контрольного журнала и применения предохранителя в подразделах 9.1.2 и 9.2.1 показаны всего несколько типичных методов, поскольку все методы были реализованы одинаково. Нам не захотелось приводить еще несколько страниц практически идентичного кода, поскольку это отвлекало бы от сути рассматриваемого вопроса.

В листинге 10.1 еще раз показан метод `Delete`, принадлежащий классу `CircuitBreakerProductRepositoryDecorator`.

Листинг 10.1. Метод Delete класса CircuitBreakerProductRepositoryDecorator

```
public void Delete(Product product)
{
    this.breaker.Guard();
    try
    {
        this.decoratee.Delete(product);
        this.breaker.Succeed();
    }
    catch (Exception ex)
    {
        this.breaker.Trip(ex);
        throw;
    }
}
```

В листинге 10.2 показано, как методы класса `CircuitBreakerProductRepositoryDecorator` похожи друг на друга. Здесь приведен только метод `Insert`, но мы уверены, что вы в состоянии представить, как будет выглядеть остальная часть реализации.

Цель этого листинга — проиллюстрировать повторяющуюся природу декораторов, используемых в качестве аспектов в текущем проекте. Единственным различием методов `Delete` и `Insert` является то, что каждый из них обращается в задекорированном хранилище к собственному методу.

Даже при успешном делегировании реализации предохранителя отдельному классу через интерфейс `ICircuitBreaker` этот лишний код нарушает DRY-принцип. И хотя он объективно неизменяем, но все равно становится обузой. При необходимо-

сти добавить к декорируемому типу новый компонент или при желании применить предохранитель к новой абстракции придется использовать все тот же лишний код. Если потребуется сопровождать такое приложение, эта повторяемость может перерасти в серьезную проблему.

Листинг 10.2. Нарушение DRY-принципа из-за повторяемости логики предохранителя

```
public void Insert(Product product)
{
    this.breaker.Guard();

    try
    {
        this.decoratee.Insert(product);
        this.breaker.Succeed();
    }
    catch (Exception ex)
    {
        this.breaker.Trip(ex);
        throw;
    }
}
```

Эта строка кода — единственное отличие этого листинга от листинга 10.1. Вместо вызова Delete в ней вызывается Insert

ПРИМЕЧАНИЕ

AOP является парадигмой, сфокусированной на борьбе с повторяемостью кода.

В примере ведения контрольного журнала из главы 9 мы установили, что нам не хочется помещать код ведения этого журнала в реализацию `SqlProductRepository`, потому что тем самым будет нарушен принцип единственной ответственности (SRP). Не хотелось бы также иметь десятки контролирующих декораторов для каждой абстракции хранилища, имеющейся в системе. Это может привести к серьезной повторяемости кода и, возможно, вызовет масштабные изменения, что нарушит принцип открытости/закрытости (OCP). Вместо этого хотелось бы декларировать, что аспект ведения контрольного журнала нужно применить к определенному набору методов всех абстракций хранилищ, имеющихся в системе, и реализовать этот аспект всего один раз.

В этой главе будут найдены инструменты, среды выполнения и архитектурные стили, позволяющие воспользоваться AOP. Рассмотрим здесь идеальную форму AOP. В следующей главе в качестве форм AOP, основанных на применении инструментария, будут рассмотрены динамический перехват и автоматическое добавление аспекта в ходе компиляции. Три основных метода AOP[1], а также их основные преимущества и недостатки перечислены в табл. 10.1.

[1] Существуют и другие методы AOP, но они либо аналогичны данным, либо не вписываются в мир среды .NET, поэтому в книге не рассматриваются. Но будьте в курсе, что у каждого рассматриваемого метода имеется множество вариаций.

Таблица 10.1. Самые распространенные методы AOP

Метод	Описание	Преимущества	Недостатки
SOLID	Применение аспектов с помощью декораторов, охватывающих многократно используемые абстракции, определенные для групп классов на основе их поведения	• Не требует никакого инструментария. • Аспекты легко реализуются. • Сфокусирован на проектировании. • Упрощена сопровождаемость системы	Трудно применять в устаревших системах
Динамический перехват	Приводит к созданию декораторов на основе абстракций приложения в ходе выполнения программы. Эти декораторы внедряются с аспектами-перехватчиками, специфичными для применяемого инструмента	• Просто добавлять к существующим или устаревшим приложениям с относительно небольшим объемом изменений при условии, что приложение уже программировалось под применение интерфейсов. • Скомпилированное приложение не связано с используемой библиотекой динамического перехвата. • Свободный доступ к качественному инструментарию	• Приводит к сильной связанности аспектов с инструментом AOP. • Теряет поддержку в ходе компиляции. • Основан на конвенции, то есть хрупок и подвержен ошибкам
Автоматическое добавление аспекта в ходе компиляции	Аспекты добавляются к приложению после компиляции. Наиболее распространенной формой является IL weaving, где внешнее инструментальное средство читает скомпилированный код сборки и изменяет его путем применения аспекта и замены исходной сборки ее модифицированным вариантом	Просто добавлять к существующим или устаревшим приложениям с относительно небольшим объемом изменений, даже если приложение не программировалось под применение интерфейсов	• Внедрение в аспекты изменчивых зависимостей вызывает временную связанность или взаимозависимость тестов. • Аспекты автоматически добавляются в ходе компиляции, делая невозможным вызов кода без примененного аспекта. Это усложняет тестирование и снижает гибкость программного продукта. • Автоматическое добавление аспекта в ходе компиляции является противоположностью DI

К динамическому перехвату и автоматическому добавлению аспекта в ходе компиляции мы вернемся в следующей главе. Но сначала более пристально взглянем на использование принципов SOLID в качестве движущей силы AOP, а начнем с краткого напоминания о том, что представляют собой эти принципы.

10.2. Принципы SOLID

Наверное, вы обратили внимание на более частое, чем обычно, использование в главе 9 и в предыдущем разделе таких понятий, как принцип единственной ответственности (Single Responsibility Principle, SRP), принцип открытости/закрытости (Open/Closed Principle, OCP) и принцип подстановки Лисков (Liskov Substitution Principle, LSP). Вместе с принципом изоляции интерфейса (Interface Segregation Principle, ISP) и принципом инверсии зависимостей (Dependency Inversion Principle, DIP) они составляют акроним SOLID. Все пять принципов, независимо друг от друга, уже рассматривались в книге, и в данном разделе приводится краткая выжимка из этой информации, поскольку усвоение этих принципов играет важную роль в изучении всей остальной главы.

ПРИМЕЧАНИЕ

> Кому не хочется создавать надежные программные продукты? Программы, способные выдержать испытание временем и обеспечить полезные средства для работы своим пользователям, — вполне достойная цель для любого программиста. Мы ввели созвучный со словом «солидный» акроним SOLID, поскольку создание качественных программных продуктов заслуживает особого внимания.

Все упомянутые паттерны и принципы признаны полезными для создания качественного и понятного кода. Основная цель раздела — связать это основополагающее руководство с DI, подчеркнув тем самым, что DI — всего лишь средство достижения цели. Поэтому мы используем технологию DI в качестве стимулятора для создания легко сопровождаемого кода.

Ни один из принципов, составляющих акроним SOLID, не является обязательным для выполнения. Это всего лишь рекомендации, которые должны помочь создавать простой и понятный код. Нам они видятся целями, помогающими принимать решения о направлении создания наших приложений. Мы всегда радуемся успехам на этом пути, но иногда приходится терпеть неудачи.

В следующих разделах дается представление о принципах SOLID и обобщается все то, что ранее говорилось об их применении. Каждый раздел является кратким обзором, поэтому примеров в них нет. К примерам мы вернемся в разделе 10.3, где выполнен подробный разбор практического примера, показывающего, почему несоблюдение принципов SOLID может создать серьезные проблемы в плане сопровождаемости программных продуктов. Пока вспомним, что собой представляют пять принципов SOLID.

10.2.1. Принцип единственной ответственности

В подразделе 2.1.3 говорилось: SRP (Single Responsibility Principle) утверждает, что у каждого класса должна быть только одна причина для внесения изменений. Нарушение этого принципа приводит к усложнению классов и затруднению их тестирования и сопровождения.

Но чаще всего бывает сложно понять, есть ли у класса несколько причин для внесения изменений. Помочь здесь может взгляд на SRP с точки зрения связности. Понятие связности определяется как функциональное родство элементов класса или модуля. Чем слабее родство, тем слабее связность, а чем слабее связность, тем выше вероятность нарушения классом принципа SRP. В разделе 10.3 связность будет рассмотрена на конкретном примере.

Придерживаться данного принципа может быть нелегко, но если используется технология DI, то одним из множества преимуществ внедрения через конструктор оказывается прояснение ситуации с нарушением принципа SRP. В примере ведения контрольного журнала из подраздела 9.1.2 придерживаться принципа SRP удавалось за счет разделения ответственности на два отдельных типа: `SqlUserRepository` только сохраняет и извлекает данные о товарах, а `AuditingUserRepositoryDecorator` концентрируется на ведении контрольного журнала в базе данных. Единственной ответственностью класса `AuditingUserRepositoryDecorator` является координация действий `IUserRepository` и `IAuditTrailAppender`.

10.2.2. Принцип открытости/закрытости

Как говорилось в подразделе 4.4.2, принцип OCP (Open/Closed Principle) предписывает создавать приложение на основе конструкции, которая не позволит вносить радикальные изменения по всей кодовой базе. Или же, как гласит сама формулировка принципа, класс должен быть открыт для расширений, но закрыт для изменений. У разработчика должна быть возможность расширять функционал системы, не испытывая при этом необходимости изменять исходный код существующих классов.

Между принципом OCP и принципом «Не повторяйся» (Don't Repeat Yourself, DRY) существует стойкая взаимосвязь, поскольку оба они нацелены на предотвращение широкомасштабных изменений. Но OCP фокусируется на коде, а DRY — на знании.

Не повторяйся

Акроним DRY, являющийся сокращением фразы Don't Repeat Yourself, придуман Энди Хантом (Andy Hunt) и Дэйвом Томасом (Dave Thomas) и появился в их книге The Pragmatic Programmer в следующей формулировке: *«Каждый фрагмент знания должен иметь единственное, однозначное, надежное представление в системе»*[1].

Разработчики трудятся в системах, где знания нестабильны. Дублирование таких знаний затрудняет поддержание их согласованности. Наше понимание как системы, так и требований изменяется, и порой довольно часто. Принцип DRY гласит, что мы должны стремиться к сосредоточению каждой части знаний в одном месте. DRY касается не только кода, но и документации.

[1] *Хант Э., Томас Д.* Программист-прагматик. Путь от подмастерья к мастеру. — М.: Лори, 2009. — С. 22.

Превратить класс в расширяемый можно множеством способов, включая виртуальные методы, внедрение стратегий и применение декораторов[1]. Но независимо от подробностей технология DI предоставляет такую возможность, позволяя выполнять компоновку объектов.

10.2.3. Принцип подстановки Лисков

В подразделе 8.1.1 говорилось, что все потребители зависимостей должны при обращении к ним придерживаться принципа LSP (Liskov Substitution Principle), поскольку каждая зависимость должна вести себя так, как определено ее абстракцией. Это позволит заменить изначально задуманную реализацию другой реализацией той же самой абстракции, не опасаясь, что это нарушит работу потребителя. Поскольку в декораторе реализуется та же самая абстракция, что и в заключенном в него классе, оригинал можно заменить декоратором, но только если последний придерживается контракта, заданного его абстракцией.

Именно это и было сделано в листинге 9.3, когда оригинал `SqlUserRepository` был заменен `AuditingUserRepositoryDecorator`. Это можно было сделать, не внося никаких изменений в код класса-потребителя `ProductService`, поскольку принципа LSP должна придерживаться любая реализация. Классу `ProductService` требуется экземпляр `IUserRepository`, и пока он общается исключительно с этим интерфейсом, подойдет любая реализация.

Принцип LSP положен в основу технологии DI. Когда потребители его не придерживаются, толку от внедрения зависимостей мало, поскольку их нельзя заменить по своему желанию, в результате чего будут утрачены многие (если не все) преимущества использования DI.

10.2.4. Принцип изоляции интерфейса

В подразделе 6.2.1 указывалось, что принципом ISP (Interface Segregation Principle) поощряется применение узкоспециализированных, а не универсальных абстракций. Когда потребитель зависит от абстракции, в которой часть компонентов остаются невостребованными, ISP нарушается.

Поначалу создается впечатление, что ISP может быть отдаленно связан с DI, но это, наверное, потому, что в основной части книги этот принцип нами игнорировался. Изменения произойдут в разделе 10.3, когда вы узнаете, что соблюдение принципа ISP имеет решающее значение для эффективного применения аспектно-ориентированного программирования.

---

[1] Расширить познания о стратегиях позволяет издание: *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 362.

10.2.5. Принцип инверсии зависимостей

Когда принцип DIP (Dependency Inversion Principle) рассматривался в подразделе 3.1.2, выяснилось, что многое, чего мы стараемся достичь, следуя технологии DI, относится и к принципу DIP. Этот принцип утверждает, что программировать нужно, исходя из абстракций, и что уровень потребления должен контролировать форму потребляемой абстракции. Потребитель должен иметь возможность определять абстракцию таким образом, чтобы она приносила ему наибольшую пользу. Если обнаружится, что вы добавляете к интерфейсу компоненты, необходимые для удовлетворения потребностей других реализаций, включая потенциальные, то вы практически наверняка нарушаете принцип DIP.

10.2.6. SOLID-принципы и перехват

Паттерны проектирования, такие как «Декоратор», и руководства, такие как SOLID-принципы, практикуются много лет и в целом считаются полезными. В следующих разделах будет дано представление о том, как они относятся к технологии DI.

Принципы SOLID не теряли своей актуальности во всех главах книги. Но когда разговор заходит о перехвате и его отношении к декораторам, выгода от соблюдения принципов SOLID занимает особое место. Одни принципы различимы хуже, чем другие, но добавление поведения (например, ведения контрольного журнала) путем использования декоратора является четким признаком применения принципов OCP и SRP, а последний позволяет создавать реализации с конкретно определенными областями видимости.

В предыдущих разделах кратко рассмотрены самые распространенные паттерны и принципы, необходимые для того, чтобы понять, как технология DI связана с другими устоявшимися методологическими принципами. Теперь, вооружившись этими знаниями, вернемся к основной цели данной главы — написанию понятного и легко сопровождаемого кода в условиях противоречивых или изменяющихся требований, а также при необходимости реализации сквозной функциональности.

10.3. Соблюдение принципов SOLID как движущая сила AOP

Из раздела 10.1 вы узнали, что основной целью AOP является соблюдение принципа DRY при реализации сквозной функциональности. В разделе 10.2 рассматривалась тесная связь принципов OCP и DRY. Их соблюдение выражается в стремлении к одной и той же цели — сведению к минимуму повторяемости кода и предотвращению его широкомасштабных изменений.

С этой позиции повторяемость кода, свидетелем которой вы были при использовании классов `AuditingUserRepositoryDecorator`, `CircuitBreakerProductRepositoryDecorator` и `SecureProductRepositoryDecorator` в главе 9 (листинги 9.2, 9.4 и 9.7), является явным признаком нарушения принципа OCP. Аспектно-ориентированное программирование стремится решить эту проблему путем выделения расширяемого

поведения (аспектов) в отдельные компоненты, которые могут легко применяться к различным реализациям.

Бытует ошибочное мнение, активно поддерживаемое поставщиками AOP-инструментов, что AOP требует специального инструментария. Мы же предпочитаем практиковать AOP в ходе обычного проектирования, при котором, прежде чем обратиться к специализированному AOP-инструментарию, подобному библиотекам динамического перехвата, используют паттерны и принципы.

В этом разделе мы именно так и поступим. Посмотрим на AOP с позиции проектирования, обращая внимание на абстракцию `IProductService`, с которой познакомились в главе 3. Проанализируем, какие SOLID-принципы были нарушены и какие нарушения породили проблемы. После этого займемся пошаговой проработкой этих нарушений, чтобы упростить сопровождаемость приложения и устранить необходимость внесения широкомасштабных изменений в будущем. Нужно заранее подготовиться к некоторому душевному дискомфорту, даже когнитивному диссонансу, поскольку будет брошен вызов вашим убеждениям о порядке разработки программных продуктов. Пристегните ремни безопасности и подготовьтесь к поездке.

10.3.1. Пример: реализация функций работы с товарами с использованием IProductService

Присмотримся к абстракции `IProductService`, созданной в главе 3 как часть доменного уровня учебного приложения электронной торговли. В листинге 10.3 этот интерфейс показан в своем изначальном определении, каким оно было в листинге 3.5.

Листинг 10.3. Интерфейс IProductService из главы 3

```
public interface IProductService
{
    IEnumerable<DiscountedProduct> GetFeaturedProducts();
}
```

Когда конструкция приложения рассматривается с позиции соблюдения принципов SOLID в целом и принципа OCP в частности, важно принять во внимание то, как приложение изменилось со временем, и, исходя из этого, предсказать будущие изменения. С учетом этого можно определить, закрыто ли приложение для внесения изменений, которые со значительной долей вероятности могут произойти в будущем.

ПРИМЕЧАНИЕ

> Чем больше опыт в областях применения приложения и разработки программных продуктов, тем выше вероятность удачного прогнозирования будущих изменений. Именно поэтому получить нужную конструкцию сразу же после запуска проекта непросто.

Важно отметить, что даже при соблюдении при проектировании принципов SOLID может настать время, когда потребуются серьезные изменения. Полностью отказаться от изменений и невозможно, и нежелательно. Кроме того, соответствовать

принципу OCP — удовольствие не из дешевых. Подбор и конструирование подходящих абстракций требует существенных усилий, а слишком большое число абстракций может значительно усложнить приложение. Ваша задача — добиться разумного баланса рисков и затрат и прийти к глобальному оптимуму.

Поскольку вам нужно наблюдать за развитием приложения, одномоментная оценка `IProductService` вряд ли принесет много пользы. К счастью, Мэри Роуэн (наша разработчица из главы 2) продолжала заниматься своим приложением электронной торговли и с того момента, когда мы в последний раз видели ее работу, реализовала несколько новых функций. Достигнутый ею прогресс показан в листинге 10.4.

Листинг 10.4. Интерфейс IProductService, получивший дальнейшее развитие

```
public interface IProductService
{
    IEnumerable<DiscountedProduct> GetFeaturedProducts();

    void DeleteProduct(Guid productId);
    Product GetProductById(Guid productId);
    void InsertProduct(Product product);
    void UpdateProduct(Product product);
    Paged<Product> SearchProducts(
        int pageIndex, int pageSize,
        Guid? manufacturerId, string searchText);
    void UpdateProductReviewTotals(
        Guid productId, ProductReview[] reviews);
    void AdjustInventory(
        Guid productId, bool decrease, int quantity);
    void UpdateHasTierPricesProperty(Product product);
    void UpdateHasDiscountsApplied(
        Guid productId, string discountDescription);
}
```

Новые функции, добавленные Мэри, пока мы изучали последние несколько глав. Как вскоре выяснится, в этом проблемном фрагменте кода целых три нарушения принципов SOLID

Нетрудно заметить, что в приложение было добавлено довольно много функций. Одни из них, например `UpdateProduct`, являются типовыми CRUD-операциями, а другие, например `UpdateHasTierPricesProperty`, используются в более сложных случаях. Третьи же, такие как `SearchProducts` и `GetProductById`, предназначены для извлечения данных.

ПРИМЕЧАНИЕ

Если функциональные возможности новых методов вам непонятны, ничего страшного: подробности этого интерфейса и сведения о том, чем занимается каждый из методов, не имеют к рассматриваемому вопросу никакого отношения.

Хотя Мэри, определяя первую версию `IProductService`, показанную в листинге 10.3, исходила из лучших побуждений, но то, что данный интерфейс нуждался в обновлении при каждой реализации новой функции для работы с товарами, — это явный признак того, что здесь что-то неладно.

Если все это экстраполировать, чтобы дать прогноз, то можно ли вскоре ожидать еще одного обновления данного интерфейса? Ответ очевиден: да. По сути, у Мэри уже есть на примете несколько функций, касающихся рекомендаций покупки сопутствующих товаров (cross-sellings), изображений товаров и обзора товаров, и это послужит причиной внесения изменений в `IProductService`[1].

Из этого можно сделать вывод, что в данном приложении новые функции, относящиеся к работе с товарами, добавляются регулярно. Поскольку это приложение электронной торговли, в этом нет ничего удивительного. Но, так как это центральная часть кодовой базы, к тому же довольно часто обновляемая, нужно усовершенствовать ее конструкцию. Проанализируем существующую конструкцию с позиции соблюдения SOLID-принципов.

10.3.2. Анализ IProductService с позиции соблюдения SOLID-принципов

Что касается пяти SOLID-принципов, рассмотренных в разделе 10.2, то приложение, созданное Мэри, нарушает три из них, а именно ISP, SRP и OCP. Начнем с первого: `IProductService` нарушает ISP.

IProductService нарушает принцип ISP

То, что `IProductService` нарушает ISP, несомненно. Из объяснений, приведенных в подразделе 10.2.4, видно, что ISP требует использования узкоспециализированных, а не универсальных абстракций. С позиции ISP `IProductService` слишком универсален. Исходя из кода, показанного в листинге 10.4, нетрудно поверить, что не будет ни одного потребителя `IProductService`, использующего все его методы. Большинство потребителей обычно применяет один метод, от силы несколько. Но как это нарушение может стать проблемой?

Проблемы в той части кодовой базы, где есть универсальные интерфейсы, выявляются непосредственно в ходе тестирования. Например, модульные тесты, предназначенные для тестирования `HomeController`, определят реализацию тестового дубликата `IProductService`, но такой тестовый дубликат должен будет реализовать все члены этого интерфейса, даже если сам `HomeController` задействует только один метод[2]. Пусть даже можно будет создать многократно используемый тестовый дубликат, но тогда нужно будет подтвердить, что `HomeController` не вызывает методы `IProductService`, не имеющие к нему отношения. В листинге 10.5 показан код имитации `IProductService`, позволяющий убедиться в отсутствии вызовов тех методов, применение которых не предусматривается.

---

[1] Приходилось ли вам видеть в интернет-магазинах списки типа «Товары, приобретенные другими покупателями» или «Сопутствующие товары»? Это и есть cross-sellings, то есть практика продажи покупателю дополнительных товаров или услуг.

[2] В листинге 3.4 показано, как принадлежащий HomeController метод Index вызывает только GetFeaturedProducts() в IProductService.

Листинг 10.5. Многократно используемая имитация базового класса IProductService

```
public abstract class MockProductService : IProductService
{
    public virtual void DeleteProduct(Guid productId)
    {
        Assert.True(false, "Should not be called.");
    }

    public virtual Product GetProductById(Guid id)
    {
        Assert.True(false, "Should not be called.");
        return null;
    }

    public virtual void InsertProduct(Product product)
    {
        Assert.True(false, "Should not be called.");
    }
    ...
}
```

Реализация всех методов выдает ошибку

Список методов продолжается. Нужна реализация всех десяти методов

Сбой всех методов реализован вызовом метода `Assert.True` со значением `false`. Метод `Assert.True` является частью среды тестирования `xUnit`[1]. Из-за передачи аргумента `false` утверждение не выполняется и текущий запущенный тест также дает сбой.

Чтобы не тратить бумагу понапрасну, в листинге 10.5 мы показали лишь небольшую часть методов, принадлежащих `MockProductService`, но полагаем, что общая картина у вас уже сложилась. Если бы интерфейс был конкретизирован под нужды `HomeController`, то заниматься реализацией такого большого списка сбойных методов вам бы не пришлось: тогда для `HomeController` верны были бы ожидания вызовов только тех методов, которые входят в его зависимости, и подобная проверка стала бы не нужна.

IProductService нарушает принцип SRP

Поскольку концептуально ISP является основой для SRP, нарушение ISP обычно является, как в данном случае, признаком несоблюдения SRP. Порой нарушения SRP трудно обнаружить, и в качестве возражения можно заявить, что реализация `ProductService` несет только одну ответственность — обрабатывает все, что касается товаров.

Но понятие «все, что касается товаров» слишком широко и расплывчато. Скорее, речь должна идти о классах, имеющих всего одну причину для внесения в них изменений. А у `ProductService` для этого определенно есть несколько причин. Например,

[1] Вызов Assert.True с аргументом false выглядит немного странно, но среде xUnit не хватает более удобного в подобной ситуации метода Assert.Fail.

необходимость внесения изменений в `ProductService` может быть вызвана любой из следующих причин:

- изменение порядка применения скидок;
- изменение способа уточнения данных о товарных запасах;
- добавление критериев поиска товаров;
- добавление новой функции для работы с товарами.

Но наличием множества причин для изменения `ProductService` дело не ограничивается, скорее всего, у методов этого класса вдобавок отсутствует связность. Выявить низкую связность несложно: достаточно проверить, просто ли будет переместить часть функциональности класса в новый класс. Чем проще это сделать, тем ниже взаимосвязанность составных частей функциональности и выше вероятность нарушения принципа SRP.

Вполне возможно, что в `UpdateHasTierPricesProperty` и в `UpdateHasDiscountsApplied` используются одинаковые зависимости, но этим все и заканчивается — между ними нет никакой связности. В результате класс, скорее всего, становится сложным, а его сопровождение — проблемным. Исходя из этого `ProductService` следует разбить на несколько классов. Но тогда возникают вопросы: на сколько классов и какие методы должны быть сгруппированы и нужно ли вообще их группировать? Прежде чем отвечать на них, выясним, как конструкция `IProductService` нарушает принцип OCP.

IProductService нарушает принцип OCP

Чтобы проверить, нарушает ли код принцип OCP, сначала нужно определить, какого рода изменений для этой части приложения можно ожидать. А после этого задать вопрос: «Станет ли данная конструкция причиной широкомасштабной модификации в случае внесения ожидаемых изменений?»

Вполне вероятно, что в течение всего срока службы приложения электронной торговли потребуется внесение двух изменений. Во-первых, нужно будет добавить новые функции (у Мэри они уже намечены). Во-вторых, Мэри все же придется решать сквозные задачи. Если принять во внимание ожидаемые изменения, ответ на вопрос будет очевиден: «Да, существующая в данный момент конструкция без значительных изменений не обойдется». Они произойдут при добавлении как новых функций, так и новых аспектов.

При добавлении новой функции для работы с товарами изменение распространяется на все реализации `IProductService`, которые будут основной реализацией `ProductService`, а также на все декораторы и тестовые дубликаты. А при добавлении новой сквозной функциональности, скорее всего, не обойдется без системных изменений: кроме нового декоратора для `IProductService`, будут добавляться также декораторы для `ICustomerService`, `IOrderService` и прочих абстракций вида `I...Service`. Поскольку каждая абстракция потенциально содержит десятки методов, код аспекта, как говорилось в разделе 10.1, будет повторяться много раз.

ПРИМЕЧАНИЕ

Количество изменений, которые придется вносить в уже существующие декораторы, растет пропорционально количеству функций в системе. Со временем это сделает добавление новых аспектов и функций более затратным, доведя стоимость добавления функций до неприемлемых значений.

В табл. 9.1 приведена сводка широкого спектра аспектов, которые вам могут понадобиться. В начале проекта можно еще и не знать, какие будут нужны. Но даже если пока не сложилось четкое представление о том, какие именно сквозные функции придется добавлять, было бы неплохо догадаться, как делает Мэри, что в ходе проработки проекта этого не избежать.

Завершение анализа IProductService

В результате нашего исследования можно прийти к заключению, что, помимо представленных в нем реализаций, код листинга 10.4 нарушает три из пяти SOLID-принципов. Хотя с позиции AOP для добавления аспектов можно воспользоваться либо динамическим перехватом (см. раздел 11.1), либо инструментами автоматического добавления аспектов в ходе компиляции (см. раздел 11.2), мы утверждаем, что этим решается только часть проблемы, а именно эффективное применение сквозной функциональности с использованием подхода, позволяющего упростить сопровождение реализующего ее кода. Использование инструментов не устраняет основных проблем проекта, которые по-прежнему усложняют сопровождение программного продукта в долгосрочной перспективе.

СОВЕТ

Хотя с ходу отвергать инструментальные методы AOP и не стоит, в первую очередь следует попытаться улучшить конструкцию приложения. Прибегать к инструментальным средствам нужно лишь тогда, когда такое улучшение не решает проблем сопровождаемости программного продукта.

При изучении подразделов 11.1.2 и 11.2.2 выяснится, что недостатки есть у обоих методов AOP. Но посмотрим, можно ли из приложения, созданного Мэри, получить легче сопровождаемую и более приверженную принципам SOLID конструкцию.

10.3.3. Усовершенствование конструкции применением SOLID-принципов

В этом разделе нам предстоит заняться пошаговым улучшением конструкции приложения, предусматривающим:

- отделение операций чтения от операций записи;
- устранение нарушений ISP и SRP с помощью разделения интерфейсов и реализаций;
- устранение нарушения OCP путем введения граничных объектов и общих интерфейсов для реализаций;

- устранение случайно допущенных нарушений LSP в ходе определения обобщенной абстракции.

Шаг 1: отделение операций чтения от операций записи

Одна из проблем конструкции, созданной Мэри, заключается в том, что большинство аспектов, применяемых к `IProductService`, требуются только подмножеству его методов. Хотя такой аспект, как безопасность, обычно применяется ко всем функциям, аспекты ведения контрольного журнала, проверки допустимости и устойчивости к сбоям обычно востребованы только некоторыми частями приложения, изменяющими его состояние. А вот задействование такого аспекта, как кэширование, может иметь смысл для методов, выполняющих чтение данных без изменения состояния. Создание декораторов можно упростить, разбив `IProductService` на интерфейсы, предназначенные только для чтения и только для записи (рис. 10.1).

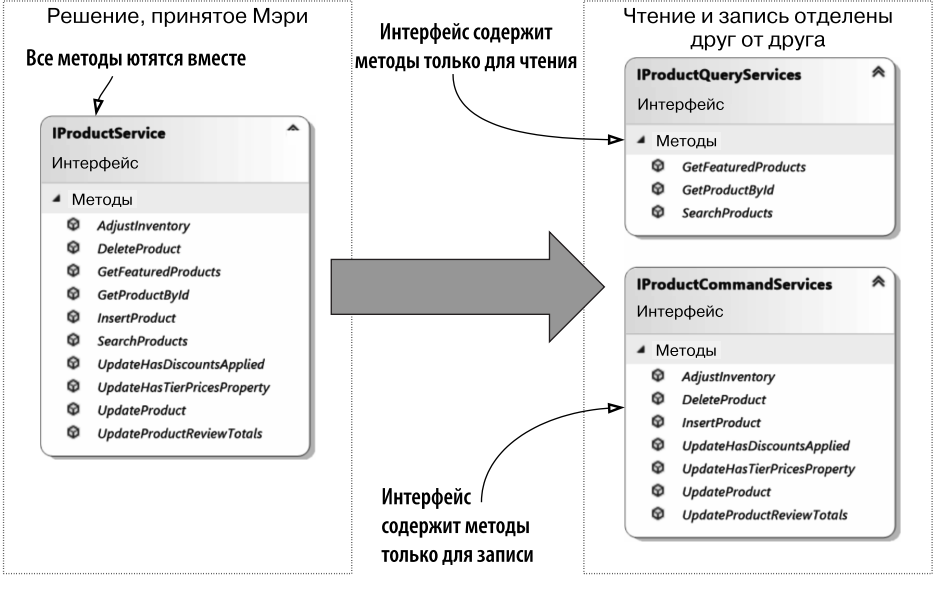


Рис. 10.1. Разбиение IProductService на абстракцию IProductQueryServices, предназначенную только для чтения, и абстракцию IProductCommandServices, предназначенную только для записи

ПРИМЕЧАНИЕ

Понятие «запрос» мы используем для операций, выполняющих только чтение состояния, но не изменяющих состояние системы, а понятие «команда» — для операций, изменяющих состояние системы, но не дающих каких-либо результатов. Эта терминология базируется на принципе разделения команд и запросов (Command-Query Separation, CQS). Мэри уже применяла CQS в отношении IProductService на уровне методов, но за счет разбиения интерфейса распространила его и на уровень интерфейса.

Разделение команд и запросов

Принцип разделения команд и запросов был придуман Бертраном Мейером (Bertrand Meyer) в книге Object-Oriented Software Construction (ISE Inc., 1988). Он существенно повлиял на объектно-ориентированное программирование, продвигая идею, что каждый метод должен делать что-то одно:

- возвращать результат, но не изменять наблюдаемое состояние системы;
- изменять состояние, но не порождать никакого значения.

Мейер называл методы, производящие значения, запросами, а методы, изменяющие состояние, — командами. Цель разделения — упростить рассуждения о методах, относя их либо к запросу, либо к команде, но не к тому и другому одновременно.

Преимущество от такого разбиения заключается в более узкой, чем прежде, специализации новых интерфейсов. При этом снижается риск стать зависимым от ненужных методов. К примеру, когда создается декоратор, применяющий транзакцию к исполняемому коду, в декорировании будет нуждаться только `IProductCommandServices`, из-за чего становится ненужной реализация методов, принадлежащих `IProductQueryServices`. Реализации становятся меньше и проще для понимания.

Хотя такое разбиение является улучшением по сравнению с исходным интерфейсом `IProductService`, новая конструкция все же не избавляет от необходимости внесения широкомасштабных изменений. Как и прежде, реализация нового метода работы с товарами повлечет за собой изменения во многих классах, имеющихся в приложении. Хотя вероятность изменения класса снизилась наполовину, внесение изменения все еще служит причиной тому, что им затрагивается примерно такое же количество классов. Это подводит нас ко второму шагу.

Шаг 2: устранение нарушений ISP и SRP с помощью разделения интерфейсов и реализаций

Поскольку разбиение излишне универсальных интерфейсов ведет нас в правильном направлении, продолжим в том же духе. Мы сконцентрируемся на `IProductCommandServices` и проигнорируем `IProductQueryServices`.

Попробуем сделать что-нибудь радикальное. Разобьем `IProductCommandServices` на несколько однокомпонентных интерфейсов. На рис. 10.2 показано, что реализация `ProductCommandServices` разделена на семь классов, у каждого из которых есть собственный однокомпонентный интерфейс.

На рис. 10.2 каждый метод интерфейса `IProductCommandServices` перемещен в отдельный интерфейс и каждому интерфейсу дан его собственный класс. В листинге 10.6 показаны определения нескольких таких интерфейсов.

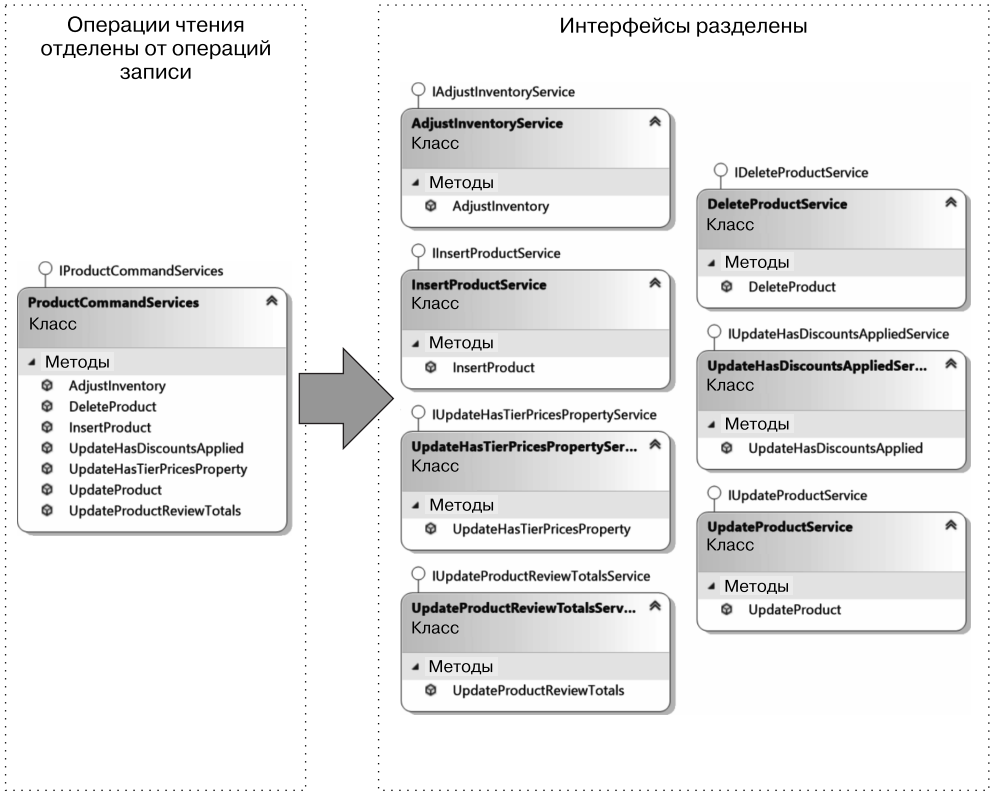


Рис. 10.2. Интерфейс IProductCommandServices, содержащий семь компонентов, заменяется семью однокомпонентными интерфейсами. Каждый интерфейс получает собственную реализацию, соответствующую предназначению

Листинг 10.6. Большой интерфейс разделен на однокомпонентные интерфейсы

```
public interface IAdjustInventoryService
{
    void AdjustInventory(Guid productId, bool decrease, int quantity);
}

public interface IUpdateProductReviewTotalsService
{
    void UpdateProductReviewTotals(Guid productId, ProductReview[] reviews);
}

public interface IUpdateHasDiscountsAppliedService
{
    void UpdateHasDiscountsApplied(Guid productId, string description);
}
...
```

Для краткости остальные четыре интерфейса не показаны

Не стоит сильно пугаться, возможно, все не так уж плохо, как кажется. Перечислим ряд неоспоримых преимуществ таких изменений.

- Каждый интерфейс стал изолированным. Ни один клиент не будет зависеть от методов, которые не использует.
- Когда создается отображение «один в один» интерфейса на реализацию, каждый вариант применения получает собственный класс. Классы становятся небольшими и узкоспециализированными — на них возлагается единственная ответственность.
- Добавление новой функции влечет за собой добавление новой пары «интерфейс — реализация». Никаких изменений в существующие классы, реализующие другие варианты использования, вносить не нужно.

Но даже притом, что новая конструкция соответствует принципам ISP и SRP, создание декораторов все-таки требует внесения широкомасштабных изменений. И дело здесь вот в чем.

- Когда интерфейс `IProductCommandServices` разбит на семь однокомпонентных интерфейсов, на каждый аспект будет приходиться семь декораторов. Получается, к примеру, что для десяти аспектов придется создавать 70 декораторов.
- Внесение изменений в уже существующий аспект потребует больших изменений по всему большому набору классов, поскольку каждый аспект распространяется на множество декораторов.

Новая конструкция приводит к тому, что каждый класс в приложении фокусируется только на одном конкретном варианте использования, что в смысле соблюдения принципов SRP и ISP можно считать весьма положительным обстоятельством. Но из-за отсутствия в них той самой унифицированности, к которой можно было бы применить аспекты, возникает необходимость создания множества декораторов с почти одинаковыми реализациями. Гораздо лучше было бы иметь возможность определить единый интерфейс для всех имеющихся в кодовой базе командных операций. Это существенно уменьшило бы повторяемость кода, относящегося к аспектам, и свело число декораторов классов к одному декоратору для каждого аспекта.

Заметить при изучении листинга 10.6 возможное сходство этих интерфейсов может быть нелегко. Все они возвращают пустой тип `void`, но у каждого имеется метод с оригинальным именем, а у каждого метода — оригинальный набор параметров. Так есть у них что-то общее, пригодное для извлечения, или нет?

Шаг 3: устранение нарушения OCP за счет использования граничных объектов

А что, если извлечь параметры метода у каждого командного метода в граничный объект (Parameter Object)? Большинство инструментов реструктуризации кода позволяют выполнить такую переделку с помощью нескольких нажатий клавиш.

ОПРЕДЕЛЕНИЕ

Граничный объект представляет собой группу параметров, естественным образом связанных друг с другом[1].

Результат такой реструктуризации показан в листинге 10.7.

Листинг 10.7. Заключение параметров метода в граничный объект

```
public interface IAdjustInventoryService
{
    void Execute(AdjustInventory command);
}

public class AdjustInventory
{
    public Guid ProductId { get; set; }
    public bool Decrease { get; set; }
    public int Quantity { get; set; }
}

public interface IUpdateProductReviewTotalsService
{
    void Execute(UpdateProductReviewTotals command);
}

public class UpdateProductReviewTotals
{
    public Guid ProductId { get; set; }
    public ProductReview[] Reviews { get; set; }
}
```

Теперь вместо приема списка параметров IAdjustInventoryService принимает всего один параметр в виде нового объекта-параметра AdjustInventory. В классе этого объекта сгруппированы все параметры метода. Имя метода изменено на более общее — Execute

AdjustInventory содержит сгруппированные параметры метода, принадлежащие IAdjustInventoryService. Это объект-параметр, не содержащий никакого поведения

Та же реструктуризация применена и к этому интерфейсу. Теперь он принимает в качестве своего единственного параметра UpdateProductReviewTotals

Теперь два параметра метода IUpdateProductReviewTotalsService сведены в группу и стали новым объектом-параметром UpdateProductReviewTotals

Важно отметить, что, хотя граничные объекты `AdjustInventory` и `UpdateProductReviewTotals` являются конкретными объектами, они все равно в то же время остаются частью своей абстракции. Как уже говорилось в подразделе 3.1.1, поскольку они просто объекты данных без поведения, сокрытие их значения за абстракцией вряд ли принесет какую-либо пользу. Если переместить реализации в другую сборку, граничные объекты останутся в той же сборке, что и их абстракция. К тому же эти извлеченные граничные объекты становятся определением командной операции. Поэтому обычно мы называем их командами.

СОВЕТ

Наличие у командных граничных объектов всего одного параметра или полное отсутствие каких-либо параметров — это нормально. У обеих команд, InsertProduct и UpdateHasTierPricesProperty, будет всего один параметр типа Product. Но вставка товара представляет собой нечто совершенно иное, чем обновление свойства товара HasTierPrices. Опять же сам по себе тип команды становится определением командной операции.

[1] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 297.

Эти реструктуризации привели к тому, что код существенно изменился: из одного интерфейса и реализации с семью методами получились семь интерфейсов и четырнадцать классов. Вы можете подумать, что мы повредились рассудком, и, возможно, захотите выбросить эту книгу в окно. Это, видимо, тот самый душевный дискомфорт, о котором мы предупреждали в самом начале раздела. Потерпите еще немного, поскольку увеличение числа классов в вашей системе может быть не таким уж и плохим, как может показаться на первый взгляд, и структуризация приведет нас к чему-нибудь полезному. Обещаем, так оно и будет.

ВНИМАНИЕ

В результате этой реструктуризации в проекте увеличивается число файлов, но так как каждый класс и интерфейс получают в проекте собственный файл, исполняемый код не изменяется. Каждый метод содержит столько же кода, сколько и раньше. Каждому варианту использования дан его собственный объект данных, и каждый класс теперь имеет дело с одним вариантом.

После реструктуризации получился следующий паттерн.

- Каждая абстракция содержит всего один метод.
- Каждый метод получил название `Execute`.
- Каждый метод возвращает пустой тип `void`.
- У каждого метода есть всего один входной параметр.

Теперь из этого паттерна можно извлечь общий интерфейс:

```
public interface ICommandService
{
    void Execute(object command);
}
```

Один интерфейс, чтобы управлять всеми!

Если сервисы команд реализуются с использованием нового интерфейса `ICommandService`, в результате получается код, показанный в листинге 10.8. Обратите внимание на то, что определением этого нового интерфейса можно, скорее всего, воспользоваться и для замены других абстракций вида `I...Service`.

На рис. 10.3 показано, что число интерфейсов сократилось с семи до одного, вернувшись к прежнему количеству. Но теперь параметры метода извлекаются в граничный объект для каждого сервиса.

Листинг 10.8. AdjustInventoryService, реализующий ICommandService

```
public class AdjustInventoryService : ICommandService
{
    readonly IInventoryRepository repository;

    public AdjustInventoryService(
        IInventoryRepository repository)
    {
        this.repository = repository;
```

Вместо IAdjustInventoryService реализуется ICommandService

Для зависимостей класса используется внедрение через конструктор

```
    }

    public void Execute(object cmd)
    {
        var command = (AdjustInventory)cmd;

        Guid id = command.ProductId;
        bool decrease = command.Decrease;
        int quantity = command.Quantity;
        ...
    }
}
```

Execute принимает значение типа object, но, поскольку известно, что AdjustInventoryService поставляется с сообщением о команде AdjustInventory, выполняется приведение типа

Получение доступа к параметрам команды и выполнение соответствующего кода. Это код, который изначально был помещен в метод AdjustInventory, принадлежащий классу ProductService

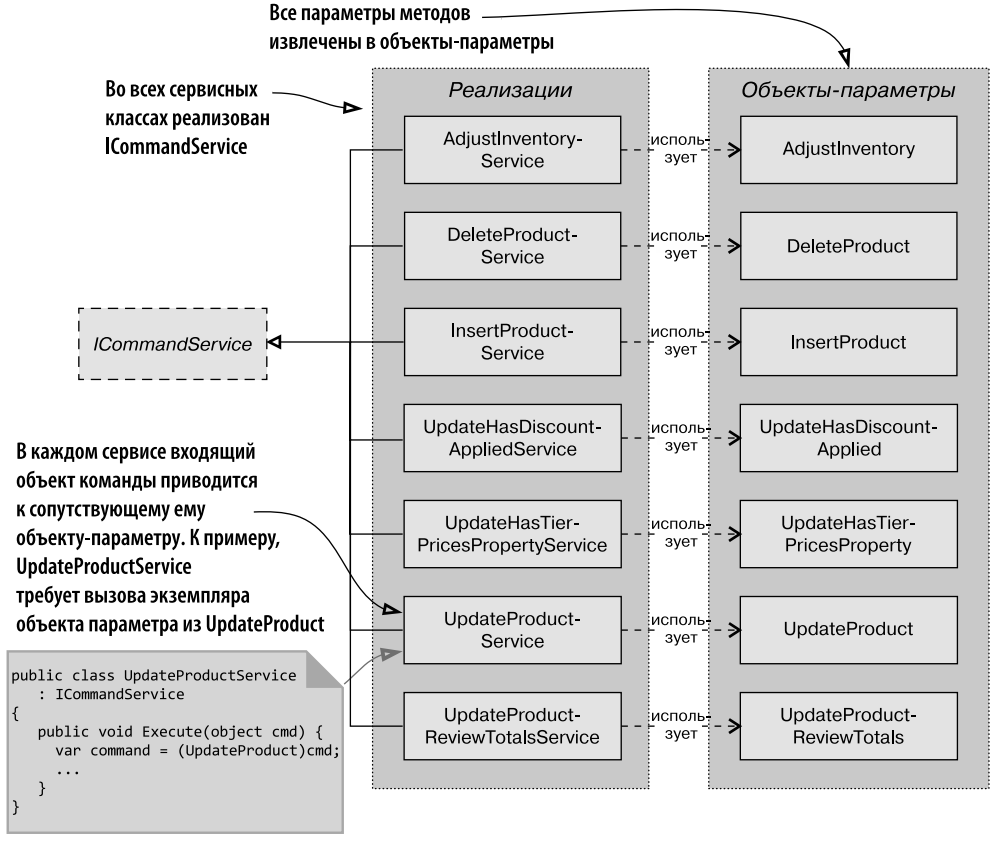


Рис. 10.3. Благодаря извлечению параметров методов в граничные объекты количество интерфейсов сократилось с семи до одного ICommandService

Как утверждалось ранее, граничные объекты являются частью абстракции. Объединение всех интерфейсов в один делает его еще более очевидным. Граничный объект становится определением варианта использования, то есть контрактом.

Потребители могут получать `ICommandService` внедренным в их конструкторы и вызывать его метод `Execute`, предоставляя соответствующий граничный объект (листинг 10.9).

Листинг 10.9. InventoryController зависит от ICommandService

```
public class InventoryController : Controller
{
    private readonly ICommandService service;

    public InventoryController(ICommandService service)
    {
        this.service = service;
    }

    [HttpPost]
    public ActionResult AdjustInventory(
        AdjustInventoryViewModel viewModel)
    {
        if (!this.ModelState.IsValid)
        {
            return this.View(viewModel);
        }

        AdjustInventory command = viewModel.Command;

        this.service.Execute(command);

        return this.RedirectToAction("Index");
    }
}
```

Внедрение ICommandService в класс контроллера MVC

AdjustInventoryViewModel оборачивает команду AdjustInventory в качестве свойства

Если данные допустимы, команда передается в ICommandService для выполнения

`AdjustInventoryViewModel` заключает в себя команду `AdjustInventory` в качестве свойства. Получается очень удобно, поскольку `AdjustInventory` является частью абстракции и содержит только данные, специфичные для варианта использования. Когда пользователь отправляет запрос обратно, команда `AdjustInventory` будет привязана к модели средой MVC вместе с окружающей ее `AdjustInventoryViewModel`.

ПРИМЕЧАНИЕ

Если вы заметили, что код листинга 10.9 нарушает LSP, нам остается только аплодировать. Вскоре мы займемся и этим нарушением.

Использование ICommandService для реализации сквозной функциональности

Наличие единого интерфейса для всех вызовов сервиса команд дает огромные преимущества. Поскольку теперь все варианты применения, изменяющие состояние приложения, реализуются этим единым интерфейсом, появляется возможность создания единого декоратора для каждого аспекта и заключения в него абсолютно

всех реализаций. В качестве доказательства в листинге 10.10 показана реализация аспекта транзакции в виде декоратора для `ICommandService`.

Листинг 10.10. Использование аспекта транзакции, основанного на ICommandService

```
public class TransactionCommandServiceDecorator
    : ICommandService
{
    private readonly ICommandService decoratee;

    public TransactionCommandServiceDecorator(
        ICommandService decoratee)
    {
        this.decoratee = decoratee;
    }

    public void Execute(object command)
    {
        using (var scope = new TransactionScope())
        {
            this.decoratee.Execute(command);

            scope.Complete();
        }
    }
}
```

Поскольку этот декоратор похож на тот код, который уже много раз встречался в главе 9, мы считаем, что он не требует особых пояснений, кроме разве что класса `TransactionScope`.

TransactionScope

Класс `System.Transactions.TransactionScope` из библиотеки `System.Transactions.dll` позволяет заключать любой произвольный фрагмент кода в транзакцию. Любой объект `DbTransaction`, созданный в период существования этой области видимости, автоматически включается в ту же транзакцию. В результате получается весьма эффективная концепция, позволяющая применять транзакции к нескольким фрагментам кода, принадлежащим одной и той же бизнес-операции, без необходимости в передаче транзакции через стек вызовов.

По сравнению с полной средой .NET среда .NET Core не поддерживает распределенные транзакции, поскольку для них требуется сервис Microsoft Distributed Transaction Coordinator (MSDTC), эквивалент которого отсутствует на платформах, отличных от Windows. Мы считаем это очень полезной особенностью, поскольку полагаем, что в целом распределенные транзакции в любом случае должны быть предотвращены. Но при использовании среды .NET Core `TransactionScope` все же можно задействовать, чтобы подключить операции транзакции к одному источнику данных.

Теперь, применяя новый декоратор, можно скомпоновать `InventoryController` путем внедрения нового `AdjustInventoryService`, перехватываемого `TransactionCommandServiceDecorator`:

```
ICommandService service =
    new TransactionCommandServiceDecorator(
        new AdjustInventoryService(repository));

new InventoryController(service);
```

Такая конструкция, по сути, исключает внесение широкомасштабных изменений как при добавлении новых функций, так и при необходимости применения новой сквозной функциональности. Теперь эта конструкция по-настоящему закрыта для изменений, потому что:

- добавление новой (командной) функции влечет за собой создание нового командного граничного объекта и поддержку реализации `ICommandService`. Ни один уже существующий класс в изменении не нуждается;
- добавление новой функции не требует ни создания новых декораторов, ни внесения изменений в уже существующие;
- добавление к приложению новой сквозной функциональности может выполняться добавлением всего лишь одного декоратора;
- изменение сквозной функциональности приводит к изменению только одного класса.

ВНИМАНИЕ

> Даже при переходе от ситуации, показанной в листинге 10.4, где было два типа (интерфейс IProductService и его реализация), к ситуации, показанной на рис. 10.3, где имеется 15 типов (один интерфейс, семь граничных объектов и семь реализаций сервисов), сопровождение приложения существенно упростилось, поскольку широкомасштабные изменения будут происходить крайне редко. Из этого можно сделать важный вывод, что само по себе количество классов не позволяет оценить легкость сопровождения программного продукта.

Некоторые разработчики против такого большого количества классов в своих системах, поскольку считают, что это усложняет перемещение по коду проекта. Но такое случается только при неподходящей структуризации проекта. В этом примере все операции, работающие с товарами, могут быть помещены в пространство имен `MyApp.Services.Products`, что позволит эффективно свести их в одну группу, подобно тому как это делалось в созданном Мэри интерфейсе `IProductService`. Теперь вместо создания группировок на уровне классов их располагают на уровне проекта, обеспечивая большое преимущество за счет того, что структура проекта сразу показывает всю суть поведения приложения.

Теперь, после устранения проанализированных нарушений SOLID-принципов, можно посчитать, что реструктуризация завершена. Но, к сожалению, внесенные изменения случайно привели к новому нарушению SOLID-принципов. Рассмотрим сложившуюся ситуацию.

Анализ нового случайного нарушения принципа LSP

Итак, при определении `ICommandService` было допущено новое случайное нарушение SOLID-принципов, а именно LSP. Это нарушение в коде `InventoryController` показано в листинге 10.9.

Как говорилось в подразделе 10.2.3, принцип LSP гласит, что вы должны иметь возможность выполнить подстановку абстракции для ее произвольной реализации, не изменяя корректность кода клиента. В соответствии с принципом LSP, поскольку `AdjustInventoryService` реализует `ICommandService`, у вас должна быть возможность заменить его другой реализацией, не нарушая корректности кода `InventoryController`. Измененная компоновка объектов для `InventoryController` показана в листинге 10.11.

Листинг 10.11. Подстановка AdjustInventoryService

```
ICommandService service =
    new TransactionCommandServiceDecorator(
        new UpdateProductReviewTotalsService(
            repository));

new InventoryController(service);
```

Вместо AdjustInventoryService внедряется UpdateProductReviewTotalsService. Код проходит компиляцию, но полностью ломает InventoryController, а это нарушение LSP

В следующем фрагменте кода показан метод `Execute` для `UpdateProductReviewTotalsService`:

```
public void Execute(object cmd)
{
    var command = (UpdateProductReviewTotals)cmd;
    ...
}
```

Это приведение типа дает сбой, когда методу Execute предоставляется команда типа AdjustInventory

`InventoryController` получает `ICommandService` внедренным в его конструктор. Он передает команду `AdjustInventory` этому внедренному `ICommandService`. Поскольку внедренным `ICommandService` является `UpdateProductReviewTotalsService`, он попытается выполнить приведение поступившей команды к типу `UpdateProductReviewTotals`. Но, так как выполнить приведение `AdjustInventory` к типу `UpdateProductReviewTotals` не представляется возможным, данное действие дает сбой. Это приводит к нарушению работы `InventoryController` и, следовательно, нарушает принцип LSP.

ПРИМЕЧАНИЕ

DI-контейнеры составляют граф объектов на основе информации о типе, извлеченной из аргументов конструктора типа. Поскольку их основной метод компоновки объектов построен именно на этом, DI-контейнеры плохо подходят для работы с неопределенными абстракциями. Следовательно, нарушения принципа LSP при использовании DI-контейнера обычно усложняют корень композиции. Иными словами, применение DI-контейнера делает нарушения LSP более очевидными, точно так же, как внедрение через конструктор делает очевидным нарушение SRP.

Конечно, можно возразить, что обеспечение правильной реализации — прерогатива корня композиции, но интерфейс `ICommandService` по-прежнему является причиной неопределенности и мешает компилятору проверить, имеет ли композиция нашего графа объектов какой-либо смысл. Нарушения LSP могут сделать систему неустойчивой. Более того, нетипизированный аргумент командного метода, потребляемый методами `Execute`, требует, чтобы каждая реализация `ICommandService` содержала код приведения типа, что само по себе может рассматриваться как проблемный код. Устраним это нарушение.

Шаг 4: устранение нарушения LSP с помощью обобщенной абстракции

Рассмотрим весьма элегантный выход из этого, казалось бы, неразрешимого тупика нашей конструкции. Для устранения проблемы нужно всего лишь изменить определение `ICommandService` (листинг 10.12).

Листинг 10.12. Обобщенная реализация ICommandService

```
public interface ICommandService<TCommand>
{
    void Execute(TCommand command);
}
```

TCommand является обобщенным типом аргумента. Он указывает тип той команды, которая будет выполняться реализацией

Можно, конечно, удивиться и спросить: чем же поможет приведение интерфейса к обобщенному виду? Чтобы прояснить ответ на данный вопрос, в листинге 10.13 показано, как реализовать `ICommandService<TCommand>`.

Листинг 10.13. AdjustInventoryService реализует ICommandService<TCommand>

```
public class AdjustInventoryService
    : ICommandService<AdjustInventory>
{
    private readonly IInventoryRepository repository;

    public AdjustInventoryService(
        IInventoryRepository repository)
    {
        this.repository = repository;
    }

    public void Execute(AdjustInventory command)
    {
        var productId = command.ProductId;

        ...
    }
}
```

Реализация ICommandService<TCommand>, показывающая, что этот класс обрабатывает сообщения AdjustInventory

Поскольку класс реализует ICommandService<AdjustInventory>, теперь его метод Execute вместо объекта принимает AdjustInventory

Поскольку теперь Execute принимает AdjustInventory напрямую, параметр command может использоваться непосредственно, без какого-либо приведения типов

ВНИМАНИЕ

> Помните, в листинге 6.9 был показан код определения абстракции IEventHandler<TEvent>? Сигнатура этой новой абстракции ICommandService<TCommand> идентична сигнатуре IEventHandler<TEvent>, и это не совпадение. Если следовать принципам SOLID, то при разработке кодовой базы такая структура возникает довольно часто и представляет собой обобщенные однокомпонентные интерфейсы, принимающие и/или возвращающие сообщения на основе своих обобщенных типов.

Многие среды и имеющиеся в Сети эталонные образцы архитектур используют разные имена для интерфейса, аналогичного показанному в предыдущих примерах. Это могут быть `IHandler<T>`, `ICommandHandler<T>`, `IMessageHandler<T>` или `IHandleMessages<T>`. Некоторые абстракции имеют асинхронную природу и возвращают задачу `Task`, а другие добавляют в качестве аргумента метода маркер отмены (Cancellation Token). Иногда метод называется `Handle` или `HandleAsync`. Имена разные, но сама идея и то влияние, которое она оказывает на степень легкости сопровождения вашего приложения, одинаковые.

Хотя дополнительная помощь в реализации при компиляции — это, несомненно, хорошо, основной причиной использования обобщенного интерфейса `ICommandService<TCommand>` является предотвращение нарушения LSP в его клиентах. Как внедрение `ICommandService<TCommand>` в `InventoryController` устраняет нарушение LSP, показано в листинге 10.14.

Листинг 10.14. InventoryController зависит от ICommandService<TCommand>

```
public class InventoryController : Controller
{
    readonly ICommandService<AdjustInventory> service;

    public InventoryController(
        ICommandService<AdjustInventory> service)
    {
        this.service = service;
    }
    public ActionResult AdjustInventory(
        AdjustInventoryViewModel viewModel)
    {
        ...
        AdjustInventory command = viewModel.Command;

        this.service.Execute(command);

        return this.RedirectToAction("Index");
    }
}
```

Внедрение определенного ICommandService<AdjustInventory> указывает на то, что нужно выполнить команду AdjustInventory. Так предотвращается случайное внедрение сервисов, обрабатывающих UpdateProductReviewTotals, или любого другого объекта типа command

Параметр service принимает в качестве объекта типа command только AdjustInventory. Предоставление сообщения другого типа невозможно, поскольку код не пройдет компиляцию

СОВЕТ

> Если ваш язык программирования не поддерживает обобщений, можно было бы в качестве приемлемого пути обхода найти вариант использования необобщенного интерфейса ICommandService в сочетании с паттерном проектирования «Посредник» (Mediator)[1]. В таком случае вводится дополнительная абстракция «Посредник», принимающая произвольные команды и внедряемая в потребители. Работой посредника будет направление предоставляемых команд нужной реализации ICommandService.

Замена необобщенного `ICommandService` обобщенным `ICommandService<TCommand>` устраняет последнее нарушение SOLID-принципов. Самое время воспользоваться преимуществами нашей новой конструкции.

Применение обработки транзакций с использованием обобщенной абстракции

Хотя область использования обобщенной однокомпонентной абстракции значительно шире, чем создание сквозной функциональности, возможность задействовать аспекты таким образом, чтобы не требовалось вносить широкомасштабные изменения, — одно из наиболее примечательных преимуществ такой конструкции. Как и в случае применения необобщенного интерфейса `ICommandService`, интерфейс `ICommandService<TCommand>` позволяет создавать для каждого аспекта всего один декоратор. В листинге 10.15 показан переделанный код декоратора транзакции из листинга 10.10, в котором задействуется новая обобщенная абстракция `ICommandService<TCommand>`.

Листинг 10.15. Реализация обобщенного аспекта транзакции

```
public class TransactionCommandServiceDecorator<TCommand>
    : ICommandService<TCommand>
{
    private readonly ICommandService<TCommand> decoratee;

    public TransactionCommandServiceDecorator(
        ICommandService<TCommand> decoratee)
    {
        this.decoratee = decoratee;
    }

    public void Execute(TCommand command)
    {
        using (var scope = new TransactionScope())
        {
            this.decoratee.Execute(command);
```

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 319.

```
            scope.Complete();
        }
    }
}
```

При использовании интерфейса `ICommandService<TCommand>` и декоратора `TransactionCommandServiceDecorator<TCommand>` корень композиции приобретает следующий вид:

```
new InventoryController(
    new TransactionCommandServiceDecorator<AdjustInventory>(
        new AdjustInventoryService(repository)));
```

Таким образом, наступает момент, когда однокомпонентная обобщенная абстракция начинает затмевать все остальное. Именно теперь мы начинаем добавлять новые аспекты сквозной функциональности.

10.3.4. Добавление новых аспектов сквозной функциональности

Все примеры сквозной функциональности, рассмотренные в разделе 9.2 (например, показанные в листингах 9.4 и 9.7), сфокусированы на применении аспектов в пределах хранилищ данных. Но в этом разделе фокус в многоуровневой архитектуре будет смещен на один уровень вверх, от доступа к данным библиотечного хранилища к `IProductService`, принадлежащему библиотеке доменного уровня.

Это вполне осознанное смещение, поскольку скоро выяснится, что хранилища не являются подходящим уровнем детализации для эффективного применения многих аспектов сквозной функциональности. Всего лишь одно бизнес-действие, определенное на доменном уровне, потенциально будет выполнять вызовы к нескольким хранилищам или же вызывать одно и то же хранилище несколько раз. Если бы, к примеру, транзакция была выполнена на уровне хранилища, это все равно означало бы, что бизнес-операция может выполняться в десятках транзакций, что способно поставить под угрозу корректность всей системы.

Как правило, одна бизнес-операция должна запускаться в одной транзакции. Этот уровень детализации распространяется не только на транзакции, но и на другие типы операций.

Бизнес-операции реализуются в доменной библиотеке, и именно в ее границах обычно и нужно применять многие аспекты сквозной функциональности. Далее перечислен ряд примеров. Этот список не исчерпывающий, но он даст вам представление о том, что можно использовать на этом уровне.

- *Ведение контрольного журнала.* Несмотря на то что эту сквозную функциональность можно обернуть вокруг хранилищ, как делалось в `AuditingUserRepositoryDecorator` в коде листинга 9.1, в таком журнале будет представлен перечень изменений отдельно взятых сущностей, а общая картина, вскрывающая причину изменения, будет смазана. Создание отчетов об изменениях отдельно взятых сущностей могло бы подойти для приложений, основанных на выполнении

CRUD-операций, но если в приложении реализуются более сложные сценарии использования, влияющие более чем на одну сущность, выгоднее переместить ведение контрольного журнала на один уровень вверх и сохранить информацию о выполненной команде. Пример ведения контрольного журнала будет показан чуть позже.

- *Регистрация событий.* Как упоминалось в подразделе 5.3.2, хорошо продуманная конструкция приложения может не допустить распространения ненужных инструкций регистрации событий по всей кодовой базе. Регистрация любой бизнес-операции с ее данными дает вам подробную информацию о вызове, что, как правило, избавляет от необходимости создания регистрационной записи в начале выполнения каждого метода.
- *Мониторинг производительности.* Поскольку 99 % времени выполнения запроса обычно затрачивается на выполнение самой бизнес-операции, `ICommandService<TCommand>` становится идеальной границей для встраивания мониторинга производительности.
- *Обеспечение безопасности.* Можно попытаться ограничить доступ на уровне хранилища, но обычно такая детализация — это лишнее, так как вам, скорее всего, потребуется ограничить доступ на уровне бизнес-операции. Команды можно снабдить информацией о требуемой роли либо разрешении, сведя тем самым обеспечение безопасности к использованию всего одного декоратора. Соответствующий пример вскоре будет приведен.
- *Обеспечение устойчивости к сбоям.* Поскольку транзакции нужно ограничить кругом бизнес-операций, то, как показано в листинге 10.15, на том же самом уровне, как правило, должны задействоваться другие аспекты устойчивости к сбоям, например аспект повторных попыток при взаимной блокировке базы данных. Такой механизм всегда должен применяться к аспекту транзакций.
- *Проверка допустимости.* Как мы показали в листингах 10.9 и 10.14, команда может стать частью данных, отправленных с веб-запросом. При сопровождении команд атрибутами проверки достоверности (Data Annotations) данные команд тоже будут проверены средой MVC[1]. В качестве особой меры безопасности можно создать декоратор, проверяющий входящие команды с использованием статического класса `Validator` из библиотеки Data Annotations[2].

В следующем разделе рассматривается способ реализации двух из этих аспектов, представляющий собой надстройку над `ICommandService<TCommand>`.

Пример: реализация аспекта ведения контрольного журнала

В листингах 9.1 и 9.2 был определен декоратор для `IUserRepository`, в котором повторно использован интерфейс `IAuditTrailAppender` из листинга 6.23. Если вместо этого задействовать ведение контрольного журнала в отношении

---

[1] System.ComponentModel.DataAnnotations — независимой от конкретной среды применения библиотекой проверки допустимости данных, разработанной компанией Microsoft.

[2] Пример такого декоратора приводится по адресу https://simpleinjector.org/aop#decoration.

`ICommandService<TCommand>`, получится просто идеальный уровень детализации, поскольку в команде содержатся все интересующие нас данные применительно к варианту сценария, которые можно будет записать. Если сопроводить эти данные и метаданные определенным объемом информации, например указать имя пользователя и текущее системное время, то, по большому счету, этого будет вполне достаточно. Декоратор ведения контрольного журнала выполнен в виде надстройки над `ICommandService<TCommand>` (листинг 10.16).

Листинг 10.16. Реализация обобщенного аспекта ведения контрольного журнала для бизнес-операций

```
public class AuditingCommandServiceDecorator<TCommand>
    : ICommandService<TCommand>
{
    private readonly IUserContext userContext;
    private readonly ITimeProvider timeProvider;
    private readonly CommerceContext context;
    private readonly ICommandService<TCommand> decoratee;

    public AuditingCommandServiceDecorator(
        IUserContext userContext,
        ITimeProvider timeProvider,
        CommerceContext context,
        ICommandService<TCommand> decoratee)
    {
        this.userContext = userContext;
        this.timeProvider = timeProvider;
        this.context = context;
        this.decoratee = decoratee;
    }

    public void Execute(TCommand command)
    {
        this.decoratee.Execute(command);
        this.AppendToAuditTrail(command);
    }

    private void AppendToAuditTrail(TCommand command)
    {
        var entry = new AuditEntry
        {
            UserId = this.userContext.CurrentUser.Id,
            TimeOfExecution = this.timeProvider.Now,
            Operation = command.GetType().Name,
            Data = Newtonsoft.Json.JsonConvert
                .SerializeObject(command)
        };

        this.context.AuditEntries.Add(entry);
        this.context.SaveChanges();
    }
}
```

Следует напомнить, что это интерфейс ITimeProvider из листинга 5.10

Кроме добавления в контрольный журнал имени пользователя и времени выполнения, декоратор сохраняет название команды, а также выполняет сериализацию представления ее данных. Эта информация собирается с использованием отображения, в данном случае задействуется известная библиотека сериализации JSON.NET (https://www.newtonsoft.com/json), преобразующая данные команды в читаемый формат JSON

ПРИМЕЧАНИЕ

В этом декораторе логика ведения контрольного журнала сочетается с логикой декорирования. Хорошо это или плохо, зависит от объема логики в декораторе, необходимости ее повторного использования другими классами, и от того, в каком модуле размещается декоратор. Поскольку теперь в него можно заключить все бизнес-операции, мы заверяем вас, что совместное применение этой логики с другими классами вряд ли имеет смысл, поэтому мы объединили эти два класса. Но из-за зависимости от CommerceContext этот декоратор должен размещаться либо на уровне доступа к данным, либо в корне композиции.

Когда Мэри запускает приложение, используя `AuditingCommandServiceDecorator<TCommand>`, декоратор выдает в таблицу контрольного журнала информацию, приведенную в табл. 10.2.

Таблица 10.2. Пример контрольного журнала

Пользователь	Время	Операция	Данные
Mary	2018-12-24 11:20	AdjustInventory	{ ProductId: "ae361...00bc", Decrease: false, Quantity: 2 }
Mary	2018-12-24 11:21	UpdateHasTierPricesProperty	{ Product: { Id: "ae361...00bc", Name: "Gruyère", UnitPrice: 48.50, IsFeatured: true } }
Mary	2018-12-24 11:25	UpdateHasDiscountsApplied	{ ProductId: "ae361...00bc", DiscountDescription: "Test" }
Mary	2018-12-24 15:11	AdjustInventory	{ ProductId: "5435...a845", Decrease: true, Quantity: 1 }
Mary	2018-12-24 15:12	UpdateProductReviewTotals	{ ProductId: "5435...a845", Reviews: [{ Rating: 5, Text: "nice!" }] }

Ранее уже говорилось, что `AuditingCommandServiceDecorator<TCommand>` получает название команды и преобразует ее в JSON-формат с помощью отражения. Хотя этот формат считается читаемым, показывать его конечным пользователям вряд ли стоит. И все же для внутреннего аудита он неплох. Использование этой информации дает возможность составить четкое представление о том, что происходит в вашей системе, кто что сделал и когда. Она даже позволит вам повторить операцию, если по каким-то причинам та даст сбой, или воспользоваться ею для проведения в системе реалистичного стресс-теста. Сериализованную информацию из этой таблицы можно опять превратить в команды, запустив их в системе.

Как говорилось в подразделе 6.3.2, работа с доменными событиями очень хорошо подходит для ведения контрольного журнала. Но данный аспект аудита позволяет лишь записывать сведения об успешно выполненных действиях. Хотя для ведения контрольного журнала сбои могут быть неважны, разработчикам интересны и такие события. Нетрудно представить, как тот же механизм можно использовать для записи таких же данных, а в случае сбоя операции включить трассировку стека.

ВНИМАНИЕ

Конструкция приложения прошла путь от модели вызова методов, подобной сервису вызова удаленных процедур, до модели передачи сообщений. Эти сообщения могут сериализоваться, выстраиваться в очередь, регистрироваться и выдаваться повторно, и обеспечить все эти возможности, применяя модель вызова методов, подобную исходной реализации IProductService из листинга 10.4, гораздо труднее.

Регистрация в качественно спроектированных приложениях требует всего нескольких строк кода

При использовании абстракции, заключающей в себя бизнес-транзакцию, подобно тому как это делает `ICommandService<TCommand>`, параметры метода становятся легко сериализуемым пакетом данных, показанным в листинге 10.16. Получается, всего один декоратор позволяет применить регистрацию для широкого диапазона методов приложения.

Этим можно удовлетворить все ваши регистрационные потребности. Но мы по собственному опыту знаем: когда приложение становится сложнее, но оно качественно спроектировано и придерживается принципов SOLID, то абстракции позволяют определить несколько декораторов, удовлетворяющих 98 % его регистрационных потребностей. Но есть и другие методы, которые требуется применять, чтобы предотвратить необходимость регистрации событий на слишком большом количестве участков приложения.

- Вместо регистрации неожиданных ситуаций, не останавливая при этом выполнения программы, стоит выдавать исключения[1].
- Вместо перехвата неожиданных исключений, регистрации события и продолжения выполнения в середине операции оставить исключение необработанным и дать ему распространиться вверх по стеку вызовов. Допуская быстрый сбой операции, вы позволите исключению быть зарегистрированным в одном месте — на вершине стека вызовов, и у пользователей не возникнет иллюзии успешного завершения их запроса.
- По возможности уменьшать объем кода методов[2]. Это повысит читаемость кода, а в случае выдачи исключений трассировка стека даст вам больше информации о пути, пройденном приложением до времени выдачи исключения.

Кроме того, этой информацией можно воспользоваться и для мониторинга производительности, дополнительно сохраняя промежуток времени сразу же за временем и деталями операции. Тогда можно будет легко отследить, выполнение каких операций со временем замедлилось. Прежде чем продемонстрировать пример нового корня композиции с применением `AuditingCommandServiceDecorator<TCommand>`,

[1] Прочтите, к примеру, статью Джеффа Этвуда (Jeff Atwood) The Problem With Logging, https://blog.codinghorror.com/the-problem-with-logging/ (2008).

[2] См. книгу: *Мартин Р. К.* Чистый код. Создание, анализ и рефакторинг. — СПб.: Питер, 2019.

поговорим о том, как можно воспользоваться пассивными атрибутами для реализации аспекта обеспечения безопасности.

Пример: реализация аспекта обеспечения безопасности

В ходе изучения сквозной функциональности в разделе 9.2 был реализован `SecureProductRepositoryDecorator`, код которого показан в листинге 9.7. Поскольку этот декоратор касался конкретно `IProductRepository`, было ясно, какой роли он должен предоставлять доступ. В этом примере доступ к методам записи `IProductRepository` был ограничен ролью администратора.

В новой обобщенной модели в один декоратор заключаются все бизнес-операции, а не только CRUD-операции с товарами. Некоторые операции должны выполняться и по запросу пользователей с другими ролями, при этом жестко заданная роль администратора для обобщенной модели не подходит. Реализовать соответствующую проверку для обеспечения безопасности в виде надстройки над абстракцией можно множеством способов, но одним из самых привлекательных является использование пассивных атрибутов.

ОПРЕДЕЛЕНИЕ

> Пассивный атрибут предоставляет не поведение, а метаданные. Пассивные атрибуты не дают сформироваться антипаттерну «Диктатор», возникновение которого обусловливается атрибутами аспектов, включающими поведение, зачастую представленное нестабильными зависимостями[1].

Если в качестве примера авторизации придерживаться обеспечения безопасности на основе ролей, можно указать пассивный атрибут `PermittedRoleAttribute` (листинг 10.17).

Листинг 10.17. Пассивный атрибут PermittedRoleAttribute

```
public class PermittedRoleAttribute : Attribute
{
    public readonly Role Role;

    public PermittedRoleAttribute(Role role)
    {
        this.Role = role;
    }
}

public enum Role
{
    PreferredCustomer,
    Administrator,
    InventoryManager
}
```

Этот пассивный атрибут позволяет оснастить классы метаданными о разрешенной роли

Заключает в себе используемое в приложении перечисление ролей Role, в котором определяется фиксированный набор ролей приложения

Перечисление Role содержит известные роли, задействованные в приложении. Впервые оно было показано в подразделе 3.1.2

[1] *Seemann M.* Passive attributes, 2014; https://blog.ploeh.dk/2014/06/13/passive-attributes/.

Этот атрибут можно использовать для оснащения команд метаданными о ролях, позволяющих выполнять операцию (листинг 10.18).

Листинг 10.18. Оснащение команд метаданными, с помощью которых обеспечивается безопасность

```
[PermittedRole(Role.InventoryManager)]
public class AdjustInventory
{
    public Guid ProductId { get; set; }
    public bool Decrease { get; set; }
    public int Quantity { get; set; }
}

[PermittedRole(Role.Administrator)]
public class UpdateProductReviewTotals
{
    public Guid ProductId { get; set; }
    public ProductReview[] Reviews { get; set; }
}
```

Пометка команды атрибутом разрешенной роли PermittedRoleAttribute при указании этой роли. В данном случае команда AdjustInventory может исполняться для пользователей, играющих роль InventoryManager, а для исполнения команды UpdateProductReviewTotals нужна авторизация исключительно в роли администратора

ВНИМАНИЕ

Обратите внимание на то, как разрешенная роль в листинге 10.18 становится частью определения команды.

Использование пассивных атрибутов, таких как `PermittedRoleAttribute`, сильно отличается от применения атрибутов аспекта, и эти отличия будут рассмотрены в разделе 11.2. По сравнению с атрибутами аспекта пассивные атрибуты отделены от того аспекта, который задействует их значения, что, как выяснится в главе 11, становится одной из основных проблем при автоматическом добавлении аспекта в ходе компиляции. У пассивных атрибутов нет непосредственной связи с аспектом. Это позволяет сразу нескольким аспектам повторно использовать метаданные и, возможно, разными способами.

СОВЕТ

Вместо повторного задействования атрибута, связанного с определенной средой выполнения приложения, следует создавать доменно-ориентированный атрибут. Например, использование принадлежащего среде ASP.NET Core атрибута [Authorize] приведет к перетаскиванию зависимости к Microsoft.AspNetCore.Authorization.dll, что неприемлемо для повторного применения домена, к примеру, в служебном приложении Windows.

Подобно рассмотренному ранее, добавление поведения, связанного с обеспечением безопасности, сводится к созданию декоратора и заключению в него настоящей реализации. Код такого декоратора показан в листинге 10.19. В нем используется `PermittedRoleAttribute`, предоставляемый, как показано в листинге 10.18, командам.

Листинг 10.19. SecureCommandServiceDecorator<TCommand>

```
public class SecureCommandServiceDecorator<TCommand>
    : ICommandService<TCommand>
{
    private static readonly Role PermittedRole = GetPermittedRole();

    private readonly IUserContext userContext;
    private readonly ICommandService<TCommand> decoratee;

    public SecureCommandServiceDecorator(
        IUserContext userContext,
        ICommandService<TCommand> decoratee)
    {
        this.decoratee = decoratee;
        this.userContext = userContext;
    }

    public void Execute(TCommand command)
    {
        this.CheckAuthorization();
        this.decoratee.Execute(command);
    }

    private void CheckAuthorization()
    {
        if (!this.userContext.IsInRole(PermittedRole))
        {
            throw new SecurityException();
        }
    }

    private static Role GetPermittedRole()
    {
        var attribute = typeof(TCommand)
            .GetCustomAttribute<PermittedRoleAttribute>();

        if (attribute == null)
        {
            throw new InvalidOperationException(
                "[PermittedRole] missing.");
        }

        return attribute.Role;
    }
}
```

Получаем роль, разрешающую выполнять эту команду

Декоратор зависит от IUserContext, позволяющего ему проверять роль текущего пользователя

Прежде чем делегировать вызов задекорированному методу, проверяем, разрешено ли пользователю выполнять эту операцию

Если пользователь не принадлежит к указанной роли, выдается исключение. Это позволяет выдать быстрый сбой операции. Регистрация исключения может быть выполнена выше по стеку вызовов

Использование отражения для получения PermittedRoleAttribute, указанного для данного типа команды

Если для этого типа команды атрибут не определен, можно предположить, что выполнение команды разрешено каждому пользователю, но это грозит нарушением мер безопасности. Вместо этого выдается исключение, заставляющее применять атрибут к каждой команде

Разновидности авторизации

Указывать авторизацию для команд и других типов сообщений можно множеством способов. Вот некоторые из них.

- Если команды неизменно доступны одной и той же роли, то вместо применения атрибута нужно рассмотреть возможность помещения команд в пространство имен,

названное созвучно этой роли. Например, административные команды можно поместить в пространство имен `MyApp.Domain.Commands.Administrator` и позволить декоратору проанализировать его. Это сделает структуру проекта интуитивно понятной, поскольку команды будут сгруппированы по разрешенным ролям.

- Вместо работы с ролями обычная модель работает с разрешениями. Они позволяют добиться более детализированной конфигурации доступа. Команда может помечаться конкретным разрешением. При этом жестко кодируется список разрешений, а не ролей, используемых в приложении, а администратору разрешается управлять связью между пользователями, ролями и разрешениями.
- После того как будет обеспечена безопасность на основе ролей, приложению может потребоваться безопасность на основе строк. В контексте приложения электронной торговли это может означать, что конкретными группами товаров могут управлять только пользователи, которые проживают в определенных регионах. Иными словами, даже когда к одной и той же роли могут относиться сразу несколько пользователей, безопасность на основе строк может сделать конкретный товар доступным одним пользователям и недоступным другим пользователям, исполняющим ту же роль[1].

Мы могли бы привести массу примеров декораторов, заключающих в себя бизнес-транзакции, но ограничены объемом книги. Кроме того, мы полагаем, что у вас уже начала складываться картина того, как применяются декораторы в виде надстройки над `ICommandService<TCommand>`. Соберем все вместе в корне композиции.

Составление графа объектов с использованием обобщенных декораторов

В предыдущих разделах были объявлены три декоратора, реализующих обеспечение безопасности, управление транзакциями и ведение контрольного журнала. Их нужно применить к настоящей реализации в корне композиции. На рис. 10.4 показано, что декораторы, включающие в себя сервис команд, подобны матрешке.

Рис. 10.4. Оснащение реального сервиса команд аспектами ведения контрольного журнала, управления транзакциями и обеспечения безопасности

Если применить все три определенных ранее декоратора к корню композиции, получится код, показанный в листинге 10.20.

[1] Чтобы получить представление о работе с безопасностью на основе строк, обратитесь к следующему онлайн-обсуждению: https://github.com/dotnetjunkie/solidservices/issues/4.

Листинг 10.20. Декорирование AdjustInventoryService

```
ICommandService<AdjustInventory> service =
    new SecureCommandServiceDecorator<AdjustInventory>(
        this.userContext,
        new TransactionCommandServiceDecorator<AdjustInventory>(
            new AuditingCommandServiceDecorator<AdjustInventory>(
                this.userContext,
                this.timeProvider,
                context,
                new AdjustInventoryService(repository))));

return new InventoryController(service);
```

Поскольку приложение ожидает получения множества реализаций `ICommandService<TCommand>`, большинству реализаций потребуются те же самые декораторы. Поэтому код листинга 10.20 даст множество повторений кода внутри корня композиции. Это одна из проблем, которые легко решаются выделением для повторяющегося создания декоратора собственного, отдельного метода (листинг 10.21).

Листинг 10.21. Извлечение композиции декораторов в многократно используемый метод

```
private ICommandService<TCommand> Decorate<TCommand>(
    ICommandService<TCommand> decoratee, CommerceContext context)
{
    return
        new SecureCommandServiceDecorator<TCommand>(
            this.userContext,
            new TransactionCommandServiceDecorator<TCommand>(
                AuditingCommandServiceDecorator<TCommand>(
                    this.userContext,
                    this.timeProvider,
                    context,
                    decoratee))));
}
```

Заключение параметра decoratee в список декораторов

Перенос декораторов в метод `Decorate` позволяет корню композиции полностью соответствовать принципу DRY. Создание `AdjustInventoryService` сокращается до одной простой строки кода:

```
var service = Decorate(new AdjustInventoryService(repository), context);

return new InventoryController(service);
```

В главе 12 будет показано, как выполняется автоматическая регистрация реализаций `ICommandService<TCommand>` и применяются декораторы, работающие с DI-контейнером. Поскольку мы практически добрались до завершения этого раздела, посвященного использованию принципов SOLID в качестве движущей силы AOP, подытожим, чего нам удалось достичь и как это связано с общей картиной разработки приложения.

10.3.5. Заключение

В этой главе была выполнена реструктуризация крупного интерфейса `IProductService` доменного уровня, который состоял из нескольких командных методов, в простую абстракцию `ICommandService<TCommand>`, где каждая команда получает собственное сообщение и соответствующую реализацию для его обработки. Эта реструктуризация не изменила исходную логику приложения, но благодаря ей концепция команд приобрела четкие очертания.

Важным выводом является то, что теперь эти доменные команды отображаются в системе как явный артефакт, а их обработчики помечены единым интерфейсом. Эта методология аналогична реализуемой в ходе работы с такими средами приложений, как ASP.NET Core MVC. Контроллеры MVC обычно определяются путем наследования от абстракции контроллера — это позволяет среде MVC находить их, используя отражение, и предоставляет для взаимодействия с ними общий API. Такая практика в конструкции приложения полезна в более широком смысле, в чем вы убедились на примере этих команд, обработчикам которых был дан общий API (один-единственный метод `Execute`). Это позволило эффективно применять аспекты без повторения кода.

Помимо команд, в системе имеются и другие артефакты, которые вы, возможно, захотите сконструировать аналогичным образом, чтобы иметь возможность применить сквозную функциональность. Весьма распространенным артефактом, заслуживающим более четкого представления, является запрос. В начале подраздела 10.3.3, после того как `IProductService` был разделен на интерфейс чтения и записи, мы сконцентрировались на `IProductCommandServices` и проигнорировали `IProductQueryServices`. Запросы заслуживают собственной абстракции. Но из-за нехватки места этот вопрос в данной книге не рассматривается[1].

Однако мы считаем, что сходство между группами связанных компонентов можно найти во многих приложениях точно так же, как это произошло в данной главе. Это может помочь более эффективно задействовать сквозную функциональность, а также предоставить вам четкий, проверенный компилятором стиль программирования.

Цель этой главы не в том, чтобы убедить вас, что применение абстракции `ICommandService<TCommand>` — именно то, как следует разрабатывать ваши приложения. При ее изучении важно было прийти к выводу, что проектирование приложений в соответствии с SOLID-принципами — это способ обеспечить их хорошую сопровождаемость. Здесь продемонстрировано, что достичь этого можно, главным образом не используя специализированный инструментарий AOP. Это важно, так как у такого инструментария имеются ограничения и проблемы, которые подробнее будут рассмотрены в следующей главе. Но, как выяснилось, определенный набор структур проектирования применим ко многим бизнес-приложениям (line-of-business, LOB), и одной из них является `ICommandService`-подобная абстракция.

---

[1] Подобная абстракция рассматривается в статье: *Deursen S. van*. Meanwhile... on the query side of my architecture, 2011; https://cuttingedge.it/blogs/steven/pivot/entry.php?id=92.

Однако это не означает, что соблюдение SOLID-принципов никогда не вызывает затруднений. Наоборот, без трудностей не обойдется. Как утверждалось ранее, стопроцентное их соблюдение просто невозможно, к тому же на это нужно время. Ваша задача, как разработчика программных средств, — найти оптимальный вариант, а увеличить вероятность этого может использование DI и SOLID в нужные моменты.

Если следовать таким общепризнанным объектно-ориентированным принципам, как SOLID, то существенно возрастет эффект от применения DI. В частности, характеризующаяся слабой связанностью природа DI позволяет использовать паттерн «Декоратор», чтобы придерживаться как принципа OCP, так и принципа SRP. Ценность этого свойства проявляется в довольно широком спектре ситуаций, поскольку оно позволяет сохранять ясность и упорядоченность кода, особенно когда речь заходит о сквозной функциональности.

Но не будем ходить вокруг да около. Создание легко сопровождаемого программного продукта дается нелегко, даже если предпринимается попытка соблюдения SOLID-принципов. Кроме того, зачастую работа ведется над такими проектами, которые не рассчитаны на испытание временем. Большие архитектурные изменения могут быть неосуществимы или опасны. Тогда единственным жизнеспособным вариантом может стать использование инструментария AOP, даже если он обеспечивает временное решение. Прежде чем начать применять такой инструментарий, важно понять, как он работает и в чем его слабые стороны, особенно в сравнении с философией проектирования, изложенной в данной главе. Именно этому и будет посвящена следующая глава.

Резюме

- Принцип единственной ответственности (SRP) гласит, что у каждого класса должен быть только один повод для внесения изменений. Это можно проследить с позиции связности. Связность определяется как функциональная родственность элементов класса или модуля. Чем ниже уровень родственности, тем ниже связность, а чем ниже связность, тем выше вероятность того, что в классе нарушен принцип SRP.
- Принцип открытости/закрытости (OCP) гласит, что конструкция приложения должна избавлять вас от необходимости внесения широкомасштабных изменений по всей кодовой базе. Тесная связь между принципами OCP и DRY заключается в том, что оба они призывают стремиться к достижению одной и той же цели.
- Принцип «Не повторяйся» (DRY) гласит, что у каждого фрагмента знаний должно быть всего одно непротиворечивое общепринятое представление в системе.
- Принцип подстановки Лисков (LSP) гласит, что каждая реализация должна вести себя так, как определено ее абстракцией. Это позволит заменить изначально запланированную реализацию другой реализацией той же самой абстракции, не беспокоясь о повреждении кода потребителя. Этот принцип положен в основу технологии DI. Когда потребители его не соблюдают, от внедрения зависимостей

мало толку, поскольку заменить зависимость по своему желанию уже не получится и будут потеряны многие, если не все преимущества DI.

- Принцип изоляции интерфейса (ISP) предписывает использование узкоспециализированных абстракций и замену ими абстракций с широким функциональным диапазоном. Когда потребитель зависит от абстракции, где часть компонентов остается невостребованной, этот принцип нарушается. Данный принцип играет решающую роль в эффективном применении аспектно-ориентированного программирования.
- Принцип инверсии зависимостей (DIP) утверждает, что программировать нужно с прицелом на создание абстракций и что форма потребляемой абстракции должна контролироваться на уровне ее потребления. Потребитель должен иметь возможность определять абстракцию наиболее выгодным для себя образом.
- Перечисленные пять принципов образуют акроним SOLID. Но ни один из них не является абсолютным понятием. Эти принципы служат руководством, помогающим создавать понятный код.
- Аспектно-ориентированное программирование (AOP) — это парадигма, сконцентрированная на применении сквозной функциональности эффективным и легко сопровождаемым способом.
- Самой востребованной техникой в AOP является SOLID. В SOLID-приложениях предотвращаются повторения как в обычном коде, так и в реализациях сквозной функциональности. Применение SOLID-технологий может избавить и от необходимости задействовать специализированный инструментарий AOP.
- Даже при соблюдении SOLID-принципов может наступить момент, когда изменения станут широкомасштабными. Абсолютная закрытость от изменений невозможна и нежелательна. Чтобы соответствовать принципу OCP, требуется приложить немало сил для поиска и разработки подходящих абстракций, но слишком большое количество последних может обернуться нежелательным усложнением приложения.
- Принцип разделения команд и запросов (CQS) — важный объектно-ориентированный принцип, утверждающий, что каждый метод должен либо возвращать результат, но при этом не изменять видимое ему состояние системы, либо изменять состояние, но не выдавать никакого значения.
- Размещение командных методов и методов-запросов в разных абстракциях упрощает реализацию сквозной функциональности, поскольку большинство аспектов нужно применять либо к командам, либо к запросам, но не к тому и другому сразу.
- Граничный объект представляет собой группу параметров, сочетаемых друг с другом естественным образом. Извлечение граничных объектов позволяет определять повторно используемую абстракцию, которая может быть реализована большой группой компонентов. Это позволяет одинаково обрабатывать данные компоненты, а применение сквозной функциональности сделать еще эффективнее.

- В отличие от абстракции компонента эти извлеченные граничные объекты становятся определением отдельной операции или варианта применения в системе.
- Хотя разбиение крупных классов на множество классов помельче с помощью граничных объектов существенно увеличивает количество классов в системе, оно может и существенно облегчить ее сопровождение. Количество классов в системе не может служить достоверным показателем уровня простоты сопровождаемости.
- Сквозную функциональность нужно применять в приложении на должном уровне детализации. Для всех CRUD-приложений, кроме самых простых, хранилища не могут служить должным уровнем детализации в большинстве вариантов применения сквозной функциональности. При соблюдении SOLID-принципов повторно используемые однокомпонентные абстракции обычно становятся теми уровнями, где нужно задействовать аспекты сквозной функциональности.

11 Аспектно-ориентированное программирование с помощью инструментальных средств

В этой главе

- Использование динамического перехвата для применения перехватчиков с привлечением сгенерированных декораторов.
- Преимущества и недостатки динамического перехвата.
- Автоматическое добавление аспектов в ходе компиляции для реализации сквозной функциональности.
- Причина, по которой автоматическое добавление аспектов в ходе компиляции является противоположностью DI.

В этой главе продолжим говорить об аспектно-ориентированном программировании (AOP). В главе 10 AOP рассматривалось в абсолютно чистой форме — исключительно как практика проектирования на основе соблюдения принципов SOLID. Здесь подходы к AOP исследуются с позиции использования инструментария. Будут рассмотрены два широко распространенных метода применения AOP: динамический перехват и автоматическое добавление аспектов в ходе компиляции.

В том случае, когда подход к разработке, рассмотренный в главе 10, будет слишком радикальным, наиболее подходящим выбором станет динамический перехват, поэтому именно о нем будем говорить в первую очередь. Динамический перехват может быть неплохим временным решением, пока не сложится подходящая ситуация, чтобы приступить к внесению улучшений, рассмотренных в предыдущей главе.

Автоматическое добавление аспектов является противоположностью применения технологии DI и рассматривается в качестве антипаттерна. Тем не менее мы считаем, что эту технологию стоит обсудить, поскольку это широко известная форма AOP и мы хотим наглядно продемонстрировать, что она является нежизнеспособной альтернативой технологии DI.

СОВЕТ

Этот инструментарий рассматривается в ограниченном объеме применительно к теме внедрения зависимостей. Дополнительные сведения об AOP на основе применения инструментов при желании можно найти в книге: *Groves M. D.* AOP in .NET. — Manning, 2013.

11.1. Динамический перехват

В листингах кода, приведенных в разделе 10.1, где была показана реализация методов `Delete` и `Insert` класса `CircuitBreakerProductRepositoryDecorator`, допущено повторение кода. Этот код еще раз дан в листинге 11.1.

Листинг 11.1. Нарушение принципа DRY (приводится повторно)

```
public void Delete(Product product)
{
    this.breaker.Guard();

    try
    {
        this.decoratee.Delete(product);
        this.breaker.Succeed
    }
    catch (Exception ex)
    {
        this.breaker.Trip(ex);
        throw;
    }
}

public void Insert(Product product)
{
    this.breaker.Guard();

    try
    {
        this.decoratee.Insert(product);
        this.breaker.Succeed();
    }
    catch (Exception ex)
    {
        this.breaker.Trip(ex);
        throw;
    }
}
```

Код этих двух методов сильно похож на шаблон. Методы различаются только вызовами Delete и Insert

При реализации декоратора в качестве аспекта труднее всего дается разработка шаблона. После этого процесс превращается в чисто механическую работу.

1. Создание нового класса декоратора.
2. Разработка на основе желаемого интерфейса.
3. Реализация каждого компонента интерфейса с применением шаблона.

Это настолько повторяемый процесс, что для его автоматизации можно воспользоваться инструментальным средством. Среди множества эффективных функций, имеющихся в среде .NET Framework, есть и возможность динамического порождения типов. Это позволяет создавать код, генерирующий полнофункциональный класс во время выполнения программы. У такого класса нет файла базового исходного кода, он компилируется непосредственно из какой-то абстрактной модели. Это позволяет автоматизировать генерацию декораторов, создаваемых во время выполнения программы. На рис. 11.1 показано, что это позволяет выполнить динамический перехват.

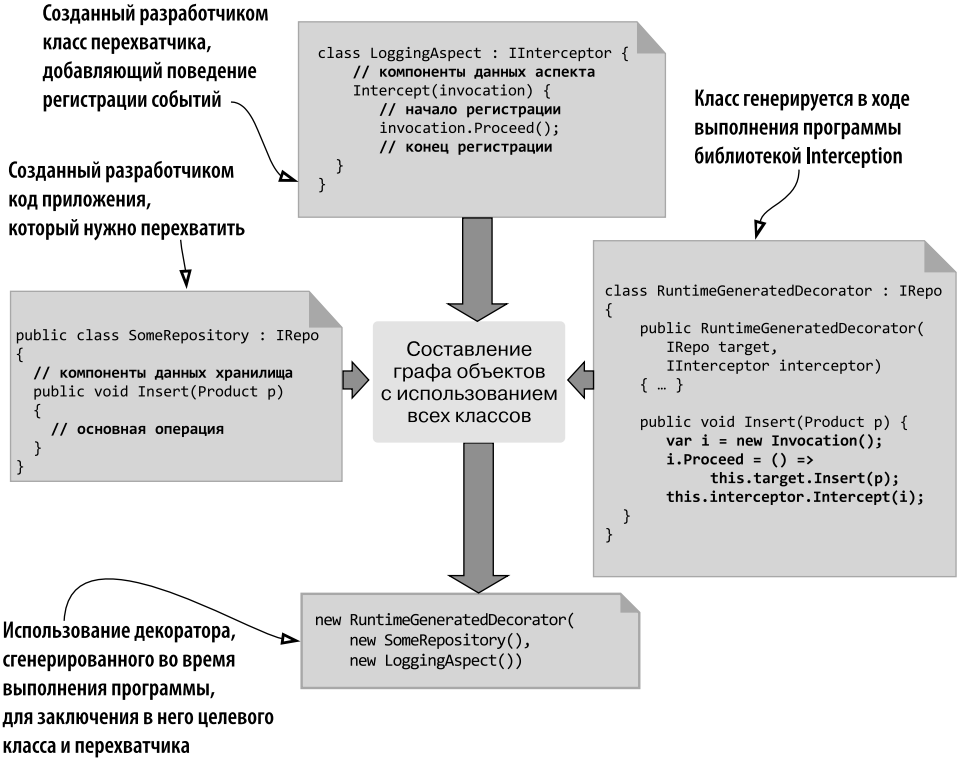


Рис. 11.1. Библиотека динамического перехвата генерирует класс декоратора во время выполнения программы. Для отдельно взятой абстракции (в данном случае IRepo) это происходит всего один раз. После завершения процесса генерации можно запросить у библиотеки перехвата создание для вас нового экземпляра этого декоратора, предоставив ей и цель, и перехватчик

После того как будет создан граф объектов для динамически сгенерированного декоратора и его зависимостей, декоратор можно применить в качестве замены реальному классу. Поскольку им реализуется реальная абстракция класса, он может быть внедрен в клиентов, использующих эту абстракцию. На рис. 11.2 дается картина потока вызовов методов, когда клиент выполняет вызов в свою перехваченную абстракцию.

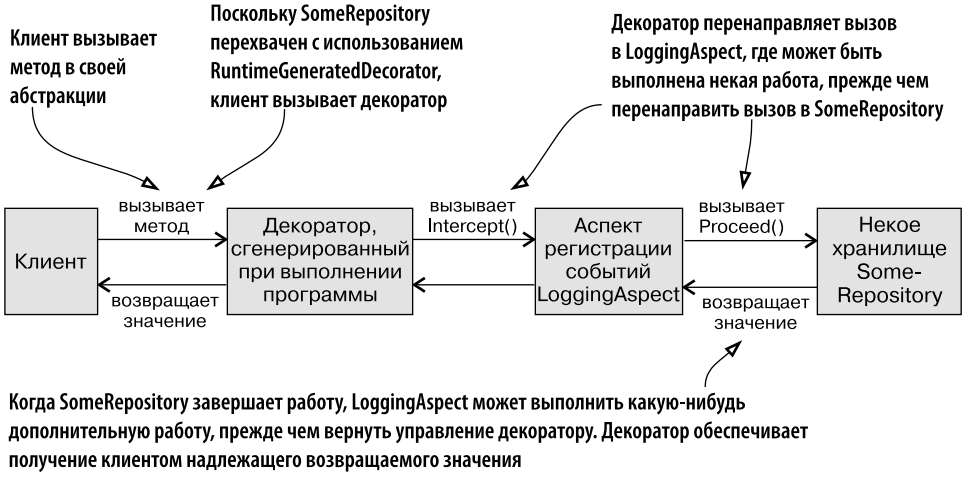


Рис. 11.2. Поток вызовов методов, возникающий, когда клиент выполняет вызов в свою перехваченную абстракцию

Чтобы воспользоваться динамическим перехватом, нужно все же создать код, реализующий аспект. Это, как показано в листинге 11.1, может быть связующий код, необходимый для аспекта предохранителя. После его создания нужно сообщить библиотеке динамического перехвата об абстракциях, к которым должен применяться аспект. Но хватит теории, рассмотрим пример.

11.1.1. Пример: перехват с использованием Castle Dynamic Proxy

В качестве кандидата для динамического перехвата вполне подойдет повторяющийся код из листинга предохранителя. Хотя код, генерирующий декораторы во время выполнения, можно написать самостоятельно, это непростая операция, к тому же для этого уже есть неплохой доступный инструментарий. Мы не станем проводить вас через утомительный процесс генерации кода вручную, а сразу перейдем к работе с инструментом. В качестве примера посмотрим, как можно сократить повторяемость кода, применив возможности перехвата, предоставляемые библиотекой Castle Dynamic Proxy.

ПРИМЕЧАНИЕ

Библиотека Castle Dynamic фактически является стандартным инструментом динамического перехвата в среде .NET. Она имеет открытый код и находится в свободном доступе. Фактически ею для выполнения динамического перехвата скрыто пользуются многие DI-контейнеры. Доступен и другой инструментарий динамического перехвата, но библиотека Castle — уже состоявшееся средство, прошедшее испытание временем, поэтому в дальнейшем будем применять именно ее.

Реализация перехватчика для предохранителя

Реализация перехватчика для Castle требует от вас реализации его интерфейса `Castle.DynamicProxy.IInterceptor`, состоящего из одного метода. В листинге 11.2 показано, как можно реализовать предохранитель из листинга 11.1. Но в отличие от него далее класс показан целиком.

Листинг 11.2. Реализация предохранителя с динамическим заместителем

```
public class CircuitBreakerInterceptor
    : Castle.DynamicProxy.IInterceptor
{
    private readonly ICircuitBreaker breaker;

    public CircuitBreakerInterceptor(
        ICircuitBreaker breaker)
    {
        this.breaker = breaker;
    }

    public void Intercept(IInvocation invocation)
    {
        this.breaker.Guard();

        try
        {
            invocation.Proceed();

            this.breaker.Succeed();
        }
        catch (Exception ex)
        {
            this.breaker.Trip(ex);
            throw;
        }
    }
}
```

Чтобы реализовать перехватчик, нужно создать интерфейс IInterceptor, определяемый Castle

Перехватчик может быть скомпонован с корнем композиции. Этот позволит воспользоваться внедрением через конструктор

Здесь реализуется только один метод, для чего применяется тот же самый код, который использовался при многократном повторении, когда вы реализовали CircuitBreakerProductRepositoryDecorator

Это указание Castle разрешить вызову перейти к задекорированному экземпляру

Основное отличие от кода листинга 11.1 заключается в том, что вместо делегирования вызова метода конкретному методу следует воспользоваться более универсальным подходом, поскольку потенциально этот код применим к любому

методу. Интерфейс `IInvocation` передается методу `Intercept` в качестве параметра, представляющего вызов метода. Он может, к примеру, представлять вызов метода вставки товара `Insert(Product)`. Одним из ключевых компонентов этого интерфейса является метод `Proceed`, поскольку он дает возможность вызову перейти к следующей реализации в стеке.

Интерфейс `IInvocation` позволяет сформировать возвращаемое значение до того, как вызову будет позволен переход. Он также делает доступной более подробную информацию о вызове метода. Из параметра вызова можно получить сведения об имени и значениях параметров метода, а также другую информацию о текущем вызове метода. Реализация перехватчика считается сложной частью задачи. Следующий шаг будет легким.

Применение перехватчика внутри корня композиции с использованием чистой технологии DI

В листинге 11.3 показано, как можно ввести `CircuitBreakerInterceptor` в состав корня композиции.

Листинг 11.3. Введение перехватчика в корень композиции[1]

```
var generator =
    new Castle.DynamicProxy.ProxyGenerator();

var timeout = TimeSpan.FromMinutes(1);

var breaker = new CircuitBreaker(timeout);

var interceptor =
    new CircuitBreakerInterceptor(breaker);

var wcfRepository = new WcfProductRepository();

IProductRepository repository = generator
    .CreateInterfaceProxyWithTarget<IProductRepository>(
        wcfRepository,
        interceptor);
```

Имеющийся в Castle ProxyGenerator может сгенерировать типы Proxy во время выполнения программы

Перехватчик принимает экземпляр ICircuitBreaker в своем конструкторе, используя внедрение через конструктор

Создание реального экземпляра IProductRepository

Требование того, чтобы Castle создавала декоратор (Proxy) на основе интерфейса IProductRepository и заключала в него как экземпляр исходного хранилища, так и только что созданный перехватчик

СОВЕТ

Чтобы не снижалась производительность, Castle.DynamicProxy.ProxyGenerator должен, как правило, создаваться всего один раз и кэшироваться на весь период выполнения приложения.

[1] По терминологии, используемой в Castle, Proxy означает «декоратор».

Несмотря на то что Castle управляет созданием декоратора `IProductRepository` и внедрением его зависимостей, из этого примера следует, что ваше приложение по-прежнему может запускаться с помощью чистой технологии DI. В следующем разделе будет проанализирован динамический перехват и рассмотрены его преимущества и недостатки.

11.1.2. Анализ динамического перехвата

Когда в предыдущей главе при рассмотрении динамического перехвата инструментальный подход к AOP сравнивался с проектным подходом, между ними обнаружилось определенное сходство.

- Каждый из них позволяет решать задачи обеспечения сквозной функциональности при программировании с прицелом на создание абстракций.
- Как и в случае применения обычных старых декораторов, перехватчики могут использовать внедрение через конструктор, что делает их вполне совместимыми с технологией DI и отвязывает от декорируемого кода. Эти характеристики позволяют без каких-либо осложнений тестировать как бизнес-код, так и ваши аспекты.
- Аспекты могут быть централизованы в корне композиции, предотвращая тем самым повторяемость кода, а в случае, когда ваше решение в Visual Studio содержит несколько приложений, позволяя аспектам применяться только в одном из двух корней композиции.

Но, несмотря на сходство, имеются и некоторые различия, делающие динамический перехват далеким от идеала. Недостатки, которые будут рассмотрены в дальнейшем, сведены в табл. 11.1.

Таблица 11.1. Недостатки динамического перехвата

Недостаток	Резюме
Потеря поддержки в ходе компиляции	Код перехвата становится сложнее кода декоратора, что затрудняет его чтение и сопровождение
Сильная связанность аспектов с инструментарием	Эта связанность усложняет тестирование и заставляет перехватчик становиться частью корня композиции, чтобы другим сборкам не требовалась зависимость от библиотеки динамического перехвата
Отсутствие универсальности применения	Аспекты могут применяться в границах только тех методов, которые имеют виртуальный или абстрактный характер, например методов, являющихся частью определения интерфейса
Базовые проблемы проектирования никуда не исчезают	В результате все равно получается система, которая в некоторых аспектах немного проще в сопровождении, но в целом существенно уступает в легкости сопровождения той конструкции, в которой строже придерживаются SOLID-принципов

Потеря поддержки в ходе компиляции

В сравнении с простыми декораторами динамический перехват включает в себя существенное количество вызовов отражений при каждом применении перехватчика. Например, при использовании Castle интерфейс `IInvocation` содержит свойство `Arguments`, которое возвращает массив экземпляров объектов, содержащий список аргументов метода. Чтение и изменение этих значений во время работы с целочисленными и булевыми типами не обходится без приведения типов и упаковки. Существенно сказываться на производительности постоянная необходимость отражения не будет. А вот типовые операции ввода-вывода, например чтение из базы данных и запись в нее, окажутся на порядок затратнее.

Но такое использование отражения усложняет создаваемые перехватчики. При обработке списка аргументов метода и возвращаемых типов придется создавать подходящее приведение типов, проверять их и, возможно, более четко уведомлять об ошибках приведения типов. Поэтому перехватчик, как правило, сложнее декоратора, что затрудняет чтение и сопровождение его кода.

Сильная связанность аспектов и инструментария

В сравнении с простыми декораторами перехватчики, созданные с помощью динамического перехвата, обладают сильной связанностью с используемой библиотекой перехвата. Наглядным примером этого может послужить перехватчик предохранителя из листинга 11.2. В нем реализуется `Castle.DynamicProxy.IInterceptor` и применяется абстракция `Castle.DynamicProxy.IInvocation`.

Как будет показано в разделе 11.2, эта сильная связанность выражена слабее, чем при автоматическом добавлении аспектов в ходе компиляции, однако она приводит к тому, что все аспекты привязываются к библиотеке Castle Dynamic Proxy. Такая связанность создает дополнительную зависимость от внешней библиотеки, которую нужно изучить, что вызывает дополнительные затраты и риски для проекта. Более подробно этот вопрос будет рассматриваться в подразделе 12.3.1.

Отсутствие универсальности применения

Поскольку динамический перехват работает за счет заключения уже существующих абстракций в динамически сгенерированные декораторы, поведение класса может быть расширено только в границах метода, принадлежащего абстракции. Приватные методы не могут быть перехвачены, потому что не являются частью интерфейса.

Это ограничение остается в силе и для практики AOP путем обычного проектирования. Но это, как правило, менее существенная проблема, поскольку абстракции проектируются таким образом, что необходимость в применении аспектов возникает в границах этих абстракций[1]. При использовании динамического перехвата приходится довольствоваться существующим положением вещей, поскольку отказ от него приведет к реализации AOP путем обычного проектирования.

---

[1] Считается, что необходимость в перехвате приватных методов является проблемным кодом.

Базовые проблемы проектирования никуда не исчезают

В главе 10 широко обсуждались конструктивные проблемы в крупном интерфейсе `IProductService` и возможности устранить их, следуя принципам SOLID. В результате выяснилось, что эти проблемы влияют на систему серьезнее, чем любые вопросы, связанные со сквозной функциональностью.

Но динамическим перехватом можно воспользоваться, когда существующая конструкция приложения вполне устраивает и хотелось бы получить возможность применить сквозную функциональность, не проводя широкую реструктуризацию. Недостаток здесь в том, что решается только часть проблемы. Будет по-прежнему получаться система, которая ненамного удобнее в сопровождении, чем существующая конструкция, и значительно менее удобная в этом плане, чем конструкция на основе соблюдения SOLID-принципов.

Динамический перехват требует программирования с прицелом на использование интерфейсов и DI-паттернов, рассмотренных в главе 4. Еще одной формой AOP, не требующей программирования с прицелом на применение интерфейсов, является автоматическое добавление аспектов в ходе компиляции. Название этой технологии звучит весьма привлекательно, но, как выяснится в дальнейшем, для технологии DI она является антипаттерном.

11.2. Автоматическое добавление аспектов в ходе компиляции

Когда разработчики создают код на языке C#, компилятор языка превращает его в код на промежуточном языке Microsoft Intermediate Language (IL). Код IL считывается компилятором Just-In-Time (JIT) среды Common Language Runtime (CLR) и тут же преобразуется в машинные инструкции, исполняемые центральным процессором[1]. Скорее всего, этот процесс в общих чертах вам уже знаком. Автоматическое добавление аспектов в ходе компиляции — широко распространенная AOP-технология, изменяющая ход компиляции.

Для чтения скомпилированной сборки, выданной компилятором C#, внесения в нее изменений и обратной записи новой сборки на диск, заменяя, по сути, исходную сборку, используется специализированный инструментарий (рис. 11.3).

Рис. 11.3. Автоматическое добавление аспектов в ходе компиляции

[1] Это упрощенное представление — в некоторых средах IL-сборка проходит интерпретацию, а не JIT-компиляцию.

Изменение исходной скомпилированной сборки в процессе посткомпиляции выполняется для того, чтобы автоматически добавить аспекты в первоначальный исходный код (рис. 11.4).

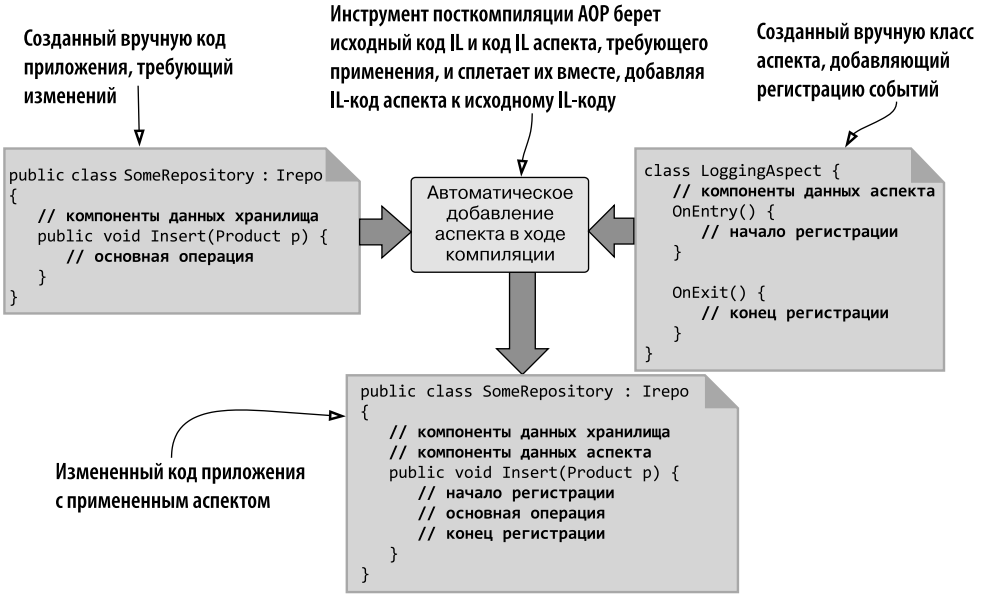


Рис. 11.4. Визуализация автоматического добавления аспектов в ходе компиляции

Но каким бы заманчивым это ни было на первый взгляд применительно к нестабильным зависимостям, автоматическое добавление аспектов в ходе компиляции сопряжено с проблемами, затрудняющими сопровождение кода. Мы считаем, что из-за недостатков, рассмотренных в данном разделе, автоматическое добавление аспектов в ходе компиляции является противоположностью технологии DI и поэтому должно считаться антипаттерном DI.

ВНИМАНИЕ

Автоматическое добавление аспектов в ходе компиляции — нежелательный метод применения AOP к нестабильным зависимостям. Следует отдавать предпочтение SOLID-принципам или, если это невозможно, вернуться к динамическому перехвату.

Как говорилось во введении, мы посчитали важным рассмотреть автоматическое добавление аспектов в ходе компиляции даже притом, что оно является антипаттерном DI. Известность этой формы AOP настолько широка, что мы должны предостеречь от ее использования. Прежде чем перейти к рассмотрению причин ее проблематичности, разберем пример.

11.2.1. Пример: применение аспекта транзакции с автоматическим добавлением аспектов в ходе компиляции

У атрибутов имеется та же особенность, что и у декораторов: хотя они могут добавлять компонент или косвенно выражать изменение его поведения, его сигнатура и исходный код остаются неизменными. В подразделе 9.2.3 с помощью декоратора задействован аспект обеспечения безопасности. А автоматическое добавление аспекта в ходе компиляции позволяет объявлять аспекты, помещая атрибуты в классы, их компоненты и даже сборки.

Использование этой концепции для применения сквозной функциональности не лишено привлекательности. Ведь хорошо было бы иметь возможность задекорировать метод или класс атрибутом `[Transaction]` или даже пользовательским атрибутом `[CircuitBreaker]` и, таким образом, применить аспект с помощью всего одной строки декларативного кода. В листинге 11.4 показано, как пользовательский атрибут аспекта `TransactionAttribute` применяется непосредственно к методам класса `SqlProductRepository`.

Листинг 11.4. Использование атрибута аспекта [Transaction] к SqlProductRepository

```
public class SqlProductRepository : IProductRepository
{
    [Transaction]
    public void Insert(Product product) ...

    [Transaction]
    public void Update(Product product) ...

    [Transaction]
    public void Delete(Guid id) ...

    public IEnumerable<Product> GetAll() ...
    ...
}
```

Пользовательский атрибут аспекта применяется ко всем методам Insert, Update и Delete, но их код остается нетронутым

Логика транзакции требуется не всем методам, поэтому данный метод на помечен атрибутом

ПРИМЕЧАНИЕ

Термин «атрибут аспекта» использован для обозначения пользовательского атрибута, объявленного в классе или его компонентах, если такой аттрибут реализует или обозначает аспект.

При наличии довольно широкого выбора инструментов автоматического добавления аспектов в ходе компиляции мы в этом разделе воспользуемся коммерческим продуктом PostSharp (https://www.postsharp.net/). В листинге 11.5 показано определение `TransactionAttribute`, выполненное с помощью PostSharp.

Листинг 11.5. Реализация аспекта TransactionAttribute с помощью PostSharp

```
[AttributeUsage(AttributeTargets.Method |
    AttributeTargets.Class |
    AttributeTargets.Assembly,
    AllowMultiple = false)]
[PostSharp.Serialization.PSerializable]
[PostSharp.Extensibility.MulticastAttributeUsage(
    MulticastTargets.Method,
    TargetMemberAttributes =
        MulticastAttributes.Instance |
        MulticastAttributes.Static)]
public class TransactionAttribute
    : PostSharp.Aspects.OnMethodBoundaryAspect
{

    public override void OnEntry(
        MethodExecutionArgs args)
    {
        args.MethodExecutionTag =
        new TransactionScope();
    }

    public override void OnSuccess(
        MethodExecutionArgs args)
    {
        var scope = (TransactionScope)
            args.MethodExecutionTag;
        scope.Complete();
    }

    public override void OnExit(
        MethodExecutionArgs args)
    {
        var scope = (TransactionScope)
            args.MethodExecutionTag;
        scope.Dispose();
    }
}
```

Атрибуты, необходимые для аспектов PostSharp

За счет наследования от принадлежащего PostSharp атрибута OnMethodBoundaryAspect аспект можно применить в границах задекорированного метода, как было бы сделано и с помощью декоратора

Реализация аспекта путем переопределения методов OnEntry, OnSuccess и OnExit

Поскольку в транзакцию нужно заключить некий произвольный фрагмент кода, требуется переопределение трех методов класса `OnMethodBoundaryAspect`, а именно `OnEntry`, `OnSuccess` и `OnExit`. В ходе выполнения `OnEntry` создается новый объект `TransactionScope`, а в ходе выполнения `OnExit` он ликвидируется. Вызов `OnExit` при этом гарантируется. PostSharp поместит его вызов в блок `finally`. Вызов метода `Complete` нужен будет только при успешном завершении операции, заключенной в транзакцию. Поэтому данный вызов реализуется в методе `OnSuccess`. Свойство `MethodExecutionTag` используется для передачи созданного объекта `TransactionScope` из метода в метод.

ПРИМЕЧАНИЕ

TransactionScope рассматривается в подразделе 10.3.3.

Если рассматривать код листинга 11.4 обособленно, то атрибуты могут показаться весьма привлекательными, но если сравнить код листинга 11.5 с тем же самым аспектом в декораторе (см. листинг 10.15), станет понятно, что в нем много шаблонного содержимого. Требуется переопределить несколько методов, применить всевозможные атрибуты и передать состояние от метода к методу.

ПРИМЕЧАНИЕ

Сравнивая TransactionCommandServiceDecorator<TCommand> из листинга 10.15 с реализацией аспекта PostSharp из листинга 11.5, мы понимаем, насколько чище написан декоратор. Он может быть написан естественным для разработчика образом. Особенно если речь идет о написании кода, содержащего блоки catch, finally или using, кода декоратора, соблюдающего SOLID-принципы, и даже кода с применением динамического перехвата, все получается проще и легче в сопровождении, чем при автоматическом добавлении аспектов в ходе компиляции, предлагаемом такими инструментами, как PostSharp.

Вероятно, с этими затратами можно было бы смириться, если бы при этом упростилось сопровождение кода, но автоматическое добавление аспектов в ходе компиляции сопряжено и с другими ограничениями, не позволяющими задействовать данную технологию для применения нестабильных зависимостей в роли сквозной функциональности.

11.2.2. Анализ автоматического добавления аспектов в ходе компиляции

Если рассматривать автоматическое добавление аспектов во взаимоотношениях с технологией DI, то для него характерны два недостатка. Данный раздел будет посвящен рассмотрению соответствующих этим недостаткам ограничений. Есть и другие недостатки, свойственные автоматическому добавлению аспектов в ходе компиляции, но те два, что приведены в табл. 11.2, выражают суть проблемы, делающей данную технологию нежелательной для внедрения зависимостей.

Таблица 11.2. Недостатки автоматического добавления аспектов в ходе компиляции с позиции внедрения зависимостей

Недостаток	Резюме
Плохая совместимость с технологией DI	Отсутствует подходящий способ внедрения нестабильных зависимостей в аспекты, автоматически добавляемые в ходе компиляции. Альтернативные варианты становятся причиной временной связанности, появления захваченных зависимостей и взаимозависимых тестов
Связанность во время компиляции	Аспекты автоматически добавляются в ходе компиляции, не позволяя вызывать код без применения аспекта. Это усложняет тестирование и уменьшает гибкость

Плохая совместимость автоматического добавления аспектов в ходе компиляции с технологией DI

Когда дело доходит до применения сквозной функциональности, постоянно сталкиваешься с работой с нестабильными зависимостями. В соответствии с изложенным в главе 1 такие зависимости находятся в центре внимания DI. В ходе работы с ними в первую очередь следует выбирать внедрение через конструктор, так как при этом происходит статическое определение списка нужных зависимостей.

К сожалению, при автоматическом добавлении аспектов в ходе компиляции использовать внедрение через конструктор невозможно. Рассмотрим листинг 11.6, где предпринимается попытка внедрения через конструктор в отношении аспекта предохранителя.

Листинг 11.6. Внедрение зависимости в аспект с помощью внедрения через конструктор

```
[PostSharp.Serialization.PSerializable]
public class CircuitBreakerAttribute
    : OnMethodBoundaryAspect
{
    private readonly ICircuitBreaker breaker;

    public CircuitBreakerAttribute(
        ICircuitBreaker breaker)
    {
        this.breaker = breaker;
    }

    public override void OnEntry(
        MethodExecutionArgs args)
    {
        this.breaker.Guard();
    }
    ...
}
```

Все остальные атрибуты для краткости опущены

Попытка внедрения через конструктор для получения нестабильной зависимости аспекта. Вскоре будет показано, что этот код неработоспособен

Использование нестабильной зависимости breaker внутри методов OnEntry, OnSuccess и OnException

Данная попытка внедрения через конструктор в этот класс аспекта не увенчалась успехом. Вспомним, что был определен атрибут, представляющий особый код, который в ходе компиляции будет автоматически добавлен в те методы, с которыми ведется работа. В среде .NET атрибуты в своих конструкторах могут располагать только элементарными типами, например строками или целыми числами.

Даже если бы у атрибутов могли быть более сложные зависимости, у вас не было бы способа предоставить экземпляру этого аспекта экземпляр `ICircuitBreaker`, потому что аспект создается в совершенно другом месте и времени — не там и не во время создания экземпляров `ICircuitBreaker`. Такие экземпляры атрибутов, как `CircuitBreakerAttribute`, создаются средой выполнения .NET, и повлиять на этот процесс невозможно. У вас нет средств внедрения зависимости в конструктор атрибута в качестве его составной части, как, например, в корень композиции:

```
[CircuitBreakerAttribute(???)]
```

Что сюда внедрить? И как?

Эта проблема не ограничивается работой с атрибутами. Даже если среда AOP задействует механизм, отличный от атрибутов, ее посткомпилятор в ходе компиляции автоматически добавляет код аспекта в ваш обычный код и делает его частью кода сборки. А вот графы ваших объектов создаются во время выполнения как часть корня композиции. Эти две модели плохо сочетаются. Внедрение через конструктор при использовании автоматического добавления аспекта в ходе компиляции невозможно.

Обойти эту проблему можно двумя способами: применением антипаттерна «Окружающий контекст» или антипаттерна «Локатор сервисов». Но у обоих есть и свои весьма существенные недостатки.

Рассмотрим, как обойти проблему с помощью антипаттерна «Окружающий контекст». В листинге 11.7 показано определение открытого статического свойства `Breaker` в аспекте предохранителя.

Листинг 11.7. Использование зависимости внутри аспекта с применением антипаттерна «Окружающий контекст»

```
public class CircuitBreakerAttribute
    : OnMethodBoundaryAspect
{
    public static ICircuitBreaker Breaker { get; set; }

    public override void OnEntry(
        MethodExecutionArgs args)
    {
        Breaker.Guard();
    }

    ...
}
```

Этот открытое статическое свойство, позволяющее устанавливать интерфейс ICircuitBreaker в корень композиции при запуске приложения. В результате получается антипаттерн «Окружающий контекст»

Как говорилось в подразделе 5.3.2, кроме прочего, «Окружающий контекст» вызывает временную связанность. Следовательно, если забыть установить свойство `Breaker`, приложение даст сбой и будет выдано исключение `NullReferenceException`, так как зависимость является обязательной.

Кроме того, поскольку единственным вариантом является однократная установка свойства во время запуска приложения, оно должно быть определено как `static`. Но это чревато другими проблемами: может получиться так, что `ICircuitBreaker` станет захваченной зависимостью, как объяснялось в подразделе 8.4.1.

Такое статическое свойство становится причиной взаимозависимости тестов, поскольку его значение остается в памяти при выполнении следующего тестового сценария. Поэтому после любого теста необходимо перенастраивать тестовые установки (Fixture Teardown)[1]. Это одно из обязательных действий, о котором нельзя

[1] Дополнительные сведения о перенастройке тестовых установок можно найти в издании: *Meszaros G.* xUnit Test Patterns: Refactoring Test Code. — Addison-Wesley, 2007. — P. 100.

забывать. Поэтому тестировать аспекты, добавляемые автоматически в ходе компиляции и использующие «Окружающий контекст» для доступа к нестабильным зависимостям, нелегко.

ПРИМЕЧАНИЕ

Негативные последствия появления захваченных зависимостей можно смягчить за счет реализации для ICircuitBreaker фабрик или заместителей, но мы полагаем, что сейчас вы уже начинаете понимать все сложности, вызываемые применением статического свойства.

Еще одним способом обхода является задействование антипаттерна «Локатор сервисов», но по сравнению с антипаттерном «Окружающий контекст» он только усугубляет ситуацию. В результате применения «Локатора сервисов» проявляются те же проблемы, связанные со взаимозависимостью тестов и временной связанностью. Кроме того, доступ аспекта к несвязанному набору нестабильных зависимостей не дает четкой картины характера его зависимостей, и он тащит за собой «Локатор сервисов» в качестве избыточной зависимости. Поскольку «Локатор сервисов» — это самый сомнительный выбор, пример приводить не будем, а сразу перейдем ко второму недостатку автоматического добавления аспектов в ходе компиляции — связанности в период компиляции.

Автоматическое добавление аспектов в ходе компиляции приводит к сильной связанности

Хотя автоматическое добавление аспектов в ходе компиляции отделяет от них исходный код, оно все же заставляет скомпилированный код получать сильную связанность с добавляемыми аспектами. Проблема заключается в том, что сквозная функциональность зачастую зависит от внешней системы. Это проявляется при написании модульных тестов, поскольку должна существовать возможность запуска модульного теста в изолированном состоянии. Требуется протестировать саму логику класса без взаимосвязи с его нестабильными зависимостями. Не нужно, чтобы модульные тесты пересекали границы процесса и сети, поскольку обмен данными с базой данных, файловой системой или другими внешними системами будет отрицательно влиять на надежность и производительность ваших тестов. Иными словами, автоматическое добавление аспектов в ходе компиляции ухудшает тестируемость.

Но даже при проведении широкомасштабных интеграционных тестов автоматическое добавление аспектов в ходе компиляции по-прежнему будет вызывать проблемы. В интеграционном тесте проверяется согласованность одной части системы с другими ее частями. Это снижает изолированность, но позволяет определить, согласованно ли отдельно взятые компоненты будут работать с другими компонентами. Если бы, к примеру, тестировалось хранилище `SqlProductRepository`, то проведение модульного теста не имело бы смысла, поскольку единственное действие

этого хранилища — запрос к базе данных. Поэтому нужно будет протестировать взаимодействие этого компонента с базой данных.

В этом случае вряд ли захочется во время тестирования применять все аспекты. К примеру, при проверке способности компонента успешно сохранять и извлекать товары используемый аспект `[CheckAuthorization]` может заставить в ходе теста пройти через определенный процесс входа в систему. Важно посмотреть, работает ли этот аспект авторизации так, как ожидается. Но, к сожалению, запуск авторизации в качестве части интеграционного теста усложняет сопровождение таких тестов и может существенно замедлить их выполнение.

Еще забавнее наблюдать за проблемой, проявляющейся при включенном кэшировании. Можно написать автоматизированный тест с намерением отправить запрос к базе данных, но фактически он никогда не будет выполнен, поскольку завершится попаданием в кэш. Именно поэтому в случае, если эти аспекты связаны с нестабильными зависимостями, требуется полный контроль над тем, какие аспекты к какому тесту и когда применяются. Автоматическое добавление аспектов в ходе компиляции чрезвычайно усложняет эту задачу.

Автоматическое добавление аспектов в ходе компиляции не подходит для использования нестабильных зависимостей

Целью применения технологии DI является управление нестабильными зависимостями путем введения в приложение швов. Это позволяет сосредоточить композицию ваших графов объектов в корне композиции.

При автоматическом добавлении аспектов в ходе компиляции все наоборот — происходит привязка нестабильных зависимостей к коду во время компиляции. Это не позволяет воспользоваться методами DI и безопасно составить полноценные графы объектов в корне композиции приложения. Именно по этой причине мы говорим, что автоматическое добавление аспектов в ходе компиляции является противоположностью технологии DI, а его реализация в отношении нестабильных зависимостей — антипаттерном.

ВНИМАНИЕ

> Мы намеренно говорили об автоматическом добавлении аспектов в ходе компиляции в сочетании с нестабильными зависимостями. С позиции DI стабильные зависимости особого интереса не представляют. Можно увидеть ценность в применении стабильных зависимостей с помощью автоматического добавления аспектов в ходе компиляции, но для нестабильных зависимостей такое добавление никакой ценности не представляет.

Нужно отдавать предпочтение соблюдению SOLID-принципов, а если это невозможно, откатываться к динамическому перехвату. На этой ноте можно оставить рассмотрение чистой технологии DI в части III и перейти к части IV, посвященной DI-контейнерам. Там вы узнаете, как с помощью DI-контейнеров избавиться от некоторых сложностей, с которыми вы можете столкнуться.

Резюме

- Динамический перехват относится к технологии аспектно-ориентированного программирования, с помощью которой автоматизируется генерация декораторов для их выдачи во время выполнения программы. Аспекты создаются в виде перехватчиков, внедряемых в декоратор, генерируемый при выполнении программы.
- У динамического перехвата проявляются следующие недостатки:
  - потеря поддержки в ходе компиляции;
  - сильная связанность аспектов с инструментарием;
  - отсутствие универсальности применения;
  - базовые проблемы проектирования никуда не исчезают.
- Динамический перехват может стать неплохим временным решением, чтобы предотвратить или отложить внесение конструктивных изменений, подобных предложенным в главе 10, и оставаться таковым до тех пор, пока не наступит пора приступить к внесению предлагаемых усовершенствований.
- Автоматическое добавление аспектов в ходе компиляции является технологией AOP, изменяющей процесс компиляции. Для изменения скомпилированной сборки путем манипуляций с кодом IL используется специальный инструментарий. Подобный метод применения AOP к нестабильным зависимостям считается нежелательным.
- В отношении технологии DI у автоматического добавления аспектов в ходе компиляции возникают следующие проблемы:
  - плохая совместимость автоматического добавления аспектов с технологией DI;
  - автоматическое добавление аспектов приводит к сильной связанности в период компиляции.
- Нужно отдавать предпочтение соблюдению SOLID-принципов, а если это невозможно, откатываться к динамическому перехвату.

Часть IV
DI-контейнеры

Предыдущие части книги были посвящены различным принципам и паттернам, в совокупности определяющим суть технологии DI. Как говорилось в главе 3, DI-контейнер является дополнительным инструментом, которым можно воспользоваться для реализации множества универсальных инфраструктур, которые иначе, при использовании чистой технологии DI, пришлось бы реализовывать самостоятельно.

Пока в этой книге мы специально не обсуждали контейнеры, то есть обучали вас применению только чистой технологии DI. Не следует считать это рекомендацией по внедрению зависимостей как таковому, мы всего лишь хотели показать технологию DI в ее самом чистом виде, не затронутом конкретным контейнерным API.

Для среды .NET имеется множество замечательных DI-контейнеров. В главе 12 говорится о том, когда следует использовать один из них, а когда — остановить выбор на чистой технологии DI. Остальные главы части IV посвящены выбору из трех DI-контейнеров, имеющих открытый код и находящихся в свободном доступе. В каждой из них подробно рассматривается API конкретного контейнера применительно к тем изменениям, речь о которых шла в части III, а также к другим вопросам, обычно пугающим начинающих разработчиков. Будут рассмотрены следующие контейнеры: Autofac (глава 13), Simple Injector (глава 14) и Microsoft.Extensions.DependencyInjection (глава 15).

Будь у нас неограниченные объем и время, мы бы включили в круг исследованных все контейнеры, но ни тем ни другим не располагаем. Все контейнеры, рассмотренные в первом издании, кроме одного, были исключены. В их числе Castle Windsor, StructureMap, String.NET, Unity и MEF. Чтобы больше узнать о них, обратитесь к копии первого издания (распространяется бесплатно с данной книгой). Кроме того, в наше поле зрения попал один из наиболее популярных DI-контейнеров — Ninject, но его мы тоже рассматривать не будем. На момент написания данной книги его доступной версии, совместимой со средой .NET Core, что стало бы условием для его включения в издание, еще не было.

Все рассматриваемые контейнеры являются проектами с открытым кодом и коротким циклом выпуска новых версий. Прежде чем приступить в этой части к рассмотрению контейнеров, в главе 12 подробно расскажем о том, что представляют собой контейнеры, чем они могут вам помочь и как принять решение, когда использовать DI-контейнер, а когда остановить свой выбор на чистой технологии DI.

Доля, которую на рынке занимает Autofac, не позволила исключить его из рассмотрения, хотя о нем уже говорилось в первом издании. Autofac — это наиболее популярный DI-контейнер для среды .NET. Ему посвящена глава 13. Несмотря на включение в перечень рассматриваемых контейнера Microsoft.Extensions.DependencyInjection (MS.DI), мы испытываем по отношению к нему определенный скепсис, поскольку его функциональность ограничена. Тем не менее мы считали своей обязанностью рассказать о нем, поскольку, прежде чем перейти к инструментарию сторонних производителей, многие разработчики используют встроенный инструментарий. О том, что можно и чего нельзя сделать с помощью MS.DI, будет сказано в главе 15.

Все главы построены по общему шаблону. При чтении одних и тех же предложений в третий раз может возникнуть ощущение дежавю. Мы считаем это преимуществом, поскольку таким образом можно быстрее найти одинаковые разделы в разных главах, если понадобится, сравнить, как реализуется конкретная функция в разных контейнерах.

Эти главы призваны вдохновить вас на выбор наиболее подходящего DI-контейнера. Если вам только предстоит сделать его, то, чтобы сравнить контейнеры, можно прочитать все три главы или только ту, которая вам интересна. Информация, представленная в части IV, была достоверна на момент написания книги, но нужно всегда обращаться к самым свежим источникам.

12 Введение в DI-контейнеры

В этой главе

- Использование конфигурационных файлов для включения позднего связывания.
- Явная регистрация компонентов в DI-контейнере с помощью конфигурации в виде кода.
- Применение соглашения о конфигурации в DI-контейнере с автоматической регистрацией.
- Выбор между чистой технологией DI и использованием DI-контейнера.

(Рассказывает Марк.) Когда я был маленьким, мы с мамой иногда делали мороженое. Случалось это нечасто, так как требовало определенных усилий и умения. Настоящее мороженое делается из легкого заварного крема, приготовляемого из сахара, яичных желтков, молока или сливок. Если эту смесь перегреть, она станет слишком густой. Даже если этого удастся избежать, на следующем этапе возникнет еще больше проблем. Если просто поставить крем в холодильник, он начнет кристаллизоваться, поэтому его нужно размешивать через равные промежутки времени, пока он не загустеет и размешивание станет невозможным. Только тогда у вас получится хорошее домашнее мороженое. Процесс этот медленный и трудоемкий, но если есть желание, необходимые ингредиенты и оборудование, с их помощью можно приготовить мороженое.

Сегодня, по прошествии 35 лет, моя свекровь делает мороженое намного чаще, чем мы с матерью в моем детстве, Причина не в том, что ей нравится заниматься этим, а в том, что ей на помощь пришел специальный технологический процесс. Сама технология не изменилась, но вместо того, чтобы периодически вынимать мороженое из морозильника и помешивать его, она использует электрическую мороженицу, выполняющую эту работу (рис. 12.1).

Рис. 12.1. Итальянская мороженица. Более совершенная технология позволяет быстрее и проще не только готовить мороженое, но и выполнять задачи программирования

DI — это прежде всего приемы программирования, но для упрощения процесса можно воспользоваться специальной технологией. В части III мы описали DI как приемы программирования. В части IV будет рассмотрена специальная технология, которой можно воспользоваться для поддержки приемов программирования DI. Ее мы называем DI-контейнерами.

ОПРЕДЕЛЕНИЕ

> DI-контейнер — это библиотека программных модулей, предоставляющая функциональные возможности внедрения зависимостей и автоматизирующая многие задачи, связанные с составлением композиции объектов, их перехватом и управлением временем жизни объектов. Это механизм, выполняющий разрешение графа объектов и управляющий им.

В этой главе будет рассмотрена сама концепция DI-контейнеров — то, как они вписываются в общую картину внедрения зависимостей, а также разобраны несколько паттернов и практических приемов их применения. Попутно рассмотрим несколько примеров.

Начнется глава с введения в DI-контейнеры, включая описание концепции под названием Auto-Wiring (автоматическое связывание), затем будет представлен раздел с вариантами конфигурации. Каждый из рассматриваемых вариантов можно изучать обособленно, но мы полагаем, что перед тем, как приступить к изучению автоматической регистрации (Auto-Registration), полезно будет ознакомиться как минимум с конфигурацией в виде кода (Configuration as Code).

Последний раздел отличается от других. Он посвящен преимуществам и недостаткам использования DI-контейнеров и помогает решать, принесет ли оно пользу вам и вашим приложениям. Мы считаем, что этот раздел важен и его обязательно следует прочесть всем, независимо от опыта работы с DI и DI-контейнерами. Его можно прочитать обособленно от других разделов, хотя сначала было бы полезно ознакомиться с разделами, посвященными Configuration as Code и Auto-Registration.

Цель этой главы — показать, что представляет собой DI-контейнер и как он согласуется с остальными паттернами и принципами, изложенными в книге. По сути, ее можно рассматривать как введение в часть IV книги. Здесь DI-контейнеры будут рассматриваться в целом, а конкретным контейнерам и их API будут посвящены следующие главы.

12.1. Введение в DI-контейнеры

DI-контейнер — это программная библиотека, позволяющая автоматизировать многие задачи, связанные с компоновкой объектов, управлением временем их жизни и их перехватом. Хотя вся нужная инфраструктура кода может быть написана с применением чистой технологии DI, это не сделает приложение особо ценным. В то же время задача компоновки объектов носит общий характер и может быть окончательно решена путем так называемой естественной подобласти (перевод взят из книги Эрика Эванса. — *Примеч. ред.*) (Generic Subdomain)[1]. Исходя из этого использование универсальной библиотеки может иметь вполне определенный смысл. Нет особой разницы, что реализуется, регистрация событий или доступ к данным, — регистрация данных приложения относится к тем задачам, которые можно решить

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — С. 364.

с помощью универсальной библиотеки ведения журнала. Это же справедливо и для составления графов объектов.

В этом разделе расскажем, как DI-контейнеры справляются с составлением графов объектов. Приведем также несколько примеров, чтобы дать общее представление о том, как могут выглядеть использование контейнера и его реализация.

12.1.1. Исследование контейнерного API, выполняющего разрешение

DI-контейнер представляет собой программную библиотеку, похожую на любую другую библиотеку программных модулей. Она предоставляет API, которым можно воспользоваться для компоновки объектов и составления графа объектов в рамках одного вызова метода. DI-контейнеры также требуют, чтобы до компоновки объектов было проведено их конфигурирование. К этому вопросу мы вернемся в разделе 12.2.

Здесь рассмотрим несколько примеров, чтобы понять, как DI-контейнеры могут выполнить разрешение графов объектов. В качестве примеров в этом разделе будут использоваться контейнеры Autofac и Simple Injector, применяемые к приложению среды ASP.NET Core MVC. Подробности, касающиеся композиции приложений этой среды, вы найдете в разделе 7.3.

Для разрешения экземпляров контроллеров можно воспользоваться DI-контейнером. Эта функция может быть реализована с помощью трех DI-контейнеров, рассматриваемых в следующих главах, но здесь будут показаны всего два примера.

Разрешение контроллеров с помощью различных DI-контейнеров

Autofac является DI-контейнером с практически шаблонным API. Если есть экземпляр контейнера Autofac, разрешение контроллера можно выполнить, предоставив запрашиваемый тип:

```
var controller = (HomeController)container.Resolve(typeof(HomeController));
```

Методу `Resolve` будет передан аргумент `typeof(HomeController)`, а обратно получен экземпляр запрошенного типа с соответствующими зависимостями. Метод `Resolve` является слабо типизированным и возвращает экземпляр `System.Object`. Это означает, как показано в примере, что требуется приведение типов к чему-то более конкретному.

API, похожие на имеющийся в Autofac, есть у многих DI-контейнеров. Соответствующий код для Simple Injector практически идентичен коду для Autofac даже притом, что экземпляры разрешаются с помощью класса `SimpleInjector.Container`. При использовании Simple Injector предыдущий пример приобретет следующий вид:

```
controller = (HomeController)container.GetInstance(typeof(HomeController));
```

Единственным реальным отличием является то, что здесь вместо метода `Resolve` применяется метод `GetInstance`. Из этих примеров можно извлечь общие очертания DI-контейнера.

Разрешение графов объектов с помощью DI-контейнеров

DI-контейнер представляет собой механизм, разрешающий графы объектов и управляющий ими. Хотя DI-контейнер занимается не только разрешением объектов, он является центральной частью любого контейнерного API. В предыдущих примерах было показано, что у контейнеров для этого имеется метод, обладающий слабой типизацией. Имена и сигнатуры могут быть разными, но в целом метод имеет следующий вид:

```
object Resolve(Type serviceType);
```

Как было показано в предыдущих примерах, поскольку возвращаемый экземпляр типизирован как `System.Object`, возвращаемое значение до его использования зачастую приходится приводить к ожидаемому типу. Многие DI-контейнеры также предлагают обобщенную версию для тех случаев, когда запрашиваемый тип уже известен в ходе компиляции. Чаще всего они выглядят следующим образом:

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

Вместо предоставления аргумента метода `Type` такое переопределение принимает параметр типа (`T`), указывающий на запрашиваемый тип. Метод возвращает экземпляр `T`. Если разрешение запрашиваемого типа выполнить невозможно, большинство контейнеров выдают исключение.

ВНИМАНИЕ

> У метода Resolve чрезвычайно эффективная и универсальная сигнатура. Можно запросить экземпляр любого типа, и код все равно пройдет компиляцию. Фактически метод Resolve соответствует сигнатуре локатора сервисов. Как говорилось в разделе 5.2, чтобы ваш DI-контейнер не стал использоваться в качестве локатора сервисов при вызове Resolve за пределами корня композиции, нужно проявить особую осмотрительность.

Если взглянуть на метод `Resolve` в отрыве от всего остального, то он может показаться неким магическим средством. С позиции компилятора от него можно потребовать разрешения экземпляров любых произвольных типов. Откуда контейнер знает, как составить запрошенный тип, включая все зависимости? А он и не знает — сначала ему нужно сообщить об этом с помощью конфигурации, отображающей абстракции на конкретные типы. К этой теме нам еще предстоит вернуться в разделе 12.2.

Если контейнер не располагает достаточными данными о конфигурации для полного создания запрошенного типа, то он обычно выдает исключение с описанием случившегося. Изучим в качестве примера следующий `HomeController`, который впервые рассматривался в листинге 3.4. Как вы, возможно, помните, он содержит зависимость типа `IProductService`:

```
public class HomeController : Controller
{
    private readonly IProductService productService;
```

```
    public HomeController(IProductService productService)
    {
        this.productService = productService;
    }
    ...
}
```

В случае неполной конфигурации Simple Injector выдает исключение с примерно таким сообщением: The constructor of type HomeController contains the parameter with name 'productService' and type IProductService, which isn't registered. Please ensure IProductService is registered or change the constructor of HomeController (Конструктор типа HomeController содержит параметр по имени 'productService', имеющий тип IProductService, который не прошел регистрацию. Убедитесь, что IProductService зарегистрирован, или измените конструктор HomeController).

Из предыдущего примера следует, что Simple Injector не может выполнить разрешение `HomeController`, поскольку в нем содержится аргумент конструктора типа `IProductService`, но контейнеру Simple Injector не было сообщено, какую реализацию следует возвратить при запросе `IProductService`. Если контейнер имеет правильную конфигурацию, он способен выполнить разрешение из запрошенного типа даже сложного графа объектов. Если что-то в конфигурации упущено, контейнер может предоставить подробную информацию о том, что это было. В следующем разделе будет дана более подробная картина того, как это делается.

12.1.2. Автоматическое связывание Auto-Wiring

DI-контейнеры выдают желаемый результат на основе статической информации, скомпилированной во все классы. Используя отражение, они способны проанализировать запрашиваемый класс и выяснить, какие зависимости нужны.

Как говорилось в разделе 4.2, предпочтительным способом применения технологии DI является внедрение через конструктор, поэтому, в принципе, в этом способе внедрения разбираются все DI-контейнеры. В частности, они составляют графы объектов, комбинируя собственную конфигурацию с информацией, извлеченной из данных о типе классов. Это называется автоматическим связыванием.

ОПРЕДЕЛЕНИЕ

> Автоматическое связывание — это возможность автоматического составления графа объектов из отображения между абстракциями и конкретными типами путем использования информации о типе, предоставленной компилятором и общеязыковой исполняющей средой (CLR).

Большинство DI-контейнеров разбирается также во внедрении через свойство, хотя некоторым из них требуется явное включение этой функции. С учетом недостатков, свойственных внедрению через свойство (рассмотрены в разделе 4.4), это весьма полезное качество. Общий алгоритм работы большинства

DI-контейнеров, применяющих автоматическое связывание графа объектов, показан на рис. 12.2.

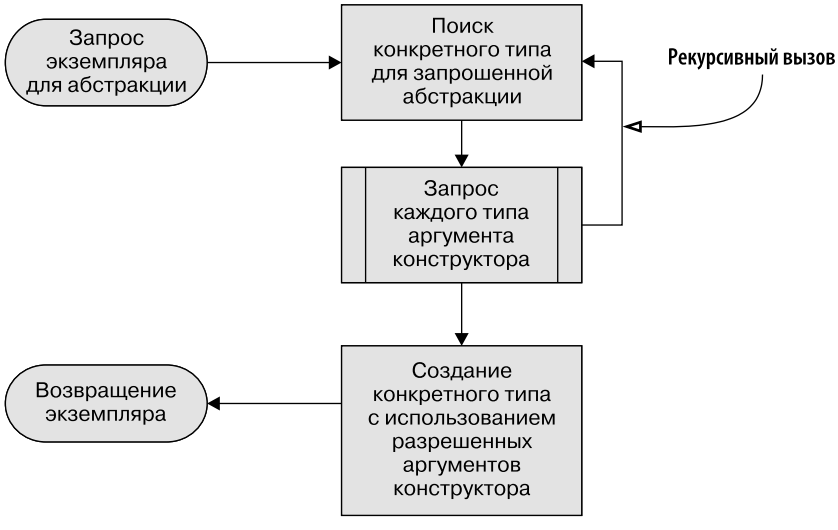


Рис. 12.2. Упрощенный поток событий при автоматическом связывании. DI-контейнер использует свою конфигурацию для поиска конкретного класса, соответствующего запрошенному типу. Затем для проверки конструктора класса применяет отражение

Исходя из увиденного, DI-контейнер ищет конкретный тип для запрошенной абстракции. Если конструктору конкретного типа требуются аргументы, запускается рекурсивный процесс, в ходе которого DI-контейнер повторяет все действия для каждого типа аргумента, пока не будут получены все аргументы конструктора. Когда процесс завершится, контейнер создаст конкретный тип, внедряя рекурсивно разрешенные зависимости.

ПРИМЕЧАНИЕ

В большинстве DI-контейнеров реализованы оптимизации, позволяющие последовательным запросам выполняться быстрее. В разных контейнерах они реализуются по-разному. Как упоминалось в подразделе 4.2.2, DI-контейнер обычно не служит причиной существенного снижения производительности приложения. Для среднестатистического приложения самым узким местом в этом плане являются операции ввода-вывода, и обычно от их оптимизации куда больше толку, чем от оптимизации компоновки объектов.

В разделе 12.2 конфигурации контейнеров будет уделено более пристальное внимание. Пока важнее всего понять, что ядром конфигурации является список отображений между абстракциями и представляющими их конкретными классами. Воспринимается это как нечто теоретическое, поэтому полагаем, что пример не будет лишним.

12.1.3. Пример: реализация упрощенного DI-контейнера, поддерживающего автоматическое связывание

Чтобы показать, как работает автоматическое связывание (Auto-Wiring), и развеять представление о работе DI-контейнеров как о чем-то волшебном, рассмотрим самый простой пример реализации DI-контейнера, способного создавать сложные графы объектов с использованием автоматического связывания.

ВНИМАНИЕ

Листинг 12.1 помечен как плохой код, что в данном контексте означает: он непригоден для реального приложения — он нужен только для того, чтобы вы могли усвоить саму концепцию. В разделе 12.3 подробнее объясним, что нужно задействовать либо чистую технологию DI, либо один из широко используемых и хорошо протестированных DI-контейнеров. Этот листинг приведен в сугубо учебных целях и применять его в свой работе ни в коем случае нельзя!

В листинге 12.1 показан код простейшей реализации DI-контейнера. Он не поддерживает управление временем жизни объектов, перехват и многие другие важные функции. Поддерживает только функцию автоматического связывания.

Листинг 12.1. Простейший DI-контейнер, поддерживающий автоматическое связывание

```
public class AutoWireContainer
{
    Dictionary<Type, Func<object>> registrations =
        new Dictionary<Type, Func<object>>();

    public void Register(
        Type serviceType, Type componentType)
    {
        this.registrations[serviceType] =
            () => this.CreateNew(componentType);
    }

    public void Register(
        Type serviceType, Func<object> factory)
    {
        this.registrations[serviceType] = factory;
    }

    public object Resolve(Type type)
    {
        if(this.registrations.ContainsKey(type))
        {
            return this.registrations[type]();
        }

        throw new InvalidOperationException(
            "No registration for " + type);
    }
```

Содержит набор отображений

Создает новую регистрацию и добавляет отображение для типа сервиса к регистрационному словарю

Можно самостоятельно предоставить контейнеру делегат Func<T>, обходя при необходимости автоматическое связывание

Разрешение полного графа объектов

```
    private object CreateNew(Type componentType)
    {
        var ctor =
            componentType.GetConstructors()[0];

        var dependencies =
            from p in ctor.GetParameters()
            select this.Resolve(p.ParameterType);

        return Activator.CreateInstance(
            componentType, dependencies.ToArray());
    }
}
```

Создание нового экземпляра компонента

В `AutoWireContainer` содержится набор регистраций. Под регистрацией понимается отображение между абстракцией (типом сервиса) и типом компонента. Абстракция представлена в виде словарного ключа, а ее значение является делегатом `Func<object>`, позволяющим сконструировать новый экземпляр компонента, реализующего абстракцию. Метод `Register` проводит новую регистрацию, сообщая контейнеру, какой компонент должен быть создан для заданного типа сервиса. Указывается только, какой компонент следует создать и как это сделать.

Метод `Register` добавляет отображение для типа сервиса к регистрационному словарю. Если нужно, метод `Register` может предоставить контейнеру делегат `Func<T>` напрямую. При этом возможность автоматического связывания будет обойдена. Вместо этого он вызовет предоставленный делегат.

Метод `Resolve` позволяет разрешить полный граф объектов. Для запрошенного `serviceType` он получает `Func<T>` из регистрационного словаря, запускает его и возвращает его значение. При отсутствии регистрации для запрошенного типа `Resolve` выдает исключение. И наконец, `CreateNew` создает новый экземпляр компонента путем перебора параметров конструктора компонента и обратного рекурсивного вызова контейнера. Это делается вызовом `Resolve` для каждого параметра с предоставлением типа параметра `Type`. Когда таким вот образом будут разрешены все зависимости типа, он конструирует сам тип, используя отражения (применяя класс `System.Activator`).

ПРИМЕЧАНИЕ

> Принадлежащий AutoWireContainer метод CreateNew содержит суть примера, показанного в листинге 12.1. Для рекурсивного обратного вызова контейнера для получения зависимостей типа и еще одного вызова с целью создания самого типа он для анализа информации о типе использует отражение. CreateNew реализует автоматическое связывание.

Экземпляр `AutoWireContainer` может быть настроен на составление произвольных графов объектов. Если вернуться к главе 3, то в листинге 3.13 `HomeController` создавался с применением чистой технологии DI. Код этого листинга повторяется в листинге 12.2. Он используется в качестве примера, чтобы продемонстрировать возможности автоматического связывания, предварительно определенные в `AutoWireContainer`.

Листинг 12.2. Составление графа объектов для HomeController с помощью чистой технологии DI

```
new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        new AspNetUserContextAdapter()));
```

Вместо самостоятельного составления графа объектов, показанного в листинге 12.2, для регистрации пяти необходимых компонентов можно воспользоваться `AutoWireContainer`. Для этого данные пять компонентов нужно отобразить на соответствующие им абстракции (табл. 12.1).

Таблица 12.1. Отображение типов для поддержки Auto-Wiring контроллера HomeController

Абстракция	Конкретный тип
HomeController	HomeController
IProductService	ProductService
IProductRepository	SqlProductRepository
CommerceContext	CommerceContext
IUserContext	AspNetUserContextAdapter

В листинге 12.3 показано, как для добавления нужного отображения, указанного в табл. 12.1, можно воспользоваться принадлежащим `AutoWireContainer` методом `Register`. Следует заметить, что в этом листинге применяется конфигурация в виде кода (Configuration as Code), рассматриваемая в подразделе 12.2.2.

Листинг 12.3. Использование AutoWireContainer для регистрации HomeController

```
var container = new AutoWireContainer();

container.Register(
    typeof(IUserContext),
    typeof(AspNetUserContextAdapter));

container.Register(
    typeof(IProductRepository),
    typeof(SqlProductRepository));

container.Register(
    typeof(IProductService),
    typeof(ProductService));

container.Register(
    typeof(HomeController),
    typeof(HomeController));

container.Register(
    typeof(CommerceContext),
    () => new CommerceContext(connectionString));
```

Создание нового экземпляра контейнера

Регистрация отображения между абстракцией и конкретным типом, не требующая указания зависимостей типа. Следует заметить: поскольку контейнер использует внутренний словарь, порядок регистрации не имеет значения

В случае с HomeController абстракция и конкретный тип относятся к одному и тому же типу. Это означает, что при каждом запросе HomeController вы получаете HomeController

Когда запрашивается CommerceContext, метод Resolve вызывает этот делегат

ПРИМЕЧАНИЕ

Когда запрашивается CommerceContext, то вместо использования Auto-Wiring вы создаете контекст самостоятельно. Здесь требуется ручное связывание, поскольку CommerceContext содержит параметр connectionString, представленный элементарным типом string.

Возможно, отображение для `HomeController` в табл. 12.1 и листинге 12.3 покажется странным, поскольку вместо отображения на абстракцию показано отображение на самого себя. Но на самом деле это вполне устоявшаяся практика, особенно когда дело касается типов, находящихся на верхушке графа объектов, например, контроллеров MVC.

Нечто подобное уже встречалось в листингах 4.4, 7.8 и 8.3, где при запросе типа `HomeController` создавался новый экземпляр `HomeController`. Основное отличие кода этих листингов от кода листинга 12.3 состоит в том, что в последнем вместо чистой технологии DI используется DI-контейнер.

ПРИМЕЧАНИЕ

Вместо самостоятельного связывания CommerceContext можно было бы попробовать применить к CommerceContext автоматическое связывание, добавив к connectionString дополнительную регистрацию. Но, поскольку connectionString принадлежит к типу String, это вызвало бы неоднозначность. Вспомним, что DI-контейнеры выполняют разрешение зависимостей на основе их типов, но компонентам приложения может потребоваться множество значений конфигурации, относящихся к типу String. В таком случае контейнер не сможет определить, какое из значений следует использовать, поскольку у всех них будет один и тот же тип. Проблема решается с помощью самостоятельного связывания CommerceContext.

Код листинга 12.3 успешно регистрирует все компоненты, необходимые для составления графа объектов `HomeController`. Теперь для создания нового экземпляра `HomeController` можно воспользоваться сконфигурированным `AutoWireContainer` (листинг 12.4).

Листинг 12.4. Использование AutoWireContainer для разрешения HomeController

```
object controller = container.Resolve(typeof(HomeController));
```

Когда для запроса нового типа `HomeController` вызывается метод `Resolve`, принадлежащий `AutoWireContainer`, контейнер рекурсивно вызывает самого себя до тех пор, пока не разрешит все требуемые зависимости. После этого создается новый экземпляр `HomeController`, а его конструктору предоставляются разрешенные зависимости. Рекурсивный процесс с использованием несколько необычного представления для визуализации рекурсивных вызовов показан на рис. 12.3. Экземпляр контейнера распространен на четыре отдельные вертикальные линии, представляющие отметки времени. У нас получается несколько уровней рекурсивных вызовов и их сворачивание в одну строку, что является нормой для диаграмм последовательности UML, но может вызвать путаницу.

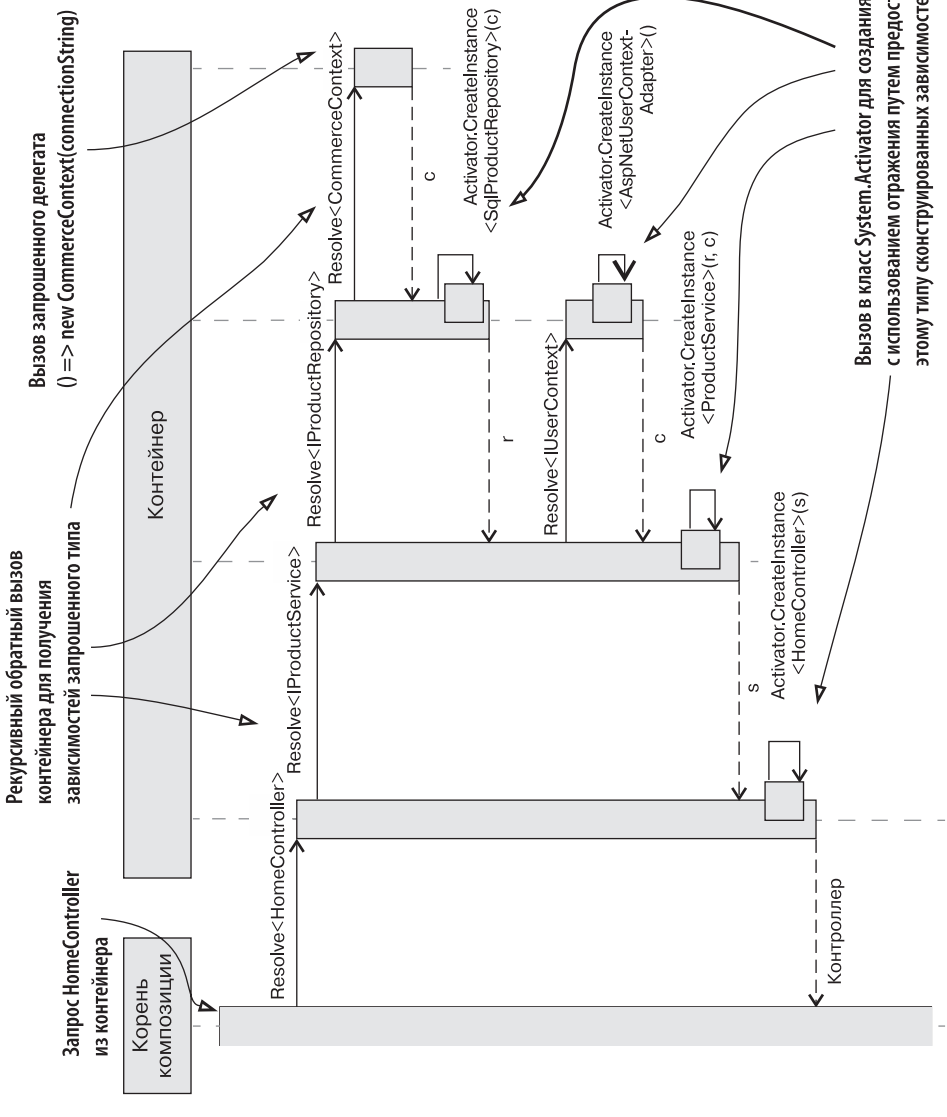


Рис. 12.3. Корень композиции запрашивает HomeController из контейнера, который выполняет рекурсивные обратные вызовы самого себя для запроса зависимостей HomeController

Когда DI-контейнер получает запрос, касающийся `HomeController`, первым делом выполняется поиск типа в его конфигурации. `HomeController` является конкретным классом, который отображается на самого себя. Затем контейнер использует отражение для проверки единственного конструктора `HomeController`, для чего применяется следующая сигнатура:

```
public HomeController(IProductService productService)
```

Поскольку у этого конструктора есть параметры, процесс при выполнении общей блок-схемы (см. рис. 12.3) требует повторения для аргумента конструктора `IProductService`. Контейнер выполняет поиск `IProductService` в своей конфигурации и обнаруживает, что он отображается на конкретный класс `ProductService`. Единственный открытый конструктор для `ProductService` имеет следующую сигнатуру:

```
public ProductService(
    IProductRepository repository,
    IUserContext userContext)
```

У конструктора все еще имеются параметры, и теперь приходится иметь дело с двумя аргументами конструктора. Контейнер обрабатывает их по очереди, начиная с интерфейса `IProductRepository`, который в соответствии с конфигурацией отображается на `SqlProductRepository`. У этого `SqlProductRepository` имеется открытый конструктор со следующей сигнатурой:

```
public SqlProductRepository(CommerceContext context)
```

Этот конструктор также не лишен параметров, поэтому для удовлетворения и конструктора `SqlProductRepository` контейнеру приходится разрешать `CommerceContext`. Но `CommerceContext` зарегистрирован в коде, показанном в листинге 12.3, с использованием следующего делегата:

```
() => new CommerceContext(connectionString)
```

Этот синтаксис для определения безымянных функций известен также как лямбда-выражение

Контейнер вызывает этот делегат, в результате чего получается новый экземпляр `CommerceContext`. На этот раз автоматическое связывание не используется.

ВНИМАНИЕ

> Начиная применять DI-контейнер, совсем не обязательно полностью отказываться от самостоятельного связывания графов объектов.

После того как у контейнера появилось приемлемое значение для `CommerceContext`, он может вызвать конструктор `SqlProductRepository`. Теперь у него есть успешно обработанный параметр `Repository` для конструктора `ProductService`, но ему нужно будет еще какое-то время сохранять это значение, а также он должен будет заняться параметром `userContext` конструктора `ProductService`. В соответствии с конфигурацией `IUserContext` отображается на конкретный класс `AspNetUserContextAdapter`, у которого имеется следующий открытый конструктор:

```
public AspNetUserContextAdapter()
```

ПРИМЕЧАНИЕ

Можно вспомнить, что в коде листинга 3.12 для AspNetUserContextAdapter вообще не был указан конструктор. Если конструктор классу не определен, компилятор C# выполняет компиляцию класса, используя публичный конструктор без параметров.

Поскольку в `AspNetUserContextAdapter` содержится конструктор без параметров, его экземпляр может быть создан без обязательного разрешения каких-либо зависимостей. Новый экземпляр `AspNetUserContextAdapter` может быть передан конструктору `ProductService`. Теперь он вместе с ранее рассмотренным `SqlProductRepository` удовлетворяет всем требованиям конструктора `ProductService` и вызывает его через отражение. И наконец, только что созданный экземпляр `ProductService` передается конструктору `HomeController`, в результате чего возвращается экземпляр `HomeController`. На рис. 12.4 показан основной рабочий процесс, представленный на рис. 12.2, отображенный на `AutoWireContainer` из листинга 12.1.

Рис. 12.4. Упрощенный рабочий процесс для автоматического связывания, отображенный на код из листинга 12.1. Конкретный тип запрашивается из словаря регистраций, параметры его конструктора получают разрешение, а конкретный тип создается с помощью его разрешенных зависимостей

Преимущества использования автоматического связывания, присущего DI-контейнерам, показанного в листинге 12.3, в отличие от чистой технологии DI, показанной в листинге 12.2, состоят в том, что применение чистой технологии требует отображения в корне композиции. При автоматическом связывании корень композиции становится более стойким к таким изменениям.

Предположим, что зависимость `CommerceContext` нужно добавить к `AspNetUserContextAdapter`, чтобы он мог делать запросы к базе данных. Изменения, которые необходимо внести в корень композиции при использовании чистой технологии DI, показаны в листинге 12.5.

Листинг 12.5. Корень композиции для измененного AspNetUserContextAdapter

```
new HomeController(
    new ProductService(
        new SqlProductRepository(
            new CommerceContext(connectionString)),
        new AspNetUserContextAdapter(
            new CommerceContext(connectionString))));
```

Теперь CommerceContext внедрен в AspNetUserContextAdapter

В то же время автоматическое связывание не вносит в корень композиции каких-либо изменений, требующихся в данном случае. `AspNetUserContextAdapter` связывается автоматически, и по причине того, что его новая зависимость `CommerceContext` уже была зарегистрирована, контейнер сможет разобраться с новым аргументом конструктора и успешно создать новый `AspNetUserContextAdapter`.

ВНИМАНИЕ

Хотя автоматическое связывание может снизить требуемый уровень сопровождения корня композиции, это еще не означает, что приоритет всегда следует отдавать применению DI-контейнеров, а не чистой технологии DI. Как говорилось ранее, случаи, при которых следует предпочесть чистую технологию DI, будут подробнее рассмотрены в разделе 12.3.

Вот так и работает автоматическое связывание, хотя DI-контейнерам следует также позаботиться об управлении временем жизни объектов и, возможно, обратиться к внедрению через свойство и к другим, более специфическим созидательным требованиям.

ВНИМАНИЕ

При наличии в графе объектов зацикленных зависимостей AutoWireContainer из листинга 12.1 вызовет исключение StackOverflowException[1]. Такие исключения считаются проблемными, так как вызывают аварийное завершение работы приложения, затрудняющее определение того места, из-за которого все произошло. Это одна из множества причин, по которой предпочтение всегда нужно отдавать одному из доступных DI-контейнеров, а не какой-нибудь самодельной реализации. Большинство современных пользующихся популярностью DI-контейнеров обнаруживают зацикленность, не прерывая рабочий процесс.

Важным является то обстоятельство, что внедрение через конструктор статически извещает о требующихся классу зависимостях, а DI-контейнер использует эту информацию для автоматического связывания при составлении сложных графов объектов. Прежде чем составлять графы объектов, контейнер должен быть сконфигурирован. Регистрировать компоненты можно различными способами.

---

[1] Зацикливание зависимостей рассматривалось в разделе 6.3.

12.2. Конфигурация DI-контейнеров

Хотя большинство действий происходит в методе `Resolve`, нужно понимать, что ваше время будет расходоваться в основном на работу с конфигурационным API DI-контейнера. Ведь разрешение графа объектов укладывается всего в один вызов метода.

Как правило, DI-контейнеры поддерживают два или три общих варианта конфигурации, показанных на рис. 12.5. Одни контейнеры не поддерживают файлы конфигурации, другие не поддерживают и автоматическую регистрацию, а вот поддержку конфигурации в виде кода (Configuration as Code) поддерживают все.

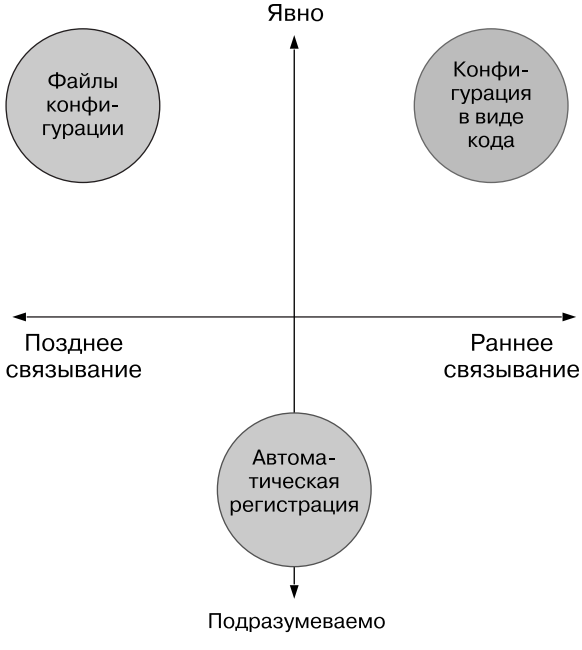


Рис. 12.5. Наиболее распространенные способы конфигурирования DI-контейнеров, показанные в зависимости от степеней явности и времени связывания

У этих трех вариантов конфигурации разные характеристики, польза от которых проявляется в тех или иных ситуациях. Как файлы конфигурации, так и конфигурация в виде кода носят явный характер, поскольку требуют от вас индивидуально регистрировать каждый компонент. А вот автоматическая регистрация имеет более предсказуемый характер, поскольку для регистрации набора компонентов задействует соглашения, соблюдающие единое правило.

При использовании конфигурации в виде кода конфигурация контейнера компилируется в сборку, а конфигурация на основе файла позволяет поддерживать позднее связывание, при котором конфигурацию можно изменять без перекомпиляции кода

приложения. В этом смысле автоматическая регистрация находится где-то посредине, поскольку от нее требуется просканировать в ходе компиляции одну сборку или же просканировать все сборки в предопределенной папке, которая ко времени компиляции может быть неизвестна. Преимущества и недостатки каждого из вариантов сведены в табл. 12.2.

Таблица 12.2. Варианты конфигурации

Стиль	Описание	Преимущества	Недостатки
Файлы конфигурации	Отображения указаны в файлах конфигурации (обычно в формате XML или JSON)	Поддержка замены без перекомпиляции	• Отсутствие проверок в ходе компиляции. • Многословность и хрупкость
Конфигурация в виде кода	Отображения задаются в коде в явном виде	• Проверки в ходе компиляции. • Высокий уровень контроля	• Отсутствие поддержки замены без перекомпиляции
Автоматическая регистрация	Для поиска подходящих компонентов с помощью отражения и для построения отображений используются правила	• Поддержка замены без перекомпиляции. • Требуется меньше усилий. • Помогает применять соглашения, позволяющие повысить последовательность кодовой базы	• Отсутствие проверок в ходе компиляции. • Пониженный уровень контроля. • Поначалу может показаться более абстрактным вариантом

Когда DI-контейнеры только появились, они использовали файлы конфигурации, чем объясняется то, почему этот вариант по-прежнему поддерживают более старые библиотеки. Но ценность данной функции была принижена в пользу более простых подходов. Поэтому DI-контейнеры, разработанные сравнительно недавно, такие как Simple Injector и Microsoft.Extensions.DependencyInjection, не имеют встроенной поддержки конфигурации на основе файлов.

ВНИМАНИЕ

После того как начато разрешение графа объектов, возвращаться назад и заниматься переконфигурацией контейнера не следует, поскольку ничего хорошего из этого не выйдет. Дело в том, что все должно делаться по паттерну «Регистрация — разрешение — высвобождение» (Register — Resolve — Release, https://mng.bz/D8Ew).

Наиболее современным, но не самым очевидным для начала вариантом является автоматическая регистрация. Из-за своей предсказуемости он может показаться абстрактнее более очевидных вариантов, поэтому рассмотрим варианты в порядке их появления, начиная с файлов конфигурации.

12.2.1. Конфигурирование контейнеров с помощью файлов конфигурации

Когда в начале 2000-х годов появились первые DI-контейнеры, в них всех в качестве механизма конфигурирования использовался язык XML (в те времена на его основе делалось многое). Чуть позже опыт работы с XML как с механизмом конфигурирования показал, что это далеко не лучший вариант.

XML страдал многословностью и хрупкостью. При конфигурировании DI-контейнера на языке XML определялись различные классы и интерфейсы, но компилятор не выдавал предупреждения в случае опечаток. Даже если имена классов указывались правильно, не было никакой гарантии, что нужная сборка окажется на пути поиска, используемого приложением.

ВНИМАНИЕ

> В последнее время приобрел популярность способ выражения конфигурационных установок с помощью формата JSON. Он проще и понятнее при чтении, чем XML, но у него те же особенности — хрупкость и многословность.

В довершение всего, по сравнению с обычным кодом XML, не слишком выразителен. Поэтому выразить определенные конфигурации в конфигурационном файле порой весьма нелегко или вовсе невозможно, в то время как их выражение в коде не представляет никаких проблем. Например, в листинге 12.3 регистрация `CommerceContext` была выполнена с использованием лямбда-выражения, которое не может быть выражено в форматах XML и JSON.

В то же время преимущество файлов конфигурации заключается в возможности изменения поведения приложения без его перекомпиляции. Ценность этого свойства проявляется при разработке программного средства, поставляемого тысячам клиентов, поскольку они получают способ подгонки приложения под свои обстоятельства. Но если создается приложение для внутреннего пользования или веб-сайт, где среда развертывания полностью контролируется, то при необходимости изменить поведение приложение проще перекомпилировать и развернуть заново.

ВНИМАНИЕ

> Файлы конфигурации — такая же часть вашего корня композиции, что и конфигурация в виде кода и автоматическая регистрация. Поэтому использование файлов конфигурации не приводит к уменьшению объема корня композиции — часть его просто перемещается в другое место. Файлы конфигурации следует применять только для тех частей конфигурации DI, которым требуется позднее связывание. Для других частей конфигурации следует задействовать конфигурацию в виде кода или автоматической регистрации.

Конфигурирование DI-контейнера с помощью файлов зачастую выполняется путем указания ему на конкретный файл конфигурации. В листинге 12.6 в качестве соответствующего примера используется контейнер Autofac.

ПРИМЕЧАНИЕ

Поскольку Autofac является единственным рассматриваемым в этой книге DI-контейнером со встроенной поддержкой файлов конфигурации, использование его в качестве примера имеет вполне определенный смысл.

В этом примере выполняется конфигурирование тех же самых классов, что и в подразделе 12.1.3. Основной частью задачи является применение конфигурации, намеченной в табл. 12.1, также нужно предоставить точно такую же конфигурацию для поддержки составления композиции класса `HomeController`. Конфигурация, необходимая для приведения приложения в рабочее состояние, дана в следующем листинге.

Листинг 12.6. Конфигурирование Autofac с помощью файла конфигурации в формате JSON

```
{
    "defaultAssembly": "Commerce.Web",
    "components": [
    {
        "services": [{
            "type":
                "Commerce.Domain.IUserContext, Commerce.Domain"
        }],
        "type":
            "Commerce.Web.AspNetUserContextAdapter"
    },
    {
        "services": [{
            "type": "Commerce.Domain.IProductRepository, Commerce.Domain"
        }],
        "type": "Commerce.SqlDataAccess.SqlProductRepository, Commerce.
            ➥ SqlDataAccess"
    },
    {
        "services": [{
            "type": "Commerce.Domain.IProductService, Commerce.Domain"
        }],
        "type":
            "Commerce.Domain.ProductService, Commerce.Domain"
    },
    {
        "type": "Commerce.Web.Controllers.HomeController"
    },
    {
        "type": "Commerce.SqlDataAccess.CommerceContext,
            ➥ Commerce.SqlDataAccess",
        "parameters": {
            "connectionString":
                "Server=.;Database=MaryCommerce;Trusted_
            ➥ Connection=True;"
        }
    }]
}
```

defaultAssembly помогает вести запись типов в сокращенной форме

Простое отображение

Если тип отображается сам на себя, массив сервисов можно опустить

Определяет строку подключения в виде значения параметра конструктора по имени connectionString

Если в этом примере в типе или интерфейсе не указать полного имени типа с упоминанием сборки, то сборкой по умолчанию будет считаться `defaultAssembly`. Для простого отображения должны использоваться полные имена типов, включая пространство имен и имя сборки. Поскольку `AspNetUserContextAdapter` не включает в себя имя сборки, Autofac ищет его в сборке `Commerce.Web`, которая была определена в качестве значения для `defaultAssembly`.

Даже в этом простом листинге кода можно увидеть, что конфигурация в формате JSON склонна к хрупкости. Простое отображение, подобное тому, что имеется между интерфейсом `IUserContext` и классом `AspNetUserContextAdapter`, требует слишком много текста в скобках и полных имен типов.

Вспомним, что `CommerceContext` получает в качестве ввода строку подключения, поэтому нужно указывать, где найти ее значение. Добавив к отображению параметры, можно указать значения по имени их параметра — в данном случае `connectionString`. Загрузка конфигурации в контейнер выполняется с помощью следующего кода (листинг 12.7).

Листинг 12.7. Чтение файлов конфигурации с помощью Autofac

```
var builder = new Autofac.ContainerBuilder();

IConfigurationRoot configuration =
    new ConfigurationBuilder()
    .AddJsonFile("autofac.json")
    .Build();

builder.RegisterModule(
    new Autofac.Configuration.ConfigurationModule(
        configuration));
```

Позволяет добавлять отображение между абстракциями и конкретными типами

Загрузка файла конфигурации autofac.json из листинга 12.6 с помощью системы конфигурирования, принадлежащей среде .NET Core

Заключение созданной конфигурации в модуль Autofac, обрабатывающий файл конфигурации и отображающий компоненты для регистрации в Autofac

Autofac является единственным DI-контейнером с поддержкой файлов конфигурации, включенным в эту книгу, но есть и другие, не рассматриваемые здесь DI-контейнеры, в которых поддерживаются файлы конфигурации. Конкретная схема для каждого контейнера своя, но общая структура практически одинакова, поскольку нужно отобразить абстракцию на реализацию.

ВНИМАНИЕ

По мере разрастания и усложнения приложения растет и объем файла конфигурации. Это может стать настоящим камнем преткновения. Дело в том, что с его помощью моделируются такие элементы кода, как классы, параметры и т. п., но без тех преимуществ, которые предоставляются компилятором, режимами отладки и т. д. В силу этого файлы конфигурации приобретают хрупкость, допущенные ошибки оказываются неочевидными, поэтому данный подход следует применять только тогда, когда требуется позднее связывание.

Файлы конфигурации непригодны для масштабирования

(Рассказывает Стивен.) Однажды я выполнял для крупного клиента работу по сопровождению продукта, содержащего код, на создание которого потребовалось свыше 100 человеко-лет программирования. Технология DI применялась там везде, что было большим плюсом. Но для поддержки составления композиции объектов в качестве DI-контейнеров использовалась среда Spring.NET, которая в то время поддерживала только файлы конфигурации формата XML. Хуже того, применяемая версия Spring.NET не поддерживала автоматическое связывание. Это требовало не только абсолютно точного определения каждого отображения в больших XML-файлах, но и указания всех зависимостей конструктора. Поскольку над кодовой базой работало более десятка команд, XML-файлы конфигурации, относящиеся к среде Spring.NET, не только были многословными, хрупкими и непростыми в сопровождении, но и периодически приводили к конфликтам слияния.

Всем разработчикам было понятно, что из-за многословности, хрупкости, отсутствия поддержки со стороны компилятора и низкой производительности при обработке таких XML-файлов конфигурации ежедневно на их разработку тратилось впустую слишком много времени. Лучше было бы с самого начала прийти к решению использовать чистую технологию DI[1]. Сама по себе эта технология не избавила бы от конфликтов слияния, но по крайней мере компилятор помог бы отловить большинство допущенных ошибок на ранней стадии.

СОВЕТ

Хотя в небольших приложениях или применительно к небольшим порциям кода вашего приложения файлы конфигурации могут использоваться весьма успешно, они не подлежат масштабированию. Не стоит брать их в качестве метода конфигурирования DI по умолчанию. После изучения раздела 12.3 станет ясно, что следует задействовать либо чистую технологию DI, либо автоматическую регистрацию.

Не следует допускать, чтобы отсутствие поддержки файлов конфигурации слишком сильно влияло на выбор DI-контейнера. Как уже говорилось, в файлах конфигурации нужно определять только компоненты, обладающие настоящим поздним связыванием, которых вряд ли наберется много. Как показано в листинге 1.2, типы могут быть загружены из файлов конфигурации с помощью всего нескольких инструкций, даже если ваши контейнеры не поддерживают такие файлы.

Недостатки в виде многословности и хрупкости могут подвигнуть вас на то, чтобы отдать предпочтение альтернативным способам конфигурирования контей-

[1] Вряд ли их выбор составления композиции объектов на основе XML казался странным, учитывая, что время начала разработки характеризовалось повсеместным использованием XML.

неров. По степени детализации и концепции на файлы конфигурации очень похожа конфигурация в виде кода, но в ней вместо файлов конфигурации используется программный код.

12.2.2. Конфигурирование контейнеров с применением конфигурации в виде кода

Наверное, самым простым способом компоновки приложения является жесткое кодирование конструкции графов объектов. Казалось бы, это противоречит самому духу DI, поскольку определяет конкретные реализации, которые должны использоваться для всех абстракций к моменту компиляции. Но если сделать это в корне композиции, будет утрачено только одно преимущество из тех, что перечислены в табл. 1.1, а именно позднее связывание.

Преимущество позднего связывания утрачивается при жесткой закодированности зависимостей, но, как говорилось в главе 1, оно не обязательно распространяется на все типы приложений. Если приложение развернуто в ограниченном количестве экземпляров в контролируемой среде, то при необходимости замены модулей его нетрудно будет перекомпилировать и развернуть заново.

«Я часто думаю, что люди проявляют излишнее усердие по определению файлов конфигурации. Зачастую простой и весьма эффективный механизм конфигурации можно создать на языке программирования»[1] (Мартин Фаулер).

При использовании конфигурации в виде кода явно указываются те же дискретные отображения, что и при использовании файлов конфигурации, только вместо XML или JSON задействуется код.

ОПРЕДЕЛЕНИЕ

> При использовании DI-контейнеров конфигурация в виде кода позволяет сохранять конфигурацию контейнера в исходном коде. Каждое отображение между абстракцией и конкретной реализацией выражается явно непосредственно в коде.

Все современные DI-контейнеры полностью поддерживают конфигурацию в виде кода в качестве преемника файлов конфигурации. Фактически в большинстве из них эта форма конфигурации представлена в качестве механизма по умолчанию, а файлы конфигурации могут выступать дополнительной функциональной возможностью. Некоторые контейнеры вообще не предлагают поддержку файлов конфигурации. API, доступный для поддержки конфигурации в виде кода, от контейнера к контейнеру различается, но общей целью по-прежнему является определение дискретных отображений между абстракциями и конкретными типами.

---

[1] *Fowler M.* Inversion of Control Containers and the Dependency Injection Pattern, 2004; https://martinfowler.com/articles/injection.html.

СОВЕТ

Если не нужно применять позднее связывание, предпочтение следует отдавать конфигурации в виде кода, а не использованию файлов конфигурации. Кроме того, что будет оказана помощь со стороны компилятора, система сборки среды Visual Studio автоматически скопирует все нужные сборки в выходную папку. Если понадобится позднее связывание, следует задействовать файлы конфигурации только для тех частей конфигурации, которые в них нуждаются, а они в целом приложении обычно составляют совсем небольшой поднабор типов.

Посмотрим, как можно сконфигурировать приложение электронной торговли, используя конфигурацию в виде кода с Microsoft.Extensions.DependencyInjection. Для этого рассмотрим пример, в котором выполняется конфигурирование в виде кода учебного приложения электронной торговли.

В подразделе 12.2.1 было показано, как можно сконфигурировать учебное приложение электронной торговли с помощью файлов конфигурации с Autofac. С этим же контейнером можно показать и применение конфигурации в виде кода, но, чтобы читать главу стало интереснее, вместо него возьмем Microsoft.Extensions.DependencyInjection. Здесь показано, что при использовании API конфигурирования компании Microsoft конфигурацию из листинга 12.6 можно выразить более компактно (листинг 12.8).

Листинг 12.8. Конфигурирование Microsoft.Extensions.DependencyInjection с помощью кода

```
var services = new ServiceCollection();

services.AddSingleton<
    IUserContext,
    AspNetUserContextAdapter>();

services.AddTransient<
    IProductRepository,
    SqlProductRepository>();

services.AddTransient<
    IProductService,
    ProductService>();

services.AddTransient<HomeController>();

services.AddScoped<CommerceContext>(
    p => new CommerceContext(connectionString));
```

Определение отображения между абстракциями и реализациями

Добавление отображений автоматического связывания между абстракциями и конкретными типами

Переопределение, принимающее конкретный тип в качестве аргумента обобщенного типа

Переопределение, позволяющее отображать абстракцию на делегат Func<T>

`ServiceCollection` является у компании Microsoft эквивалентом принадлежащего контейнеру Autofac `ContainerBuilder`, где определяются отображения между абстракциями и реализациями. Методы `AddTransient`, `AddScoped` и `AddSingleton` используются для добавления отображений автоматического связывания между абстракциями и конкретными типами применительно к конкретным жизненным циклам. Эти методы обобщенные, что выражается в более сжатом коде с дополнительным преимуществом получения некоторой дополнительной проверки в ходе компиляции. В случае, когда конкретный тип отображается сам на себя вместо ото-

бражения абстракции на конкретный тип, используется удобное переопределение, принимающее конкретный тип в качестве аргумента обобщенного типа. И так же, как в примере `AutoWireContainer` из листинга 12.1, API этого DI-контейнера содержит переопределение, позволяющее отображать абстракцию на делегат `Func<T>`.

ПРИМЕЧАНИЕ

Если в этом видится что-то знакомое, не стоит удивляться: концептуально код практически идентичен учебному коду из листинга 12.3. Там нами была заложена основа концепции автоматического связывания.

В листинге 12.8 мы позволили себе продемонстрировать регистрацию компонентов с помощью трех наиболее распространенных жизненных циклов: Singleton, Transient и Scoped. В следующих главах будет более подробно показано, как сконфигурировать жизненный цикл для каждого контейнера.

Сравните этот код с кодом листинга 12.6 и заметьте, насколько он компактнее, притом что делает абсолютно то же самое. Простое отображение, подобное отображению `IProductService` на `ProductService`, выражено одним вызовом метода.

Конфигурация в виде кода не только намного компактнее конфигураций, выраженных в виде файлов конфигурации, но и пользуется поддержкой компилятора. Аргументы типов, использованные в листинге 12.8, представляют реальные типы, проверяемые компилятором. Обобщения идут еще дальше, потому что применение общих ограничений типа, таких как в API Microsoft, позволяет компилятору проверять, соответствует ли предоставленный конкретный тип абстракции. Если преобразование невозможно, код не будет скомпилирован.

Хотя задействовать конфигурацию в виде кода просто и безопасно, она все же требует более тщательного сопровождения, чем хотелось бы. При каждом добавлении к приложению нового типа нужно не забыть его зарегистрировать, а многие регистрации оказываются на поверку похожими друг на друга. Эта проблема решается за счет автоматической регистрации.

12.2.3. Конфигурирование контейнеров по соглашению с использованием автоматической регистрации

Если рассматривать регистрацию из листинга 12.8, то с несколькими строками кода в вашем проекте вполне можно было бы смириться. Но, когда проект разрастается, увеличивается и число регистраций, необходимых для конфигурирования DI-контейнера. Со временем будут появляться многочисленные регистрации, похожие друг на друга. Они будут соответствовать практически одному и тому же паттерну. В листинге 12.9 показано, как могут выглядеть повторяющиеся регистрации.

Листинг 12.9. Повторения в регистрациях при использовании конфигурации в виде кода

```
services.AddTransient<IProductRepository, SqlProductRepository>();
services.AddTransient<ICustomerRepository, SqlCustomerRepository>();
services.AddTransient<IOrderRepository, SqlOrderRepository>();
services.AddTransient<IShipmentRepository, SqlShipmentRepository>();
services.AddTransient<IImageRepository, SqlImageRepository>();
```

```
services.AddTransient<IProductService, ProductService>();
services.AddTransient<ICustomerService, CustomerService>();
services.AddTransient<IOrderService, OrderService>();
services.AddTransient<IShipmentService, ShipmentService>();
services.AddTransient<IImageService, ImageService>();
```

Повторяющееся написание кода регистрации похоже на нарушение DRY-принципа. К тому же данный фрагмент инфраструктуры кода представляется крайне непродуктивным, не добавляющим приложению особой ценности. Можно было бы сэкономить время и снизить вероятность появления ошибок, если бы удалось автоматизировать регистрацию компонентов при условии, что они будут придерживаться определенного соглашения. Многие DI-контейнеры предоставляют возможность автоматической регистрации, позволяющую вам вводить собственные соглашения и применять соглашения по конфигурации (Convention over Configuration).

ОПРЕДЕЛЕНИЕ

Автоматическая регистрация дает контейнеру возможность регистрировать компоненты путем сканирования одной или нескольких сборок на предмет реализации желаемых абстракций на основе конкретного соглашения. Иногда автоматическую регистрацию называют пакетной регистрацией (Batch Registration) или сканированием сборок (Assembly Scanning).

Соглашения по конфигурации

Концепция соглашения по конфигурации — набирающая популярность архитектурная модель. Вместо написания и сопровождения весьма объемного кода конфигурации можно принять соглашения, распространяющиеся на кодовую базу. Хорошим примером простого соглашения может послужить способ нахождения средой ASP.NET Core MVC контроллеров, основанный на их именах[1].

- Поступает запрос на контроллер по имени `Home`.
- Фабрика контроллеров по умолчанию выполняет в списке известных пространств имен сквозной поиск класса `HomeController`. Если такой класс будет найден, фиксируется совпадение.
- Фабрика контроллеров по умолчанию передает тип класса активатору контроллера, который создает экземпляр контроллера.

Соглашение здесь состоит в том, что контроллер должен иметь название `[ИмяКонтроллера]Controller`.

Соглашения могут применяться не только к контроллерам среды ASP.NET Core MVC. Чем больше добавлять соглашений, тем шире будут возможности автоматизации различных частей конфигурации контейнера.

[1] Описание этого соглашения по поиску MVC-контроллеров приводится в упрощенном виде. В реальности все гораздо сложнее (см. https://mng.bz/lED8).

СОВЕТ

> Соглашения по конфигурации дают больше преимуществ, чем просто поддержка конфигурации DI. Они придают вашему коду более высокую степень согласованности, поскольку он работает автоматически, если придерживается соглашений.

В действительности же может потребоваться сочетание автоматической регистрации с конфигурацией в виде кода или с файлами конфигурации, поскольку подогнать каждый отдельно взятый компонент под предметное соглашение может и не получиться. Но чем сильнее удастся подогнать кодовую базу под соглашения, тем легче будет ее сопровождение.

Autofac поддерживает автоматическую регистрацию, но мы подумали, что для конфигурирования учебного приложения электронной торговли с использованием соглашений было бы интереснее воспользоваться еще одним DI-контейнером. Поскольку нам хотелось бы приводить в примерах только контейнеры, рассматриваемые в книге, а также из-за отсутствия в Microsoft.Extensions.DependencyInjection средств автоматической регистрации воспользуемся для иллюстрирования данной концепции контейнером Simple Injector.

Возвращаясь к коду листинга 12.9: вы, наверное, согласитесь, что регистрация различных компонентов доступа к данным выглядит повторяющейся. Нельзя ли обозначить все это каким-нибудь соглашением? Все пять конкретных типов хранилищ, упомянутых в коде листинга 12.9, имеют ряд общих характеристик.

- Все они определены в одной и той же сборке.
- Имя каждого конкретного класса заканчивается на `Repository`.
- В каждом из них реализован один интерфейс.

Представляется, что соответствующее соглашение будет выявлять это сходство путем сканирования рассматриваемой сборки и регистрировать все классы, соответствующие соглашению. Но даже притом, что Simple Injector поддерживает автоматическую регистрацию, соответствующий API фокусируется на регистрации группы типов, совместно использующих один и тот же интерфейс. Сам по себе API этого контейнера не позволяет обозначить данное соглашение из-за отсутствия единого интерфейса, описывающего эту группу хранилищ.

На первый взгляд подобное упущение может показаться досадным, но определение пользовательского LINQ-запроса в качестве надстройки над API отражения среды .NET, который обычно проще написать, обеспечивает более высокую степень гибкости и освобождает от необходимости изучения другого API, при условии, что вы знакомы с LINQ и API отражения среды .NET. Соответствующее соглашение с использованием LINQ-запроса показано в листинге 12.10.

Каждый из классов, прошедших через фильтры `where` в ходе последовательного перебора, должен быть зарегистрирован за своим интерфейсом. Например, поскольку интерфейсом для `SqlProductRepository` служит `IProductRepository`, должно получиться отображение `IProductRepository` на `SqlProductRepository`.

Листинг 12.10. Соглашение для сканирования хранилищ с помощью Simple Injector

```
var assembly =
    typeof(SqlProductRepository).Assembly;

var repositoryTypes =
    from type in assembly.GetTypes()
    where !type.Abstract
    where type.Name.EndsWith("Repository")
    select type;

foreach (Type type in repositoryTypes)
{
    container.Register(
        type.GetInterfaces().Single(), type);
}
```

Выбор сборки для соглашения

Определение LINQ-запроса на выявление всех типов в сборке, отвечающих критерию конкретности и оканчивающихся на слово Repository

Последовательный перебор результата LINQ-запроса с целью регистрации каждого типа

По этому конкретному соглашению сканируется сборка, содержащая компоненты доступа к данным. Ссылку на эту сборку можно получить множеством способов, но самый простой будет заключаться в выборе репрезентативного типа, такого как `SqlProductRepository`, и получении из него сборки в соответствии с тем, что показано в листинге 12.10. Можно также выбрать другой класс или найти сборку по имени.

ПРИМЕЧАНИЕ

При использовании контейнера Microsoft.Extensions.DependencyInjection код соглашения из листинга 12.10 будет практически идентичным. Другим будет только тело цикла foreach, поскольку только там будет вызываться API DI-контейнера.

Если сравнивать это соглашение с четырьмя регистрациями, показанными в листинге 12.9, можно подумать, что преимущества от данного соглашения весьма незначительны. Так и есть: поскольку в текущем примере всего четыре компонента доступа к данным, то за счет соглашения объем инструкций в коде возрастет. Но масштабируется это соглашение намного лучше. Как только оно будет создано, можно будет без всяких дополнительных усилий справиться с сотнями компонентов.

Под соглашения можно подвести и другие отображения из листингов 12.6 и 12.8, но нам это не принесет особой пользы. В качестве примера: со следующим соглашением можно зарегистрировать все сервисы:

```
var assembly = typeof(ProductService).Assembly;

var serviceTypes =
    from type in assembly.GetTypes()
    where !type.Abstract
    where type.Name.EndsWith("Service")
    select type;

foreach (Type type in serviceTypes)
{
    container.Register(type.GetInterfaces().Single(), type);
}
```

По этому соглашению идентифицированная сборка сканируется на наличие конкретных классов, имена которых заканчиваются на `Service`, и каждый тип закрепляется за реализуемым им интерфейсом. Получается, что `ProductService` закрепляется за интерфейсом `IProductService`, но, поскольку в данный момент других совпадений с этим соглашением нет, особой пользы это не приносит. Как показано в листинге 12.9, приступать к формулировке соглашения имеет смысл только при добавлении дополнительных сервисов.

Самостоятельное определение соглашений с использованием LINQ имеет смысл для типов, выведенных из собственного интерфейса, в чем ранее можно было убедиться на примере хранилищ. Но, когда начинается регистрация типов, основанных на обобщенном интерфейсе, подробно рассмотренном в подразделе 10.3.3, эта стратегия начинает довольно быстро рассыпаться — запрос обобщенных типов путем отражения ни к чему хорошему не приводит[1].

Именно поэтому API автоматической регистрации в Simple Injector построен вокруг регистрации типов на основе обобщенной абстракции, такой как интерфейс `ICommandService<TCommand>` из листинга 10.12. Simple Injector позволяет регистрировать все реализации `ICommandService<TCommand>` в одной строчке кода (листинг 12.11).

Листинг 12.11. Автоматическая регистрация реализаций на основе обобщенной абстракции

```
Assembly assembly = typeof(AdjustInventoryService).Assembly;

container.Register(typeof(ICommandService<>), assembly);
```

ПРИМЕЧАНИЕ

ICommandService<> является синтаксисом C# для определения открытой обобщенной версии, что достигается пропуском обобщенного аргумента типа TCommand.

Предоставляя список сборок одному из своих переопределений метода `Register`, контейнер Simple Injector выполняет последовательный перебор этих сборок с целью поиска необобщенных, конкретных типов, реализующих `ICommandService<TCommand>`, закрепляя каждый тип за конкретным интерфейсом `ICommandService<TCommand>`. Обобщенный аргумент типа `TCommand` заполняется фактическим типом.

ОПРЕДЕЛЕНИЕ

Обобщенный тип, в котором обобщенные аргументы типа заполнены (например, ICommandService<AdjustInventory>), называется закрытым обобщением (closed generic). По аналогии с этим при наличии только самого определения обобщенного типа (например, ICommandService<TCommand>) такой тип называется открытым обобщением.

---

[1] Приходится искать реализации обобщенного интерфейса, рассматривать типы, реализующие несколько интерфейсов, регистрировать все декораторы для всех реализаций и т. д.

В приложении с четырьмя реализациями `ICommandService<TCommand>` предыдущий вызов API будет эквивалентен конфигурации в виде кода, показанной в листинге 12.12.

Листинг 12.12. Зарегистрированные реализации с использованием конфигурации в виде кода

```
container.Register(typeof(ICommandService<AdjustInventory>),
    typeof(AdjustInventoryService));
container.Register(typeof(ICommandService<UpdateProductReviewTotals>),
    typeof(UpdateProductReviewTotalsService));
container.Register(typeof(ICommandService<UpdateHasDiscountsApplied>),
    typeof(UpdateHasDiscountsAppliedService));
container.Register(typeof(ICommandService<UpdateHasTierPricesProperty>),
    typeof(UpdateHasTierPricesPropertyService));
```

Но последовательный перебор списка сборок для поиска подходящих типов — не единственное, что можно получить из API автоматической регистрации контейнера Simple Injector. Еще одним эффективным свойством является регистрация обобщенных декораторов, подобных показанному в листингах 10.15, 10.16 и 10.19. Вместо самостоятельного выстраивания иерархии декораторов, как это делалось в листинге 10.21, Simple Injector позволяет декораторам применяться с помощью переопределений его метода `RegisterDecorator` (листинг 12.13).

Листинг 12.13. Регистрация обобщенных декораторов с использованием автоматической регистрации

```
container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(AuditingCommandServiceDecorator<>));

container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(TransactionCommandServiceDecorator<>));

container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(SecureCommandServiceDecorator<>));
```

RegisterDecorator предоставляется с открытым обобщенным типом сервиса ICommandService<TCommand> и открытой обобщенной реализацией для декоратора. Используя эту информацию, Simple Injector заключает каждый тип ICommandService<TCommand>, который он же разрешает в подходящий декоратор

Simple Injector применяет декораторы в порядке регистрации, следовательно, в отношении кода листинга 12.13 декоратор ведения регистрационного журнала заключается в декоратор транзакции, который, в свою очередь, заключается в декоратор обеспечения безопасности, в результате чего получается граф объектов, идентичный показанному в листинге 10.21.

Регистрация открытых обобщенных типов может рассматриваться как разновидность автоматической регистрации, поскольку всего один вызов метода `RegisterDecorator` может привести к тому, что декоратор будет применен к множеству регистраций[1].

[1] На фоне этого происходит множество событий. Например, если декоратор содержит ограничения обобщенного типа, Simple Injector автоматически определяет применимость декоратора к заданной регистрации на основе этих ограничений типа. Если бы это делалось вручную, занятие стало бы утомительным и не застрахованным от ошибок.

Без этой формы автоматической регистрации для обобщенных классов декораторов вам пришлось бы, как показано в листинге 12.14, индивидуально регистрировать каждую закрытую версию каждого декоратора для всех закрытых реализаций `ICommandService<TCommand>`.

Листинг 12.14. Регистрация обобщенных декораторов с использованием конфигурации в виде кода

```
container.RegisterDecorator(
    typeof(ICommandService<AdjustInventory>),
    typeof(AuditingCommandServiceDecorator<AdjustInventory>));
container.RegisterDecorator(
    typeof(ICommandService<AdjustInventory>),
    typeof(TransactionCommandServiceDecorator<AdjustInventory>));

container.RegisterDecorator(
    typeof(ICommandService<AdjustInventory>),
    typeof(SecureCommandServiceDecorator<AdjustInventory>));
container.RegisterDecorator(
    typeof(ICommandService<UpdateProductReviewTotals>),
    typeof(AuditingCommandServiceDecorator<UpdateProductReviewTotals>));
container.RegisterDecorator(
    typeof(ICommandService<UpdateProductReviewTotals>),
    typeof(TransactionCommandServiceDecorator<UpdateProductReviewTotals>));
container.RegisterDecorator(
    typeof(ICommandService<UpdateProductReviewTotals>),
    typeof(SecureCommandServiceDecorator<UpdateProductReviewTotals>));

container.RegisterDecorator(
    typeof(ICommandService<UpdateHasDiscountsApplied>),
    typeof(AuditingCommandServiceDecorator<UpdateHasDiscountsApplied>));
container.RegisterDecorator(
    typeof(ICommandService<UpdateHasDiscountsApplied>),
    typeof(TransactionCommandServiceDecorator<UpdateHasDiscountsApplied>));
container.RegisterDecorator(
    typeof(ICommandService<UpdateHasDiscountsApplied>),
    typeof(SecureCommandServiceDecorator<UpdateHasDiscountsApplied>));
...
```

Для краткости остальные регистрации здесь не приводятся

Код в этом листинге слишком громоздок и не застрахован от ошибок. Кроме того, он может стать причиной разрастания корня композиции в геометрической прогрессии.

СОВЕТ

Наиболее существенным недостатком автоматической регистрации является частичная потеря контроля. Нужно, чтобы была возможность автоматического связывания каждого компонента, выбранного средством автоматической регистрации. Когда есть конкретный компонент, требующий ручного связывания, он должен быть исключен из автоматической регистрации во избежание ошибок.

В системе, соблюдающей SOLID-принципы, создается множество небольших узкоспециализированных классов, но существующие классы не слишком сильно изменяются, что упрощает их сопровождение. Автоматическая регистрация избавляет корень композиции от постоянных обновлений. Эта эффективная технология может сделать DI-контейнер невидимым. После создания подходящих соглашений изменение конфигурации контейнера может понадобиться в самых крайних случаях.

12.2.4. Смешивание и сопоставление подходов к конфигурированию

До сих пор были показаны три различных подхода к конфигурированию DI-контейнера:

- файлы конфигурации;
- конфигурация в виде кода;
- автоматическая регистрация.

Ни один из них не исключает применения любого из остальных подходов. Можно выбрать смешивание автоматической регистрации с определенными отображениями абстрактных типов на конкретные типы и даже смешивание всех трех подходов, чтобы получилась отчасти автоматическая регистрация, отчасти конфигурация в виде кода и отчасти конфигурация, помещенная в файлы для получения позднего связывания.

Нужно взять за правило сначала выбирать автоматическую регистрацию, дополненную конфигурацией в виде кода для особых случаев. Файлы конфигурации нужно попридержать для случаев, когда понадобится возможность изменения реализации без перекомпиляции приложения, что случается намного реже, чем вы себе можете представить.

Теперь, после изучения способов конфигурирования DI-контейнера и разрешения графов объектов с помощью одного из них, нужно получить представление о том, как все это использовать. Знать, как применять DI-контейнер, — это одно, но понимать, когда это следует делать, — совершенно другое.

12.3. Когда следует использовать DI-контейнер

В предыдущих частях книги в качестве метода построения композиции объектов применялась исключительно чистая технология DI. Это делалось не только в учебных целях. С применением только чистой технологии DI можно создавать целые приложения.

В разделе 12.2 речь шла о различных методах конфигурирования DI-контейнеров и о том, как автоматическая регистрация может упростить сопровождение корня композиции. Но применение DI-контейнеров влечет за собой дополнительные расходы, к тому же приходится мириться с недостатками, которых нет при чистой

технологии DI. DI-контейнеры, за редким исключением, имеют открытый код, то есть распространяются бесплатно. Но поскольку время работы программистов — наиболее дорогостоящая часть разработки программных продуктов, все, что увеличивает время на разработку и сопровождение программных продуктов, относится к издержкам, о чем мы здесь и поговорим.

В этом разделе мы сопоставим все преимущества и недостатки, чтобы можно было принять взвешенное решение о том, когда следует применять DI-контейнер, а когда — остановить свой выбор на чистой технологии DI. Начнем с аспекта, который часто упускают из виду, задействуя такие библиотеки, как DI-контейнеры, — с сопутствующих издержек и рисков.

12.3.1. Использование сторонних библиотек сопряжено с издержками и рисками

Когда библиотека бесплатная, разработчики зачастую игнорируют прочие издержки, связанные с ее применением. DI-контейнер может рассматриваться как стабильная зависимость (см. подраздел 1.3.1), поэтому с позиции внедрения зависимостей его использование не составляет никаких проблем. Но есть и другие опасения, требующие изучения. Работа с DI-контейнером, как и с другой сторонней библиотекой, сопряжена с издержками и рисками.

Самой очевидной издержкой, связанной с любой библиотекой, является ее освоение: чтобы научиться пользоваться новой библиотекой, нужно время. Приходится изучать ее API, поведение, причуды и ограничения. Когда работа ведется в команде, большинству ее членов придется разобраться в том, как работать с библиотекой в разных ситуациях. Если о том, как с ней обращаться, будет знать кто-то один, на первых порах это позволит сэкономить время, но такая практика станет помехой при долговременном ведении проекта[1].

Поведение библиотеки с ее странностями и ограничениями может не в полной мере соответствовать вашим потребностям. Библиотека может быть сориентирована на модель, отличную от той, на которой построен ваш программный продукт[2]. Обычно это обнаруживается при изучении способов ее использования. После применения ее к вашей кодовой базе может выясниться, что требуется реализация различных обходных путей. Иногда оказывается, что овчинка не стоит выделки.

Оценить экономию от использования в проекте новой библиотеки в денежном выражении будет довольно трудно, поскольку затраты на ее изучение зачастую не поддаются реальному определению. Совокупное время, потраченное на изучение API сторонней библиотеки, не относится к времени, потраченному на создание самого приложения, а следовательно, входит в реальные издержки.

---

[1] Это часто называют фактором автобуса. Под ним понимается минимальное число представителей команды, внезапно покидающих проект (сбитых автобусом), прежде чем он застопорится из-за недостатка знающего или компетентного персонала.

[2] Позиция создателя библиотеки может позволить вам и вашей команде научиться чему-либо, но она запросто может противоречить вашему мнению.

Помимо непосредственных издержек на освоение работы с библиотекой, существуют также риски, связанные с зависимостью от нее. Один из них связан с приостановкой ее сопровождения разработчиками[1]. Это накладывает на проект дополнительные издержки, поскольку может заставить перейти на использование других библиотек. В таком случае вновь потребуются рассмотренные ранее расходы плюс издержки на повторную миграцию и тестирование приложения.

СОВЕТ

В силу перечисленных издержек и рисков библиотеки для проекта нужно выбирать с особой осмотрительностью. Чтобы снизить риски при запуске нового проекта, стоит ограничить количество внешних библиотек, с которыми следует ознакомиться вашей команде.

Все это может прозвучать как аргумент против применения внешних библиотек, но это не так. Без них вы вряд ли могли бы работать продуктивно, поскольку пришлось бы заново изобретать колесо. Если отказаться от внешних библиотек, такие же библиотеки придется создавать самостоятельно, что зачастую способно ухудшить ситуацию. (И нам, разработчикам, свойственно недооценивать время, необходимое для написания, тестирования и сопровождения подобного программного продукта.)

Но в случае с DI-контейнерами ситуация несколько иная. Дело в том, что альтернативой использованию библиотеки DI-контейнера является не создание собственной библиотеки, а применение чистой технологии DI.

Не стоит создавать собственный DI-контейнер

При изучении кода листинга 12.1 может показаться, что DI-контейнер можно написать в несколько строк кода. Хотя в листинге 12.1 показаны лишь первые шаги написания DI-контейнера, есть веская причина, по которой он помечен как плохой код.

Код в листинге 12.1 является простейшей реализацией, в которой, как говорилось ранее, отсутствует множество критически важных возможностей. Полноценный DI-контейнер должен поддерживать управление временем жизни объектов, перехват, автоматическую регистрацию и средства обнаружения зацикленности зависимостей, четко сообщать об ошибках конфигурации, иметь правильно спроектированные точки расширения, опираться на качественно разработанную документацию и уметь делать многое другое. С такой работой за пару недель не справиться.

[1] Этот риск связан не только со сторонними библиотеками. Даже у Microsoft, одной из организаций, долго поддерживающих свои технологии, случаются нарушение совместимости и отказ от технологий. В качестве примеров можно привести Workflow Foundation, Silverlight, Visual Studio LightSwitch, Windows Phone и Windows RT. Сейчас, похоже, Microsoft вновь стремится оказывать долгосрочную поддержку. Однако ни в чем и никогда не нужно быть абсолютно уверенными.

Исходя из собственного опыта, я (рассказывает Стивен) скажу, что могут пройти годы, прежде чем такая библиотека станет стабильной и отработанной. Хотя для вас это может стать отличным способом повышения своего мастерства как разработчика, это не поможет вашему проекту или вашей компании, поскольку вы должны быть сконцентрированы на производстве коммерческих ценностей[1].

Но это не означает, что вам вообще не следует создавать новые библиотеки с открытым кодом, такие как DI-контейнер. Инновации являются важным аспектом нашей индустрии, и создание новых библиотек способствует ее развитию. Иногда нужны совершенно новые идеи, порой это означает, что следует создавать новые библиотеки и среды выполнения, основанные на этих идеях. Но все же вы должны быть осмотрительны, тратя средства вашего нанимателя на эти цели, поскольку расходы превысят первоначальные предположения.

Как говорилось в разделе 4.1, взаимодействие с DI-контейнером должно ограничиваться корнем композиции. Это снизит риск его вынужденной замены. Но даже в этом случае на замену DI-контейнера и ознакомление с новым API и философией конструкции может уйти много времени.

Основным преимуществом применения чистой технологии DI является простота изучения. Вам не нужно осваивать API какого-либо DI-контейнера, и хотя отдельные классы по-прежнему задействуют DI, как только будет определен корень композиции, станет понятно, что происходит и как строятся графы объектов. Несмотря на то что новые среды IDE сглаживают эту проблему, новому разработчику в команде может оказаться сложно разобраться в построенном графе объектов и найти реализацию зависимости класса при использовании DI-контейнера.

С применением чистой технологии DI проблем становится меньше, поскольку конструкция графа объектов жестко закодирована в корне композиции. Помимо простоты освоения, преимуществом чистой технологии DI будет то, что она быстрее реагирует на ошибки, вкравшиеся в вашу композицию объектов. Посмотрим, почему так происходит.

12.3.2. Чистая технология Di позволяет быстрее реагировать на происходящее

Такие технологии DI-контейнеров, как автоматическое связывание и автоматическая регистрация, зависят от использования отражения. Следовательно, в ходе выполнения программы DI-контейнер будет анализировать аргументы конструктора, задействуя отражение или даже анализируя готовые сборки, чтобы найти типы на основе соглашений для построения полного графа объектов. Следовательно, ошибки

[1] Создание и сопровождение Simple Injector, а также поддержка сообщества пользователей позволили мне приобрести все те знания, которые со временем сделали меня соавтором этой книги.

конфигурации обнаруживаются только во время выполнения, когда происходит разрешение графа объектов. По сравнению с чистой DI-технологией DI-контейнер выполняет роль компилятора по проверке кода.

ВНИМАНИЕ

> Чистая технология DI имеет существенное преимущество, которое зачастую упускают из виду: она является строго типизированной. Это позволяет компилятору выдавать реакцию на корректность кода, причем ни с чем не сравнимую по быстроте.

При правильном структурировании корня композиции, в котором создание экземпляров с жизненными циклами Singleton и Scoped выполняется раздельно (см., к примеру, листинги 8.10 и 8.13), у компилятора появляется возможность обнаружить захваченные зависимости, рассмотренная в подразделе 8.4.1.

Как говорилось в подразделе 3.2.2, по причине строгой типизации у чистой технологии DI имеется и еще одно преимущество: она рисует четкую картину относительно структуры графа объектов приложения. С началом применения DI-контейнера все это тут же теряется.

Но строгая типизация имеет как положительные, так и отрицательные стороны, поскольку, как говорилось в подразделе 12.1.3, она также означает, что при каждой переделке конструктора нарушается корень композиции. Если есть библиотека (доменной модели, утилиты, компонента доступа к данным и т. д.), совместно применяемая несколькими приложениями, может понадобиться сопровождение сразу нескольких корней композиции. Насколько это обременительно, зависит от частоты переделки конструкторов, но нам попадались проекты, где такое случалось по нескольку раз за день. Когда над одним проектом трудятся несколько разработчиков, это запросто может приводить к конфликтам слияния, на устранение которых требуется время.

Хотя при использовании чистой технологии DI компилятор реагирует быстро, объем доступных ему проверок ограничен. Он может отчитаться о зависимостях, утраченных из-за изменений, внесенных в конструкторы, и о некоторых распространенных захваченных зависимостях, но, помимо всего остального, он не сможет определить:

- сбой в вызове конструкторов из-за выдачи исключений из тела конструктора (например, при срабатывании граничных операторов);
- ликвидируются ли ликвидируемые компоненты, когда выходят из области видимости;
- повторное (случайное) создание классов с предопределенным жизненным циклом Singleton или Scoped в другой части корня композиции, возможно, с другим жизненным циклом[1].

---

[1] Эти дефекты часто называют разорванными жизненными циклами (Torn Lifestyles, https://simpleinjector.org/diatl) и двусмысленными жизненными циклами (Ambiguous Lifestyles, https://simpleinjector.org/diaal).

При использовании чистой технологии DI размер корня композиции растет линейно по отношению к размеру приложения. Когда приложение невелико, его корень композиции также будет небольшим. Это сделает корень композиции ясным и простым в сопровождении, а ранее перечисленные дефекты будет проще заметить. Но с ростом корня композиции увеличивается и вероятность пропустить их.

Сгладить эту проблему может DI-контейнер. Большинство DI-контейнеров автоматически выявляют ликвидируемые компоненты и способны обнаруживать самые распространенные опасности наподобие захваченных зависимостей[1].

12.3.3. Вердикт: когда следует использовать DI-контейнер

Если воспользоваться имеющейся в DI-контейнере конфигурацией в виде кода (рассмотренной в подразделе 12.2.2), явно регистрирующей буквально каждый компонент, применяющий API контейнера, будет утрачена быстрая реакция, свойственная строгой типизации. В то же время, скорее всего, благодаря автоматическому связыванию сократятся издержки на сопровождение. Но при введении каждого нового класса потребуется регистрация, являющаяся линейным расширением, а вам придется изучать конкретный API используемого контейнера. Но, даже если вы уже знакомы с этим API, остается риск его замены в дальнейшем. При этом можно больше потерять, чем приобрести.

По сути, если умело обращаться с DI-контейнером, им можно воспользоваться для определения набора соглашений при автоматической регистрации (в соответствии с изложенным в подразделе 12.2.3). Этими соглашениями устанавливается набор правил, которые следует соблюдать при написании кода, и пока это делается, код сохраняет работоспособность. Контейнер уходит на второй план, и необходимость обращаться к нему возникает крайне редко.

ВНИМАНИЕ

> Использование соглашения по конфигурации с помощью автоматической регистрации может свести к минимуму — почти до нуля — объем работ по сопровождению корня композиции.

На освоение автоматической регистрации требуется время, и она слабо типизируема, но, если все сделать правильно, она позволяет сконцентрироваться на полезном для практики коде, а не на инфраструктуре. Дополнительный плюс заключается в создании механизма положительной обратной связи, вынуждающего команду писать код, соответствующий соглашениям. Компромисс между чистой технологией DI и использованием DI-контейнера представлен на рис. 12.6.

---

[1] Все три DI-контейнера, рассматриваемые в этой книге, способны в той или иной степени обнаруживать захваченные зависимости.

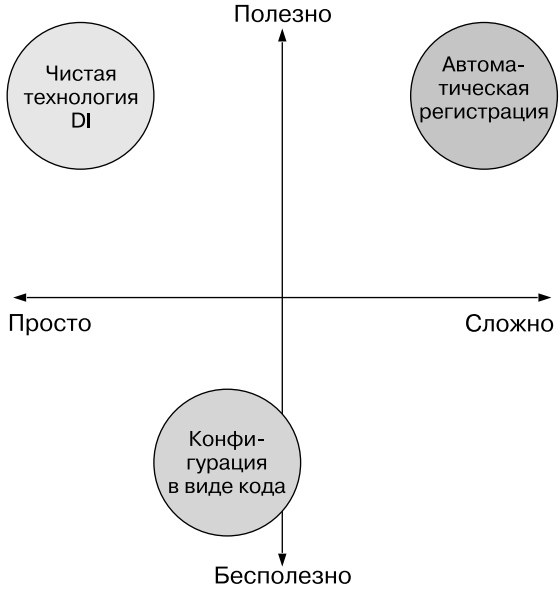


Рис. 12.6. Чистая технология DI ценна своей простотой, а DI-контейнер может быть либо полезен, либо нет в зависимости от способа применения. При умелом использовании (с автоматической регистрацией) DI-контейнер может стать средством, обеспечивающим наилучшее соотношение пользы и затрат

Из подраздела 12.2.4 известно, что ни один из доступных подходов не исключает применения наряду с ним каких-либо других подходов. Можно найти такой корень композиции, в котором сочетаются все стили конфигурации, но он должен быть сориентирован либо на чистую технологию DI, возможно, с несколькими типами с поздней привязкой, либо на автоматическую регистрацию с дополнительным лимитированным объемом конфигурации в виде кода, чистой технологии DI и файлов конфигурации. Применять корень композиции, сориентированный на использование конфигурации в виде кода, не имеет смысла, и этого нужно избегать.

Возникает вопрос: когда следует выбирать чистую технологию DI, а когда воспользоваться автоматической регистрацией? К сожалению, конкретного ответа на него у нас нет. Все зависит от величины проекта, индивидуального и командного опыта работы с DI-контейнером и осознаваемой степени риска.

Но в целом следует применять чистую технологию DI для небольшого корня композиции и переходить к автоматической регистрации, как только станут возникать проблемы с его сопровождением. Более крупные приложения, имеющие множество классов, подпадающих под несколько соглашений, могут извлечь вполне определенную выгоду из автоматической регистрации[1].

---

[1] Мы также склонны продвигать идею сравнительно небольших приложений. Это неизбежно приводит к понятию ограниченных контекстов (Bounded Contexts). См. книгу: *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011. — С. 298.

Автомагия

Я (рассказывает Марк) работал однажды на клиента, для которого применял в кодовой базе автоматическую регистрацию, основанную на соглашениях. Другие разработчики были от этого не в восторге, поскольку сочли данный подход излишне автоматизированным (автомагическим). Они освоили DI и использовали разработку на основе тестирования (TDD), но не были заинтересованы в применении DI-контейнера, поскольку не знали его API.

Зачастую соглашения работали как следует. Когда разработчики вводили новые классы или интерфейсы, DI-контейнер обнаруживал новые типы и корректно выполнял их конфигурирование. Но временами разработчики, и я в том числе, при реализации функции отходили от соглашений. В таких случаях соглашения нуждались в корректировке.

Другие разработчики не понимали, как работать с API DI-контейнера, и не стремились научиться, поэтому реализация необходимых изменений ложилась на мои плечи. Я превратился в некий критический ресурс, а иногда и в узкое место для всего проекта. Покидая проект, я предполагал, что команда откажется от DI-контейнера и заменит его чистой технологией DI. Когда через год я вернулся, то не удивился тому, что именно так они и сделали. И не могу сказать, что я их осудил за это.

Мы не можем подсказать, какой именно DI-контейнер следует выбрать. Выбор контейнера подразумевает его оценку не только с чисто технической стороны. Нужно брать в расчет также приемлемость модели лицензирования, степень доверия сотрудникам или организациям, разрабатывающим и сопровождающим DI-контейнер, то, как этот контейнер вписывается в IT-стратегию вашей организации, и т. д. Поиск подходящего DI-контейнера не должен ограничиваться контейнерами, перечисленными в этой книге. В свободном выборе имеется множество превосходных DI-контейнеров, к примеру, для платформы .NET.

При грамотном использовании DI-контейнер может стать очень полезным средством. Важнее всего понять, что применение DI никоим образом не зависит от DI-контейнера. Приложение может быть создано из многих классов и модулей со слабой связанностью, и ни один из этих модулей не будет даже подозревать о существовании контейнера. Наиболее эффективный способ оставить код приложения в неведении о каком-либо DI-контейнере заключается в том, чтобы использовать последний только в корне композиции. Тем самым вы избежите случайного применения антишаблона «Локатор сервисов», поскольку контейнер будет задействован лишь в небольшой изолированной области кода.

Используемый таким образом DI-контейнер становится механизмом, занимающимся частью архитектуры приложения. Он составляет графы объектов на основе своей конфигурации. Особый эффект может быть получен от применения соглашения по конфигурации. При соответствующей реализации контейнер может взять на себя составление графов объектов, позволяя вам сосредоточиться на создании новых функций. Контейнер будет автоматически обнаруживать новые классы,

соответствующие установленным соглашениям, и открывать к ним доступ для потребителей. В трех заключительных главах книги рассматриваются DI-контейнеры Autofac (глава 13), Simple Injector (глава 14) и Microsoft.Extensions.DependencyInjection (глава 15).

Резюме

- DI-контейнер представляет собой библиотеку, обеспечивающую функциональные возможности внедрения зависимостей. Это механизм, разрешающий графы объектов и управляющий ими.
- Технология DI ни в коей мере не зависит от использования DI-контейнера, который является полезным, но дополнительным инструментом.
- Автоматическое связывание обеспечивает возможность автоматического составления графа объектов из отображений между абстракциями и конкретными типами с помощью информации о типе, предоставленной компилятором и общеязыковой средой выполнения (CLR).
- Внедрение через конструктор приводит к статическому объявлению требования классом зависимостей, а DI-контейнеры используют эту информацию для автоматического связывания сложных графов объектов.
- Автоматическое связывание повышает устойчивость корня композиции к изменениям.
- Приступая к работе с DI-контейнером, не нужно полностью отказываться от ручного составления графов объектов. Когда удобнее применять ручное связывание, оно может стать частью процесса конфигурирования.
- При использовании DI-контейнера применяются три стиля конфигурирования: файлы конфигурации, конфигурация в виде кода и автоматическая регистрация.
- Файлы конфигурации являются такой же частью корня композиции, как и конфигурация в виде кода и автоматическая регистрация. Поэтому применение файлов конфигурации не приводит к уменьшению корня композиции — он всего лишь перемещается.
- По мере увеличения размера и сложности приложения увеличивается и файл конфигурации. Хрупкость и непрозрачность в сфере выявления ошибок, свойственные файлам конфигурации, имеют тенденцию усугубляться, поэтому данный подход к конфигурированию следует использовать только при необходимости в позднем связывании.
- Не позволяйте отсутствию поддержки работы с файлами конфигурации влиять на выбор DI-контейнера. Для загрузки типов из файлов конфигурации достаточно всего нескольких простых инструкций.
- Конфигурация в виде кода позволяет сохранять конфигурацию контейнера в исходном коде. Каждое отображение между абстракцией и конкретной реализацией выражается в явном виде и непосредственно в коде. Этому методу следует от-

давать предпочтение перед применением файлов конфигурации, если только вам не нужно получить позднее связывание.

- Соглашение по конфигурации представляет собой применение соглашений к коду с целью упрощения регистрации.
- Автоматическая регистрация представляет собой автоматизацию регистрации компонентов в контейнере путем сканирования одной или большего количества сборок на наличие реализаций нужных абстракций, что является формой соглашения по конфигурации.
- Автоматическая регистрация помогает избежать постоянного обновления корня композиции, поэтому она предпочтительнее конфигурации в виде кода.
- Использование таких внешних библиотек, как DI-контейнеры, приводит к издержкам и рискам, например к издержкам, связанным с освоением нового API, и к риску заброшенности библиотеки.
- Не стоит разрабатывать собственный DI-контейнер. Либо воспользуйтесь одним из существующих, надежно протестированных и находящихся в свободном доступе DI-контейнеров, либо довольствуйтесь применением чистой технологии DI. Создание и сопровождение соответствующей библиотеки требует немалых усилий, не направленных на получение прибыли.
- Большим преимуществом чистой технологии DI является строгая типизация. Она позволяет компилятору живо реагировать на все нарушения, получая наиболее быстрый отклик.
- Чистую технологию DI можно применять для корня композиции небольшого объема, а когда сопровождение таких корней композиции станет проблематичным, переключаться на автоматическую регистрацию. Более крупные приложения с большим количеством классов, подпадающих под несколько соглашений, могут извлечь из автоматической регистрации весьма существенную выгоду.

13 DI-контейнер Autofac

В этой главе

- Работа с основным API регистрации Autofac.
- Управление временем жизни компонентов.
- Конфигурирование сложных API.
- Конфигурирование последовательностей, декораторов и компоновщиков.

В предыдущих главах рассматривались общие паттерны и принципы, применимые к технологии DI, но нам только предстоит подробно изучить порядок их задействования при использовании конкретного DI-контейнера и привести нескольких примеров. В этой главе будет показано, как общие паттерны проецируются на Autofac. Чтобы извлечь из этого практическую выгоду, нужно освоить материал предыдущих глав.

Autofac — полноценный DI-контейнер, предлагающий тщательно спроектированный и надежный API. Он появился в конце 2007 года и на момент написания книги был самым популярным контейнером[1].

[1] Официальная статистика по использованию DI-контейнеров не ведется, поэтому оценка основана на среднем количестве NuGet-загрузок в день.

В этой главе будет рассмотрен способ использования Autofac с целью применения принципов и паттернов, представленных в частях I–IV. Глава разбита на четыре раздела. Можно изучать их по отдельности, но прочитать раздел 13.1 обязательно, чтобы понять все остальные разделы, а в разделе 13.4 есть ссылки на некоторые методы и классы, рассмотренные в разделе 13.3.

Эта глава должна помочь приступить к работе с Autofac, а также разобраться с проблемами, чаще всего встречающимися при ежедневном использовании этого контейнера. Она не является полным курсом по Autofac — для этого понадобилось бы еще несколько глав, а то и целая книга. Более подробное знакомство с Autofac лучше всего начать с домашней страницы по адресу https://autofac.org.

13.1. Введение в Autofac

В этом разделе расскажем, где можно получить Autofac, что именно будет получено и как приступить к работе с контейнером. Кроме этого, дадим общие сведения о его конфигурации. Основная информация, которая пригодится для начала работы с контейнером, приведена в табл. 13.1.

Таблица 13.1. Краткое знакомство с Autofac

Вопрос	**Ответ**
Где его получить?	Из среды Visual Studio его можно получить через NuGet. Пакет называется Autofac. Кроме этого, NuGet-пакет может быть загружен из хранилища GitHub (https://github.com/autofac/Autofac/releases)
Какие платформы поддерживаются?	.NET 4.5 (без .NET Core SDK) и .NET Standard 1.1 (.NET Core 1.0, Mono 4.6, Xamarin.iOS 10.0, Xamarin.Mac 3.0, Xamarin.Android 7.0, UWP 10.0, Windows 8.0, Windows Phone 8.1). Старые сборки, поддерживающие .NET 2.0 и Silverlight, доступны через историю NuGet
Сколько он стоит?	Нисколько. Это средство с открытым кодом
Какая у него лицензия?	MIT License
Где можно получить помощь?	Коммерческую поддержку можно получить от компаний, связанных с разработчиками Autofac. Варианты по адресу https://autofac.readthedocs.io/en/latest/support.html. Что же касается некоммерческой поддержки, то Autofac по-прежнему является программным продуктом с открытым кодом с разрастающейся экосистемой, поэтому вполне возможно (но не гарантировано) получить помощь от публикаций в Stack Overflow по адресу https://stackoverflow.com или на официальном форуме https://groups.google.com/group/autofac
На какой версии основан материал данной главы?	4.9.0-beta1

Использование Autofac не особо отличается от применения других DI-контейнеров, рассматриваемых в следующих главах. Использование этого контейнера,

равно как и контейнеров Simple Injector и Microsoft.Extensions.DependencyInjection, является двухэтапным процессом (рис. 13.1). Сначала выполняется конфигурирование `ContainerBuilder`, а затем конфигурация задействуется при создании контейнера для разрешения компонентов.

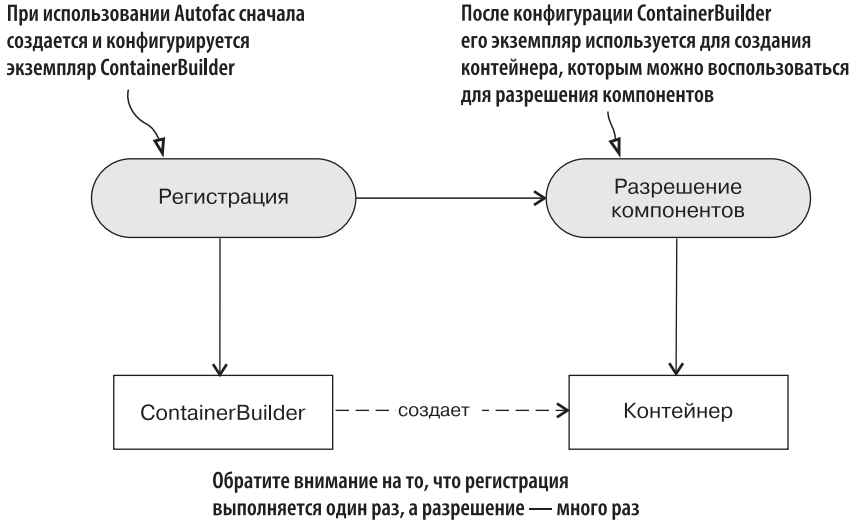


Рис. 13.1. Схема применения Autofac предусматривает сначала конфигурирование контейнера, а затем разрешение компонентов

Усвоив материал этого раздела, вы получите четкое представление об общей схеме работы с Autofac и сможете использовать этот контейнер в сценариях с надлежащим поведением, где все компоненты будут соответствовать корректным DI-паттернам, подобным внедрению через конструктор. Начнем с самого простого сценария и посмотрим, как можно разрешить объекты, применяя контейнер Autofac.

13.1.1. Разрешение объектов

Основной сервис любого DI-контейнера заключается в составлении графов объектов. В этом разделе будет рассмотрен API, позволяющий составить графы объектов с помощью Autofac.

По умолчанию Autofac требует от вас зарегистрировать все соответствующие компоненты, прежде чем появится возможность их разрешения. Но поведение поддается конфигурированию. Один из самых простых способов применения Autofac показан в листинге 13.1.

Листинг 13.1. Самый простой способ использования Autofac

```
var builder = new ContainerBuilder();

builder.RegisterType<SauceBéarnaise>();
```

```
IContainer container = builder.Build();

ILifetimeScope scope = container.BeginLifetimeScope();

SauceBéarnaise sauce = scope.Resolve<SauceBéarnaise>();
```

На рис. 13.1 показано, что для конфигурирования компонентов нужен экземпляр `ContainerBuilder`. Конкретный класс `SauceBéarnaise` регистрируется с помощью `builder`, поэтому, когда запрашивается создание контейнера, получающийся контейнер конфигурируется с классом `SauceBéarnaise`. Именно это и позволяет разрешить класс `SauceBéarnaise` из контейнера.

Но при использовании Autofac разрешение исходит не из самого корневого контейнера, а из области видимости времени жизни (lifetime scope). Более подробно эта область видимости и причины недопустимости разрешения из корневого контейнера рассмотрены в подразделе 13.2.1.

ВНИМАНИЕ

> При работе с Autofac непосредственное разрешение из корневого контейнера считается порочной практикой. Это легко может вызвать утечки памяти или ошибки параллелизма. Разрешение всегда должно выполняться из области видимости времени жизни.

Если компонент `SauceBéarnaise` не зарегистрировать, попытка разрешения приведет к выдаче исключения `ComponentNotRegisteredException` со следующим сообщением: The requested service "Ploeh.Samples.MenuModel.SauceBéarnaise" has not been registered. To avoid this exception, either register a component to provide the service, check for service registration using IsRegistered(), or use the ResolveOptional() method to resolve an optional dependency (Запрошенный сервис Ploeh.Samples.MenuModel.SauceBéarnaise не был зарегистрирован. Чтобы избежать выдачи исключения, нужно либо зарегистрировать компонент для предоставления сервиса, либо проверить регистрацию сервиса с помощью IsRegistered(), либо воспользоваться методом ResolveOptional() для разрешения опциональной зависимости).

Autofac способен не только разрешить конкретные типы с конструкторами без параметров, но и выполнить автоматическое связывание типа с другими зависимостями. Все эти зависимости должны быть зарегистрированы. Программировать при этом стоит в основном с прицелом на использование интерфейсов, поскольку таким образом создается слабое связывание. Для поддержки такого стиля программирования Autofac позволяет отображать абстракции на конкретные типы.

Отображение абстракций на конкретные типы

Поскольку корневые типы вашего приложения обычно разрешаются в их конкретные типы, слабая связанность требует отображения абстракций на конкретные типы. Создание экземпляров на основе таких отображений является основным сервисом, предлагаемым любым DI-контейнером, но отображение все же должно

быть определено. В данном примере интерфейс `IIngredient` отображается на конкретный класс `SauceBéarnaise`, что позволяет успешно выполнить разрешение `IIngredient`:

```
var builder = new ContainerBuilder();

builder.RegisterType<SauceBéarnaise>()
    .As<IIngredient>();                    ← Отображение конкретного типа на абстракцию

IContainer container = builder.Build();

ILifetimeScope scope = container.BeginLifetimeScope();

IIngredient sauce = scope.Resolve<IIngredient>();   ← Разрешение класса SauceBéarnaise
```

Метод `As<T>` дает возможность отобразить конкретный тип на определенную абстракцию. Благодаря предыдущему вызову `As<IIngredient>()` теперь `SauceBéarnaise` может быть разрешен как `IIngredient`.

Экземпляр `ContainerBuilder` используется для регистрации типов и определения отображений. Метод `RegisterType` позволяет регистрировать конкретный тип.

Из кода листинга 13.1 видно, что при желании зарегистрировать только класс `SauceBéarnaise` на этом можно и остановиться. А можно продолжить и применить метод `As` для определения порядка регистрации конкретного типа[1].

ВНИМАНИЕ

> В отличие от Simple Injector и Microsoft.Extensions.DependencyInjection здесь между типами, определенными RegisterType и методами As, ограничения обобщенных типов не действуют. Это позволяет отображать несовместимые типы. Код станет компилироваться, но во время его выполнения, когда ContainerBuilder создаст контейнер, будет выдано исключение.

Во многих случаях ваши потребности сводятся к обобщенному API. Хотя он и не обеспечивает такого же уровня безопасности, как использование некоторых других DI-контейнеров, все же при таком способе конфигурации контейнера код остается разборчивым. Однако бывают ситуации, когда требуется менее типизированный способ разрешения сервисов. Autofac предоставляет и такую возможность.

Разрешение слабо типизированных сервисов

В некоторых случаях применение обобщенного API недопустимо, поскольку во время разработки подходящий тип неизвестен. Вы располагаете лишь экземпляром `Type`, но все же хотите получить экземпляр этого типа. Пример такой ситуации при-

[1] При использовании Autofac все начинается с конкретного типа и его отображения на абстракцию. Большинство других DI-контейнеров делают наоборот: начинают с абстракции и отображают ее на конкретный тип.

веден в разделе 7.3, где рассматривался принадлежащий среде ASP.NET Core MVC класс `IControllerActivator`. Соответствующий метод имел следующий вид:

```
object Create(ControllerContext context);
```

В листинге 7.8 было показано, что `ControllerContext` захватывает `Type` контроллера, который можно извлечь, воспользовавшись свойством `ControllerTypeInfo`, принадлежащим свойству `ActionDescriptor`:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
```

Поскольку имеется только экземпляр `Type`, использовать обобщенный метод `Resolve<T>` нельзя и придется прибегнуть к слабо типизированному API. Autofac предлагает слабо типизированное переопределение метода `Resolve`, позволяющее реализовать метод `Create` следующим образом:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
return scope.Resolve(controllerType);
```

Слабо типизированное переопределение метода `Resolve` позволяет передать переменную `controllerType` непосредственно в Autofac. Как правило, это означает, что возвращаемое значение придется привести к некой абстракции, поскольку слабо типизированный метод `Resolve` возвращает `object`. Но в случае `IControllerActivator` этого не требуется, поскольку среда ASP.NET Core MVC не требует от контроллера реализации интерфейса или базового класса.

Независимо от того, какое именно переопределение `Resolve` используется, Autofac гарантирует, что он возвратит экземпляр запрошенного типа или выдаст исключение, если зависимости не смогут быть удовлетворены. После должного конфигурирования всех нужных зависимостей Autofac может выполнить автоматическое связывание требуемого типа.

В предыдущем примере областью видимости является экземпляр `Autofac.ILifetimeScope`. Чтобы получилось разрешение запрошенного типа, сначала должны быть сконфигурированы все слабосвязанные зависимости. Существует множество способов конфигурирования Autofac, обзор наиболее распространенных из них будет дан в следующем разделе.

13.1.2. Конфигурирование ContainerBuilder

В разделе 12.2 говорилось, что DI-контейнер может быть сконфигурирован несколькими концептуально различными путями. Варианты показаны на рис. 12.5 — это файлы конфигурации, конфигурация в виде кода и автоматическая регистрация. На рис. 13.2 они приведены еще раз.

Базовый API конфигурации сконцентрирован на коде и поддерживает как конфигурацию в виде кода, так и автоматическую регистрацию на основе соглашений. Поддержка файлов конфигурации может быть подключена с помощью NuGet-пакета `Autofac.Configuration`. Autofac поддерживает все три подхода и позволяет смешивать их в пределах одного контейнера. В этом разделе будут показаны способы использования каждого из трех типов источников конфигурации.

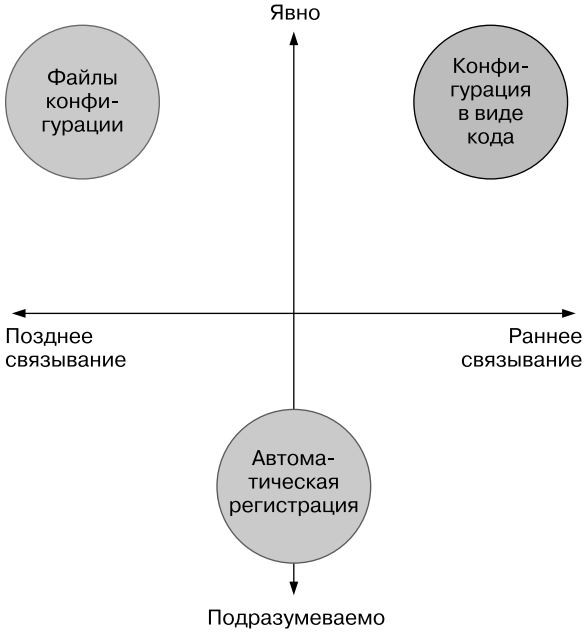


Рис. 13.2. Наиболее распространенные способы конфигурирования DI-контейнеров, расставленные по шкалам их явности-неявности и времени связывания

Конфигурирование ContainerBuilder с помощью конфигурации в виде кода

Краткое знакомство со строго типизированным API конфигурации Autofac состоялось в разделе 13.1. Здесь он будет изучен подробнее.

Все конфигурации в Autofac задействуют API, предоставленный классом `ContainerBuilder`, хотя большинство применяемых методов являются методами расширения. Один из наиболее часто используемых — показанный ранее метод `RegisterType`:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
```

Регистрация `SauceBéarnaise` в качестве `IIngredient` скрывает конкретный класс, поэтому в дальнейшем разрешение `SauceBéarnaise` с этой регистрацией невозможно. Но это легко исправить за счет использования переопределения метода `As`, которое позволяет указать конкретный тип, отображающийся на более чем один зарегистрированный тип:

```
builder.RegisterType<SauceBéarnaise>().As<SauceBéarnaise, IIngredient>();
```

Вместо регистрации класса лишь как `IIngredient` его можно зарегистрировать и как его самого, и как реализуемый им интерфейс. Это позволит контейнеру вы-

полнять разрешение запросов как `SauceBéarnaise`, так и `IIngredient`. В качестве альтернативы вызовы метода `As` можно также выстроить в цепочку:

```
builder.RegisterType<SauceBéarnaise>()
    .As<SauceBéarnaise>()
    .As<IIngredient>();
```

Это даст такой же результат, что и в предыдущем примере. Разница между двумя регистрациями заключается в используемом стиле.

Три обобщенных переопределения метода `As` позволяют указать один, два или три типа. Если нужно указать больше типов, существует необобщенное переопределение, которым можно воспользоваться для указания стольких типов, сколько хочется.

Рваные жизненные циклы

В этом разделе было показано, как можно вызвать `As` несколько раз для регистрации компонента в качестве нескольких типов сервиса. В следующем примере это показано еще раз:

```
builder.RegisterType<SauceBéarnaise>()
    .As<SauceBéarnaise>()
    .As<IIngredient>();
```

Можно подумать, что это эквивалент следующего кода:

```
builder.RegisterType<SauceBéarnaise>();
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
```

Но первый пример не эквивалентен второму. Это становится ясно при замене жизненного цикла Transient, к примеру, на Singleton.

```
builder.RegisterType<SauceBéarnaise>().SingleInstance();
builder.RegisterType<SauceBéarnaise>().As<IIngredient>()
    .SingleInstance();
```

Вопреки ожиданию того, что под время жизни контейнера подпадет только один экземпляр `SauceBéarnaise`, разбиение регистрации заставляет Autofac создать отдельные экземпляры под каждый вызов `RegisterType`. Поэтому жизненный цикл `SauceBéarnaise` должен считаться *рваным*.

ВНИМАНИЕ

Предоставление методу As нескольких типов сервисов не эквивалентно выполнению нескольких вызовов RegisterType. Для каждого вызова будет создаваться его собственная область кэш-памяти, что может привести к разорванности времени жизни при выборе жизненного цикла, отличного от Transient.

В реальных приложениях всегда имеется более одной отображаемой абстракции, поэтому конфигурировать нужно несколько отображений. Это делается несколькими вызовами `RegisterType`:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
builder.RegisterType<Course>().As<ICourse>();
```

В этом примере `IIngredient` отображается на `SauceBéarnaise`, а `ICourse` — на `Course`. Здесь нет перекрытия типов, поэтому происходящее должно быть вполне очевидным. Одну и ту же абстракцию также можно зарегистрировать несколько раз:

```
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
builder.RegisterType<Steak>().As<IIngredient>();
```

Здесь дважды регистрируется `IIngredient`. Если выполняется разрешение `IIngredient`, получается экземпляр `Steak`. Последняя регистрация выигрывает, но предыдущая при этом не забывается. Autofac хорошо справляется с несколькими конфигурациями одной и той же абстракции (вернемся к этой теме в разделе 13.4).

Для конфигурирования Autofac существуют и более совершенные методы, но с помощью показанных здесь можно сконфигурировать целое приложение. Однако, чтобы не тратить слишком много усилий на явное сопровождение конфигурации контейнера, можно обратиться к подходу, более тесно связанному с соглашениями, и воспользоваться автоматической регистрацией.

Конфигурирование ContainerBuilder с помощью автоматической регистрации

Регистрации во многих случаях будут похожи друг на друга. Сопровождение регистраций способно превратиться в весьма утомительное занятие, а явная регистрация буквально каждого компонента, как выяснилось в подразделе 12.3.3, может стать крайне нерациональным подходом.

Рассмотрим библиотеку, содержащую множество реализаций `IIngredient`. Можно сконфигурировать каждый класс по отдельности, но тогда получится множество похожих вызовов метода `RegisterType`. Хуже того, чтобы при каждом добавлении реализация `IIngredient` стала доступной, ее также нужно явно регистрировать с помощью `ContainerBuilder`. Рациональнее было бы заявить, что должны быть зарегистрированы все реализации `IIngredient`, найденные в заданной сборке.

Сделать это можно с помощью метода `RegisterAssemblyTypes`. Он позволяет в одной инструкции указать сборку и сконфигурировать все выбранные из нее классы. Для получения экземпляра `Assembly` можно воспользоваться репрезентативным классом (в данном случае `Steak`):

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

builder.RegisterAssemblyTypes(ingredientsAssembly).As<IIngredient>();
```

Метод `RegisterAssemblyTypes` возвращает такой же интерфейс, как и метод `RegisterType`, поэтому доступны многие такие же варианты конфигурации.

Это очень важная особенность, означающая, что для автоматической регистрации не нужно изучать новый API.

В предыдущем примере для регистрации всех типов в сборке в качестве сервисов `IIngredient` использовался метод `As`. В нем без каких-либо условий конфигурировались все реализации интерфейса `IIngredient`, но вы можете предоставлять фильтры, позволяющие выбрать только один поднабор. Сканирование на основе соглашения, при котором добавляются только классы, имена которых начинаются с `Sauce`, имеет следующий вид:

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

builder.RegisterAssemblyTypes(ingredientsAssembly)
    .Where(type => type.Name.StartsWith("Sauce"))
    .As<IIngredient>();
```

Для определения критерия выбора при регистрации всех типов в сборке можно воспользоваться предикатом. В этом случае единственным отличием от предыдущего примера кода будет включение метода `Where`, выбирающего только типы, имена которых начинаются с `Sauce`.

Существует множество других методов, позволяющих применять различные критерии выбора. Метод `Where` предоставляет фильтр, позволяющий только пропускать типы через предикат соответствия, но есть также метод `Except`, работающий противоположным образом.

Кроме выбора из сборки приемлемых типов, автоматическая регистрация занимается также определением нужного отображения. В предыдущих примерах для регистрации всех выбранных типов за этим интерфейсом использовался метод `As`, для которого этот интерфейс указывался. Но иногда могут потребоваться другие соглашения.

Предположим, вместо интерфейсов применяются классы на основе абстракций и нужно зарегистрировать все имеющиеся в сборке типы, имена которых заканчиваются на `Policy`. Для этого существует несколько других переопределений метода `As`, включая и такое, где в качестве входных данных принимается `Func<Type, Type>`:

```
Assembly policiesAssembly = typeof(DiscountPolicy).Assembly;

builder.RegisterAssemblyTypes(policiesAssembly)
    .Where(type => type.Name.EndsWith("Policy"))
    .As(type => type.BaseType);
```

СОВЕТ

> Рассматривайте RegisterAssemblyTypes в качестве множественного числа RegisterType.

Блок кода, предоставляемый методу `As`, можно использовать для каждого отдельно взятого типа, имя которого заканчивается на `Policy`. Тем самым гарантируется, что все классы с суффиксом `Policy` будут зарегистрированы за их базовым классом

и при условии запроса на разрешение базового класса контейнер станет разрешать его в тип, отображаемый этим соглашением. Регистрация на основе соглашений с применением Autofac на удивление несложна и задействует API, являющийся практически зеркальным отражением API, предоставляемого оригинальным методом `RegisterType`.

Автоматическая регистрация обобщенных абстракций с помощью AsClosedTypesOf

При изучении главы 10 рассматривалась переделка большого и крайне неудачного интерфейса `IProductService` в интерфейс `ICommandService<TCommand>`, показанный в листинге 10.12. Посмотрим на эту абстракцию еще раз:

```
public interface ICommandService<TCommand>
{
    void Execute(TCommand command);
}
```

Как известно из главы 10, каждый граничный объект команды представлял собой вариант использования и для каждого такого варианта, за исключением каких-либо декораторов, реализующих сквозную функциональность, предусматривалась одна реализация. В качестве примера был представлен сервис `AdjustInventoryService` (см. листинг 10.8). В нем был создан вариант использования «корректировка товарных запасов». В листинге 13.2 этот класс показан еще раз.

Листинг 13.2. AdjustInventoryService из главы 10

```
public class AdjustInventoryService : ICommandService<AdjustInventory>
{
    private readonly IInventoryRepository repository;

    public AdjustInventoryService(IInventoryRepository repository)
    {
        this.repository = repository;
    }

    public void Execute(AdjustInventory command)
    {
        var productId = command.ProductId;

        ...
    }
}
```

В любой довольно сложной системе могут реализовываться сотни вариантов использования. Такие системы можно рассматривать в качестве идеальных кандидатов на автоматическую регистрацию. С применением Autofac вряд ли можно придумать что-то еще более простое, чем то, что показано в листинге 13.3.

Листинг 13.3. Автоматическая регистрация реализаций ICommandService<TCommand>

```
Assembly assembly = typeof(AdjustInventoryService).Assembly;

builder.RegisterAssemblyTypes(assembly)
    .AsClosedTypesOf(typeof(ICommandService<>));
```

Как и в предыдущих листингах, для выбора классов из имеющейся сборки используется метод `RegisterAssemblyTypes`. Но вместо метода `As` вызывается метод `AsClosedTypesOf` и предоставляется открытый обобщенный интерфейс `ICommandService<TCommand>`.

С помощью открытого обобщенного интерфейса Autofac выполняет последовательный перебор списка типов сборки и регистрирует все типы, реализующие закрытую обобщенную версию `ICommandService<TCommand>`. Это означает, к примеру, что регистрируется `AdjustInventoryService`, поскольку в нем реализуется `ICommandService<AdjustInventory>`, являющийся закрытой обобщенной версией `ICommandService<TCommand>`.

Метод `RegisterAssemblyTypes` принимает массив `params` экземпляров `Assembly`, поэтому для одного соглашения можно предоставить сколько угодно сборок. Вряд ли можно будет увидеть что-либо странное в сканировании папки на наличие сборок и в предоставлении их всех для реализации дополнительной функциональности. Тогда надстройки можно будет добавлять без перекомпиляции основного приложения. Это один из способов реализации позднего связывания, еще одним способом является использование файлов конфигурации.

Конфигурирование ContainerBuilder с помощью файлов конфигурации

Если нужно изменить регистрации контейнеров без перекомпиляции приложения, то вполне приемлемым вариантом может стать применение файлов конфигурации. Из подраздела 12.2.1 известно, что файлы конфигурации должны задействоваться только для тех типов DI-конфигураций, которые требуют позднего связывания, а для остальных типов и всех прочих частей конфигурации нужно выбирать конфигурацию в виде кода или автоматической регистрации.

Самый естественный способ использования файлов конфигурации — их встраивание в стандартный файл конфигурации .NET-приложения. Но если нужно изменить конфигурацию Autofac независимо от стандартного файла `.config`, то наряду с этой возможностью можно воспользоваться отдельным файлом конфигурации. Независимо от желаемого варианта API будет практически одним и тем же.

ПРИМЕЧАНИЕ

Поддержка конфигурации Autofac реализована в отдельной сборке. Чтобы воспользоваться этой функцией, нужно добавить ссылку на сборку Autofac.Configuration (https://mng.bz/1Q4V).

Имея ссылку на `Autofac.Configuration`, для считывания регистраций компонентов из стандартного файла `.config` можно обратиться к `ContainerBuilder`:

```
var configuration = new ConfigurationBuilder()
    .AddJsonFile("autofac.json")
    .Build();

builder.RegisterModule(
    new ConfigurationModule(configuration));
```

Загрузка файла autofac.json с помощью системы конфигурации среды .NET Core. По умолчанию файл конфигурации будет находиться в корневом каталоге приложения

Заключение созданной конфигурации в модуль Autofac, обрабатывающий файл конфигурации и отображающий в Autofac регистрации на основе файла. Этот модуль добавляется в сборщик с помощью RegisterModule

Рассмотрим простой пример отображения интерфейса `IIngredient` на класс `Steak`:

```
{
  "defaultAssembly": "Ploeh.Samples.MenuModel",
  "components": [
  {
    "services": [{
      "type": "Ploeh.Samples.MenuModel.IIngredient"
    }],
    "type": "Ploeh.Samples.MenuModel.Steak"
  }]
}
```

Конструкция defaultAssembly позволяет записывать типы в краткой форме. Если в ссылке на тип или интерфейс не было указано имя типа с конкретизацией сборки, предполагается, что он будет в исходной сборке

Простое отображение IIngredient на Steak. Тип указан с использованием полного имени, но если он определен в исходной сборке, имя сборки, как в данном случае, можно опустить

Чтобы Autofac мог найти определенный тип, его имя должно входить в пространство имен. Поскольку оба типа находятся в исходной сборке `Ploeh.Samples.MenuModel`, ее имя в данном случае может быть опущено. Хотя атрибут `defaultAssembly` необязательный, он полезен тем, что избавляет от необходимости набора большого объема текста, если в одной и той же сборке находится определение множества типов.

Элемент `components` представляет собой JSON-массив элементов `component`. В предыдущем примере содержался один компонент, но можно добавить любое их количество. В каждом из них нужно указать конкретный тип с атрибутом `type`. Это единственный обязательный атрибут. Для отображения класса `Steak` на `IIngredient` можно воспользоваться дополнительным атрибутом `services`.

Использование файла конфигурации подходит для тех случаев, когда нужно изменить конфигурацию одного или нескольких компонентов без перекомпиляции приложения, но, поскольку он хрупок, его применение следует ограничить исключительно такими случаями. Для основной части конфигурации контейнера следует задействовать либо автоматическую регистрацию, либо конфигурацию в виде кода.

СОВЕТ

Не забывайте, что победителем всегда будет самая последняя конфигурация типа! Этой особенностью можно воспользоваться для переопределения жестко закодированной конфигурации с помощью конфигураций в формате XML. Для этого нужно не забыть считать XML-конфигурацию после того, как были сконфигурированы любые другие компоненты.

В этом разделе был представлен DI-контейнер Autofac и продемонстрированы основные механизмы: способы конфигурирования `ContainerBuilder` и последующее использование созданного контейнера для разрешения сервисов. Это разрешение выполняется довольно просто за счет всего лишь одного вызова метода `Resolve`, поэтому вся сложность заключается в конфигурировании контейнера. Оно может выполняться несколькими различными способами, включая императивный код и файлы конфигурации.

Пока мы рассматривали наиболее общий API, не охваченными остались гораздо более сложные области. Одной из наиболее важных тем является управление временем жизни компонентов.

13.2. Управление временем жизни

Управление временем жизни, включая наиболее распространенные концептуальные жизненные циклы Singleton, Scoped и Transient, рассматривалось в главе 8. В Autofac поддерживаются несколько иные жизненные циклы, позволяющие настраивать время жизни всех сервисов. В качестве составной части API доступны жизненные циклы, показанные в табл. 13.2.

ПРИМЕЧАНИЕ

В Autofac жизненные циклы называются областями видимости экземпляров.

Таблица 13.2. Области видимости экземпляров (жизненные циклы) в Autofac

Название в Autofac	Паттерн	Комментарии
Per-dependency	Transient	Это область видимости экземпляров по умолчанию. Экземпляры отслеживаются контейнером
Single instance	Singleton	Экземпляры ликвидируются при ликвидации контейнера
Per-lifetime scope	Scoped	Связывает время жизни компонентов с областью видимости времени жизни (см. подраздел 13.2.1)

СОВЕТ

Жизненный цикл по умолчанию Transient считается самым безопасным, но не всегда самым эффективным. Singleton более эффективен для потокобезопасных сервисов, но нужно не забыть явно зарегистрировать эти сервисы.

Реализации Transient и Singleton в Autofac являются эквивалентами обычных жизненных циклов, рассмотренных в главе 8, поэтому подробно говорить о них в этой главе не будем. Вместо этого в данном разделе покажем, как можно определить жизненные циклы для компонентов как в коде, так и с помощью файлов конфигурации. Рассмотрим также используемую в Autofac концепцию областей видимости времени жизни и то, как они могут применяться для реализации жизненного цикла Scoped. Изучив раздел, вы сможете воспользоваться жизненными циклами Autofac в собственном приложении. Для начала рассмотрим способы конфигурирования области видимости экземпляров применительно к компонентам.

13.2.1. Конфигурирование областей видимости экземпляров

В этом разделе будут рассмотрены способы управления в Autofac областями видимости экземпляров применительно к компонентам. Конфигурирование областей видимости экземпляров является частью регистрации компонентов, определить их можно как с помощью кода, так и с использованием файла конфигурации. Рассмотрим все по порядку.

Конфигурирование областей видимости экземпляров с помощью кода

Определение областей видимости экземпляров является частью регистраций, выполняемых в отношении экземпляра `ContainerBuilder`. Делается это довольно просто:

```
builder.RegisterType<SauceBéarnaise>().SingleInstance();
```

Этой строкой кода выполняется конфигурация конкретного класса `SauceBéarnaise` в качестве одиночки (Singleton), в результате чего при каждом запросе `SauceBéarnaise` возвращается один и тот же экземпляр. Если нужно отобразить абстракцию на конкретный класс с определенным жизненным циклом, можно воспользоваться обычным методом `As` и поместить вызов метода `SingleInstance` где угодно. Функционально эти две регистрации являются эквивалентными:

```
builder
    .RegisterType<SauceBéarnaise>()
    .As<IIngredient>()
    .SingleInstance();
```

```
builder
    .RegisterType<SauceBéarnaise>()
    .SingleInstance()
    .As<IIngredient>();
```

Заметьте, единственное различие состоит в том, что мы поменяли местами вызовы методов `As` и `SingleInstance`. Нам ближе последовательность, показанная слева, поскольку вызовы методов `RegisterType` и `As` формируют отображение между конкретным классом и абстракцией. Когда они стоят рядом, регистрацию легче прочитать, а затем в качестве модификации отображения можно указать и область видимости экземпляра.

Хотя Transient является областью видимости экземпляра, используемой по умолчанию, ее можно указать и явным образом. Следующие два примера эквивалентны друг другу:

```
builder
    .RegisterType<SauceBéarnaise>();
```

```
builder
    .RegisterType<SauceBéarnaise>()
    .InstancePerDependency();
```

Конфигурирование области видимости экземпляра для регистраций на основе соглашений выполняется с использованием того же метода, что и при единичных регистрациях:

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

builder.RegisterAssemblyTypes(ingredientsAssembly).As<IIngredient>()
    .SingleInstance();
```

Для определения области видимости экземпляра для всех регистраций в соглашении можно воспользоваться `SingleInstance` и другими родственными ему методами. В предыдущем примере все регистрации `IIngredient` были определены с жизненным циклом Singleton. Точно так же, как регистрацию компонентов можно проводить с использованием не только кода, но и файла конфигурации, в обоих случаях можно выполнять и конфигурирование области видимости экземпляра.

Конфигурирование областей видимости экземпляров с помощью файлов конфигурации

Когда возникает потребность в определении компонентов в файле конфигурации, в нем же следует выполнять и конфигурацию областей видимости экземпляров. В противном случае окажется, что все компоненты получат один и тот же жизненный цикл, используемый по умолчанию. Это нетрудно сделать в качестве части конфигурационной схемы, показанной в подразделе 13.1.2. Для объявления жизненного цикла можно воспользоваться опциональным атрибутом `instancescope` (листинг 13.4).

Листинг 13.4. Применение опционального атрибута instance-scope

```
{
    "defaultAssembly": "Ploeh.Samples.MenuModel",
    "components": [
    {
        "services": [{
            "type": "Ploeh.Samples.MenuModel.IIngredient"
        }],
        "type": "Ploeh.Samples.MenuModel.Steak",
        "instance-scope": "single-instance"
    }]
}
```

Добавление к конфигурации области видимости Singleton-экземпляра

Единственное отличие данного кода от примера из подраздела 13.1.2 заключается в добавлении атрибута `instance-scope`, настраивающего экземпляр на жизненный

цикл Singleton. Когда атрибут `instancescope` опущен, используется значение `per-dependency`, которое в Autofac является эквивалентом жизненного цикла Transient.

Настройка области видимости экземпляров компонентов как в коде, так и в файле не особо сложна. Во всех случаях это делается в сугубо декларативной манере. Хотя конфигурирование, как правило, не вызывает особого труда, не следует забывать, что некоторые жизненные циклы связаны с присутствием в системе долгоживущих объектов, потребляющих ресурсы на всем протяжении своего существования.

13.2.2. Высвобождение компонентов

Из подраздела 8.2.2 известно, что после завершения работы с объектами важно высвободить их. В Autofac нет явно определенного метода `Release`, вместо него применяется понятие области видимости времени жизни. Эта область может рассматриваться как одноразовая копия контейнера. На рис. 13.3 показано, что ею определяется граница, в пределах которой компоненты можно использовать повторно.

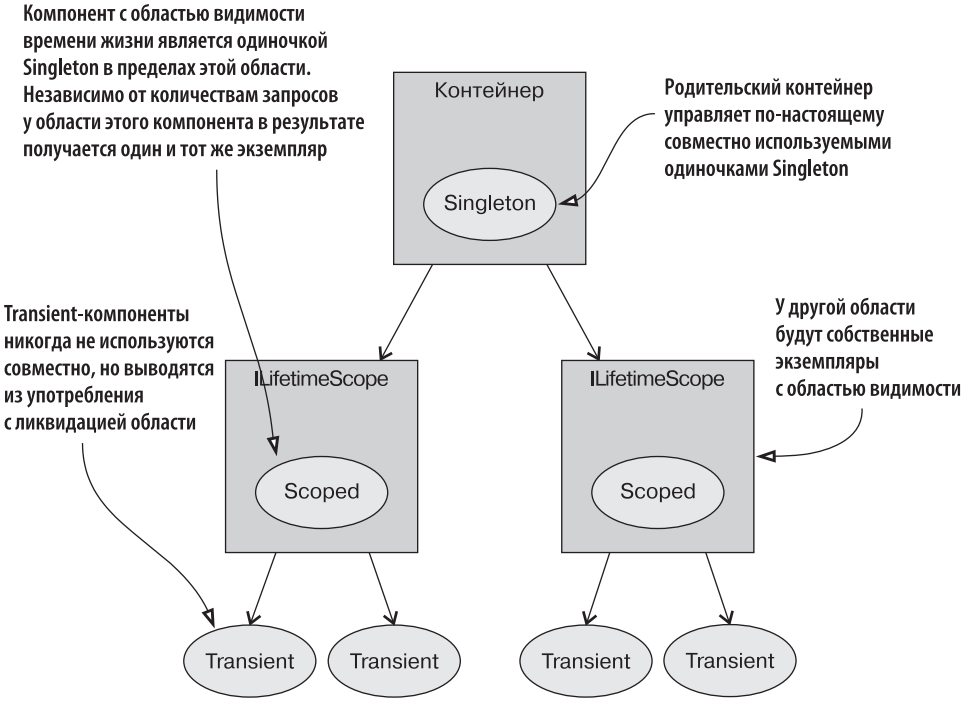


Рис. 13.3. В Autofac области видимости времени жизни ведут себя как контейнеры, способные совместно использовать компоненты в течение ограниченного периода времени или с определенной целью

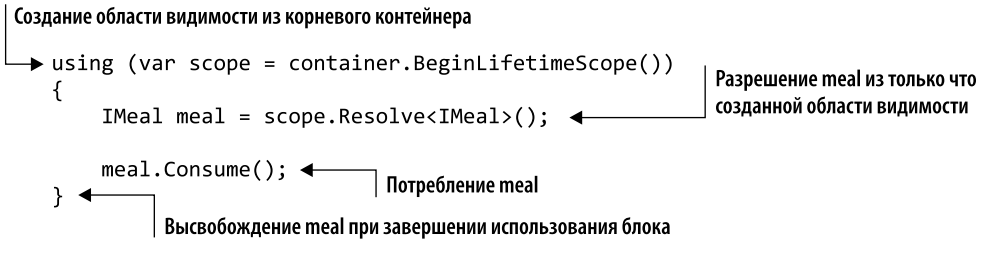
Область видимости времени жизни определяется производным контейнером, которым можно воспользоваться в определенный период времени или с определенной

целью, наиболее очевидный пример — веб-запрос. Область видимости порождается контейнером, поэтому она наследует все Singleton-элементы, отслеживаемые родительским контейнером, но при этом область видимости выступает также в качестве контейнера локальных Singleton-компонентов. Когда из области видимости времени жизни запрашивается компонент с определяемым ею временем жизни, всегда выдается один и тот же экземпляр. Отличие от настоящих Singleton-компонентов состоит в том, что при запросе из второй области видимости выдается другой экземпляр.

ПРИМЕЧАНИЕ

Transient-компоненты действуют предсказуемо независимо от того, были они разрешены из корневого контейнера или из области видимости времени жизни.

Одной из важных особенностей областей видимости времени жизни является возможность правильного высвобождения компонентов, как только область видимости завершит свое существование. Новая область видимости создается с помощью метода `BeginLifetimeScope` и высвобождает все соответствующие компоненты путем вызова своего метода `Dispose`:



Новая область видимости создается из контейнера вызовом метода `BeginLifetimeScope`. Возвращаемым значением реализуется `IDisposable`, следовательно, его можно заключить в блок `using`. Поскольку им также реализуется тот же самый интерфейс, который реализуется самим контейнером, областью видимости можно воспользоваться для разрешения компонентов точно так же, как и самим контейнером.

Когда работа с областью видимости времени жизни будет завершена, ее можно будет ликвидировать. Это происходит автоматически при выходе из блока `using`, но можно также выбрать явную ликвидацию, вызвав метод `Dispose`. При ликвидации области видимости высвобождаются также все компоненты, созданные областью видимости времени жизни. В приведенном примере это означает, что высвобождается граф объектов `meal`.

Зависимости компонента всегда разрешаются в области видимости времени жизни компонента или ниже ее. Например, если нужна Transient-зависимость, внедренная в Singleton-компонент, то она берется из корневого контейнера, даже если Singleton-компонент был разрешен из вложенной области видимости времени жизни. Тогда Transient-зависимость станет отслеживаться в корневом контейнере и окажется застрахована от ликвидации, когда будет ликвидироваться область видимости времени жизни. В противном случае Singleton-потребитель выйдет из

строя, поскольку он останется в корневом контейнере, продолжая зависеть от уже ликвидированного компонента.

ПРИМЕЧАНИЕ

Следует помнить, что высвобождение ликвидируемого компонента не эквивалентно его ликвидации. Это сигнал контейнеру, что данный компонент может быть выведен из обращения. Если это Scoped-компонент, он будет ликвидирован, однако если это Singleton-компонент, он останется активным до тех пор, пока не будет ликвидирован корневой контейнер.

Способ конфигурирования Singleton- или Transient-компонентов уже был показан в этом разделе. Конфигурирование компонента таким образом, чтобы его экземпляр был привязан к области видимости времени жизни, выполняется похожим способом:

```
builder.RegisterType<SauceBéarnaise>()
    .As<IIngredient>()
    .InstancePerLifetimeScope();
```

Метод InstancePerLifetimeScope используется по аналогии с методами SingleInstance и InstancePerDependency, чтобы заявить, что время жизни компонента должно следовать за той областью видимости времени жизни, в которой создается экземпляр

ВНИМАНИЕ

Autofac отслеживает большинство компонентов, даже ликвидируемые Transient-компоненты, поэтому важно разрешать все компоненты из области видимости времени жизни и ликвидировать область видимости после использования. Разрешение экземпляров из корневого контейнера приводит к постоянному возвращению одного и того же экземпляра. Это вызывает ошибки одновременных вычислений, если находящийся в такой области компонент не является потокобезопасным. Когда корневой контейнер применяется для разрешения ликвидируемого Transient-компонента, то, даже если при каждом вызове Resolve создаются новые экземпляры, они не ликвидируются, чтобы была возможность сделать это при ликвидации контейнера. Поскольку корневой контейнер не будет ликвидироваться до тех пор, пока не будет остановлено приложение, возникнут утечки памяти.

По своей природе Singleton-компоненты не высвобождаются до истечения времени жизни самого контейнера. Если же контейнер вам больше не нужен, можно высвободить и эти компоненты. Это делается ликвидацией самого контейнера:

```
container.Dispose();
```

В общем-то, это не так важно, как ликвидация области видимости, поскольку время жизни контейнера тесно коррелирует с временем жизни поддерживаемого им приложения. Обычно контейнер сохраняется, пока выполняется приложение, поэтому его ликвидация произойдет только по завершении работы приложения. В этом случае память будет восстановлена операционной системой.

На этом наш тур по управлению временем жизни в Autofac завершается. Компоненты могут быть сконфигурированы со смешанными областями видимости элементов, и это справедливо даже при регистрации сразу нескольких реализаций одной и той же абстракции. До сих пор контейнеру позволялось связывать зависимости, заранее предполагая, что этот процесс будет выполняться автоматически. Но бывают и исключения. В следующем разделе рассмотрим работу с классами, экземпляры которых должны создаваться особым способом.

13.3. Регистрация сложных API

Пока мы рассматривали только возможности конфигурирования компонентов, использующих внедрение через конструктор. Одним из многочисленных преимуществ подобного внедрения является то, что такие DI-контейнеры, как Autofac, могут легко разобраться в способах составления композиции и создания всех классов, фигурирующих в графе зависимостей. Но эта ясность утрачивается, когда API ведут себя не лучшим образом.

В этом разделе покажем, как работать с элементарными аргументами конструкторов и статическими фабриками. Это требует особого внимания. Начнем с рассмотрения классов, принимающих в качестве аргументов конструктора такие элементарные типы, как строки и целые числа.

13.3.1. Конфигурирование элементарных зависимостей

Пока в потребителей внедряются абстракции, не возникает никаких проблем. Но ситуация усложняется, когда конструктор зависит от элементарного типа, такого как строка, число или перечисление. Именно это и происходит с реализациями доступа к данным, принимающими в качестве параметра конструктора строку подключения. Однако сам вопрос гораздо шире и касается всех строк и чисел.

Концептуально регистрация строки или числа в качестве компонента контейнера не всегда логична. Но в Autofac такая возможность существует. Рассмотрим в качестве примера следующий конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В данном случае `Spiciness` является перечислением:

```
public enum Spiciness { Mild, Medium, Hot }
```

СОВЕТ

Общее правило гласит, что перечисления относятся к проблемному коду и должны быть переструктурированы в полиморфные классы[1]. Но для данного примера они вполне подходят.

[1] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 71.

Если нужно, чтобы все потребители `Spiciness` использовали одно и то же значение, то `Spiciness` и `ChiliConCarne` можно зарегистрировать независимо друг от друга. Как это сделать, показано в данном фрагменте:

```
builder.Register<Spiciness>(c => Spiciness.Medium);
builder.RegisterType<ChiliConCarne>().As<ICourse>();
```

Когда в дальнейшем будет выполняться разрешение `ChiliConCarne`, то значением `Spiciness` для него станет `Medium`, что произойдет и со всеми другими компонентами с зависимостью от `Spiciness`. Если нужен более конкретный контроль взаимоотношений `ChiliConCarne` и `Spiciness`, можно воспользоваться методом `WithParameter`. Поскольку параметру `Spiciness` требуется предоставить конкретное значение, можно применить переопределение `WithParameter`, принимающее имя и значение параметра:

```
builder.RegisterType<ChiliConCarne>().As<ICourse>()
    .WithParameter(
        "spiciness",          <-- Имя параметра
        Spiciness.Hot);       <-- Внедряемое значение
```

В обоих рассматриваемых здесь вариантах для предоставления компоненту конкретного значения используется автоматическое связывание. Из раздела 13.4 будет известно, что у него есть свои достоинства и недостатки. Но более удобным решением является извлечение элементарных зависимостей в граничные объекты.

В подразделе 10.3.3 объяснялось, как введение граничных объектов позволило смягчить нарушение принципа открытости/закрытости, вызванное применением `IProductService`. Граничные объекты отлично подходят и для устранения неопределенности.

Например, свойство `Spiciness` (острота) может быть описано более общим понятием вкуса (flavoring). `Flavoring` может включать и другие свойства, например соленость. Иными словами, `Spiciness` (острота) и `ExtraSalty` (соленость) можно заключить в класс вкуса `Flavoring`[1]:

```
public class Flavoring
{
    public readonly Spiciness Spiciness;
    public readonly bool ExtraSalty;

    public Flavoring(Spiciness spiciness, bool extraSalty)
    {
        this.Spiciness = spiciness;
        this.ExtraSalty = extraSalty;
    }
}
```

[1] С кулинарной точки зрения Марку не нравится пересоленность (extra saltiness). Аппетитная еда не должна быть пересолена.

СОВЕТ

Из подраздела 10.3.3 известно, что наличие у граничных объектов всего одного параметра — это вполне нормальное явление. Целью этого является устранение неопределенности, и не только на техническом уровне. Имя такого граничного объекта может послужить описанием того, чем ваш код занят на функциональном уровне, что весьма элегантно делается в классе Flavoring.

Теперь, с введением граничного объекта `Flavoring`, облегчается автоматическое связывание любой реализации `ICourse`, требующей указания какого-либо вкуса:

```
var flavoring = new Flavoring(Spiciness.Medium, extraSalty: true);
builder.RegisterInstance<Flavoring>(flavoring);

builder.RegisterType<ChiliConCarne>().As<ICourse>();
```

Сейчас имеется единственный экземпляр класса `Flavoring`. И `Flavoring` становится для экземпляров `ICourse` объектом конфигурации. Поскольку в наличии будет только один экземпляр `Flavoring`, его можно зарегистрировать в Autofac, воспользовавшись `RegisterInstance`.

ПРИМЕЧАНИЕ

Не внедряйте граничные объекты, действие которых в качестве объектов конфигурации распространяется на все приложение. Лучше применять более конкретизированные граничные объекты, содержащие только значения, необходимые конкретному потребителю. Тогда станет понятнее, какие значения конфигурации используются компонентом, что упростит тестирование. В общем смысле внедрение объектов конфигурации, действие которых распространяется на все приложение, нарушает принцип изоляции интерфейса.

Извлечение элементарных зависимостей в граничные объекты должно превалировать над ранее рассмотренными вариантами, поскольку такие объекты устраняют неопределенность как на функциональном, так и на техническом уровне. Но при этом требуется вносить изменения в конструктор компонента, что не всегда возможно. В таком случае подходящим выбором может стать использование метода `WithParameter`.

13.3.2. Регистрация объектов с помощью блоков кода

Еще одним вариантом создания компонента с элементарным значением является использование метода `Register`. Он позволяет предоставить делегат, создающий компонент:

```
builder.Register<ICourse>(c => new ChiliConCarne(Spiciness.Hot));
```

Метод `Register` нам уже встречался при рассмотрении регистрации `Spiciness` в подразделе 13.3.1. Здесь же при каждом разрешении сервиса `ICourse` со `Spiciness Hot` вызывается конструктор `ChiliConCarne`.

ПРИМЕЧАНИЕ

Метод Register безопасен с точки зрения типов, но он отменяет автоматическое связывание.

Что же касается класса `ChiliConCarne`, то у вас есть выбор между автоматическим связыванием и использованием блока кода. Для других классов могут действовать более строгие ограничения: их экземпляры не могут создаваться через публичный конструктор. Вместо этого для создания экземпляров типа приходится задействовать фабрику. Для DI-контейнеров это всегда проблематично, поскольку изначально они обходятся публичными конструкторами. Рассмотрим следующий конструктор для открытого класса `JunkFood`:

```
internal JunkFood(string name)
```

Хотя класс `JunkFood` может быть публичным, конструктор является внутренним. На самом деле в данном случае экземпляры `JunkFood` должны создаваться путем применения статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static JunkFood Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С позиции контейнера Autofac мы имеем дело с проблематичным API, поскольку в отношении статичных фабрик каких-то однозначных и устоявшихся соглашений нет. Здесь требуется помощь, получить которую можно за счет предоставления блока кода, выполняемого контейнером для создания экземпляра:

```
builder.Register<IMeal>(c => JunkFoodFactory.Create("chicken meal"));
```

На этот раз метод `Register` применяется для создания компонента путем вызова статической фабрики внутри блока кода. При наличии такой конструкции `JunkFoodFactory.Create` вызывается при каждом разрешении `IMeal`, возвращая соответствующий результат.

Если все сводится к написанию кода для создания экземпляра, то чем же это лучше вызова данного кода напрямую? Использование блока кода внутри вызова метода `Register` полезно тем, что:

- *создается отображение* `IMeal` *на* `JunkFood`. Это позволяет классам-потребителям сохранять слабую связанность;
- *сохраняется возможность конфигурирования области видимости экземпляра*. Хотя блок кода будет вызываться для создания экземпляра, при каждом запросе экземпляра этого может и не происходить. Вызов осуществляется по умолчанию, но если изменить область видимости экземпляра на Singleton, блок кода будет вызван только один раз, а результат будет кэширован для последующего повторного применения.

В этом разделе был показан порядок использования Autofac для работы с более сложными порождающими API. Для связывания конструкторов с сервисами с целью поддержания видимости автоматического связывания можно воспользоваться методом `WithParameter`, а для более типобезопасного подхода — методом `Register`. Мы еще не рассматривали работу с несколькими компонентами, поэтому теперь уделим внимание и этому направлению.

13.4. Работа с несколькими компонентами

В подразделе 12.1.2 говорилось, что в DI-контейнерах делается ставка на определенность, а вот неопределенность создает для них существенные трудности. При внедрении через конструктор лучше работать с единственным конструктором, чем с несколькими переопределяемыми конструкторами, поскольку так создается четкое представление о том, какой конструктор следует использовать, когда нет выбора. То же самое относится и к отображению абстракций на конкретные типы. Если попытаться отобразить на одну и ту же абстракцию сразу несколько конкретных типов, возникнет неопределенность.

Несмотря на крайнюю нежелательность пребывания в неопределенности, зачастую все же приходится работать с несколькими реализациями одной и той же абстракции[1]. Это может быть вызвано следующими ситуациями.

- Разные потребители используют разные конкретные типы.
- Зависимости представлены последовательностями.
- Применяются декораторы или компоновщики.

В данном разделе будут последовательно рассмотрены все эти случаи и то, как Autofac справляется с ними. Когда вопрос будет закрыт, вы сможете регистрировать и разрешать компоненты, даже если в работе будут сразу несколько реализаций одной и той же абстракции. Сначала рассмотрим способы обеспечения более четкого контроля, чем тот, который доступен при использовании автоматического связывания.

13.4.1. Выбор из нескольких кандидатов

При всех своих удобстве и эффективности автоматическое связывание не позволяет контролировать ситуацию в целом. Пока все абстракции четко отображаются на конкретные типы, не возникает никаких проблем. Но как только вводится сразу несколько реализаций одного и того же интерфейса, вскрывается вся неприглядность неопределенности. Сначала вспомним, как Autofac справляется с несколькими регистрациями одной и той же абстракции.

---

[1] Фактически наличие множества абстракций при всего лишь одной реализации служит признаком проблемной конструкции, подпадающей под принцип повторного использования абстракций. См. статью: *Gorman J.* Reused Abstractions Principle (RAP), 2010; http://www.codemanship.co.uk/parlezuml/blog/?postid=934.

Конфигурирование нескольких реализаций одного и того же сервиса

Из подраздела 13.1.2 известно, что несколько реализаций одного и того же интерфейса могут быть зарегистрированы следующим образом:

```
builder.RegisterType<Steak>().As<IIngredient>();
builder.RegisterType<SauceBéarnaise>().As<IIngredient>();
```

В данном примере классы `Steak` и `SauceBéarnaise` регистрируются в качестве сервиса `IIngredient`. Выигрывает последняя регистрация, поэтому, если `Ingredient` разрешается через `scope.Resolve<IIngredient>()`, будет получен экземпляр `SauceBéarnaise`.

СОВЕТ

> Экземпляр по умолчанию для этого типа определяется последней регистрацией данного сервиса. Если вы не хотите, чтобы регистрация имела приоритет над предыдущими регистрациями, можете зарегистрировать тип с помощью .PreserveExistingDefaults().

Можно также потребовать от контейнера разрешения всех компонентов `IIngredient`. Специального метода для этого в Autofac нет, ставка делается на типы отношения (https://mng.bz/P429). Под типом отношения понимается тип, показывающий отношение, которое может интерпретироваться контейнером. В качестве примера можно воспользоваться `IEnumerable<T>`, чтобы показать, что нужны все сервисы заданного типа:

```
IEnumerable<IIngredient> ingredients =
    scope.Resolve<IEnumerable<IIngredient>>();
```

Обратите внимание на то, что здесь используется обычный метод `Resolve`, но мы запрашиваем `IEnumerable<IIngredient>`. Autofac интерпретирует это в качестве соглашения и выдает все имеющиеся у него компоненты `IIngredient`.

СОВЕТ

> В качестве альтернативы IEnumerable<T> можно запросить массив. Результат будет таким же — в обоих случаях будут получены все компоненты запрошенного типа.

При регистрации компонентов каждой регистрации можно дать имя, которым впоследствии воспользоваться для выбора между различными компонентами. Этот процесс показан в следующем фрагменте кода:

```
builder.RegisterType<Steak>().Named<IIngredient>("meat");
builder.RegisterType<SauceBéarnaise>().Named<IIngredient>("sauce");
```

Как всегда, сначала используется метод `RegisterType`, но вместо следующего за ним метода `As` применяется метод `Named`, указывающий тип сервиса и его имя.

Это позволяет разрешить поименованные сервисы путем предоставления такого же имени методу `ResolveNamed`:

```
IIngredient meat = scope.ResolveNamed<IIngredient>("meat");
IIngredient sauce = scope.ResolveNamed<IIngredient>("sauce");
```

ПРИМЕЧАНИЕ

> Поименованный компонент не считается компонентом по умолчанию. Если зарегистрировать только поименованные компоненты, разрешить экземпляр сервиса по умолчанию будет невозможно. Но ничто не мешает зарегистрировать и компонент по умолчанию (безымянный), воспользовавшись для этого методом As. Это можно сделать даже в той же самой инструкции путем встраивания метода в цепочку.

Поименованные компоненты со строками довольно часто встречаются в DI-контейнерах. Autofac позволяет также идентифицировать компоненты с произвольными ключами:

```
object meatKey = new object();
builder.RegisterType<Steak>().Keyed<IIngredient>(meatKey);
```

Ключом может стать любой объект, которым впоследствии можно воспользоваться для разрешения компонента:

```
IIngredient meat = scope.ResolveKeyed<IIngredient>(meatKey);
```

Если разрешение сервисов всегда должно происходить в единственном корне композиции, то ожидать встречи с такой неопределенностью на этом уровне не нужно. Если придется вызывать метод `Resolve` с указанным именем или ключом, следует разобраться, можно ли изменить используемый подход, сделав его менее неопределенным. Поименованными или снабженными ключом экземплярами можно воспользоваться для выбора из множества альтернатив при конфигурировании зависимостей для заданного сервиса.

Регистрация поименованных зависимостей

При всех достоинствах автоматического связывания иногда требуется переопределить обычное поведение, чтобы более четко управлять тем, какие зависимости куда направляются, или обратиться к неопределенному API. Рассмотрим в качестве примера следующий конструктор:

```
public ThreeCourseMeal(ICourse entrée, ICourse mainCourse, ICourse dessert)
```

Здесь имеются три идентично типизированные зависимости, в каждой из которых представлена своя концепция. В большинстве случаев нужно отобразить каждую из зависимостей на отдельно взятый тип. Порядок регистрации отображений `ICourse` показан в листинге 13.5.

Листинг 13.5. Регистрация поименованной перемены блюд

```
builder.RegisterType<Rillettes>().Named<ICourse>("entrée");
builder.RegisterType<CordonBleu>().Named<ICourse>("mainCourse");
builder.RegisterType<MousseAuChocolat>().Named<ICourse>("dessert");
```

Здесь регистрируются три поименованных компонента, при этом `Rilettes` отображается на экземпляр по имени `entrée`, `CordonBleu` — на экземпляр `mainCourse`, а `MousseAuChocolat` — на экземпляр `dessert`. Учитывая эту конфигурацию, теперь можно зарегистрировать класс `ThreeCourseMeal` с поименованными регистрациями.

Это на удивление непростая задача. Сначала в листинге 13.6 будет показано, как это выглядит, а затем пример будет разобран по частям, чтобы понять, что в нем происходит.

Листинг 13.6. Переопределение автоматического связывания

Метод WithParameter предоставляет значения параметра для конструктора ThreeCourseMeal. Одно из его переопределений получает два аргумента

```
builder.RegisterType<ThreeCourseMeal>().As<IMeal>()
    .WithParameter(
        (p, c) => p.Name == "entrée",
        (p, c) => c.ResolveNamed<ICourse>("entrée"))
    .WithParameter(
        (p, c) => p.Name == "mainCourse",
        (p, c) => c.ResolveNamed<ICourse>("mainCourse"))
    .WithParameter(
        (p, c) => p.Name == "dessert",
        (p, c) => c.ResolveNamed<ICourse>("dessert"));
```

Предикат, сопоставляющий параметр конструктора с указанным именем, в данном случае это mainCourse

Разрешение значения, внедряемого в параметр конструктора, в данном случае это desert

Разберемся, что здесь происходит. Переопределение метода `WithParameter` охватывает класс `ResolvedParameter`, имеющий следующий конструктор:

```
public ResolvedParameter(
    Func<ParameterInfo, IComponentContext, bool> predicate,
    Func<ParameterInfo, IComponentContext, object> valueAccessor);
```

Параметр предиката является тестом, определяющим, должен ли быть вызван делегат `valueAccessor`. Когда предикат возвращает `true`, для предоставления значения для параметра вызывается `valueAccessor`. Оба делегата получают один и тот же ввод — информацию о параметре в форме объекта `ParameterInfo` и `IComponentContext`, которым можно воспользоваться для разрешения других компонентов. Когда контейнер Autofac использует экземпляры `ResolvedParameter`, он предоставляет оба этих значения при вызове делегатов.

Как показано в листинге 13.6, полученная регистрация страдает многословием. Но благодаря двум самостоятельно созданным вспомогательным методам можно добиться существенного упрощения регистрации:

```
builder.RegisterType<ThreeCourseMeal>().As<IMeal>()
    .WithParameter(Named("entrée"), InjectWith<ICourse>("entrée"))
    .WithParameter(Named("mainCourse"), InjectWith<ICourse>("mainCourse"))
    .WithParameter(Named("dessert"), InjectWith<ICourse>("dessert"));
```

Введение вспомогательных методов `Named` и `InjectWith<T>` упрощает регистрацию, устраняет многословие и в то же время позволяет разобраться с тем, что происходит. Все становится похожим на поэзию (или на бутылку выдержанного вина): «О создай свой обед из трех блюд (`ThreeCourseMeal`) с параметром по имени (`Named`) `entrée` (закуска), для которого внедряется (`InjectedWith`) блюдо (`ICourse`) по имени `entrée` (закуска)».

Два новых метода показаны в следующем фрагменте кода:

```
Func<ParameterInfo, IComponentContext, bool> Named(string name)
{
    return (p, c) => p.Name == name;
}

Func<ParameterInfo, IComponentContext, object> InjectWith<T>(string name)
{
    return (p, c) => c.ResolveNamed<T>(name);
}
```

При вызове оба метода создают новый делегат, оборачивающий предоставленный аргумент `name`. Иногда не остается ничего иного, как применить метод `WithParameter` буквально для каждого параметра конструктора, но бывает и так, что можно воспользоваться соглашением.

Разрешение поименованных компонентов с помощью соглашения

При более тщательном изучении кода листинга 13.6 можно заметить повторяющийся паттерн. При каждом вызове метод `WithParameter` работает только с одним параметром конструктора, то же самое делает и каждый метод `valueAccessor`: для разрешения компонента `ICourse` с тем же именем, что и у параметра, он использует `IComponentContext`.

Называть компонент по имени параметра конструктора не обязательно, но когда он так назван, можно воспользоваться этим соглашением и упростить код листинга 13.6. Как это сделать, показано в листинге 13.7.

Листинг 13.7. Переопределение автоматического связывания с помощью соглашения

```
builder.RegisterType<ThreeCourseMeal>().As<IMeal>()
    .WithParameter(
        (p, c) => true,
        (p, c) => c.ResolveNamed(p.Name, p.ParameterType));
```

Как ни странно, но со всеми тремя параметрами конструктора класса `ThreeCourseMeal` можно работать в рамках одного вызова `WithParameter`. Сделать это можно, указав, что этот экземпляр будет работать с любым параметром, который выдаст ему контейнер Autofac. Поскольку этот метод используется только для конфигурирования класса `ThreeCourseMeal`, соглашение применяется в пределах этой ограниченной области видимости.

Поскольку предикат всегда возвращает `true`, второй блок кода будет вызван для всех трех параметров конструктора. Во всех трех случаях он для разрешения компонента

с тем же именем, что и у параметра, станет обращаться к `IComponentContext`. Функционально это то же самое, что выполнял код листинга 13.6.

ВНИМАНИЕ

Идентификация параметров по их именам удобна, но небезопасна для реструктуризации. Если переименовать параметр, можно испортить конфигурацию (в зависимости от инструмента реструктуризации).

Как и в случае с листингом 13.6, можно создать упрощенную версию листинга 13.7. Но пусть это станет упражнением для читателя.

Переопределение автоматического связывания путем явного отображения параметров на поименованные компоненты — универсальное решение. Это можно сделать, даже если конфигурирование поименованных компонентов является частью корня композиции, а потребитель находится в совершенно другой части, поскольку имя — единственный идентификатор, связывающий поименованный компонент с параметром. Такая возможность есть всегда, но если приходится управлять довольно большим количеством имен, код может стать ненадежным. Когда главной целью использования имен является устранение неопределенности, стоит разработать собственный API. Зачастую общая конструкция от этого только выигрывает.

В следующем разделе будет показано, как можно воспользоваться более определенным и гибким подходом, допускающим любое количество перемен блюд. Для этого нужно изучить порядок работы Autofac со списками и последовательностями.

13.4.2. Связывание последовательностей

В подразделе 6.1.1 рассматривалось внедрение через конструктор в качестве системы предупреждения о нарушении принципа единственной ответственности. Выяснилось, что вместо того, чтобы признать чрезмерное внедрение через конструктор слабостью данного паттерна, следует удовлетвориться тем, что при этом выявляются проблемы самой конструкции.

В процессе работы с DI-контейнерами в условиях неопределенности наблюдается такая же взаимосвязь. В общем-то, DI-контейнеры не способны достойно справиться с неопределенностью. Конечно, такой качественный контейнер, как Autofac, можно заставить ее преодолеть, но выглядит это весьма неуклюже. А зачастую и является признаком того, что конструкцию кода можно усовершенствовать.

СОВЕТ

Если, конфигурируя конкретную часть API с помощью Autofac, вы испытываете определенные трудности, отступите на шаг и переосмыслите конструкцию с точки зрения применения паттернов и принципов, представленных в этой книге. Чаще всего затруднения с конфигурированием вызваны тем, что конструкция приложения не соответствует этим паттернам и нарушает принципы. Улучшение общей конструкции не только облегчает сопровождение приложения, но и упрощает конфигурирование Autofac.

Чтобы не чувствовать скованности, применяя Autofac, нужно смириться с предлагаемыми им условиями и позволить ему привести вас к разработке более качественной и согласованной конструкции. В этом разделе будет рассмотрен пример, показывающий способ избавления от неопределенности, а также демонстрирующий, как Autofac справляется с последовательностями, массивами и списками.

Переход к более рациональной перемене блюд за счет устранения неопределенности

В подразделе 13.4.1 было показано, как `ThreeCourseMeal` со свойственной ему неопределенностью заставил отказаться от автоматического связывания и вместо него воспользоваться методом `WithParameter`. Это должно подтолкнуть к пересмотру конструкции API. Например, простое обобщение наводит на мысль о реализации `IMeal`, получающего произвольное число экземпляров `ICourse`, вместо трех перемен блюд, как было в классе `ThreeCourseMeal`:

```
public Meal(IEnumerable<ICourse> courses)
```

Заметьте, что вместо требования в конструкторе трех отдельных экземпляров `ICourse` одна-единственная зависимость от экземпляра `IEnumerable<ICourse>` позволяет предоставить классу `Meal` любое количество перемен блюд — от нуля до... полного изобилия! Тем самым решается вопрос с неопределенностью, поскольку теперь мы имеем дело только с одной зависимостью. Кроме того, за счет предоставления всего одного универсального класса, способного смоделировать различные типы трапез, от простого перекуса с одной переменой блюд до изысканного обеда на 12 перемен, улучшаются API и реализация компонента.

В этом разделе будут рассмотрены способы конфигурирования Autofac для связывания экземпляров `Meal` с соответствующими зависимостями `ICourse`. Усвоив этот материал, вы сможете составить четкое представление о доступных вариантах конфигурирования экземпляров с последовательностями зависимостей.

Автоматическое связывание последовательностей

Autofac неплохо разбирается в последовательностях, поэтому, если нужно воспользоваться всеми зарегистрированными компонентами заданного сервиса, автоматическое связывание по-прежнему окажется работоспособным. В качестве примера сервис `IMeal` можно сконфигурировать следующим образом:

```
builder.RegisterType<Rillettes>().As<ICourse>();
builder.RegisterType<CordonBlue>().As<ICourse>();
builder.RegisterType<MousseAuChocolat>().As<ICourse>();

builder.RegisterType<Meal>().As<IMeal>();
```

Заметьте, что это абсолютно стандартное отображение конкретного типа на абстракцию. Autofac автоматически разбирается с конструктором `Meal` и определяет, что правильным действием станет разрешение всех компонентов `ICourse`.

При разрешении `IMeal` будет получен экземпляр `Meal` с компонентами `ICourse`: `Rillettes`, `CordonBleu` и `MousseAuChocolat`.

Autofac обрабатывает последовательности в автоматическом режиме и, пока не указано иное, делает то, чего от него ожидают: разрешает последовательность зависимостей для всех зарегистрированных компонентов заданного типа. Дополнительные действия потребуются, только если понадобится явным образом выбрать некоторые компоненты из более широкого набора. Посмотрим, как это можно сделать.

Выбор ограниченного числа компонентов из более широкого набора

Стратегия, по умолчанию используемая Autofac для внедрения всех компонентов, зачастую оказывается правильной политикой, но на рис. 13.4 показано, что иногда из более широкого набора зарегистрированных компонентов требуется выбрать лишь некоторые.

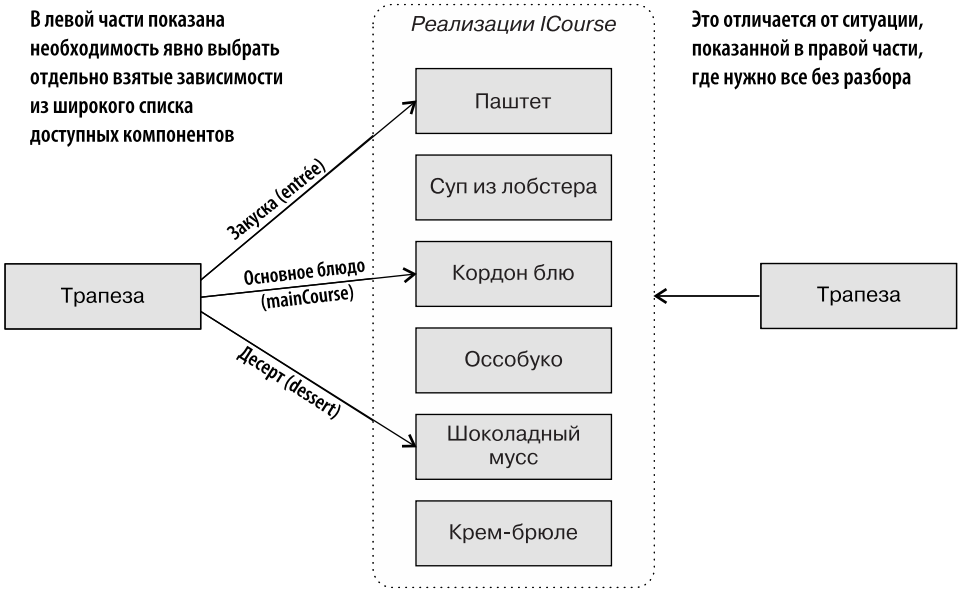


Рис. 13.4. Выбор компонентов из широкого набора зарегистрированных компонентов

ПРИМЕЧАНИЕ

Потребность во внедрении поднабора из полной коллекции не относится к типовым сценариям, но показывает порядок удовлетворения более сложных потребностей.

Когда ранее контейнеру Autofac было позволено автоматически связать все сконфигурированные экземпляры, это соответствовало ситуации, изображенной

в правой части рисунка. Если же нужно зарегистрировать компонент так, как показано в левой части, намеченные к использованию компоненты должны быть определены явно. Чтобы получить желаемый результат, можно воспользоваться методом `WithParameter`, что, собственно, и делалось в коде листингов 13.6 и 13.7. Теперь мы имеем дело с конструктором `Meal`, получающим только один параметр. В листинге 13.8 показан способ реализации той части `WithParameter`, которая предоставляет значения для явного выбора поименованных компонентов из `IComponentContext`.

Листинг 13.8. Внедрение в последовательность поименованных компонентов

```
builder.RegisterType<Meal>().As<IMeal>()
    .WithParameter(
        (p, c) => true,
        (p, c) => new[]
        {
            c.ResolveNamed<ICourse>("entrée"),
            c.ResolveNamed<ICourse>("mainCourse"),
            c.ResolveNamed<ICourse>("dessert")
        });
```

В подразделе 13.4.1 было показано, что метод `WithParameter` в качестве входных параметров принимает два делегата. Первый является предикатом, используемым для определения того, должен ли задействоваться второй делегат. В данном случае решено немного полениться и возвратить `true`. Как известно, у класса `Meal` только один параметр конструктора, поэтому все сработает как надо. Но если впоследствии класс `Meal` будет переделан для получения второго параметра конструктора, это может и не сработать. С позиции безопасности лучше явно проверить тип параметра.

Второй делегат предоставляет значение для параметра. Для разрешения трех поименованных компонентов в массив используется `IComponentContext`. В результате получается массив экземпляров `ICourse`, совместимый с `IEnumerable<ICourse>`.

Autofac разбирается в последовательностях. Этот контейнер все сделает правильно, если только не потребуется из всех сервисов заданного типа явно выбрать отдельно взятые компоненты. Автоматическое связывание работает не только с отдельными экземплярами, но и с последовательностями, и контейнер отображает последовательность на все сконфигурированные экземпляры соответствующего типа. Возможно, менее интуитивно понятным является рассматриваемое далее использование нескольких экземпляров одной и той же абстракции, но оно относится к паттерну проектирования «Декоратор».

13.4.3. Связывание декораторов

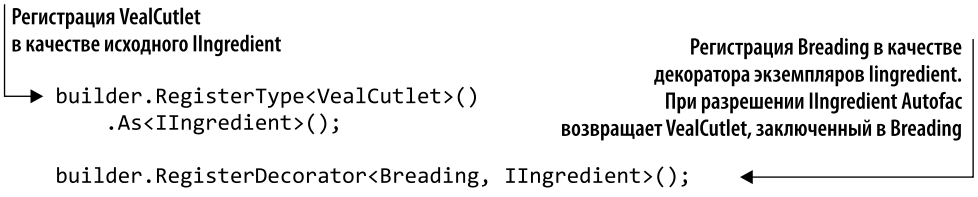
В подразделе 9.1.1 рассматривались преимущества, получаемые от использования паттерна проектирования «Декоратор» при решении проблем применения сквозной функциональности. Согласно определению декораторы вводят в обращение несколько типов одной и той же абстракции. Имеются по крайней мере две реализации абстракции: сам декоратор и декорируемый тип. Если выстроить декораторы в цепочку, можно получить больше абстракций. Это еще один пример наличия

нескольких регистраций одного и того же сервиса. В отличие от того, что говорилось в предыдущих разделах, концептуально эти регистрации не равны, скорее всего, они являются зависимостями друг для друга.

Существует несколько стратегий применения декораторов в Autofac, например ранее рассмотренный метод `WithParameter` или использование блоков кода (см. подраздел 13.3.2). Здесь мы сконцентрируемся на методах `RegisterDecorator` и `RegisterGenericDecorator`, поскольку они облегчают конфигурирование декораторов.

Декорирование необобщенных абстракций с помощью RegisterDecorator

В Autofac имеется встроенная поддержка декораторов с помощью метода `RegisterDecorator`. Порядок его использования для применения `Breading` к `VealCutlet` показан в следующем примере:



Из главы 9 известно, что блюдо кордон блю получается, если сделать в отбивной котлете надрез, начинить ее ветчиной, сыром и чесноком, а потом запанировать и поджарить. В следующем примере показано, как декоратор `HamCheeseGarlic` добавляется между `VealCutlet` и декоратором `Breading`:

```
builder.RegisterType<VealCutlet>()
    .As<IIngredient>();

builder.RegisterDecorator<HamCheeseGarlic,
    IIngredient>();

builder.RegisterDecorator<Breading, IIngredient>();
```

Добавление нового декоратора

За счет размещения этой новой регистрации перед регистрацией `Breading` декоратор `HamCheeseGarlic` первым заключается во внешний декоратор. В результате получается граф объектов, эквивалентный следующей версии, выполненной по технологии чистого DI:

```
new Breading(
    new HamCheeseGarlic(
        new VealCutlet()));
```

VealCutlet обернут в HamCheeseGarlic,
который, в свою очередь, обернут в Breading

ПРИМЕЧАНИЕ

Autofac применяет декораторы в порядке их регистрации.

Имеющийся в Autofac метод `RegisterDecorator` позволяет легко выстроить декораторы в цепочку. Так же легко можно регистрировать обобщенные декораторы, что и будет показано далее.

Декорирование обобщенных абстракций с помощью метода RegisterGenericDecorator

В ходе изучения главы 10 было определено несколько обобщенных декораторов, подходящих для применения к любой реализации `ICommandService<TCommand>`. Далее в этой главе мы отложим в сторону все наши ингредиенты и перемены блюд и рассмотрим способы регистрации обобщенных декораторов с помощью Autofac. В листинге 13.9 показано, как зарегистрировать все реализации `ICommandService<TCommand>` с тремя декораторами, представленными в разделе 10.3.

Листинг 13.9. Декорирование обобщенных абстракций, прошедших автоматическую регистрацию

```
builder.RegisterAssemblyTypes(assembly)
    .AsClosedTypesOf(typeof(ICommandService<>));

builder.RegisterGenericDecorator(
    typeof(AuditingCommandServiceDecorator<>),
    typeof(ICommandService<>));

builder.RegisterGenericDecorator(
    typeof(TransactionCommandServiceDecorator<>),
    typeof(ICommandService<>));

builder.RegisterGenericDecorator(
    typeof(SecureCommandServiceDecorator<>),
    typeof(ICommandService<>));
```

Как было показано в листинге 13.3, в коде листинга 13.9 для регистрации произвольных реализаций `ICommandService<TCommand>` используется метод `RegisterAssemblyTypes`. Но, чтобы зарегистрировать обобщенные декораторы, Autofac предоставляет другой метод — `RegisterGenericDecorator`. Результат, полученный при создании конфигурации в коде листинга 13.9, который уже рассматривался в подразделе 10.3.4, показан на рис. 13.5.

Рис. 13.5. Оснащение реального сервиса команд аспектами ведения контрольного журнала, управления транзакциями и обеспечения безопасности

Конфигурирование декораторов может выполняться и другими способами, но в этом разделе мы рассматриваем только применение методов Autofac, специально разработанных для решения этой задачи. Autofac позволяет работать с несколькими экземплярами разными способами: компоненты

можно зарегистрировать в качестве альтернатив друг другу, в качестве равнозначных элементов, разрешаемых в виде последовательностей, или в виде иерархии декораторов. Во многих случаях Autofac сам определяет, что нужно сделать, но, если нужен более строгий контроль над ситуацией, вы всегда можете явно указать, как должны быть скомпонованы сервисы.

Понятное на интуитивном уровне использование нескольких экземпляров одной и той же абстракции — это внедрение зависимостей в потребителей, полагающихся на последовательности таких зависимостей. Декораторы, однако, тоже будут хорошим примером. К тому же есть и третий, возможно, несколько неожиданный случай выхода нескольких экземпляров на первый план — применение паттерна проектирования «Компоновщик» (Composite).

13.4.4. Связывание компоновщиков

Паттерн проектирования «Компоновщик» уже рассматривался в ряде случаев. Например, в подразделе 6.1.2 создан сервис `CompositeNotificationService` (см. листинг 6.4), в котором не только реализован сервис `INotificationService`, но и сформирована последовательность реализаций последнего.

Связывание необобщенных компоновщиков

Посмотрим, как в Autofac можно зарегистрировать компоновщики, подобные `CompositeNotificationService` из главы 6. В листинге 13.10 этот класс показан еще раз.

Листинг 13.10. Компоновщик CompositeNotificationService из главы 6

```
public class CompositeNotificationService : INotificationService
{
    private readonly IEnumerable<INotificationService> services;

    public CompositeNotificationService(
        IEnumerable<INotificationService> services)
    {
        this.services = services;
    }

    public void OrderApproved(Order order)
    {
        foreach (INotificationService service in this.services)
        {
            service.OrderApproved(order);
        }
    }
}
```

Когда компоновщик внедряется с последовательностью поименованных экземпляров, его регистрацию нужно добавлять в качестве регистрации по умолчанию:

```
builder.RegisterType<OrderApprovedReceiptSender>()
    .Named<INotificationService>("service");
builder.RegisterType<AccountingNotifier>()
    .Named<INotificationService>("service");
builder.RegisterType<OrderFulfillment>()
    .Named<INotificationService>("service");

builder.Register(c =>
    new CompositeNotificationService(
        c.ResolveNamed<IEnumerable<INotificationService>>("service")))
    .As<INotificationService>();
```

Здесь под одним именем `service` регистрируются три реализации `INotificationService`, для чего используется имеющееся в Autofac API автоматического связывания. В то же время `CompositeNotificationService` регистрируется с помощью делегата. Внутри делегата компоновщик создается с использованием `new` в ручном режиме, и в него внедряется `IEnumerable<INotification Service>`. Указанием имени сервиса разрешаются предыдущие поименованные регистрации.

Поскольку со временем количество сервисов уведомлений, скорее всего, увеличится, автоматическая регистрация может уменьшить нагрузку на корень композиции. Используя метод `RegisterAssemblyTypes`, можно превратить прежний список регистраций в простую однострочную инструкцию (листинг 13.11).

Листинг 13.11. Регистрация CompositeNotificationService

```
builder.RegisterAssemblyTypes(assembly)
    .Named<INotificationService>("service");

builder.Register(c =>
    new CompositeNotificationService(
        c.ResolveNamed<IEnumerable<INotificationService>>("service")))
    .As<INotificationService>();
```

Кажущаяся простота обманчива. `RegisterAssemblyTypes` зарегистрирует любую необобщенную реализацию `INotificationService`. При попытке запуска предыдущего кода в зависимости от того, в какой сборке находится ваш компоновщик, Autofac может выдать примерно такое исключение: Circular component dependency detected: CompositeNotificationService ->INotificationService[] ->CompositeNotificationService ->INotificationService[] ->CompositeNotificationService (Обнаружена зацикленная зависимость компонентов: CompositeNotificationService ->INotificationService[] ->CompositeNotificationService ->INotificationService[] ->CompositeNotificationService).

Autofac обнаруживает зацикленность зависимостей (она подробно рассматривалась в разделе 6.3). К счастью, сообщение об исключении довольно четкое. В нем дается описание того, что `CompositeNotificationService` зависит от `INotificationService[]`. И правда, `CompositeNotificationService` включает в себя последовательность `INotificationService`, но сама эта последовательность опять же содержит `CompositeNotificationService`. Значит, `CompositeNotificationService` является элементом последовательности, внедряемым в `CompositeNotificationService`. Такой граф объектов построить невозможно.

`CompositeNotificationService` становится частью последовательности, поскольку имеющийся в Autofac метод `RegisterAssemblyTypes` регистрирует все найденные им необобщенные реализации `INotificationService`. В данном случае `CompositeNotificationService` был помещен в ту же сборку, что и все другие реализации.

Устранить проблему можно несколькими способами. Самым простым решением будет перемещение компоновщика в другую сборку, например ту, что содержит корень композиции. Это не даст методу `RegisterAssemblyTypes` выбрать тип по причине того, что он предоставляется с конкретным экземпляром `Assembly`. Еще один вариант заключается в отфильтровывании `CompositeNotificationService` из списка. Изящнее всего это можно сделать с помощью метода `Except`:

```
builder.RegisterAssemblyTypes(assembly)
    .Except<CompositeNotificationService>()
    .Named<INotificationService>("service");
```

Но перемещение может потребоваться не только для классов компоновщика. Это может коснуться и любого декоратора. Сделать это нетрудно, но при большом количестве реализаций декоратора лучше было бы запросить информацию о типе, чтобы определить, представляет данный тип декоратор или нет. В следующем примере показано, как можно отфильтровать также декораторы с помощью самостоятельно созданного вспомогательного метода `IsDecoratorFor`:

```
builder.RegisterAssemblyTypes(assembly)
    .Except<CompositeNotificationService>()
    .Where(type => !IsDecoratorFor<INotificationService>(type))
    .Named<INotificationService>("service");
```

В следующем примере показан метод `IsDecoratorFor`:

```
private static bool IsDecoratorFor<T>(Type type)
{
    return typeof(T).IsAssignableFrom(type) &&
        type.GetConstructors()[0].GetParameters()
            .Any(p => p.ParameterType == typeof(T));
}
```

Метод `IsDecoratorFor` предполагает, что у типа имеется только один конструктор. Тип считается декоратором, когда в нем реализуется заданная абстракция `T`, а его конструктору также требуется `T`.

Связывание обобщенных компоновщиков

В подразделе 13.4.3 было показано, как использование имеющегося в Autofac метода `RegisterGenericDecorator` превращает регистрацию обобщенных декораторов в детскую забаву. В этом разделе будет рассмотрен способ регистрации компоновщиков для обобщенных абстракций.

В подразделе 6.1.3 класс `CompositeEventHandler<TEvent>` (см. листинг 6.12) назван реализацией компоновщика последовательности реализаций `IEventHandler<TEvent>`.

Посмотрим, можно ли зарегистрировать компоновщик с заключенными в него реализациями обработчиков событий.

Начнем с автоматической регистрации обработчиков событий. Исходя из ранее увиденного, сделаем это с помощью метода `RegisterAssemblyTypes`:

```
builder.RegisterAssemblyTypes(assembly)
    .As(type =>
        from interfaceType in type.GetInterfaces()
        where interfaceType.IsClosedTypeOf(typeof(IEventHandler<>))
        select new KeyedService("handler", interfaceType));
```

В этом примере используется переопределение метода `As`, позволяющее предоставить последовательность экземпляров `Autofac.Core.KeyedService`. Класс `KeyedService` представляет собой небольшой объект данных, в котором сочетаются ключ и тип сервиса.

Autofac пропускает любые найденные им в сборке типы через этот метод `As`. Для поиска интерфейса реализации типа, являющегося закрытой обобщенной версией `IEventHandler<TEvent>`, можно воспользоваться LINQ-запросом. Для большинства имеющихся в сборке типов этот запрос не выдаст никаких результатов, поскольку в большинстве типов не реализуется `IEventHandler<TEvent>`. Для них регистрация в `ContainerBuilder` не добавляется.

Но при всей своей сложности обобщенные компоновщики и декораторы в отфильтровывании не нуждаются. `RegisterAssemblyTypes` выбирает только необобщенные реализации. Такие обобщенные типы, как `CompositeEventHandler<TEvent>`, не создадут никаких проблем и не должны отфильтровываться или перемещаться в другую сборку. И это хорошо, поскольку совсем не весело было бы, если бы нужно было создавать версию `IsDecoratorFor`, способную обрабатывать обобщенные абстракции.

Остается только регистрация для `CompositeEventHandler<TEvent>`. Поскольку это обобщенный тип, задействовать переопределение метода `Register`, принимающего предикат, невозможно. Вместо него берется метод `RegisterGeneric`. Он позволяет создавать отображение обобщенной реализации на ее абстракцию, что похоже на происходящее с методом `RegisterGenericDecorator`. Чтобы внедрить последовательность поименованных регистраций в аргумент конструктора компоновщика, нужно еще раз воспользоваться универсальным методом `WithParameter`:

```
builder.RegisterGeneric(typeof(CompositeEventHandler<>))
    .As(typeof(IEventHandler<>))
    .WithParameter(
        (p, c) => true,
        (p, c) => c.ResolveNamed("handler", p.ParameterType));
```

Поскольку в `CompositeEventHandler<TEvent>` содержится один параметр конструктора, применение регистрации ко всем параметрам упрощается за счет того, что предикату позволено возвращать значение `true`.

Делегаты `WithParameter` вызываются при запросе закрытого `IEventHandler<TEvent>`. Таким образом, вызывая их, можно получить тип параметра конструктора путем

вызова `p.ParameterType`. Например, если запрошен `IEventHandler<OrderApproved>`, типом параметра будет `IEnumerable<IEventHandler<OrderApproved>>`. Передавая этот тип методу `ResolveNamed` с обработчиком имени последовательности, Autofac разрешает ранее зарегистрированную последовательность поименованных экземпляров, реализующих `IEvent Handler<OrderApproved>`.

Хотя в регистрации декораторов нет ничего сложного, к компоновщикам это не относится. Пока еще с прицелом на использование паттерна проектирования «Компоновщик» Autofac не разрабатывался. Возможно, в следующей версии ситуация изменится к лучшему.

На этом рассмотрение DI-контейнера Autofac завершается. В следующей главе сосредоточимся на DI-контейнере Simple Injector.

Резюме

- DI-контейнер Autofac предлагает полноценный API и справляется со множеством непростых ситуаций, которые часто возникают при использовании DI-контейнеров.
- Похоже, что для Autofac наиболее важна четкость выражения задуманного. Он не пытается строить предположения о сути вашего замысла, а предлагает простой и понятный API, предоставляющий варианты явного включения функций.
- Autofac четко разграничивает конфигурирование и потребление контейнера. Конфигурирование компонентов выполняется с помощью экземпляра `ContainerBuilder`, но сам `ContainerBuilder` не может выполнять разрешение компонентов. Когда конфигурирование с помощью `ContainerBuilder` завершится, оно применяется для создания `IContainer`, которым можно воспользоваться для разрешения компонентов.
- При использовании Autofac выполнять разрешение непосредственно из корневого контейнера не принято. Это легко может привести к утечкам памяти или ошибкам одновременных вычислений. Разрешение всегда нужно выполнять из области видимости времени жизни.
- Autofac поддерживает стандартные жизненные циклы Transient, Singleton и Scoped.
- Autofac дает возможность работать с неопределенными конструкторами и типами за счет API, позволяющего предоставлять блоки кода. Благодаря этому можно выполнять любой код, создающий сервис.

14 DI-контейнер Simple Injector

В этой главе

- Работа с основным API регистрации Simple Injector.
- Управление временем жизни компонентов.
- Конфигурирование сложных API.
- Конфигурирование последовательностей, декораторов и компоновщиков.

В предыдущей главе рассматривался DI-контейнер Autofac, созданный Николасом Блумхардтом (Nicholas Blumhardt) в 2007 году. Тремя годами позже Стивен создал Simple Injector, который будет изучен в данной главе. Мы дадим Simple Injector ту же трактовку, что и Autofac. Вы увидите, как можно воспользоваться Simple Injector для применения принципов и паттернов, представленных в частях I–III.

Эта глава разбита на четыре раздела. Можно изучать их по отдельности, но прочитать раздел 14.1 обязательно, чтобы понять все остальные разделы, а в разделе 14.4 есть ссылки на некоторые методы и классы, рассмотренные в разделе 14.3.

Эту главу можно изучить отдельно от остальных глав части IV, если вы нацелены исключительно на Simple Injector, или наряду с другими главами, чтобы сравнивать DI-контейнеры.

Хотя данная глава не претендует на роль исчерпывающего руководства по контейнеру Simple Injector, представленной в ней информации достаточно для начала

работы с ним. Эта глава содержит информацию о том, как можно справиться с наиболее острыми проблемами, которые могут возникнуть при использовании Simple Injector. Дополнительные сведения об этом контейнере можно получить на главной странице Simple Injector, расположенной по адресу https://simpleinjector.org.

14.1. Введение в Simple Injector

Из этого раздела станет известно, где можно будет получить Simple Injector, что именно будет получено и как начать использовать этот контейнер. Кроме этого, будут рассмотрены общие вопросы его конфигурации. Основная информация, которая, возможно, пригодится для начала работы с контейнером, приведена в табл. 14.1.

Таблица 14.1. Краткое знакомство с Simple Injector

Вопрос	Ответ
Где его получить?	Из среды Visual Studio его можно получить через NuGet. Пакет называется SimpleInjector
Какие платформы поддерживаются?	.NET 4.0 и .NET Standard 1.0 (.NET Core 1.0, Mono 4.6, Xamarin.iOS 10.0, Xamarin.Mac 3.0, Xamarin.Android 7.0, UWP 10.0, Windows 8.0, Windows Phone 8.1)
Сколько он стоит?	Нисколько. Это средство с открытым кодом
Какая у него лицензия?	MIT License
Где можно получить помощь?	Гарантированная поддержка отсутствует, но помощь с большой вероятностью можно получить на официальном форуме https://simpleinjector.org/forum или задавая вопросы в Stack Overflow по адресу https://stackoverflow.com/
На какой версии основан материал данной главы?	4.4.3

По большому счету, использование Simple Injector не слишком отличается от применения других DI-контейнеров. Как и в случае с DI-контейнерами Autofac (рассмотрен в главе 13) и Microsoft.Extensions.DependencyInjection (о нем речь пойдет в главе 15), использование этого контейнера представляет собой двухэтапный процесс (рис. 14.1).

Из материала главы 13 можно вспомнить, что для облегчения этого двухэтапного процесса в Autofac используется класс `ContainerBuilder`, создающий `IContainer`. А вот в Simple Injector регистрация и разрешение объединены в одном экземпляре `Container`. И все же он вынуждает проводить регистрацию в два этапа, не позволяя выполнять какие-либо явно выраженные регистрации после разрешения первого сервиса.

Хотя разрешение такое же, как везде, используемый в Simple Injector API регистрации сильно отличается от того, как работает большинство DI-контейнеров. Благодаря своей конструкции и реализации он исключает большинство подводных камней, зачастую становящихся причиной ошибок. Эти подводные камни уже рас-

сматривались в книге, поэтому в данной главе речь пойдет о следующих различиях между Simple Injector и другими DI-контейнерами.

- Области видимости являются охватывающими, поэтому графы объектов всегда разрешаются из самого контейнера, что позволяет предотвратить возникновение ошибок использования памяти и одновременных вычислений.
- Последовательности регистрируются через другой API, чтобы избежать случайного дублирования регистраций от переопределения друг друга.
- Элементарные типы не могут регистрироваться напрямую, чтобы регистрации не стали неопределенными.
- Графы объектов могут быть проверены для выявления распространенных ошибок конфигурации, например захваченных зависимостей.

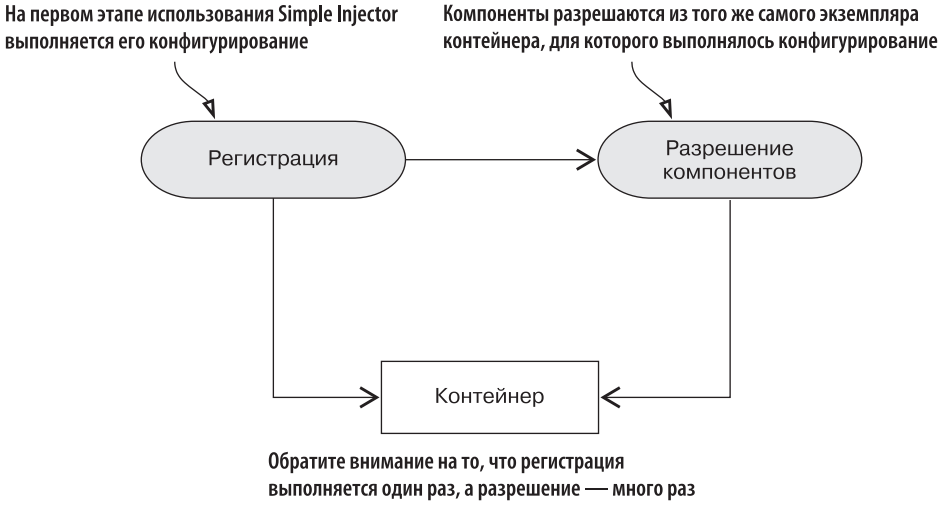


Рис. 14.1. Схема применения Simple Injector. Сначала выполняется конфигурирование контейнера Container, а затем, с использованием того же экземпляра контейнера, из него выполняется разрешение компонентов

Усвоив материал данной главы, вы получите достаточное представление об общей схеме применения Simple Injector и сможете использовать его в сценариях с надлежащим поведением, где все компоненты придерживаются правильных DI-паттернов, таких как внедрение через конструктор. Начнем с самого простого сценария и посмотрим, как можно выполнить разрешение объектов с помощью контейнера Simple Injector.

14.1.1. Разрешение объектов

Основным сервисом любого DI-контейнера является построение графа объектов. В этом разделе будет рассмотрен API, позволяющий составлять графы объектов с помощью Simple Injector.

Если помните, при рассмотрении разрешения компонентов с помощью Autofac говорилось, что этот контейнер требует от вас регистрации всех компонентов до их разрешения. С Simple Injector все происходит по-другому: если запрашивается конкретный тип с конструктором без параметров, конфигурация не нужна. В листинге 14.1 показан один из самых простых вариантов применения Simple Injector.

Листинг 14.1. Простейший вариант использования Simple Injector

```
var container = new Container();   <-- Создание контейнера

SauceBéarnaise sauce =
    container.GetInstance<SauceBéarnaise>();   <-- Разрешение конкретного экземпляра
```

Благодаря экземпляру `SimpleInjector.Container` можно воспользоваться обобщенным методом `GetInstance` с целью получения экземпляра конкретного класса `SauceBéarnaise`. Поскольку этот класс имеет конструктор без параметров, Simple Injector автоматически создает его экземпляр. Никакой явной конфигурации контейнера здесь не требуется.

ПРИМЕЧАНИЕ

> Метод GetInstance<T> является эквивалентом имеющегося в Autofac метода Resolve<T>.

Как говорилось в подразделе 12.1.2, автоматическое связывание предоставляет возможность автоматического составления графа объектов с помощью информации о типах. Поскольку в Simple Injector поддерживается автоматическое связывание, то даже в отсутствие конструктора без параметров этот контейнер может создавать экземпляры без конфигураций при условии, что все задействованные параметры конструктора являются конкретными типами и все параметры во всем дереве имеют конечные ответвления с конструкторами без параметров. Рассмотрим в качестве примера следующий конструктор `Mayonnaise`:

```
public Mayonnaise(EggYolk eggYolk, SunflowerOil oil)
```

Хотя рецепт майонеза слегка упрощен, предположим, что как `EggYolk`, так и `SunflowerOil` являются конкретными классами с конструкторами без параметров. Хотя у самого `Mayonnaise` нет конструктора без параметров, Simple Injector создает его без какой-либо конфигурации:

```
var container = new Container();
Mayonnaise mayo = container.GetInstance<Mayonnaise>();
```

Это срабатывает благодаря тому, что Simple Injector способен разобраться в том, как создать все нужные параметры конструктора. Но, как только вводится слабое связывание, требуется конфигурирование Simple Injector путем отображения абстракций на конкретные типы.

Отображение абстракций на конкретные типы

Временами способность Simple Injector к автоматическому связыванию конкретных типов может пригодиться, но слабое связывание требует от вас отображения абстракций на конкретные типы. Создание экземпляров, основанное на таком отображении, является основной услугой, предлагаемой любым DI-контейнером, но отображение все же требует определения. В этом примере интерфейс `IIngredient` отображается на конкретный класс `SauceBéarnaise`, что позволяет выполнить успешное разрешение `IIngredient`:

```
var container = new Container();

container.Register<IIngredient, SauceBéarnaise>();

IIngredient sauce =
    container.GetInstance<IIngredient>();
```

Отображение абстракции на конкретную реализацию

Разрешение SauceBéarnaise в качестве IIngredient

Экземпляр `Container` используется для регистрации типов и определения отображений. Здесь обобщенный метод `Register` позволяет абстракции быть отображенной на конкретную реализацию. Это дает возможность зарегистрировать конкретный тип. Благодаря ранее сделанному вызову `Register` теперь `SauceBéarnaise` может быть разрешен как `IIngredient`.

ПРИМЕЧАНИЕ

> Метод Register<TService, TImplementation> содержит ограничения обобщенного типа. Это означает, что несовместимое отображение типов будет отловлено компилятором.

Во многих случаях можно обойтись применением обычного API. Но все же бывают ситуации, когда нужен способ разрешения сервисов с еще более слабой типизацией. Имеется и такая возможность.

Разрешение сервисов с еще более слабой типизацией

Иногда применить обычный API невозможно, поскольку в ходе разработки о подходящем типе еще ничего не известно. Имеется лишь экземпляр `Type`, но все равно хотелось бы получить экземпляр подходящего типа. Соответствующий пример был показан в разделе 7.3, где рассматривался принадлежащий среде ASP.NET Core MVC класс `IControllerActivator`. Метод имел следующий вид:

```
object Create(ControllerContext context);
```

Исходя из показанного в листинге 7.8, `ControllerContext` захватывает принадлежащий контроллеру `Type`, который можно извлечь, воспользовавшись свойством `ControllerTypeInfo`, принадлежащим свойству `ActionDescriptor`:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
```

Поскольку в нашем распоряжении имеется лишь экземпляр `Type`, воспользоваться обобщенным методом `GetInstance<T>` нельзя и приходится прибегать к API с более слабой типизацией. Simple Injector предлагает переопределение метода `GetInstance` с более слабой типизацией, позволяющее реализовать метод `Create`:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
return container.GetInstance(controllerType);
```

Переопределение `GetInstance` с более слабой типизацией позволяет передавать переменную `controllerType` непосредственно в Simple Injector. Обычно это означает, что тип возвращаемого значения необходимо привести к некой абстракции, поскольку метод `GetInstance` с более слабой типизацией возвращает объект `object`. Но в случае с `IControllerActivator` это не нужно, поскольку среда ASP.NET Core MVC не требует от контроллеров реализации какого-либо интерфейса или базового класса.

Независимо от того, какое переопределение метода `GetInstance` используется, Simple Injector гарантирует, что возвратит экземпляр запрошенного типа или выдаст исключение, если имеются зависимости, удовлетворить которые невозможно. При приемлемом конфигурировании всех нужных зависимостей Simple Injector может выполнить автоматическое связывание запрошенного типа.

Чтобы иметь возможность разрешения запрошенного типа, все слабосвязанные зависимости следует предварительно сконфигурировать. Конфигурирование Simple Injector можно выполнить несколькими способами, наиболее популярный будет рассмотрен в следующем подразделе.

14.1.2. Конфигурирование контейнера

Из раздела 12.2 известно, что конфигурировать DI-контейнер можно несколькими концептуально различными способами. Обзор возможных вариантов представлен на рис. 12.5, это файлы конфигурации, конфигурация в виде кода и автоматическая регистрация. Все эти варианты показаны также на рис. 14.2.

Основной API конфигурирования Simple Injector сконцентрирован на применении кода и поддерживает как конфигурацию в виде кода, так и автоматическую регистрацию на основе соглашений. Конфигурация на основе файлов полностью исключена. Это не должно стать препятствием для работы Simple Injector, поскольку, как известно из главы 12, данного метода конфигурирования следует избегать. Если ваше приложение все же требует позднего связывания, то конфигурирование на основе применения файлов, как выяснится далее в этом разделе, нетрудно добавить и своими силами.

Simple Injector позволяет смешивать все три подхода. В этом разделе будут рассмотрены способы использования каждого из трех источников конфигурации.

Конфигурирование контейнера с помощью конфигурации в виде кода

В разделе 14.1 было получено некоторое представление об имеющемся в Simple Injector конфигурационном API со строгой типизацией. Здесь он будет рассмотрен подробнее.

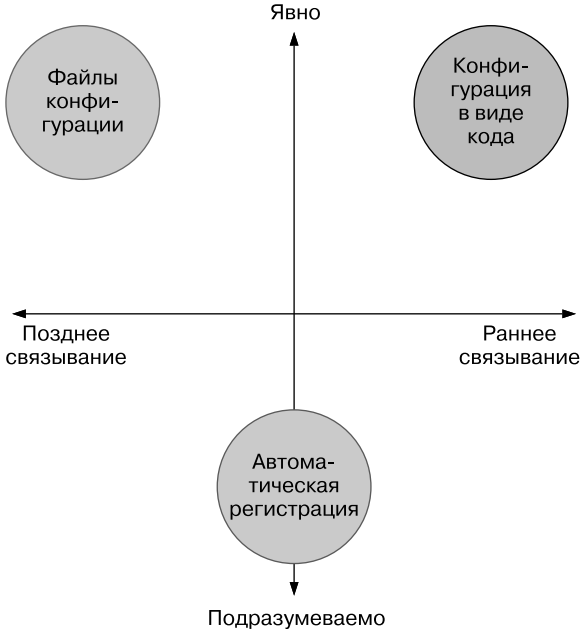


Рис. 14.2. Наиболее распространенные способы конфигурирования DI-контейнеров, расставленные по шкалам их явности-неявности и времени связывания

Все конфигурации в Simple Injector используют API, предоставленный классом `Container`. Чаще всего задействуется уже встречавшийся нам метод `Register`:

```
container.Register<IIngredient, SauceBéarnaise>();
```

Из-за потребности вести программирование с прицелом на создание интерфейсов большинство ваших компонентов будут зависеть от абстракций. Следовательно, основная часть компонентов станет регистрироваться под соответствующей им абстракцией. Если тип компонента находится на вершине графа объектов, то нет ничего необычного в том, что он разрешается по своему типу, а не по абстракции. К примеру, по конкретному типу разрешаются MVC-контроллеры.

СОВЕТ

Несмотря на то что Simple позволяет выполнять разрешение конкретных незарегистрированных типов, нужно обеспечить явную регистрацию самых верхних типов. Тогда Simple Injector сможет проверить полный граф объектов, включая эти корневые типы. Зачастую это позволяет обнаружить ошибки конфигурации, которые в противном случае были бы скрыты. Подробнее проверка рассматривается в подразделе 14.2.4.

Как правило, тип регистрируется либо по своей абстракции, либо по конкретному типу, но никак не вместе. Однако из этого правила есть исключение. В Simple Injector

регистрация компонента как по конкретному типу, так и по абстракции выполняется простым добавлением еще одной регистрации:

```
container.Register<IIngredient, SauceBéarnaise>();
container.Register<SauceBéarnaise>();
```

Вместо регистрации класса только в качестве `IIngredient` его можно зарегистрировать как самого по себе, так и в качестве реализуемого интерфейса. Это позволит контейнеру выполнять разрешения запросов в отношении как `SauceBéarnaise`, так и `IIngredient`.

ПРИМЕЧАНИЕ

> В подразделе 13.1.2 рассматривалась опасность рваных жизненных циклов при двойном вызове RegisterType для какого-то компонента в контейнере Autofac. При использовании Simple Injector такой проблемы нет. Внутреннее устройство Simple Injector таково, что он автоматически предотвращает дублирование регистраций SauceBéarnaise и не дает превращать жизненный цикл этого компонента в рваный.

В настоящих приложениях всегда имеется более одной абстракции, которые требуется отображать, следовательно, нужно отконфигурировать несколько отображений. Это делается с помощью нескольких вызовов `Register`:

```
container.Register<IIngredient, SauceBéarnaise>();
container.Register<ICourse, Course>();
```

В этом примере выполняется отображение `IIngredient` на `SauceBéarnaise` и `ICourse` на `Course`. Перекрытия типов здесь нет, поэтому происходящее не вызывает никаких сомнений. Что получится, если одну и ту же абстракцию зарегистрировать несколько раз?

```
container.Register<IIngredient, SauceBéarnaise>();
container.Register<IIngredient, Steak>();
```

Выдает исключение

Здесь происходит двойная регистрация `IIngredient`, причем при второй регистрации выдается исключение со следующим сообщением: Type IIngredient has already been registered. If your intention is to resolve a collection of IIngredient implementations, use the Collection. Register overloads. For more information, see https://simpleinjector.org/coll1 (Тип IIngredient уже был зарегистрирован. Если вы хотели разрешить коллекцию реализаций IIngredient, нужно было воспользоваться переопределением Collection.Register. Дополнительную информацию можно найти по адресу https://simpleinjector.org/coll1).

В отличие от большинства других DI-контейнеров Simple Injector, как показывает предыдущий фрагмент кода, не позволяет выстраивать цепочку регистраций для создания последовательности типов. Его API явно отделяет регистрацию последовательностей от отображения отдельно взятых абстракций[1]. Вместо выполнения

[1] Подробные причины этого раскрываются в публикации, размещенной по адресу https://simpleinjector.org/separate-collections.

нескольких вызовов `Register` Simple Injector заставляет воспользоваться методами регистрации свойства `Collection`, такими как метод `Collection.Register`:

```
container.Collection.Register<IIngredient>(
    typeof(SauceBéarnaise),
    typeof(Steak));
```

ПРИМЕЧАНИЕ

> Там, где мы используем понятие «последовательность», в Simple Injector применяется термин «коллекция». В большинстве случаев они могут заменять друг друга.

В этом примере в одном вызове регистрируются сразу все ингредиенты. Для добавления реализаций в последовательность ингредиентов можно также воспользоваться методом `Collection.Append`:

```
container.Collection.Append<IIngredient, SauceBéarnaise>();
container.Collection.Append<IIngredient, Steak>();
```

Предыдущие регистрации позволяют любому компоненту, зависящему от `IEnumerable<IIngredient>`, получить внедренную последовательность ингредиентов. Simple Injector неплохо справляется с несколькими конфигурациями одной и той же абстракции (к этой теме мы вернемся в разделе 14.4).

Доступны и более совершенные варианты конфигурирования Simple Injector, но с помощью показанных здесь методов можно сконфигурировать целое приложение. Чтобы избавиться от лишней работы по конфигурированию контейнера, нужно обратиться к подходам, основанным на применении соглашений, и воспользоваться автоматической регистрацией.

Конфигурирование контейнера с помощью автоматической регистрации

Регистрации во многих случаях будут похожи друг на друга. Сопровождение регистраций может превратиться в весьма утомительное занятие, а явная регистрация буквально каждого компонента, как выяснилось в подразделе 12.3.3, может стать крайне нерациональным подходом.

Рассмотрим библиотеку, содержащую множество реализаций `IIngredient`. Можно сконфигурировать каждый класс по отдельности, но тогда получится постоянно меняющийся список экземпляров `Type`, предоставленный методу `Collection.Register`. Хуже того, чтобы при каждом добавлении реализация `IIngredient` стала доступной, ее также нужно явно регистрировать с помощью `Container`. Следовало бы заявить, что должны быть зарегистрированы все реализации `IIngredient`, найденные в заданной сборке.

Такая возможность предоставляется при использовании переопределений методов `Register` и `Collection.Register`. Конкретные переопределения позволяют указывать сборку и конфигурировать все выбранные классы из нее, задействуя для

этого всего одну инструкцию. Чтобы получить экземпляр сборки `Assembly`, можно воспользоваться репрезентативным классом, в данном случае `Steak`:

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

container.Collection.Register<IIngredient>(ingredientsAssembly);
```

В предыдущем примере выполняется безусловное конфигурирование всех реализаций интерфейса `IIngredient`, но вы можете предоставить фильтры, позволяющие выбрать определенный поднабор этих реализаций. Сканирование на основе соглашений, позволяющее добавить только те классы, имена которых начинаются с `Sauce`, выглядит следующим образом:

```
Assembly assembly = typeof(Steak).Assembly;

var types = container.GetTypesToRegister<IIngredient>(assembly)
    .Where(type => type.Name.StartsWith("Sauce"));

container.Collection.Register<IIngredient>(types);
```

Здесь используется метод `GetTypesToRegister`, выполняющий поиск типов без их регистрации. Это позволяет при фильтрации выбора применять предикат. Теперь методу `Collection.Register` вместо списка сборок можно предоставить список экземпляров `Type`.

Кроме выбора из сборки приемлемых типов, у автоматической регистрации есть и другая функция — определение нужного отображения. В предыдущих примерах, чтобы зарегистрировать все выбранные типы за этим интерфейсом, применялся метод `Collection.Register`. Но иногда могут потребоваться другие соглашения. Предположим, что вместо интерфейсов используются базовые абстрактные классы и нужно зарегистрировать за базовым типом все имеющиеся в сборке типы, чьи имена заканчиваются на `Policy`:

```
Assembly policiesAssembly = typeof(DiscountPolicy).Assembly;

var policyTypes =
    from type in policiesAssembly.GetTypes()
    where type.Name.EndsWith("Policy")
    select type;

foreach (Type type in policyTypes)
{
    container.Register(type.BaseType, type);
}
```

Получение всех имеющихся в сборке типов

Применение фильтра по суффиксу Policy

Регистрация каждого компонента политики за его базовым классом

В этом примере почти не применяется API Simple Injector. Вместо этого для фильтрации и получения ожидаемых типов используется отражение и API LINQ, которые предоставляет среда .NET.

Несмотря на ограниченность имеющегося в Simple Injector API на основе соглашений, все же за счет использования API среды .NET регистрация на основе соглашений выполняется удивительно легко. Существующий в Simple Injector API на

основе соглашений фокусируется в основном на регистрации последовательностей и обобщенных типов. Но, когда речь заходит об обобщениях, ситуация меняется, поэтому в Simple Injector есть явно выраженная поддержка регистрации типов на основе обобщенных абстракций, которая будет рассмотрена в следующем подразделе.

Автоматическая регистрация обобщенных абстракций

В ходе изучения главы 10 рассматривалась переделка большого, крайне неприглядного интерфейса `IProductService` в интерфейс `ICommandService<TCommand>`, показанный в листинге 10.12. Посмотрим на эту абстракцию еще раз:

```
public interface ICommandService<TCommand>
{
    void Execute(TCommand command);
}
```

Как известно из главы 10, все командные граничные объекты представляли тот или иной вариант использования и для каждого такого варианта предусматривалась всего одна реализация. В качестве примера в листинге 10.8 был показан сервис `AdjustInventoryService`. В нем реализован вариант применения «корректировка товарных запасов». В листинге 14.2 этот класс показан еще раз.

Листинг 14.2. AdjustInventoryService из главы 10

```
public class AdjustInventoryService : ICommandService<AdjustInventory>
{
    private readonly IInventoryRepository repository;

    public AdjustInventoryService(IInventoryRepository repository)
    {
        this.repository = repository;
    }

    public void Execute(AdjustInventory command)
    {
        var productId = command.ProductId;

        ...
    }
}
```

В любой сложной системе могут реализовываться сотни вариантов использования. Такие системы можно рассматривать в качестве идеальных кандидатов для автоматической регистрации. С применением Simple Injector вряд ли можно придумать что-либо еще более простое, чем то, что показано в листинге 14.3.

Листинг 14.3. Автоматическая регистрация реализаций ICommandService<TCommand>

```
Assembly assembly = typeof(AdjustInventoryService).Assembly;

container.Register(typeof(ICommandService<>), assembly);
```

В отличие от листинга 14.2, в котором использовался метод `Collection.Register`, здесь снова задействован метод `Register`. Дело в том, что у запрашиваемого командного сервиса всегда будет только одна реализация, а внедрять последовательность командных сервисов вам не потребуется.

С помощью открытого обобщенного интерфейса Simple Injector выполняет последовательный перебор элементов списка всех типов, имеющихся в сборке, и регистрирует типы, реализующие закрытую обобщенную версию `ICommandService<TCommand>`. Это означает, что класс `AdjustInventoryService` проходит регистрацию, потому что в нем реализуется `ICommandService<AdjustInventory>`, являющийся закрытой обобщенной версией `ICommandService<TCommand>`.

Но будут зарегистрированы не все реализации `ICommandService<TCommand>`. Simple Injector пропускает открытые обобщенные реализации, декораторы и компоновщики, поскольку зачастую для них требуется особая регистрация. Этот вопрос будет рассмотрен в разделе 14.4.

Метод `Register` принимает массив `params` экземпляров `Assembly`, поэтому под одно соглашение можно предоставить сколько угодно сборок. Вряд ли можно увидеть что-то странное в сканировании папки на наличие сборок и предоставлении их всех для реализации дополнительной функциональности. Тогда надстройки можно будет добавлять без перекомпиляции основного приложения (посмотрите, к примеру, публикацию https://simpleinjector.org/registering-plugins-dynamically). Это один из способов реализации позднего связывания, а еще одним способом является использование файлов конфигурации.

Конфигурирование контейнера с помощью файлов конфигурации

Когда требуется изменить конфигурацию без перекомпиляции приложения, вполне приемлемым вариантом может стать применение файлов конфигурации. Самым естественным способом работы с файлами конфигурации является их встраивание в стандартный файл конфигурации .NET-приложения. Но если нужно изменить конфигурацию Simple Injector независимо от стандартного файла .config, то наряду с этим можно воспользоваться отдельным файлом конфигурации.

СОВЕТ

> Как известно из подраздела 12.2.1, файлы конфигурации должны использоваться только для тех типов DI-конфигураций, которые требуют позднего связывания, а для всех остальных типов и прочих частей конфигурации следует предпочесть конфигурацию в виде кода или автоматической регистрации.

Как говорилось в начале раздела, явной поддержки конфигурации на основе файлов в Simple Injector нет. Но за счет применения встроенной в среду .NET системы конфигурирования с загрузкой регистраций из файла конфигурации проблем не возникает. Для этого можно определить собственную структуру конфигурации, отображающую абстракции на реализации. Рассмотрим простой пример, в котором интерфейс `IIngredient` отображается на класс `Steak` (листинг 14.4).

Листинг 14.4. Простое отображение IIngredient на Steak с использованием файла конфигурации

```
{
    "registrations": [
        {
            "service":
                "Ploeh.Samples.MenuModel.IIngredient, Ploeh.Samples.MenuModel",
            "implementation":
                "Ploeh.Samples.MenuModel.Steak, Ploeh.Samples.MenuModel"
        }
    ]
}
```

ПРИМЕЧАНИЕ

Структура этого примера конфигурации очень похожа на структуру в контейнере Autofac, поскольку это вполне естественный формат.

Элемент `registrations` представляет собой JSON-массив элементов `registration`. В предыдущем примере содержалась одна регистрация, но можно добавить любое количество элементов регистрации. В каждом элементе нужно указать конкретный тип с атрибутом `implementation`. Для отображения класса `Steak` на `IIngredient` можно воспользоваться опциональным атрибутом `service`.

Используя встроенную в среду .NET Core систему конфигурирования, можно загрузить файл конфигурации и выполнить последовательный перебор его записей. После чего определенные регистрации добавляются в контейнер:

```
var config = new ConfigurationBuilder()
    .AddJsonFile("simpleinjector.json")
    .Build();

var registrations = config
    .GetSection("registrations").GetChildren();

foreach (var reg in registrations)
{
    container.Register(
        Type.GetType(reg["service"]),
        Type.GetType(reg["implementation"]));
}
```

Загрузка файла конфигурации simpleinjector.json с использованием системы конфигурирования среды .NET Core. При установках по умолчанию файл конфигурации будет находиться в корневом каталоге приложения

Загрузка списка регистраций из файла конфигурации

Последовательный перебор элементов списка регистраций. Добавление каждой регистрации к Simple Injector с использованием предоставляемого сервиса и типов реализации в соответствии с определениями в файле конфигурации

Вариант с использованием файла конфигурации подходит для тех случаев, когда нужно изменить конфигурацию одного или нескольких компонентов без перекомпиляции приложения, но, так как он хрупок, его применение следует ограничить исключительно такими случаями. Для основной части конфигурации контейнера

следует задействовать либо автоматическую регистрацию, либо конфигурацию в виде кода.

В этом разделе был представлен DI-контейнер Simple Injector и продемонстрированы основные механизмы: способы конфигурирования контейнера `Container` и последующее использование созданного контейнера для разрешения сервисов. Это разрешение выполняется довольно просто — за счет всего лишь одного вызова метода `GetInstance`, поэтому вся сложность заключается в конфигурировании контейнера. Оно может выполняться несколькими различными способами, включая императивный код и файлы конфигурации.

Пока мы рассматривали наиболее общий API, неохваченными остались гораздо более сложные области. Одной из наиболее важных тем является управление временем жизни компонентов.

14.2. Управление временем жизни

Управление временем жизни, включая наиболее распространенные концептуальные жизненные циклы: Singleton, Scoped и Transient, рассматривалось в главе 8. В Simple Injector поддерживается отображение на эти три жизненных цикла. Жизненные циклы, показанные в табл. 14.2, доступны в качестве части API.

Таблица 14.2. Жизненные циклы в Simple Injector

Название в Simple Injector	Паттерн	Комментарии
Transient	Transient	Это жизненный цикл по умолчанию. Transient-экземпляры не отслеживаются контейнером и поэтому никогда не ликвидируются. Используя свои диагностические сервисы, Simple Injector предупреждает, если ликвидируемый компонент зарегистрирован в качестве Transient-экземпляра
Singleton	Singleton	Ликвидация происходит при ликвидации самого контейнера
Scoped	Scoped	Шаблон для жизненных циклов, допускающих ограничения области видимости экземпляров. Жизненный цикл Scoped определяется базовым классом ScopedLifestyle, при этом существует несколько реализаций ScopedLifestyle. Чаще всего для приложений среды .NET Core используется AsyncScopedLifestyle.Экземпляры отслеживаются по времени жизни областей видимости и ликвидируются при ликвидации самой области видимости

СОВЕТ

Самым безопасным, но не всегда самым эффективным является жизненный цикл Transient. Более результативным вариантом для потокобезопасных сервисов является жизненный цикл Singleton, но при этом не следует забывать о явной регистрации таких сервисов.

Реализации Transient и Singleton в Simple Injector являются эквивалентами общих жизненных циклов, рассмотренных в главе 8, поэтому здесь потратим на них не слишком много времени. Вместо этого в данном разделе поговорим о способах определения жизненных циклов для компонентов, выражаемых в коде. Рассмотрим также применяемое в Simple Injector понятие охватывающей области видимости (ambient scoping) и раскроем способность последней упрощать работу с контейнером. Затем изучим существующую в Simple Injector возможность проверки и диагностики его конфигурации с целью предотвращения самых распространенных ошибок конфигурирования. Усвоив материал раздела, вы сможете воспользоваться жизненными циклами Simple Injector в собственном приложении. Начнем со способов конфигурирования жизненных циклов для компонентов.

14.2.1. Конфигурирование жизненных циклов

В этом разделе рассмотрим способы управления жизненными циклами средствами Simple Injector. Конфигурирование жизненных циклов является частью регистрации компонентов. Все делается очень просто:

```
container.Register<SauceBéarnaise>(Lifestyle.Singleton);
```

В примере конкретный класс `SauceBéarnaise` настраивается на жизненный цикл Singleton, поэтому при каждом запросе `SauceBéarnaise` будет возвращаться один и тот же экземпляр. Если нужно отобразить абстракцию на конкретный класс с определенным жизненным циклом, можно воспользоваться обычным переопределением метода `Register` с двумя обычными аргументами, предоставив ему `Lifestyle.Singleton`:

```
container.Register<IIngredient, SauceBéarnaise>(Lifestyle.Singleton);
```

Хотя Transient является жизненным циклом по умолчанию, его можно задать и явно. Эти два примера при конфигурации по умолчанию эквивалентны[1]:

```
container.Register<IIngredient, SauceBéarnaise>(
    Lifestyle.Transient);

container.Register<IIngredient, SauceBéarnaise>();
```

Явное предоставление регистрации жизненного цикла Transient

Transient является жизненным циклом по умолчанию, и указание на его применение может быть опущено

Конфигурирование жизненных циклов для регистраций на основе соглашений может выполняться несколькими способами. Например, при регистрации

[1] Жизненный цикл по умолчанию может быть изменен установкой для Container.Options.DefaultLifestyle другого, отличного от Lifestyle.Transient, значения.

последовательности один из вариантов — предоставить методу `Collection.Register` список экземпляров `Registration`:

```
Assembly assembly = typeof(Steak).Assembly;

var types = container.GetTypesToRegister<IIngredient>(assembly);

container.Collection.Register<IIngredient>(
    from type in types
    select Lifestyle.Singleton.CreateRegistration(type, container));
```

`Lifestyle.Singleton` можно использовать для определения жизненного цикла всех регистраций в соглашении. В данном примере для всех регистраций `IIngredient` определен жизненный цикл Singleton, поскольку все они предоставлены переопределению метода `Collection.Register` в качестве экземпляра `Registration`[1].

ПРИМЕЧАНИЕ

> Registration является классом контейнера Simple Injector, отвечающим за создание дерева выражений, описывающего создание типа на основе его жизненного цикла. Объекты Registration создаются контейнером Simple Injector при вызове большинства переопределений метода Register с помощью внутреннего механизма контейнера, но их можно создать и напрямую. Данная возможность пригодится в подобных сценариях.

Когда дело доходит до конфигурирования жизненных циклов для компонентов, можно реализовать множество вариантов. Во всех случаях это делается в сугубо декларативной манере. Хотя конфигурирование, как правило, выполняется довольно просто, не следует забывать, что некоторые жизненные циклы связаны с наличием в системе долгоживущих объектов, потребляющих ресурсы на всем протяжении своего существования.

14.2.2. Высвобождение компонентов

Как известно из подраздела 8.2.2, когда работа с объектами завершилась, важно высвободить их. Так же как в Autofac, в Simple Injector нет явно определенного метода `Release`, вместо него используются понятия областей видимости (scopes). Область видимости может рассматриваться как кэш для конкретного запроса. На рис. 14.3. показано, как ею определяется граница, в пределах которой компоненты могут применяться повторно.

Областью видимости `Scope` определяется кэш, которым можно воспользоваться в определенный период времени или с определенной целью. Наиболее очевидным примером может послужить веб-запрос. Когда запрашивается `Scope`, то запрашива-

---

[1] Еще один интересный вариант — переопределение имеющегося в Simple Injector значения ILifestyleSelectionBehavior по умолчанию. См. https://simpleinjector.org/xtpls.

ется имеющийся в ней компонент, и запрашивающий всегда получает один и тот же экземпляр. От настоящих Singleton-элементов это отличается тем, что при запросе из второй области видимости будет получен другой экземпляр.

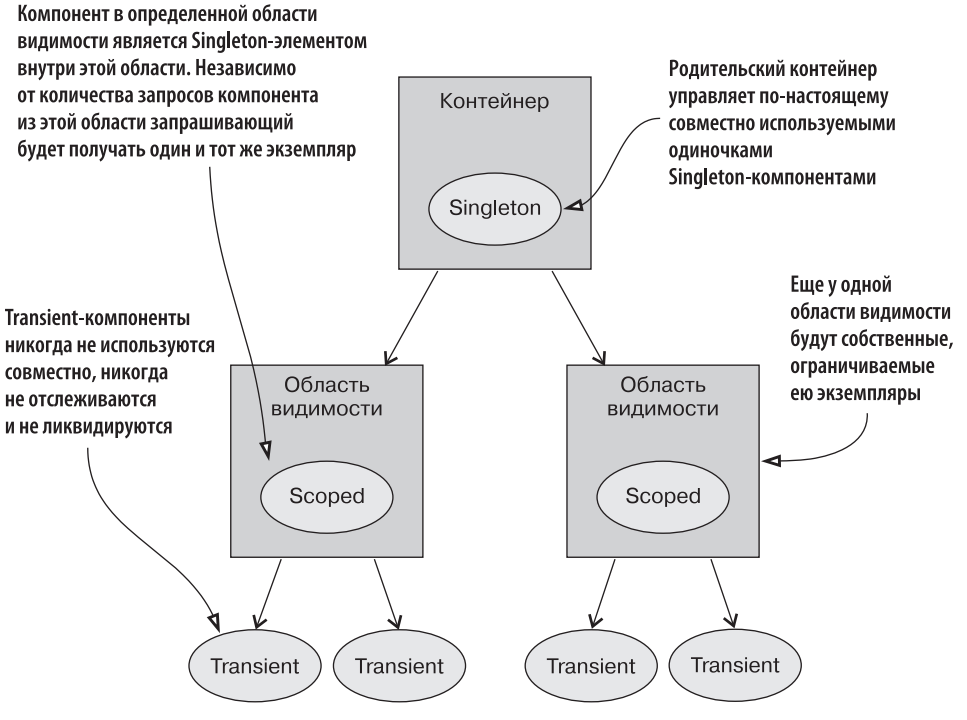


Рис. 14.3. Имеющаяся в Simple Injector область видимости Scope действует как кэш для конкретного запроса, который позволяет совместно использовать компоненты в определенный период времени или с определенной целью

Важной особенностью областей видимости является возможность правильного высвобождения компонентов при их выходе из употребления. Новая область видимости создается с помощью метода `BeginScope` конкретной реализации `ScopedLifestyle`, а с вызовом ее метода `Dispose` все соответствующие компоненты высвобождаются:

```
using (AsyncScopedLifestyle.BeginScope(container))
{
    IMeal meal = container.GetInstance<IMeal>();

    meal.Consume();
}
```

Создание области видимости для контейнера

Разрешение meal из контейнера в контексте созданной области видимости

Потребление meal

Высвобождение meal за счет прекращения использования блока

В этом примере показано, как вместо разрешения из экземпляра `Scope` `IMeal` разрешается из экземпляра `Container`. Это не опечатка — контейнер автоматически знает, в какой активной области видимости он действует. В следующем разделе этот вопрос рассматривается подробнее.

СОВЕТ

> Simple Injector содержит несколько NuGet-пакетов, помогающих объединить его с самыми распространенными средами выполнения приложений. Некоторые из этих пакетов автоматически обеспечивают заключение веб-запроса в область видимости. Чтобы определить самый подходящий способ интеграции Simple Injector с выбранной вами средой выполнения, обратитесь к руководству (https://simpleinjector.org/integration).

В предыдущем примере новая область видимости создавалась вызовом метода `BeginScope` в отношении соответствующего жизненного цикла Scoped. Возвращаемое значение реализует `IDisposable`, поэтому его можно заключить в блок `using`.

Когда работа с областью видимости завершится, ее можно ликвидировать вместе с блоком `using`. Это происходит автоматически по выходу из блока, можно также выбрать ее явную ликвидацию, вызвав метод `Dispose`. Когда ликвидируется область видимости, высвобождаются и все компоненты, созданные за время ее существования. В примере это означает, что высвобождается граф объектов `meal`.

ПРИМЕЧАНИЕ

> Следует понимать, что высвобождение ликвидируемого компонента — это не то же самое, что его ликвидация. Это сигнал контейнеру, что данный компонент может быть ликвидирован. Если это Scoped-компонент, он будет ликвидирован, если же его жизненный цикл — Singleton, он будет активно существовать до ликвидации контейнера.

Ранее в этом разделе было показано, как компоненты конфигурируются под жизненные циклы Singleton или Transient. Конфигурирование компонента под жизненный цикл Scoped выполняется аналогично:

```
container.Register<IIngredient, SauceBéarnaise>(Lifestyle.Scoped);
```

Чтобы задать компоненту время жизни, совпадающее по продолжительности с временем существования области видимости, в которой был создан его экземпляр, то подобно тому, как использовались значения `Lifestyle.Singleton` и `Lifestyle.Transient`, можно применить значение `Lifestyle.Scoped`. Но простой вызов приведет к выдаче контейнером исключения со следующим сообщением: To be able to use the Lifestyle.Scoped property, please ensure that the container is configured with a default scoped lifestyle by setting the Container.Options.DefaultScopedLifestyle property with the required scoped lifestyle for your type of application. For more information, see https://simpleinjector.org/scoped (Чтобы можно было воспользоваться свойством Lifestyle.Scoped, пожалуйста, убедитесь, что контейнер настроен на использование жизненного цикла с областью види-

мости по умолчанию установкой свойству Container.Options.DefaultScopedLifestyle значения жизненного цикла с областью видимости, требующейся вашему приложению. Дополнительную информацию можно найти по адресу https://simpleinjector.org/scoped).

Чтобы получить возможность применять значение `Lifestyle.Scoped`, Simple Injector нужна установка значения свойству `Container.Options.DefaultScopedLifestyle`. В Simple Injector имеется несколько реализаций `ScopedLifestyle`, которые порой учитывают специфику среды выполнения. Это означает, что требуется явное конфигурирование реализации `ScopedLifestyle`, наиболее соответствующее типу приложения. Для приложений ASP.NET Core подходящим `ScopedLifestyle` является `AsyncScopedLifestyle`, конфигурирование под который имеет следующий вид:

```
var container = new Container();

container.Options.DefaultScopedLifestyle =
    new AsyncScopedLifestyle();

container.Register<IIngredient, SauceBéarnaise>(
    Lifestyle.Scoped);
```

Прежде чем выполнять в контейнере какие-либо регистрации, нужно зарегистрировать AsyncScopedLifestyle в качестве жизненного цикла с областью видимости по умолчанию

Для создания регистрации с жизненным циклом с областью видимости уже можно воспользоваться значением Lifestyle.Scoped

СОВЕТ

> Поскольку в Simple Injector используется понятие охватывающих областей видимости, разрешение всегда может проводить непосредственно из Container, не опасаясь случайных утечек памяти, возникающих, если это делается с помощью таких DI-контейнеров, как Autofac. Если Scoped-зависимость будет разрешаться из контейнера при отсутствии активной области видимости, Simple Injector выдаст исключение с подробным описанием причины.

По своей природе Singleton-компоненты никогда не высвобождаются до истечения времени жизни самого контейнера. Если же контейнер вам больше не нужен, можно высвободить и эти компоненты. Это делается ликвидацией самого контейнера:

```
container.Dispose();
```

На практике это не настолько важно, как ликвидация области видимости, поскольку время жизни контейнера тесно коррелирует с временем жизни поддерживаемого им приложения. Обычно контейнер сохраняется, пока выполняется приложение, поэтому он ликвидируется только при завершении работы приложения. В этом случае память будет восстановлена операционной системой.

Ранее говорилось, что при использовании Simple Injector объекты всегда разрешаются из контейнера, а не из области видимости. Работоспособность такого подхода обусловливается тем, что области видимости в Simple Injector являются охватывающими. Охватывающие области видимости рассмотрим в следующем разделе.

14.2.3. Охватывающие области видимости

Предыдущий пример создания и ликвидации области видимости при использовании Simple Injector показывает, что экземпляры всегда могут разрешаться из `Container`, даже если это касается Scoped-экземпляров. В следующем примере эта особенность показана еще раз:

```
using (AsyncScopedLifestyle.BeginScope(container))
{
    IMeal meal = container.GetInstance<IMeal>();
    meal.Consume();
}
```

При использовании Simple Injector разрешение всегда проводится из контейнера

Этим раскрывается интересная особенность Simple Injector: экземпляры областей видимости являются охватывающими и глобально доступны в своем рабочем контексте. Соответствующее поведение раскрывается в листинге 14.5.

Листинг 14.5. Охватывающие области видимости в Simple Injector

```
var container = new Container();

container.Options.DefaultScopedLifestyle =
    new AsyncScopedLifestyle();

Scope scope1 = Lifestyle.Scoped
    .GetCurrentScope(container);

using (Scope scope2 =
    AsyncScopedLifestyle.BeginScope(container))
{
    Scope scope3 = Lifestyle.Scoped
        .GetCurrentScope(container);
}

Scope scope4 = Lifestyle.Scoped
    .GetCurrentScope(container);
```

Запрос текущей активной области видимости для сконфигурированного жизненного цикла с областью видимости, в данном случае для AsyncScopedLifestyle. Поскольку активной области видимости еще нет, этот метод возвращает значение null

Когда при наличии активной области видимости вызывается метод GetCurrentScope, он возвращает эту область видимости. В данном случае значение scope3 будет равно значению scope2

После ликвидации область видимости становится незарегистрированной. Из-за этого вызов GetCurrentScope опять возвращает null

Это поведение похоже на поведение класса `TransactionScope`, принадлежащего среде .NET (https://mng.bz/jrQP). Когда операция охватывается `TransactionScope`, все подключения к базе данных в рамках данной операции автоматически становятся частью этой же транзакции.

ВНИМАНИЕ

Охватывающие области видимости не следует путать с антипаттерном «Окружающий контекст», который предоставляет коду приложения, находящемуся за пределами корня композиции, глобальный доступ к нестабильной зависимости или к ее поведению. Поскольку область видимости DI-контейнера используется только внутри корня композиции, у охватывающих областей видимости не возникает проблем, свойственных антипаттерну «Окружающий контекст».

Вообще-то метод `GetCurrentScope` практически не применяется. Когда дело доходит до разрешения экземпляров, `Container` задействует его скрытно в ваших интересах. И все же он четко демонстрирует, что экземпляры `Scope` могут быть извлечены из контейнера и окажутся доступными.

Такие реализации `ScopedLifestyle`, как только что использованная `AsyncScopedLifestyle`, сохраняют созданный ими экземпляр `Scope` для последующего применения, что позволяет извлекать его в том же самом контексте. Конкретная реализация `ScopedLifestyle` определяет, выполняется ли код в том же контексте. Например, `AsyncScopedLifestyle` сохраняет `Scope` на внутреннем уровне в `System.Threading.AsyncLocal<T>`(https://mng.bz/WeD1). Это позволяет областям видимости перетекать от метода к методу, даже если асинхронный метод, как показано в следующем примере, продолжает выполняться в другом потоке:

```
using (AsyncScopedLifestyle.BeginScope(container))
{
    IMeal meal = container.GetInstance<IMeal>();

    await meal.Consume();

    meal = container.GetInstance<IMeal>();
}
```

Этот асинхронный вызов может привести к тому, что остальной код метода продолжит выполнение в другом потоке

Граф объектов гарантированно будет разрешен в той же области видимости

Возможно, поначалу охватывающие области видимости покажутся чем-то необычным, но их использование зачастую упрощает работу с Simple Injector. Например, не нужно беспокоиться о возникновении утечек памяти при разрешениях из контейнера, поскольку Simple Injector четко справляется с ситуацией, блюдя ваши интересы. Экземпляры `Scope` никогда не будут кэшироваться в корневом контейнере, а вот с другими контейнерами, рассматриваемыми в данной книге, в этом следует проявлять осмотрительность. Еще одна область, в которой преуспел Simple Injector, — это возможность обнаружения наиболее распространенных ошибок конфигурации.

14.2.4. Диагностика контейнера для выявления наиболее распространенных проблем, связанных с временем жизни

По сравнению с чистой технологией DI регистрация и построение графа объектов в DI-контейнерах имеют более неявный характер. При этом легче допустить случайную ошибку конфигурирования контейнера. Поэтому у многих DI-контейнеров имеется функция, позволяющая проходить по всем регистрациям и проверять, могут ли они разрешаться, и контейнер Simple Injector не исключение.

Но возможность разрешения графа объектов не гарантирует правильности конфигурации, что может быть проиллюстрировано ловушкой с захваченной зависимостью (рассмотрена в подразделе 8.4.1), возникающей из-за неправильной конфигурации времени жизни компонента. Фактически большинство ошибок, возникающих в ходе работы с DI-контейнерами, являются следствием неверной конфигурации времени жизни.

Из-за того, что ошибки конфигурирования DI-контейнера часто встречаются и трудно отслеживаются, Simple Injector позволяет проверять свою конфигурацию, выходя за рамки простого создания экземпляров графов объектов, поддерживаемого большинством DI-контейнеров. Более того, Simple Injector сканирует графы объектов на наличие самых распространенных ошибок конфигурирования, одной из которых является захваченная зависимость.

Поэтому этап конфигурирования двухэтапного процесса Simple Injector (см. рис. 14.1) состоит из двух подэтапов. Этот процесс показан на рис. 14.4.

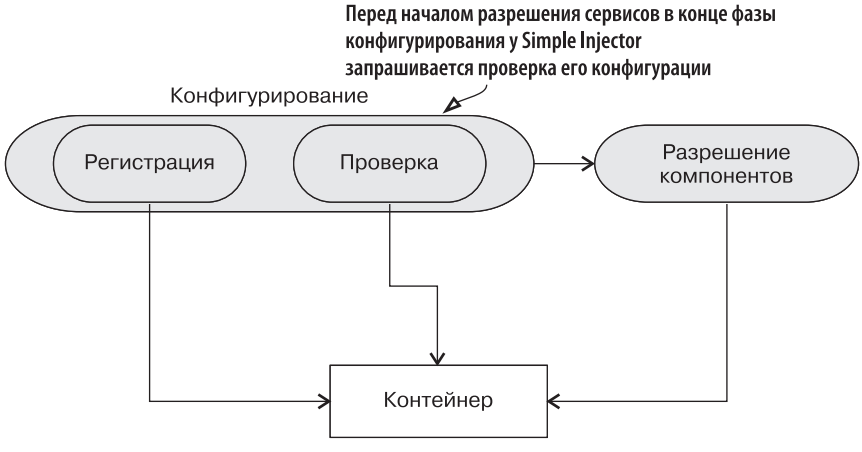


Рис. 14.4. Схема использования Simple Injector состоит из его конфигурирования, включая проверку, с последующим разрешением компонентов

Как показано в листинге 14.6, самый простой способ позволить Simple Injector провести диагностику и выявить ошибки конфигурирования заключается в вызове принадлежащего `Container` метода `Verify`.

Листинг 14.6. Проверка контейнера

```
var container = new Container();

container.Register<IIngredient, Steak>();

container.Verify();
```

Регистрация всех компонентов для полнофункционального приложения

Вызов Verify сразу же за последней регистрацией для обеспечения проверки всех регистраций

Предоставление контейнеру возможности обнаружения захваченных зависимостей

Одной из обнаруживаемых Simple Injector ошибок конфигурирования является захваченная зависимость. Посмотрим, как можно заставить метод `Verify` заметить захваченную зависимость, используя ингредиент `Mayonnaise` из подраздела 14.1.1. В его конструкторе имеется две зависимости:

```
public Mayonnaise(EggYolk eggYolk, SunflowerOil oil)
```

В листинге 14.7 регистрируется `Mayonnaise` с двумя зависимостями. Но здесь допускается ошибка конфигурирования, в результате которой `Mayonnaise` настраивается на жизненный цикл Singleton, а его зависимость `EggYolk` — на жизненный цикл Transient.

Листинг 14.7. Создание контейнеру условий для обнаружения захваченной зависимости

```
var container = new Container();

container.Register<EggYolk>(Lifestyle.Transient);
container.Register<Mayonnaise>(Lifestyle.Singleton);
container.Register<SunflowerOil>(Lifestyle.Singleton);

container.Verify();
```

Из-за того что яичный желток быстро становится не нужен, соответствующий компонент регистрируется с жизненным циклом Transient

Mayonnaise зависит от EggYolk, но совершенно случайно регистрируется с жизненным циклом Singleton

Поскольку все ранее сделанное является ошибкой конфигурирования, этот метод выдает исключение

При вызове метода `Register` контейнер Simple Injector выполняет только ряд элементарных проверок. В том числе проверяется, что тип не абстрактный, что у него есть открытый конструктор и т. п. Такие проблемы, как захваченные зависимости, на этом этапе не выявляются, поскольку регистрации могут проводиться в произвольном порядке. Например, в листинге 14.7 ингредиент `SunflowerOil` зарегистрирован после `Mayonnaise`, хотя `Mayonnaise` от него зависит. Это вполне допустимо. Проверку можно выполнить только после завершения конфигурирования. При запуске этого примера кода вызов функции `Verify` завершается сбоем с выдачей исключения, показывающего следующее сообщение:

```
The configuration is invalid. The following diagnostic warnings were reported:
— [Lifestyle Mismatch] Mayonnaise (Singleton) depends on EggYolk (Transient). See the Error
property for detailed information about the warnings. Please see https:// simpleinjector.org/diagnostics
how to fix problems and how to suppress individual warnings.
```

(Конфигурация неверна. Выданы следующие диагностические предупреждения:
— [Несоответствие жизненных циклов] Mayonnaise (Singleton) зависит от EggYolk (Transient). Для получения подробной информации о предупреждениях см. свойство Error. Пожалуйста, для изучения способов устранения проблем и подавления отдельных сообщений обратитесь к следующему источнику: https:// simpleinjector.org/diagnostics).

ПРИМЕЧАНИЕ

В Simple Injector захваченные зависимости называются несоответствием жизненных циклов (lifestyle mismatches).

Здесь интересно отметить то, что Simple Injector не позволяет Transient-зависимостям внедряться в Singleton-потребители. В этом он полностью противоположен контейнеру Autofac. При использовании Autofac подразумевается, что продолжительность жизни Transient-зависимостей равна времени существования

их потребителя, следовательно, такая ситуация никогда не рассматривается как захваченная зависимость. Поэтому жизненный цикл Transient называется в Autofac `InstancePerDependency`, чем довольно точно описывается поведение контейнера: ожидается, что каждая потребляемая зависимость, настроенная на жизненный цикл Transient, получит собственный экземпляр. Поэтому в качестве захваченной зависимости контейнер Autofac обнаруживает только внедрение Scoped-компонентов в Singleton-потребителей.

Временами рассмотренное здесь поведение полностью отвечает вашим интересам, но в большинстве случаев с ним трудно согласиться. Чаще всего ожидается, что у Transient-компонентов небольшая продолжительность жизни, а внедрение их в Singleton-потребители заставит компонент жить так же долго, как и само приложение. Поэтому девиз Simple Injector таков: «Лучше перестраховаться, чем сожалеть», исходя из которого и выдается исключение. Иногда, когда ситуация кажется более приемлемой, может возникать потребность в подавлении подобных предупреждений.

Подавление предупреждений при отдельных регистрациях

В случае, когда срок годности `EggYolk` нужно проигнорировать, Simple Injector позволяет подавлять проверку конкретной регистрации (листинг 14.8).

Листинг 14.8. Подавление диагностического предупреждения

```
var container = new Container();

Registration reg = Lifestyle.Transient
    .CreateRegistration<EggYolk>(container);

reg.SuppressDiagnosticWarning(
    DiagnosticType.LifestyleMismatch,
    justification: "I like to eat rotten eggs.");

container.AddRegistration(typeof(EggYolk), reg);

container.Register<Mayonnaise>(Lifestyle.Singleton);
container.Register<SunflowerOil>(Lifestyle.Singleton);

container.Verify();
```

Создание Transient-регистрации для EggYolk

Подавление диагностического предупреждения о захваченной зависимости с объяснением причины («Люблю есть тухлые яйца»)

Добавление регистрации к контейнеру

В `SuppressDiagnosticWarning` содержится обязательный аргумент обоснования `justification`. Он не используется `SuppressDiagnosticWarning`, но служит напоминанием, не позволяющим забыть задокументировать причину подавления предупреждения.

ПРИМЕЧАНИЕ

Описание всех доступных диагностических проверок можно найти по адресу https://simpleinjector.org/diagnostics.

На этом наш тур по управлению временем жизни в Simple Injector завершается. Компоненты могут быть сконфигурированы со смешанными жизненными циклами, и это справедливо даже при регистрации сразу нескольких реализаций одной и той же абстракции.

До сих пор контейнеру позволялось связывать зависимости, исходя из предположения, что все компоненты будут использовать внедрение через конструктор. Но бывают и исключения. В следующем разделе рассмотрим работу с классами, экземпляры которых должны создаваться особым образом.

14.3. Регистрация сложных API

Пока мы рассматривали только возможности конфигурирования компонентов, использующих внедрение через конструктор. Одним из многочисленных преимуществ подобного внедрения является то, что такие DI-контейнеры, как Simple Injector, могут легко разобраться в составлении и создании всех классов в графе зависимостей. Но эта ясность исчезает, когда поведение API складывается не самым лучшим образом.

В этом разделе будет показано, как работать с элементарными аргументами конструкторов и статическими фабриками. Все это требует особого внимания. Начнем с рассмотрения классов, принимающих в качестве аргументов конструктора элементарные типы, такие как строки и целые числа.

14.3.1. Конфигурирование элементарных зависимостей

Пока в потребителей внедряются абстракции, не возникает никаких проблем. Но ситуация усложняется, когда конструктор зависит от элементарного типа, такого как строка, число или перечисление. Именно это и происходит с реализациями доступа к данным, принимающими в качестве параметра конструктора строку подключения. Однако сам вопрос гораздо шире и касается всех строковых и числовых типов.

Концептуально регистрация строки или числа в качестве компонента контейнера не всегда сообразуется со здравым смыслом. В частности, когда используется автоматическое связывание, регистрация элементарных типов приводит к неопределенности. Возьмем, к примеру, строку. Одному компоненту может требоваться строка подключения к базе данных, а другому — путь доступа к файлу. Они концептуально разные, но, поскольку автоматическое связывание работает путем выбора зависимостей на основе их типа, возникает неопределенность. Поэтому Simple Injector блокирует регистрацию элементарных зависимостей. Рассмотрим следующий конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` является перечислением:

```
public enum Spiciness { Mild, Medium, Hot }
```

СОВЕТ

Общее правило гласит, что перечисления относятся к проблемному коду и должны быть переструктурированы в полиморфные классы[1]. Но для данного примера они являются вполне подходящей структурой.

Можно попробовать зарегистрировать `ChiliConCarne`, как показано в следующем примере. Но этот код работать не будет!

```
container.Register<ICourse, ChiliConCarne>();
```

При запуске данной строки кода будет выдано исключение со следующим сообщением: The constructor of type ChiliConCarne contains parameter 'spiciness' of type Spiciness, which cannot be used for constructor injection because it's a value type (В конструкторе типа ChiliConCarne содержится параметр 'spiciness', имеющий тип Spiciness, который не может использоваться для внедрения через конструктор, поскольку относится к типам значений).

Если нужно разрешить `ChiliConCarne` со средним (medium) значением остроты `Spiciness`, следует отказаться от автоматического связывания и вместо него воспользоваться делегатом[2]:

```
container.Register<ICourse>(() => new ChiliConCarne(Spiciness.Medium));
```

ПРИМЕЧАНИЕ

Метод Register является типобезопасным, но отключает автоматическое связывание.

Недостаток использования делегатов заключается в необходимости изменения регистрации при изменении конструктора `ChiliConCarne`. Например, когда к конструктору `ChiliConCarne` добавляется зависимость `IIngredient`, регистрация требует обновления:

```
container.Register<ICourse>(() =>
    new ChiliConCarne(
        Spiciness.Medium,
        container.GetInstance<IIngredient>()));
```

Регистрация делегата, создающего при вызове ChiliConCarne

Обратный вызов в контейнер для получения Iingredient и его самостоятельного внедрения в конструктор

---

1 *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 71.

2 Simple Injector позволяет переопределять свое поведение по умолчанию, проявляющееся в запрещении элементарных типов, путем замены реализации IDependencyInjectionBehavior по умолчанию. Но данная тема выходит за рамки книги. Подробности найдете по адресу https:/simpleinjector.org/xtppi.

Из-за необходимости дополнительного сопровождения корня композиции и отсутствия автоматического связывания использование делегатов не позволяет Simple Injector проверять правильность отношений между `ChiliConCarne` и его зависимостью `IIngredient`. Делегат скрывает факт существования этой зависимости. Неприятности из-за этого возникают нечасто, однако усложняется диагностика проблем, вызванных ошибками конфигурирования. Из-за этих недостатков более удобным решением будет извлечение элементарных зависимостей в граничные объекты.

14.3.2. Извлечение элементарных зависимостей в граничные объекты

В подразделе 10.3.3 рассматривался способ снижения остроты последствий нарушения принципа открытости/закрытости в результате применения `IProductService` за счет введения граничных объектов. Но граничные объекты отлично подходят и для устранения неопределенности. Например, острота `Spiciness` нового блюда может быть описана более общим понятием вкуса (flavoring). `Flavoring` может включать другие свойства, допустим соленость, то есть и `Spiciness`, и соленость можно заключить в класс вкуса `Flavoring`:

```
public class Flavoring
{
    public readonly Spiciness Spiciness;
    public readonly bool ExtraSalty;

    public Flavoring(Spiciness spiciness, bool extraSalty)
    {
        this.Spiciness = spiciness;
        this.ExtraSalty = extraSalty;
    }
}
```

Как известно из подраздела 10.3.3, наличие у граничных объектов всего одного параметра — вполне нормальное явление. Это помогает устранить неопределенность, и не только на техническом уровне. Имя граничного объекта может служить описанием того, чем ваш код занят на функциональном уровне, что вполне изящно делается в классе `Flavoring`. С введением граничного объекта `Flavoring` появляется возможность автоматически связывать любую реализацию `ICourse`, требующую указания какого-либо вкуса:

```
var flavoring = new Flavoring(Spiciness.Medium, extraSalty: true);
container.RegisterInstance<Flavoring>(flavoring);

container.Register<ICourse, ChiliConCarne>();
```

Этот код создает единственный экземпляр класса `Flavoring`, который становится объектом конфигурации для блюд. Поскольку в наличии будет только один экземпляр `Flavoring`, его можно зарегистрировать в Simple Injector, воспользовавшись `RegisterInstance`.

СОВЕТ

Не внедряйте граничные объекты, действие которых в качестве объектов конфигурации распространяется на все приложение. Лучше применять более конкретизированные граничные объекты, содержащие только значения, необходимые конкретному потребителю. Тогда станет понятнее, какие значения конфигурации используются компонентом, что упростит тестирование. В целом, внедрение объектов конфигурации, действие которых распространяется на все приложение, является нарушением принципа изоляции интерфейса.

Следует отказаться от рассмотренных ранее вариантов и предпочесть извлечение элементарных зависимостей в граничные объекты — они устраняют неопределенность как на функциональном, так и на техническом уровне. Но это требует внесения изменений в конструктор компонента, что не всегда возможно. В таком случае приемлемым вариантом может стать регистрация делегата.

14.3.3. Регистрация объектов с помощью блоков кода

Из предыдущего раздела известно, что один из способов создания компонента с элементарным значением заключается в использовании метода `Register`. Он позволяет предоставить делегат, создающий компонент. Покажем эту регистрацию еще раз:

```
container.Register<ICourse>(() => new ChiliConCarne(Spiciness.Hot));
```

Конструктор `ChiliConCarne` вызывается с `Hot Spiciness` при каждом разрешении сервиса `ICourse`. Но тут Simple Injector не вычисляет аргументы конструктора — вы создаете код вызова конструктора самостоятельно с помощью блока кода.

Что касается классов приложения, то у вас есть выбор между автоматическим связыванием и использованием блока кода. Но на другие классы накладывается больше ограничений: их экземпляры не могут создаваться с помощью открытого конструктора. Вместо этого для создания экземпляров типа приходится задействовать какую-либо фабрику. Для DI-контейнеров это всегда проблематично, поскольку изначально они обходятся открытыми конструкторами.

СОВЕТ

Simple Injector способен создавать экземпляры внутренних классов при условии, что их конструктор определен как public. Но это поведение можно переопределить, заменив реализацию IConstructorResolutionBehavior по умолчанию (https://simpleinjector.org/xtpcr).

Рассмотрим следующий конструктор для публичного класса `JunkFood`:

```
internal JunkFood(string name)
```

Хотя класс `JunkFood` может быть публичным, конструктор является внутренним. В следующем примере показано, что экземпляры `JunkFood` должны создаваться с помощью статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static JunkFood Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С позиции контейнера Simple Injector мы имеем дело с проблемным API, поскольку в отношении статичных фабрик однозначных и устоявшихся соглашений нет. Здесь требуется помощь, получить которую можно, предоставив блок кода, выполняемого контейнером для создания экземпляра:

```
container.Register<IMeal>(() => JunkFoodFactory.Create("chicken meal"));
```

На этот раз метод `Register` используется для создания компонента путем вызова статической фабрики внутри блока кода. `JunkFoodFactory.Create` станет вызываться при каждом разрешении `IMeal`, возвращая соответствующий результат.

Когда все сводится к написанию кода для создания экземпляра, возникает вопрос: чем же это лучше вызова данного кода напрямую? За счет применения блока кода внутри вызова метода `Register` извлекается такая польза.

- *Создается отображение* `IMeal` *на* `JunkFood`. Это позволяет классам-потребителям сохранять слабую связанность.
- *Сохраняется возможность конфигурирования жизненных циклов*. Хотя блок кода будет вызываться для создания экземпляра, при каждом запросе экземпляра это может и не происходить. Вызов осуществляется по умолчанию, но если изменить жизненный цикл экземпляра на Singleton, блок кода будет вызван только один раз, а результат кэширован для повторного использования в дальнейшем

В этом разделе показан порядок работы Simple Injector с довольно сложными API. Чтобы реализовать более типобезопасный подход, метод `Register` можно задействовать с блоком кода. Мы еще не рассматривали работу с несколькими компонентами, поэтому сейчас поговорим об этом.

14.4. Работа с несколькими компонентами

Из подраздела 12.1.2 известно, что в DI-контейнерах делается ставка на определенность, а неопределенность создает для них существенные трудности. При внедрении через конструктор лучше воспользоваться единственным конструктором, чем несколькими переопределяемыми конструкторами, поскольку при этом создается четкое представление о том, какой конструктор следует использовать при отсутствии выбора. То же самое относится и к отображению абстракций на конкретные типы. Если попытаться отобразить на одну и ту же абстракцию сразу несколько конкретных типов, возникнет неопределенность.

ПРИМЕЧАНИЕ

Когда в классе имеется сразу несколько переопределяемых конструкторов, в большинстве контейнеров для выбора нужного конструктора используется тот или иной эвристический алгоритм. В отличие от них Simple Injector в таком случае выдает исключение с пояснением, что данное определение типа страдает неопределенностью. Хотя такое поведение, как утверждалось в подразделе 4.2.3, может быть переопределено (см. https://simpleinjector.org/xtpcr), мы советуем избегать создания компонентов с несколькими конструкторами.

Несмотря на крайнюю нежелательность пребывания в неопределенности, зачастую все же приходится работать с несколькими реализациями одной и той же абстракции[1]. Это может быть вызвано следующими обстоятельствами.

- Разные потребители используют разные конкретные типы.
- Зависимости представлены последовательностями.
- Применяются декораторы или компоновщики.

В данном разделе будут рассмотрены все эти случаи и то, как Simple Injector справляется с каждым из них. Когда вопрос будет закрыт, вы сможете регистрировать и разрешать компоненты, даже если в работе будет сразу несколько реализаций одной и той же абстракции. Сначала рассмотрим способы обеспечения более четкого контроля над ситуациями неопределенности.

14.4.1. Выбор из нескольких кандидатов

При всех своих удобстве и эффективности автоматическое связывание не позволяет полностью контролировать ситуацию. Пока все абстракции четко отображаются на конкретные типы, не возникает никаких проблем. Но как только вводится несколько реализаций одного и того же интерфейса, вскрывается вся неприглядность неопределенности. Сначала вспомним, как Simple Injector справляется с несколькими регистрациями одной и той же абстракции.

Конфигурирование нескольких реализаций одного и того же сервиса

Из подраздела 14.1.2 известно, что несколько реализаций одного и того же интерфейса могут быть зарегистрированы следующим образом:

```
container.Collection.Register<IIngredient>(
    typeof(SauceBéarnaise),
    typeof(Steak));
```

---

[1] Фактически множество абстракций при всего лишь одной реализации служит признаком проблемной конструкции, подпадающей под принцип повторного использования абстракций. См. статью: *Gorman J.* Reused Abstractions Principle (RAP), 2010; http://www.codemanship.co.uk/parlezuml/blog/?postid=934.

В данном примере регистрация классов `Steak` и `SauceBéarnaise` выполняется в качестве последовательности сервисов `IIngredient`. У контейнера можно запросить разрешение всех `IIngredient`-компонентов. В Simple Injector для этого есть специальный метод `GetAllInstances`, который получает `IEnumerable` со всеми зарегистрированными ингредиентами. Это выглядит следующим образом:

```
IEnumerable<IIngredient> ingredients =
    container.GetAllInstances<IIngredient>();
```

Вместо этого можно запросить у контейнера разрешение всех `IIngredient`-компонентов с помощью метода `GetInstance`:

```
IEnumerable<IIngredient> ingredients =
    container.GetInstance<IEnumerable<IIngredient>>();
```

Заметьте, что запрашивается `IEnumerable<IIngredient>`, но используется обычный метод `GetInstance`. Simple Injector интерпретирует это как соглашение и выдает вам все имеющиеся у него `IIngredient`-компоненты.

СОВЕТ

> В качестве альтернативы IEnumerable<T> можно также запросить такие абстракции, как IList<T>, ICollection<T>, IReadOnlyList<T> и IReadOnlyCollection<T>. Результат неизменен: во всех случаях будут получены все компоненты запрошенного типа.

Зачастую при наличии нескольких реализаций конкретной абстракции есть и потребитель, зависящий от последовательности. И все же иногда компоненты должны работать с фиксированным набором или поднабором зависимостей одной и той же абстракции, что и рассмотрим далее.

Устранение неопределенности с помощью условных регистраций

При всех достоинствах автоматического связывания иногда требуется переопределить обычное поведение, чтобы более четко управлять тем, какие зависимости куда направляются, также возникает необходимость обратиться к неопределенному API. Рассмотрим в качестве примера следующий конструктор:

```
public ThreeCourseMeal(ICourse entrée, ICourse mainCourse, ICourse dessert)
```

В данном случае имееется три идентично типизированные зависимости, в каждой из которых представлена одна из трех концепций. В большинстве случаев нужно отобразить каждую из зависимостей на отдельно взятый тип. Чаще всего в DI-контейнерах типовым решением проблем такого рода являются регистрации с использованием ключей или имен, как было показано для Autofac в главе 13. А в Simple Injector типовое решение заключается в изменении регистрации зависимости, а не потребителя. Возможный порядок выбора регистрации отображений `ICourse` показан в листинге 14.9.

Листинг 14.9. Регистрация перемен блюд на основе имен параметров конструктора

```
container.Register<IMeal, ThreeCourseMeal>();

container.RegisterConditional<ICourse, Rillettes>(
    c => c.Consumer.Target.Name == "entrée");

container.RegisterConditional<ICourse, CordonBleu>(
    c => c.Consumer.Target.Name == "mainCourse");

container
    .RegisterConditional<ICourse, MousseAuChocolat>(
    c => c.Consumer.Target.Name == "dessert");
```

ThreeCourseMeal создан с использованием обычной регистрации с автоматическим связыванием

Три перемены блюд регистрируются условно на основе цели типа потребления. Целью может быть свойство или параметр конструктора. В данном случае целями являются параметры конструктора ThreeCourseMeal

Детальнее разберемся в том, что здесь происходит. Метод `RegisterConditional` принимает значение `Predicate<PredicateContext>`, что позволяет ему определить, должна регистрация внедряться в потребитель или нет. В нем используется следующая сигнатура:

```
public void RegisterConditional<TService, TImplementation>(
    Predicate<PredicateContext> predicate)
    where TImplementation : class, TService
    where TService : class;
```

`System.Predicate<T>` — это тип делегата среды .NET. Значение предиката будет вызываться контейнером Simple Injector. Если предикат возвращает `true`, используется регистрация для данного потребителя. В противном случае Simple Injector ожидает, что в другой условной регистрации будет делегат, возвращающий `true`. Если найти регистрацию невозможно, он выдает исключение, потому что в таком случае невозможно сконструировать граф объектов. Он также выдает исключение, когда имеется сразу несколько подходящих регистраций.

У Simple Injector строгий характер: он никогда не строит предположений о вашем выборе (об этом уже говорилось применительно к компонентам с несколькими конструкторами). Но это означает, что Simple Injector всегда вызывает все предикаты всех подходящих условных регистраций, чтобы найти вероятные перекрывающиеся регистрации. Возможно, это покажется неэффективным, но данные предикаты вызываются, только когда компонент разрешается в первый раз. Любое последующее разрешение содержит всю доступную информацию, а следовательно, выполняется довольно быстро.

ВНИМАНИЕ

Идентификация параметров по именам довольно удобна, но не позволяет безопасно проводить реструктуризацию. Если параметр переименован, конфигурация может стать неработоспособной (все зависит от инструментария реконфигурации).

Переопределяя автоматическое связывание с помощью условно зарегистрированных компонентов, Simple Injector получает возможность строить граф объектов, не возвращаясь к регистрации блока кода, рассмотренной в подразделе 14.3.3. Польза от этого в ходе работе с Simple Injector заключается в получении рассмотренных ранее возможностей диагностики. Использование блока кода ослепляет контейнер, из-за этого ошибки конфигурирования могут слишком долго остаться нераспознанными.

В следующем подразделе покажем, как можно применить более определенный и гибкий подход, допускающий любое количество перемен блюд. Для этого нужно изучить порядок работы Simple Injector со списками и последовательностями.

14.4.2. Связывание последовательностей

В подразделе 6.1.1 рассматривалось внедрение через конструктор в качестве системы предупреждения о нарушении принципа единственной ответственности. Выяснилось, что вместо того, чтобы признать чрезмерное внедрение через конструктор слабостью данного паттерна, следует удовлетвориться тем, что при этом выявляются проблемы самой конструкции.

В ходе работы с DI-контейнерами в условиях неопределенности наблюдается такая же взаимосвязь. В общем-то DI-контейнеры не способны достойно справиться с неопределенностью. Такой качественный контейнер, как Simple Injector, можно заставить с ней справиться, но выглядит это весьма неуклюже. А зачастую и является признаком того, что конструкцию кода можно усовершенствовать.

СОВЕТ

> Если, конфигурируя конкретную часть API с помощью Simple Injector, вы испытываете определенные трудности, отступите на шаг и переосмыслите конструкцию с точки зрения применения паттернов и принципов, представленных в этой книге. Чаще всего затруднения с конфигурированием вызваны тем, что конструкция приложения не соответствует паттернам и нарушает принципы. Улучшение общей конструкции не только облегчает сопровождение приложения, но и упрощает конфигурирование Simple Injector.

Чтобы не чувствовать скованности, применяя Simple Injector, нужно смириться с предлагаемыми им условиями и позволить ему привести вас к разработке более рациональной и последовательной конструкции. В этом разделе будет рассмотрен пример, показывающий, как можно избавиться от неопределенности, и демонстрирующий порядок работы Simple Injector с последовательностями.

Переход к более рациональной перемене блюд за счет устранения неопределенности

В подразделе 14.4.1 было показано, как `ThreeCourseMeal` со свойственной ему неопределенностью заставил вас усложнить регистрацию. Это должно подтолкнуть к пересмотру конструкции API. Простое обобщение наводит на мысль о реализации

IMeal, получающего произвольное число экземпляров ICourse вместо всего трех перемен блюд, как было в классе ThreeCourseMeal:

```
public Meal(IEnumerable<ICourse> courses)
```

Заметьте, что вместо требования в конструкторе трех отдельных экземпляров ICourse одна-единственная зависимость от экземпляра IEnumerable<ICourse> позволяет предоставить классу Meal любое количество перемен блюд — от нуля до... полного изобилия! Тем самым решается вопрос с неопределенностью, поскольку теперь мы имеем дело лишь с одной зависимостью. Кроме того, за счет предоставления всего одного универсального класса, способного смоделировать различные типы трапез, от простого перекуса с одной переменой блюд до изысканного обеда на 12 перемен, улучшаются API и реализация компонента.

В этом разделе будут рассмотрены способы конфигурирования Simple Injector для связывания экземпляров Meal с соответствующими зависимостями ICourse. Усвоив этот материал, вы сможете составить четкое представление о доступных вариантах конфигурирования экземпляров с последовательностями зависимостей.

Автоматическое связывание последовательностей

Simple Injector неплохо разбирается в последовательностях, поэтому при необходимости воспользоваться всеми зарегистрированными компонентами заданного сервиса автоматическое связывание по-прежнему окажется работоспособным. В качестве примера данный набор настроенных экземпляров ICourse можно сконфигурировать с сервисом IMeal следующим образом:

```
container.Register<IMeal, Meal>();
```

Заметьте, что это стандартное отображение абстракции на конкретный тип. Simple Injector автоматически разбирается с конструктором Meal и определяет, что правильным действием станет разрешение всех компонентов ICourse. При разрешении IMeal будет получен экземпляр Meal с компонентами ICourse. Но от вас все же потребуется зарегистрировать последовательность компонентов ICourse, например, с помощью автоматической регистрации:

```
container.Collection.Register<ICourse>(assembly);
```

Simple Injector обрабатывает последовательности в автоматическом режиме и, пока не указано иное, делает то, что от него ожидается: разрешает последовательность зависимостей для всех регистраций заданной абстракции. Дополнительные действия потребуются, только если понадобится явно выбрать некоторые компоненты из более широкого набора. Посмотрим, как это можно сделать.

Выбор ограниченного числа компонентов из более широкого набора

Стратегия, по умолчанию используемая Simple Injector для внедрения всех компонентов, зачастую представляет собой правильную политику, но на рис. 14.5 показано, что иногда из довольно широкого набора зарегистрированных компонентов требуется выбрать лишь некоторые.

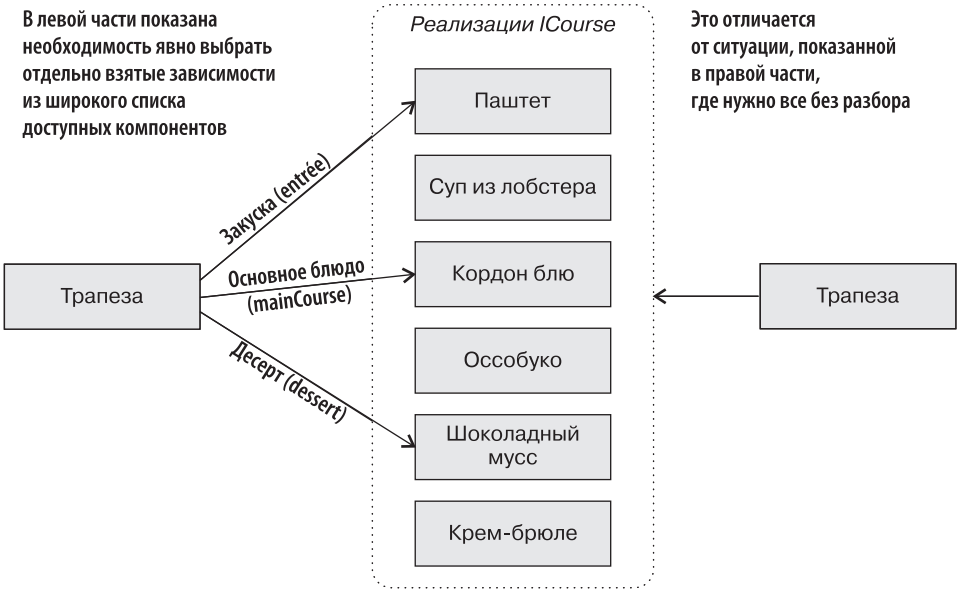


Рис. 14.5. Выбор компонентов из широкого набора зарегистрированных компонентов

ПРИМЕЧАНИЕ

Потребность во внедрении поднабора из полной коллекции не относится к типовым сценариям, но показывает порядок удовлетворения более сложных потребностей.

Когда ранее контейнеру Simple Injector разрешено было провести автоматическую регистрацию и автоматическое связывание всех сконфигурированных экземпляров, это соответствовало ситуации, изображенной в правой части рисунка. Если же нужно зарегистрировать компонент так, как показано в левой части, намеченные к использованию компоненты должны быть определены явно. Чтобы получить желаемый результат, можно воспользоваться методом `Collection.Create`, что позволит создать поднабор последовательности. Способ внедрения поднабора последовательности в потребителя показан в листинге 14.10.

Метод `Collection.Create` позволяет создавать последовательность заданной абстракции. Сама последовательность в контейнере зарегистрирована не будет, но это можно сделать с помощью метода `Collection.Register`. Многократным вызовом `Collection.Create` в отношении одной и той же абстракции можно создать несколько последовательностей, и в каждой из них, как показано в листинге 14.10, могут быть разные поднаборы.

Каким бы странным это ни казалось, но вызов `Collection.Create` в листинге 14.10 не приводит в тот же самый момент к созданию блюда. В данном случае последовательность имеет вид потока данных. Разрешение экземпляров начнется

только с началом перебора последовательности. Благодаря такому поведению поднабор последовательности может быть безопасно и совершенно безвредно внедрен в Singleton-компонент `Meal`. Более подробно потоки данных будут рассматриваться в подразделе 14.4.5.

Листинг 14.10. Внедрение поднабора последовательности в потребитель

```
IEnumerable<ICourse> coursesSubset1 =
    container.Collection.Create<ICourse>(
        typeof(Rillettes),
        typeof(CordonBleu),
        typeof(MousseAuChocolat));

IEnumerable<ICourse> coursesSubset2 =
    container.Collection.Create<ICourse>(
        typeof(CeasarSalad),
        typeof(ChiliConCarne),
        typeof(MousseAuChocolat));

container.RegisterInstance<IMeal>(
    new Meal(sourcesSubset1));
```

Создание последовательности из трех перемен блюд

Создание еще одной последовательности с другим поднабором перемен блюд

Создание одного экземпляра Meal внедрением первой последовательности и отображением его на IMeal

Simple Injector изначально разбирается в последовательностях. Этот контейнер автоматически все сделает правильно, если только не потребуется из всех сервисов заданного типа явно выбрать отдельно взятые компоненты.

Автоматическое связывание работает не только с отдельными экземплярами, но и с последовательностями, и контейнер отображает последовательность на все сконфигурированные экземпляры соответствующего типа. Возможно, менее интуитивно понятным является рассматриваемое далее использование нескольких экземпляров одной и той же абстракции, но это относится уже к паттерну проектирования «Декоратор».

14.4.3. Связывание декораторов

В подразделе 9.1.1 рассматривались преимущества, получаемые от использования паттерна проектирования «Декоратор» при решении проблем применения сквозной функциональности. Согласно определению декораторы вводят в обращение несколько типов одной и той же абстракции. Имеется по крайней мере две реализации абстракции: сам декоратор и декорируемый тип. Если выстроить декораторы в цепочку, можно получить больше абстракций. Это еще один пример нескольких регистраций одного и того же сервиса. В отличие от того, что говорилось в предыдущих разделах, концептуально данные регистрации не равны — скорее всего, они являются зависимостями друг для друга.

В Simple Injector имеется встроенная поддержка регистрации декораторов, для чего используется метод `RegisterDecorator`. В этом разделе будет рассмотрена регистрация как необобщенных, так и обобщенных абстракций. Начнем с первых.

Декорирование необобщенных абстракций

Воспользовавшись методом `RegisterDecorator`, вы получите возможность легко и просто зарегистрировать декоратор. В следующем примере показано, как этот метод используется для применения `Breading` к `VealCutlet`:

```
var c = new Container();

c.Register<IIngredient, VealCutlet>();

c.RegisterDecorator<IIngredient, Breading>();
```

Регистрация VealCutlet в качестве IIngredient

Регистрация Breading в качестве декоратора Iingredient. При разрешении Iingredient Simple Injector возвращает VealCutlet, заключенный в Breading

Из главы 9 известно, что блюдо кордон блю получается, если сделать в отбивной котлете надрез, начинить ее ветчиной, сыром и чесноком, а потом запанировать и поджарить. В следующем примере показано, как декоратор `HamCheeseGarlic` добавляется между `VealCutlet` и декоратором `Breading`:

```
var c = new Container();

c.Register<IIngredient, VealCutlet>();

c.RegisterDecorator<IIngredient, HamCheeseGarlic>();

c.RegisterDecorator<IIngredient, Breading>();
```

Добавление декоратора

За счет помещения этой новой регистрации перед регистрацией `Breading` декоратор `HamCheeseGarlic` первым заключается во внешний декоратор. В результате получается граф объектов, эквивалентный следующей версии, выполненной по технологии чистого DI:

```
new Breading(
    new HamCheeseGarlic(
        new VealCutlet()));
```

VealCutlet обернут в HamCheeseGarlic, который, в свою очередь, обернут в Breading

ПРИМЕЧАНИЕ

Декораторы применяются в порядке их регистрации.

Имеющийся в Simple Injector метод `RegisterDecorator` позволяет легко выстроить декораторы в цепочку, а кроме того, в этом контейнере могут применяться обобщенные декораторы, что и будет показано далее.

Декорирование обобщенных абстракций

В ходе изучения главы 10 было определено несколько обобщенных декораторов, подходящих для применения к любой реализации `ICommandService<TCommand>`. Далее в этой главе мы отодвинем в сторону все наши ингредиенты и перемены

блюд и рассмотрим способы регистрации обобщенных декораторов с помощью Simple Injector. В листинге 14.11 показано, как зарегистрировать все реализации `ICommandService<TCommand>` с тремя декораторами, представленными в разделе 10.3.

Листинг 14.11. Декорирование обобщенных абстракций, прошедших автоматическую регистрацию

```
container.Register(
    typeof(ICommandService<>), assembly);

container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(AuditingCommandServiceDecorator<>));

container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(TransactionCommandServiceDecorator<>));

container.RegisterDecorator(
    typeof(ICommandService<>),
    typeof(SecureCommandServiceDecorator<>));
```

Регистрация произвольных реализаций ICommandService<TCommand>

Регистрация обобщенных декораторов

Как и в листинге 14.3, здесь используется переопределение метода `Register`, предназначенное для регистрации произвольного числа реализаций `ICommandService<TCommand>` путем сканирования сборок. Для регистрирования декораторов применяется метод `RegisterDecorator`, принимающий два экземпляра `Type`. Результат конфигурирования, выполненного кодом листинга 14.11, показан на рис. 14.6 (он уже рассматривался в подразделе 10.3.4).

Рис. 14.6. Оснащение реального сервиса команд аспектами ведения контрольного журнала, управления транзакциями и обеспечения безопасности

Что же касается имеющейся в Simple Injector поддержки декораторов, то это лишь верхушка айсберга. Несколько переопределений `RegisterDecorator` позволяют создавать декораторы, исходя из тех или иных условий, как ранее рассмотренное переопределение метода `RegisterConditional`, показанное в листинге 14.9. Но разговор об этой и других функциональных возможностях выходит за рамки данной книги[1].

Simple Injector позволяет работать с несколькими экземплярами декораторов несколькими разными способами. Компоненты можно регистрировать в качестве

[1] Подробности можно найти по адресу https://simpleinjector.org/aop.

альтернатив друг другу, в качестве равнозначных элементов, разрешаемых в виде последовательностей, или в виде иерархии декораторов. Во многих случаях Simple Injector сам определяет, что нужно сделать. Но если нужен более строгий контроль, вы всегда можете явно определить порядок компоновки сервисов.

В этом разделе основное внимание уделялось методам Simple Injector, предназначенным для конфигурирования декораторов. Хотя наиболее понятными на интуитивном уровне пользователями нескольких экземпляров одной и той же абстракции являются потребители, полагающиеся на последовательности зависимостей, хорошим примером могут стать и декораторы. Но есть и третий, возможно, несколько неожиданный случай выхода нескольких экземпляров на первый план — применение паттерна проектирования «Компоновщик» (Composite).

14.4.4. Связывание компоновщиков

В этой книге паттерн проектирования «Компоновщик» уже рассматривался в ряде случаев. Например, в подразделе 6.1.2 создавался сервис `CompositeNotificationService` (см. листинг 6.4), в котором не только был реализован сервис `INotificationService`, но и объединялись в последовательность его реализации.

Связывание необобщенных компоновщиков

Посмотрим, как в Simple Injector можно зарегистрировать компоновщики, подобные `CompositeNotificationService` из главы 6. В листинге 14.12 этот класс показан еще раз.

Листинг 14.12. Компоновщик CompositeNotificationService из главы 6

```
public class CompositeNotificationService : INotificationService
{
    private readonly IEnumerable<INotificationService> services;

    public CompositeNotificationService(
        IEnumerable<INotificationService> services)
    {
        this.services = services;
    }

    public void OrderApproved(Order order)
    {
        foreach (INotificationService service in this.services)
        {
            service.OrderApproved(order);
        }
    }
}
```

Поскольку API, имеющийся в Simple Injector, отделяет регистрацию последовательностей от регистраций без последовательностей, регистрировать компоновщиков становится проще простого. Зарегистрировать компоновщик можно заодно с регистрацией его зависимостей в качестве последовательности:

```
container.Collection.Register<INotificationService>(
    typeof(OrderApprovedReceiptSender),
    typeof(AccountingNotifier),
    typeof(OrderFulfillment),
);

container.Register<INotificationService, CompositeNotificationService>();
```

В предыдущем примере с помощью `Collection.Register` три реализации `INotificationService` зарегистрированы в качестве последовательности. А регистрация `CompositeNotificationService` выполнена в виде одиночного сервиса, не содержащего последовательности. Все типы в Simple Injector автоматически связываются. Благодаря предыдущей регистрации, когда выполняется разрешение `INotificationService`, получается граф объектов, похожий на тот, который вышел бы при использовании чистой технологии DI:

```
return new CompositeNotificationService(new INotificationService[]
{
    new OrderApprovedReceiptSender(),
    new AccountingNotifier(),
    new OrderFulfillment()
});
```

Поскольку число уведомительных сервисов со временем, скорее всего, возрастет, нагрузку на корень композиции можно уменьшить, применив автоматическую регистрацию с помощью переопределения метода `Collection.Register`, принимающего сборку `Assembly`. Тогда вы сможете превратить прежний список типов в простую строку кода:

```
container.Collection.Register<INotificationService>(assembly);

container.Register<INotificationService, CompositeNotificationService>();
```

Можно вспомнить, что в Autofac (рассматривался в главе 13) подобная конструкция не работала, поскольку в нем компоновщик автоматически регистрируется в виде части последовательности. Но в Simple Injector такого не случается. Его метод `Collection.Register` автоматически отфильтровывает компоновщики любого типа и не дает им зарегистрироваться в виде части последовательности.

Но классы компоновщиков не единственные, которые Simple Injector автоматически удаляет из списка. Этот контейнер таким же способом находит и декораторы. Такое поведение существенно облегчает работу с декораторами и компоновщиками в Simple Injector. То же самое можно сказать и о работе с обобщенными компоновщиками.

Связывание обобщенных компоновщиков

В подразделе 14.4.2 было показано, как использование имеющегося в Simple Injector метода `RegisterDecorator` превращает регистрацию обобщенных декораторов в детскую забаву. В этом разделе будет рассмотрен способ регистрации компоновщиков для обобщенных абстракций.

В подразделе 6.1.3 класс `CompositeEventHandler<TEvent>` (из листинга 6.12) был указан в качестве реализации компоновщика последовательности реализаций `IEventHandler<TEvent>`. Посмотрим, можно ли зарегистрировать компоновщик с заключенными в него реализациями обработчиков событий. Начнем с автоматической регистрации обработчиков событий:

```
container.Collection.Register(typeof(IEventHandler<>), assembly);
```

В отличие от регистрации реализаций `ICommandService<T>` в листинге 14.3 здесь вместо `Register` используется метод `Collection.Register`. Дело в том, что для конкретного типа событий может существовать сразу несколько обработчиков. Следовательно, нужно явно заявить, что вы знаете о возможных дополнительных реализациях для одного и того же типа события. Если же случайно вместо `Collection.Register` вызвать метод `Register`, Simple Injector выдаст исключение с сообщением примерно следующего содержания: In the supplied list of types or assemblies, there are 3 types that represent the same closed-generic type IEventHandler<OrderApproved>. Did you mean to register the types as a collection using the Collection.Register method instead? Conflicting types: OrderApprovedReceiptSender, AccountingNotifier and OrderFulfillment (В предоставленном списке типов или сборок имеются три типа, представляющие один и тот же закрытый обобщенный тип IEventHandler<OrderApproved>. Может быть, вы намеревались зарегистрировать типы как коллекцию, воспользовавшись вместо данного метода методом Collection.Register? Конфликтующие типы: OrderApprovedReceiptSender, AccountingNotifier и OrderFulfillment).

Просто здорово, что в этом сообщении сказано, что вам, скорее всего, следует воспользоваться `Collection.Register`, а не `Register`. Но, вполне возможно, случайно был добавлен выбранный вами неверный тип. Как говорилось ранее, в случае неопределенности Simple Injector заставляет прояснить ситуацию, что помогает обнаруживать ошибки.

Остается только зарегистрировать `CompositeEventHandler<TEvent>`. Поскольку `CompositeEventHandler<TEvent>` относится к обобщенным типам, придется воспользоваться переопределением метода `Register`, принимающим аргументы `Type`:

```
container.Register(
    typeof(IEventHandler<>),
    typeof(CompositeEventHandler<>));
```

Поскольку цель составителя — скрыть существование последовательности, он регистрируется в виде одного отображения, не имеющего отношения к последовательностям

Используя эту регистрацию, когда запрошена конкретная закрытая абстракция `IEventHandler<TEvent>` (например, `IEventHandler<OrderApproved>`), Simple

Injector определяет точный тип создаваемого `CompositeEventHandler<TEvent>`. В данном случае с этим нет никаких проблем, поскольку в результате запроса `IEventHandler<OrderApproved>` выполняется разрешение `CompositeEventHandler<OrderApproved>`. В других случаях определение точного закрытого типа может стать довольно сложным процессом, но Simple Injector прекрасно справится и с этим.

Работать с последовательностями в Simple Injector довольно легко. Но, когда дело доходит до разрешения и внедрения последовательностей, он ведет себя намного интереснее других DI-контейнеров. Как уже упоминалось, Simple Injector обрабатывает последовательности в виде потока данных.

14.4.5. Последовательности и потоки данных

В разделе 14.1 была зарегистрирована следующая последовательность ингредиентов:

```
container.Collection.Register<IIngredient>(
    typeof(SauceBéarnaise),
    typeof(Steak));
```

Ранее уже было показано, что у контейнера можно запросить разрешение всех компонентов `IIngredient`, воспользовавшись либо `GetAllInstances`, либо `GetInstance`. В следующем примере вновь применяется метод `GetInstance`:

```
IEnumerable<IIngredient> ingredients =
    container.GetInstance<IEnumerable<IIngredient>>();
```

Можно полагать, что вызов `GetInstance<IEnumerable<IIngredient>>()` приведет к созданию экземпляров обоих классов, но это абсолютно не так. При разрешении или внедрении `IEnumerable<T>` Simple Injector не заполняет последовательность сразу всеми ингредиентами. Вместо этого `IEnumerable<T>` ведет себя как поток данных[1]. Это означает, что возвращаемый `IEnumerable<IIngredient>` является объектом, способным производить в ходе итерации новые экземпляры `IIngredient`. Это похоже на потоковые данные с диска, получаемые при выполнении метода `System.IO.FileStream`, или из базы данных, получаемые при выполнении метода `System.Data.SqlClient.SqlDataReader`, когда данные поступают небольшими порциями, а не извлекаются заранее в полном объеме.

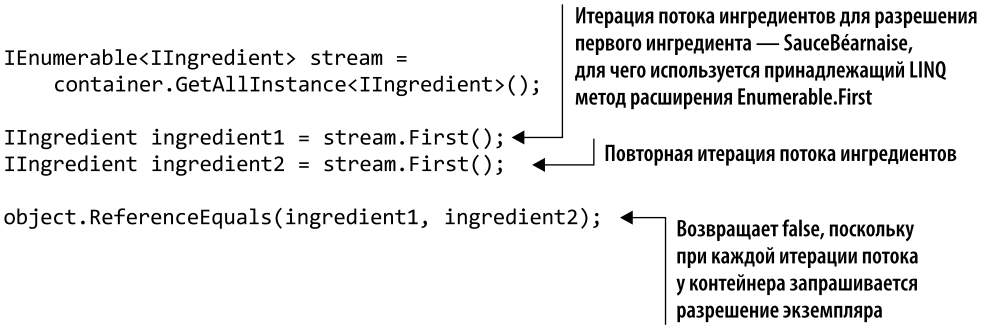
ПРИМЕЧАНИЕ

> Насколько нам известно, Simple Injector — единственный DI-контейнер, создающий из последовательностей абстракций потоки данных.

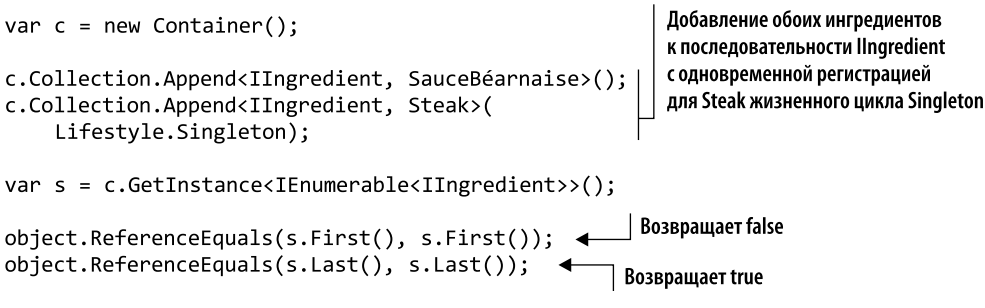
В следующем примере показано, как многократная итерация потока может создавать новые экземпляры:

---

[1] В действительности все абстракции, представленные последовательностями, например IList<T> и ICollection<T>, ведут себя в Simple Injector как потоки данных.



Когда поток данных проходит итерацию, выполняется обратный вызов в контейнер для разрешения элементов последовательности, основанной на соответствующем им жизненном цикле. Следовательно, если тип зарегистрирован с жизненным циклом Transient, то, как показано в предыдущем примере, всегда создаются новые экземпляры. Когда жизненным циклом типа является Singleton, всякий раз возвращается один и тот же экземпляр:



ПРИМЕЧАНИЕ

Вызов метода First останавливает итерацию потока после возвращения первого экземпляра, то есть создается только SauceBéarnaise, а Steak не создается. Но удивительным может быть то, что в результате вызова метода Last создаются не первый и последний элементы, а лишь последний элемент, что в ходе работы с потоками данных может стать весьма неожиданным обстоятельством.

Это вызвано оптимизацией в Enumerable.Last в сочетании с объектом, возвращаемым Simple Injector. В возвращенной последовательности реализуется IList<T>. Когда последовательность рассматривается в качестве потока данных, это может показаться странным, но такое возможно, поскольку число элементов в последовательности фиксируется после завершения фазы конфигурирования. В методе Enumerable.Last имеется оптимизация для IList<T>, позволяющая ему запрашивать с помощью индексатора List<T> только последний элемент, не нуждаясь в итерации всего списка.

Хотя потоковая передача не характерна для DI-контейнеров, у нее есть ряд интересных преимуществ. Во-первых, при внедрении потока данных в контейнер

внедрение потока как такового практически не требует ресурсов, поскольку в этот момент не создается никаких экземпляров[1]. Это качество окажется полезным, когда список элементов станет очень большим и за время жизни потребителя ему понадобятся не все элементы. Возьмем, к примеру, следующую реализацию компоновщика `ILogger` (листинг 14.13). Она является вариантом компоновщика из листинга 8.22, но в данном случае компоновщик останавливает регистрацию событий сразу после того, как успешно отработает один из заключенных в него регистраторов.

Листинг 14.13. Компоновщик, обрабатывающий лишь часть внедренного потока данных

```
public class CompositeLogger : ILogger                    ← Реализация ILogger
{
    private readonly IEnumerable<ILogger> loggers;

    public CompositeLogger(
        IEnumerable<ILogger> loggers)                     ← Зависит от IEnumerable<ILogger>
    {
        this.loggers = loggers;
    }

    public void Log(LogEntry entry)
    {
        foreach (ILogger logger in this.loggers)          ← Итерация последовательности
        {                                                   регистраторов loggers
            try
            {
                logger.Log(entry);
                break;                                    ← Выход из цикла, как только logger
            }                                               не выдаст исключение
            catch { }                                     ← Игнорирование любого исключения, выданного
        }                                                   регистратором logger, и продолжение работы
    }                                                       со следующим регистратором logger,
}                                                           имеющимся в последовательности
```

Исходя из показанного в подразделе 14.4.4, зарегистрировать `CompositeLogger` и последовательность реализаций `ILogger` можно следующим образом:

```
container.Collection.Register<ILogger>(assembly);
container.Register<ILogger, CompositeLogger>(Lifestyle.Singleton);
```

В данном случае `CompositeLogger` регистрируется с жизненным циклом Singleton, поскольку там нет состояния и единственная зависимость, `IEnumerable<ILogger>`, сама имеет жизненный цикл Singleton. Эффект от того, что у последовательностей `CompositeLogger` и `ILogger` жизненный цикл Singleton состоит в том, что внедрение `CompositeLogger` обходится практически без ресурсных затрат. Даже когда потребитель вызывает в своей зависимости метод `Log`, то, как правило, создается только первая реализация, имеющаяся в последовательности `ILogger`, а не все реализации сразу.

Вторым преимуществом последовательностей, превращенных в поток данных, является то, что, пока сохраняется только ссылка на `IEnumerable<ILogger>`, как по-

[1] Сам поток имеет жизненный цикл Singleton и создается только один раз.

казано в листинге 14.13, элементы последовательности никогда не смогут случайно стать захваченными зависимостями. Это уже было продемонстрировано в предыдущем примере. `CompositeLogger` с жизненным циклом Singleton может иметь совершенно безопасную зависимость от `IEnumerable<ILogger>`, поскольку там также используется жизненный цикл Singleton, несмотря на то что создаваемые этим компонентом сервисы могут такого жизненного цикла и не иметь.

В этом разделе был показан порядок работы с несколькими компонентами, такими как последовательности, декораторы и компоновщики. На этом изучение Simple Injector завершается. В следующей главе займемся контейнером Microsoft.Extensions.DependencyInjection.

Резюме

- Simple Injector является современным DI-контейнером, предлагающим довольно полный набор функций, но его API сильно отличается от интерфейсов большинства DI-контейнеров. Вот как выглядят некоторые из его особенностей:
  - области видимости являются охватывающими;
  - последовательности регистрируются с помощью `Collection.Register`, а не добавлением новых регистраций одной и той же абстракции;
  - последовательности ведут себя как потоки данных;
  - контейнер может быть диагностирован на предмет нахождения самых распространенных изъянов конфигурации.
- Важной общей темой для Simple Injector является строгость. Он не пытается строить догадки о вашем замысле, а старается не допустить ошибок конфигурирования и обнаружить их благодаря особенностям своего API и имеющимся средствам диагностики.
- Simple Injector обеспечивает четкое разделение регистрации и разрешения. Как для регистрации, так и для разрешения используется один и тот же экземпляр `Container`, однако после первого применения `Container` блокируется.
- Поскольку Simple Injector делает области видимости охватывающими, непосредственное разрешение из корневого контейнера считается надлежащим приемом и всецело приветствуется: оно не приводит к утечкам памяти или к ошибкам одновременных вычислений.
- Simple Injector поддерживает стандартные жизненные циклы Transient, Singleton и Scoped.
- В Simple Injector имеется весьма эффективная поддержка регистрации последовательностей, декораторов, компоновщиков и их обобщенных версий.

15 DI-контейнер Microsoft.Extensions.DependencyInjection

В этой главе

- Работа в Microsoft.Extensions.DependencyInjection с API регистрации.
- Управление временем жизни компонентов.
- Конфигурирование сложных API.
- Конфигурирование последовательностей, декораторов и компоновщиков.

С появлением среды ASP.NET Core Microsoft выпустила собственный DI-контейнер — Microsoft.Extensions.DependencyInjection, являющийся частью среды выполнения Core. В этой главе название контейнера будет сокращено до MS.DI.

Microsoft создала MS.DI, чтобы упростить управление зависимостями как для среды, так и для сторонних разработчиков компонентов, работающих со средой ASP.NET Core. В намерения Microsoft входило определение DI-контейнера с минимальным, наиболее малочисленным набором общих функциональных признаков, которому могли бы соответствовать все остальные DI-контейнеры.

В этой главе MS.DI будет дана та же трактовка, что и Autofac и Simple Injector. Вы увидите, как может использоваться MS.DI с целью применения тех принципов и паттернов, которые рассматривались в частях I–III. Несмотря на то что MS.DI

интегрирован со средой ASP.NET Core, он может действовать и отдельно от нее, поэтому в данной главе будет рассмотрен именно в таком качестве.

В ходе изучения главы обнаружится, что функциональные возможности MS.DI настолько ограничены, что мы считаем его непригодным для разработки любого более или менее солидного приложения, построенного на принципах слабой связанности и соответствующего паттернам и принципам, рассмотренным в данной книге. Если MS.DI нельзя приспособить к чему-то серьезному, зачем посвящать ему целую главу? Главная причина в том, что MS.DI на первый взгляд настолько похож на другие DI-контейнеры, что нужно потратить немало времени, чтобы понять, в чем разница между ним и более солидными DI-контейнерами. Поскольку он является частью .NET Core, то, если вы не осознаете все его ограничения, у вас может возникнуть соблазн воспользоваться этим встроенным контейнером. Цель главы — выяснить, в чем суть этих ограничений, чтобы вы могли принять обоснованное решение.

ПРИМЕЧАНИЕ

Если MS.DI неинтересен и уже принято решение воспользоваться другим DI-контейнером, то можете не читать эту главу.

Эта глава разбита на четыре раздела. Можно изучать их по отдельности, но первый раздел прочитать обязательно, чтобы понять все остальные, а в разделе 15.4 есть ссылки на некоторые методы и классы, рассмотренные в разделе 15.3.

Эту главу можно рассмотреть отдельно от других глав части IV, если вы нацелены исключительно на изучение MS.DI, или же ее можно читать наряду с другими главами, чтобы сравнивать DI-контейнеры. Основное внимание в этой главе уделяется тому, как MS.DI относится к моделям и принципам, рассмотренным в частях I–III, и как они в ней реализуются.

15.1. Введение в Microsoft.Extensions.DependencyInjection

Из этого раздела вы узнаете, где можно получить MS.DI, что именно будет получено и как приступить к использованию этого контейнера. Кроме этого, будут рассмотрены общие вопросы его конфигурации. Основная информация, которая, возможно, пригодится для начала работы с контейнером, приведена в табл. 15.1.

Таблица 15.1. Краткое знакомство с Microsoft.Extensions.DependencyInjection

Вопрос	Ответ
Где его получить?	Он автоматически включается при создании нового приложения ASP.NET Core, но его можно и самостоятельно добавить к приложениям другого типа. Из Visual Studio его можно получить с помощью NuGet. Пакет называется Microsoft .Extensions.DependencyInjection

Продолжение ⇩

Таблица 15.1 *(продолжение)*

Вопрос	Ответ
Какие платформы поддерживаются?	.NET Standard 2.0 (.NET Core 2.0, .NET Framework 4.6.1, Mono 5.4, Xamarin.iOS 10.14, Xamarin.Android 8.0, UWP 10.0.16299)
Сколько он стоит?	Нисколько. Это средство с открытым кодом
Какая у него лицензия?	Apache License, Version 2.0
Где можно получить помощь?	Поскольку это официальный продукт Microsoft .NET, у него есть гарантированная коммерческая поддержка по адресу https://www.microsoft.com/net/support/policy. Для получения некоммерческой негарантированной поддержки лучше обратиться за помощью на Stack Overflow по адресу https://stackoverflow.com/
На какой версии основан материал данной главы?	2.1.0

По большому счету, использование MS.DI не слишком отличается от использования контейнера Autofac, рассмотренного в главе 13. Применение этого контейнера представляет собой двухэтапный процесс (рис. 15.1). Но, по сравнению с Simple Injector, в ходе работы с MS.DI этот двухэтапный процесс носит четко выраженный характер: сначала выполняется конфигурирование `ServiceCollection`, а когда оно завершится, результат задействуется для создания `ServiceProvider`, который может применяться для разрешения компонентов.

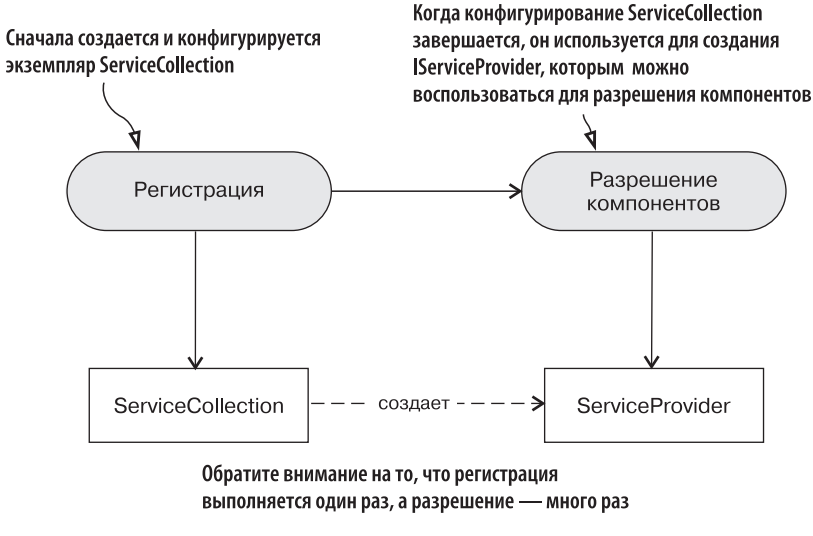


Рис. 15.1. Схема использования Microsoft.Extensions.DependencyInjection заключается в том, что сначала выполняется его конфигурирование, а затем проводится разрешение компонентов

Усвоив материал данной главы, вы получите достаточное представление об общей схеме применения MS.DI и сможете приступить к использованию его в сценариях с надлежащим поведением, где все компоненты придерживаются правильных DI-паттернов, таких как внедрение через конструктор. Начнем с самого простого сценария и посмотрим, как можно выполнить разрешение объектов с помощью контейнера MS.DI.

15.1.1. Разрешение объектов

Основным сервисом любого DI-контейнера является построение графа объектов. В этом разделе будет рассмотрен API, позволяющий составлять графы объектов с помощью MS.DI. Этот контейнер требует зарегистрировать все компоненты до их разрешения. В листинге 15.1 показан один из самых простых вариантов использования MS.DI.

Листинг 15.1. Простейший вариант применения MS.DI

```
var services = new ServiceCollection();

services.AddTransient<SauceBéarnaise>();

ServiceProvider container =
    services.BuildServiceProvider(validateScopes: true);
IServiceScope scope = container.CreateScope();

SauceBéarnaise sauce =
    scope.ServiceProvider.GetRequiredService<SauceBéarnaise>();
```

На основе показанного на рис. 15.1 можно предположить, что для конфигурирования компонентов понадобится экземпляр `ServiceCollection`. Имеющийся в MS.DI `ServiceCollection` является эквивалентом `ContainerBuilder` из Autofac.

Здесь выполняется регистрация конкретного класса `SauceBéarnaise` с сервисами, чтобы при запросе на создание контейнера получающийся контейнер был сконфигурирован с классом `SauceBéarnaise`. Это опять же позволит разрешить класс `SauceBéarnaise` из контейнера. Если компонент `SauceBéarnaise` не зарегистрировать, попытка разрешения закончится выдачей исключения `InvalidOperationException` со следующим сообщением: No service for type 'Ploeh.Samples.MenuModel.SauceBéarnaise' has been registered (Никаких сервисов для типа 'Ploeh.Samples.MenuModel.SauceBéarnaise' не зарегистрировано).

ПРИМЕЧАНИЕ

> При создании приложения ASP.NET Core хостинговая среда сама создает ServiceCollection. Вам останется только воспользоваться этим, как показано в листинге 7.7. Но в данной главе MS.DI будет рассматриваться наравне с другими DI-контейнерами, то есть мы покажем приемы его использования в менее интегрированной среде.

Из показанного в листинге 15.1 следует, что в ходе работы с MS.DI разрешение никогда не выполняется из самого корневого контейнера, а для него используется `IServiceScope`. Более подробно `IServiceScope` рассматривается в подразделе 15.2.1.

ВНИМАНИЕ

> Работая с MS.DI, следует избегать разрешения из корневого контейнера, так как это легко может вызвать утечки памяти или ошибки одновременных вычислений. Вместо этого, как показано в листинге 15.1, разрешение всегда нужно выполнять из области видимости.

Для обеспечения безопасности `ServiceProvider` нужно всегда создавать с использованием переопределения метода `BuildServiceProvider` с аргументом `validateScopes`, установленным в `true`, что соответствует примеру, приведенному в листинге 15.1. Это позволит избежать случайного разрешения Scoped-экземпляров из корневого контейнера. С появлением среды ASP.NET Core 2.0 `validateScopes` автоматически устанавливается этой средой в `true`, когда приложение запускается в среде разработки, но проверку лучше включать и вне среды разработки. То есть придется вызывать `BuildServiceProvider(true)` самостоятельно.

Контейнер MS.DI способен не только разрешать конкретные типы с конструкторами без параметров, но и выполнять автоматическое связывание типа с другими зависимостями. Все эти зависимости должны быть зарегистрированы. В основном вы будете стремиться к программированию с прицелом на интерфейсы, поскольку это позволит ввести слабое связывание. Для поддержки этого стиля программирования MS.DI позволяет отображать абстракции на конкретные типы.

Отображение абстракций на конкретные типы

В соответствии с показанным в листинге 15.1 корневые типы нашего приложения, как правило, разрешаются в их конкретные типы, а слабое связывание требует отображения абстракций на конкретные типы. Создание экземпляров, основанное на таких отображениях, — это основной сервис, предлагаемый любым DI-контейнером, но определение отображения возлагается на вас самих. В данном примере интерфейс `IIngredient` отображается на конкретный класс `SauceBéarnaise`, что позволяет успешно выполнить разрешение `IIngredient`:

```
var services = new ServiceCollection();

services.AddTransient<IIngredient, SauceBéarnaise>();

var container = services.BuildServiceProvider(true);

IServiceScope scope = container.CreateScope();

IIngredient sauce = scope.ServiceProvider
    .GetRequiredService<IIngredient>();
```

Отображение конкретного типа на определенную абстракцию

Разрешение SauceBéarnaise в качестве IIngredient

Здесь метод `AddTransient` позволяет конкретному типу отображаться на определенную абстракцию с использованием жизненного цикла Transient. Благодаря

предыдущему вызову `AddTransient` теперь `SauceBéarnaise` может быть разрешен как `IIngredient`.

Во многих случаях можно обойтись применением обычного API. Но все же бывают ситуации, когда нужен способ разрешения сервисов с еще более слабой типизацией. Имеется и такая возможность.

Разрешение сервисов с еще более слабой типизацией

Иногда применить обычный API невозможно, поскольку в ходе разработки о подходящем типе ничего не известно. Имеется только экземпляр `Type`, но все равно хочется получить экземпляр подходящего типа. Соответствующий пример был показан в разделе 7.3, где рассматривался принадлежащий среде ASP.NET Core MVC класс `IControllerActivator`. Метод имел следующий вид:

```
object Create(ControllerContext context);
```

В соответствии с показанным в листинге 7.8 `ControllerContext` захватывает принадлежащий контроллеру `Type`, который можно извлечь с помощью свойства `ControllerTypeInfo`, принадлежащего свойству `ActionDescriptor`:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
```

Поскольку в вашем распоряжении есть только экземпляр `Type`, пользоваться обобщенными методами нельзя и приходится прибегать к API с более слабой типизацией. MS.DI предлагает переопределение метода `GetRequiredService` с более слабой типизацией, позволяющее реализовать метод `Create`:

```
Type controllerType = context.ActionDescriptor.ControllerTypeInfo.AsType();
return scope.ServiceProvider.GetRequiredService(controllerType);
```

Переопределение `GetRequiredService` с более слабой типизацией позволяет передавать переменную `controllerType` непосредственно в MS.DI. Обычно это означает, что тип возвращаемого значения необходимо привести к некой абстракции, поскольку метод `GetRequiredService` с более слабой типизацией возвращает объект `object`. Но в случае с `IControllerActivator` этого не нужно, так как среда ASP.NET Core MVC не требует от контроллеров реализации какого-либо интерфейса или базового класса.

Независимо от того, какое переопределение метода `GetRequiredService` используется, MS.DI гарантирует, что им будет возвращен экземпляр запрошенного типа или выдано исключение, если имеются зависимости, удовлетворение которых невозможно. При приемлемом конфигурировании всех нужных зависимостей MS.DI может выполнить автоматическое связывание запрошенного типа.

ПРИМЕЧАНИЕ

> В качестве альтернативы GetRequiredService есть еще метод GetService. Когда запрошенный тип не может быть разрешен, GetRequiredService выдает исключение, а GetService возвращает вместо этого значение null. Когда ожидается возвращение экземпляра (почти всегда), предпочтение следует отдавать методу GetRequiredService.

Чтобы иметь возможность выполнить разрешение запрошенного типа, следует предварительно сконфигурировать все слабосвязанные зависимости. Рассмотрим способы, позволяющие сконфигурировать MS.DI.

15.1.2. Конфигурирование ServiceCollection

Из раздела 12.2 известно, что конфигурировать DI-контейнер можно несколькими концептуально различными способами. Возможные варианты (см. рис. 12.5): файлы конфигурации, конфигурация в виде кода и автоматическая регистрация. Все они показаны также на рис. 15.2.

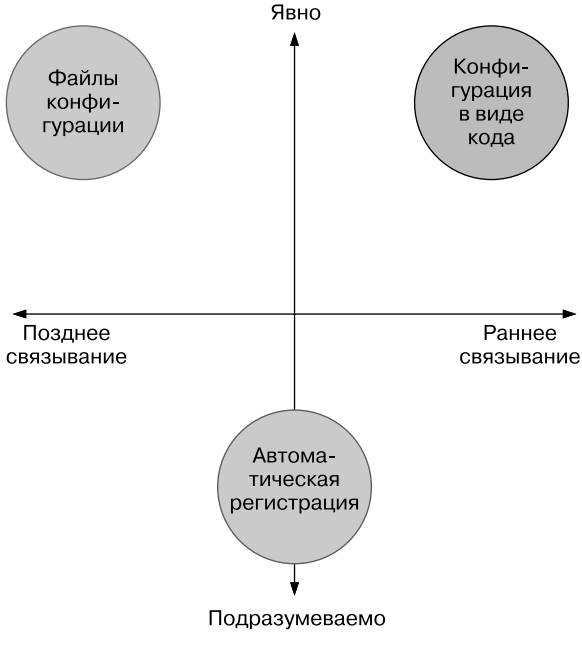


Рис. 15.2. Наиболее распространенные способы конфигурирования DI-контейнеров, расставленные по шкалам их явности-неявности и времени связывания

ВНИМАНИЕ

MS.DI сконструирован с прицелом на использование конфигурации в виде кода и не содержит API, поддерживающий файлы конфигурации или автоматическую регистрацию.

Несмотря на отсутствие API автоматической регистрации, все же можно в какой-то мере сканировать сборки с помощью .NET LINQ-запросов и API отражения. Но прежде, чем перейти к этому, изучим имеющийся в MS.DI API конфигурации в виде кода.

Конфигурирование ServiceCollection с помощью конфигурации в виде кода

В подразделе 15.1.1 уже было получено некоторое представление об имеющемся в MS.DI конфигурационном API со строгой типизацией. Здесь рассмотрим его подробнее.

Все конфигурации в MS.DI задействуют API, представленный классом `ServiceCollection`, но большинство методов являются методами расширения. Одним из самых широко используемых является `AddTransient`, который вы уже видели:

```
services.AddTransient<IIngredient, SauceBéarnaise>();
```

При регистрации `SauceBéarnaise` в качестве `IIngredient` конкретный класс скрыт, что не позволяет больше выполнять разрешение `SauceBéarnaise` с этой регистрацией. Но ситуацию можно исправить, заменив регистрацию следующим кодом:

```
services.AddTransient<SauceBéarnaise>();
services.AddTransient<IIngredient>(
    c => c.GetRequiredService<SauceBéarnaise>());
```

Регистрация делегата, совершающего обратный вызов в контейнер для разрешения ранее зарегистрированного конкретного типа

Вместо выполнения регистрации для `IIngredient` с помощью переопределения метода `AddTransient`, допускающего автоматическое связывание, регистрируется блок кода, который, будучи вызванным, перенаправляет вызов на регистрацию конкретного `SauceBéarnaise`.

Рваные жизненные циклы

В этом разделе рассмотрено, как можно вызвать `AddTransient` несколько раз для регистрации компонента в качестве нескольких типов сервиса. В следующем примере это показано еще раз:

```
services.AddTransient<SauceBéarnaise>();
services.AddTransient<IIngredient>(
    c => c.GetRequiredService<SauceBéarnaise>());
```

Можно подумать, что это является эквивалентом следующего кода:

```
services.AddTransient<SauceBéarnaise>();
services.AddTransient<IIngredient, SauceBéarnaise>();
```

Но первый пример не эквивалентен второму. Это становится ясно при замене жизненного цикла Transient, к примеру, на Singleton:

```
services.AddSingleton<SauceBéarnaise>();
services.AddSingleton<IIngredient, SauceBéarnaise>();
```

Вопреки ожиданию того, что под время жизни контейнера подпадет только один экземпляр `SauceBéarnaise`, разбиение регистрации заставляет MS.DI создать отдельные экземпляры для каждого вызова `AddSingleton`. Поэтому жизненный цикл `SauceBéarnaise` должен считаться рваным.

ВНИМАНИЕ

Каждое обращение к методам AddScoped или AddSingleton приводит к созданию его собственного уникального кэша. Поэтому наличие нескольких вызовов Add... может вылиться в создание нескольких экземпляров на одну область видимости или один контейнер. Чтобы не допустить этого, нужно регистрировать делегат, разрешающий конкретный экземпляр.

В реальных приложениях всегда более одной отображаемой абстракции, поэтому конфигурировать нужно несколько отображений. Это делается несколькими вызовами одного из методов `Add...`:

```
services.AddTransient<IIngredient, SauceBéarnaise>();
services.AddTransient<ICourse, Course>();
```

В этом примере `IIngredient` отображается на `SauceBéarnaise`, а `ICourse` — на `Course`. Здесь нет перекрытия типов, поэтому происходящее должно быть вполне очевидным. Но одну и ту же абстракцию также можно зарегистрировать несколько раз:

```
services.AddTransient<IIngredient, SauceBéarnaise>();
services.AddTransient<IIngredient, Steak>();
```

Здесь дважды регистрируется `IIngredient`. Если выполняется разрешение `IIngredient`, получается экземпляр `Steak`. Последняя регистрация выигрывает, но предыдущая при этом не забывается. MS.DI хорошо справляется с несколькими конфигурациями одной и той же абстракции (к этой теме вернемся в разделе 15.4).

Для конфигурирования MS.DI существуют и более совершенные методы, но с помощью показанных здесь можно сконфигурировать целое приложение. Однако, чтобы не тратить слишком много усилий на явное сопровождение конфигурации контейнера, можно обратиться к подходу, более тесно связанному с соглашениями, и воспользоваться автоматической регистрацией.

Конфигурирование Service Collection с помощью автоматической регистрации

Регистрации во многих случаях будут похожи друг на друга. Сопровождение регистраций может превратиться в весьма утомительное занятие, а явная регистрация буквально каждого компонента, как выяснилось в подразделе 12.3.3, может стать совершенно нерациональным подходом.

Рассмотрим библиотеку, содержащую множество реализаций `IIngredient`. Можно сконфигурировать каждый класс по отдельности, но тогда получится постоянно меняющийся список экземпляров `Type`, предоставляемый методам `Add...`. Хуже того, чтобы при каждом добавлении новая реализация `IIngredient` стала доступной, ее также нужно явно регистрировать с контейнером. Разумнее было бы заявить, что должны быть зарегистрированы все реализации `IIngredient`, найденные в заданной сборке.

Как уже говорилось, в MS.DI не содержится API автоматической регистрации. Следовательно, все придется выполнять самостоятельно. Отчасти это возможно, и в этом разделе на простом примере будет показано, как это делается, но более подробно возможности и ограничения будут рассматриваться в разделе 15.4. Посмотрим, как можно зарегистрировать последовательность регистраций `IIngredient`:

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

var ingredientTypes =
    from type in ingredientsAssembly.GetTypes()
    where !type.IsAbstract
    where typeof(IIngredient).IsAssignableFrom(type)
    select type;

foreach (var type in ingredientTypes)
{
    services.AddTransient(typeof(IIngredient), type);
}
```

Сканирование на основе соглашения

Регистрация каждого типа на основе интерфейса IIngredient

В предыдущем примере без каких-либо условий выполнялось конфигурирование всех реализаций интерфейса `IIngredient`, но вы можете предоставить фильтры, позволяющие выбрать только один поднабор. Сканирование на основе соглашения, при котором добавляются только те классы, имена которых начинаются с `Sauce`, имеет следующий вид:

```
Assembly ingredientsAssembly = typeof(Steak).Assembly;

var ingredientTypes =
    from type in ingredientsAssembly.GetTypes()
    where !type.IsAbstract
    where typeof(IIngredient).IsAssignableFrom(type)
    where type.Name.StartsWith("Sauce")
    select type;

foreach (var type in ingredientTypes)
{
    services.AddTransient(typeof(IIngredient), type);
}
```

Удаление классов, имена которых не начинаются на Sauce

Помимо выбора из сборки подходящих типов, `AutoRegistration` определяет правильное отображение. В предыдущих примерах метод `AddTransient` использовался с определенным интерфейсом для регистрации за ним всех выбранных типов.

Но иногда нужно воспользоваться и другими соглашениями. Предположим, что вместо интерфейсов применяются классы на основе абстракций и нужно зарегистрировать за их базовым типом все имеющиеся в сборке типы, чьи имена заканчиваются на `Policy`:

```
Assembly policiesAssembly = typeof(DiscountPolicy).Assembly;

var policyTypes =
    from type in policiesAssembly.GetTypes()
```

Получение всех типов, имеющихся в сборке

```
    where type.Name.EndsWith("Policy")
    select type;

foreach (var type in policyTypes)
{
    services.AddTransient(type.BaseType, type);
}
```

Фильтрование по суффиксу Policy

Регистрация каждого policy-компонента за его базовым классом

Хотя MS.DI не содержит API, работающих на основе соглашений, но использование API, имеющихся в среде .NET, все еще позволяет выполнять регистрацию на основе соглашений. Но, как мы покажем далее, ситуация становится иной, когда речь заходит об обобщениях.

Автоматическая регистрация обобщенных абстракций

При изучении главы 10 рассматривалась переделка большого, крайне неудачного интерфейса `IProductService` в интерфейс `ICommandService<TCommand>`, показанный в листинге 10.12. Посмотрим на эту абстракцию еще раз:

```
public interface ICommandService<TCommand>
{
    void Execute(TCommand command);
}
```

Как известно из главы 10, каждый командный граничный объект представлял тот или иной вариант использования и для каждого такого варианта предусматривалась всего одна реализация. В качестве примера был представлен сервис `AdjustInventoryService`, показанный в листинге 10.8. В нем был реализован вариант применения «корректировка товарных запасов». В листинге 15.2 этот класс показан еще раз.

Листинг 15.2. AdjustInventoryService из главы 10

```
public class AdjustInventoryService : ICommandService<AdjustInventory>
{
    private readonly IInventoryRepository repository;

    public AdjustInventoryService(IInventoryRepository repository)
    {
        this.repository = repository;
    }

    public void Execute(AdjustInventory command)
    {
        var productId = command.ProductId;

        ...
    }
}
```

В любой сложной системе могут реализовываться сотни вариантов использования. Такие системы можно рассматривать в качестве идеальных кандидатов для автоматической регистрации. Но из-за отсутствия в MS.DI поддержки автоматиче-

ской регистрации для ее запуска приходится создавать изрядное количество кода. Пример приведен в листинге 15.3.

Листинг 15.3. Автоматическая регистрация реализаций ICommandService<TCommand>

```
Assembly assembly = typeof(AdjustInventoryService).Assembly;

var mappings =
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where !type.IsGenericType

    from i in type.GetInterfaces()
    where i.IsGenericType
    where i.GetGenericTypeDefinition()
        == typeof(ICommandService<>)
    select new { service = i, type };

foreach (var mapping in mappings)
{
    services.AddTransient(
        mapping.service,
        mapping.type);
}
```

Выбор конкретных типов

Выбор необобщенных типов

Выбор типов, реализующих ICommandService<TCommand>

Регистрация типов за их интерфейсами

Как и в предыдущих листингах, здесь можно воспользоваться существующей в среде .NET возможностью делать LINQ-запросы и задействовать API отражения, позволяющие выбрать классы из предоставленной сборки. С помощью открытого обобщенного интерфейса выполняется последовательный перебор списка типов, входящих в сборку, и регистрируются все типы, реализующие закрытую обобщенную версию `ICommandService<TCommand>`. Это, к примеру, означает, что `AdjustInventoryService` регистрируется, потому что он реализует `ICommandService<AdjustInventory>` — закрытую обобщенную версию `ICommandService<TCommand>`.

ВНИМАНИЕ

> В коде листинга 15.1 множество недостатков. Например, при случайной реализации одного и того же закрытого обобщенного интерфейса для нескольких классов регистрация даст молчаливый сбой. Код с радостью зарегистрирует все реализации. В случае запроса сервиса команд, где для заданного типа имеется несколько реализаций, разрешаться будет последняя регистрация. Но одна из основных проблем заключается в том, что неизвестно, какая из регистраций последняя, и ситуация может измениться даже после перезапуска приложения![1]

В этом разделе был представлен DI-контейнер MS.DI и продемонстрированы основные механизмы — способы конфигурирования `ServiceCollection` и последующее

[1] Порядок следования элементов в списке типов, возвращенном методом Assembly.GetType(), не определен. Он может измениться после перекомпиляции приложения или даже его перезапуска.

использование созданного `ServiceProvider` для разрешения сервисов. Это разрешение выполняется с помощью всего одного вызова метода `GetRequiredService`, поэтому сложность заключается в конфигурировании контейнера. API поддерживает в первую очередь конфигурацию в виде кода, хотя отчасти над ним может быть надстроена автоматическая регистрация. Но в дальнейшем выяснится, что отсутствие поддержки автоматической регистрации обернется применением слишком сложного и трудносопровождаемого кода. Пока мы рассматривали лишь наиболее общий API, не охваченной осталась гораздо более сложная область — управление временем жизни компонентов.

15.2. Управление временем жизни

Управление временем жизни, включая наиболее распространенные концептуальные жизненные циклы Singleton, Scoped и Transient, рассматривалось в главе 8. MS.DI поддерживает все три жизненных цикла и позволяет настраивать время жизни всех сервисов. Жизненные циклы, показанные в табл. 15.2, доступны в качестве части API.

Таблица 15.2. Жизненные циклы Microsoft.Extensions.DependencyInjection

Название, используемое в Microsoft	Паттерн	Комментарии
Transient	Transient	Экземпляры отслеживаются контейнером и ликвидируются
Singleton	Singleton	Экземпляры ликвидируются при ликвидации контейнера
Scoped	Scoped	Экземпляры используются повторно в пределах одного и того же IServiceScope. Экземпляры отслеживаются по времени жизни области видимости и ликвидируются, когда ликвидируется область видимости

Имеющиеся в MS.DI реализации Transient и Singleton — эквиваленты общих жизненных циклов, рассмотренных в главе 8, поэтому слишком много времени в этой главе им уделять не будем. Вместо этого в данном разделе рассмотрим способы определения жизненных циклов для компонентов, выражаемые в коде. Усвоив материал раздела, вы сможете воспользоваться жизненными циклами MS.DI в собственном приложении. Начнем со способов конфигурирования областей видимости экземпляров, применяемых для компонентов.

15.2.1. Конфигурирование жизненных циклов

В этом разделе рассмотрим способы управления жизненными циклами средствами MS.DI. Конфигурирование жизненных циклов является частью регистрации компонентов. Все делается очень просто:

```
services.AddSingleton<SauceBéarnaise>();
```

В данном примере конкретный класс `SauceBéarnaise` настраивается на жизненный цикл Singleton, поэтому при каждом запросе `SauceBéarnaise` будет возвращаться один и тот же экземпляр. Если нужно отобразить абстракцию на конкретный класс с определенным жизненным циклом, можно воспользоваться обычным переопределением метода `AddSingleton` с двумя обычными аргументами:

```
services.AddSingleton<IIngredient, SauceBéarnaise>();
```

Когда дело доходит для конфигурирования жизненных циклов компонентов, то оказывается, что в MS.DI выбор невелик по сравнению с имеющимся в других DI-контейнерах. Все делается в сугубо декларативной форме. Как правило, конфигурирование выполняется довольно просто, однако не следует забывать, что некоторые жизненные циклы связаны с присутствием в системе долгоживущих объектов, потребляющих ресурсы на всем протяжении своего существования.

15.2.2. Высвобождение компонентов

Из подраздела 8.2.2 известно, что при завершении работы с объектами важно выполнить их высвобождение. Так же как в Autofac и Simple Injector, в MS.DI нет явно определенного метода `Release`, вместо него используются области видимости (scopes). Область видимости может рассматриваться как кэш для конкретного запроса. На рис. 15.3 показано, как ею определяется граница, в пределах которой компоненты могут применяться повторно.

Областью видимости `IServiceScope` определяется кэш, которым можно воспользоваться в определенный период или с определенной целью. Наиболее очевидный пример — веб-запрос. Когда из области видимости `IServiceScope` запрашивается Scoped-компонент, запрашивающий всегда получает один и тот же экземпляр. Отличие от настоящих Singleton-элементов заключается в тем, что при запросе из второй области видимости будет получен другой экземпляр.

Важной особенностью областей видимости является возможность правильно высвободить компоненты при выходе области видимости из употребления. Новая область видимости создается с помощью метода `CreateScope` конкретной реализации `IServiceProvider`, а с вызовом ее метода `Dispose` все соответствующие компоненты высвобождаются:

```
using (IServiceScope scope = container.CreateScope())
{
    IMeal meal = scope.ServiceProvider
        .GetRequiredService<IMeal>();

    meal.Consume();

}
```

Создание области видимости из корневого контейнера

Разрешение meal из только что созданной области видимости

Потребление meal

Высвобождение meal за счет прекращения использования блока

Новая область видимости создается из контейнера путем вызова метода `CreateScope`. Возвращаемое значение реализует `IDisposable`, позволяя заключать его в блок `using`.

Поскольку в `IServiceScope` содержится свойство `ServiceProvider`, реализующее такой же интерфейс, какой реализует сам контейнер, область видимости можно применять для разрешения компонентов точно таким же способом, как при использовании самого контейнера.

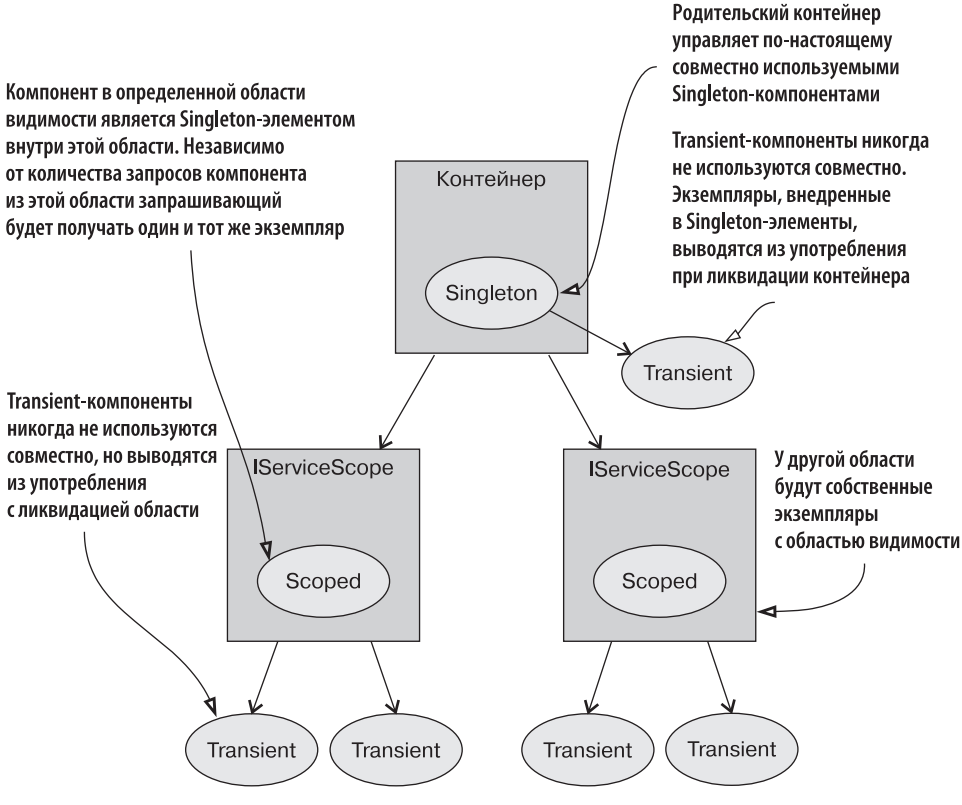


Рис. 15.3. Области видимости Microsoft.Extensions.DependencyInjection являются контейнерами, способными совместно использовать компоненты в определенный период времени или с определенной целью

Когда работа с областью видимости завершится, ее можно ликвидировать. При использовании блока `using` это происходит автоматически по выходу из блока, но можно также выбрать ее явную ликвидацию, вызвав метод `Dispose`. Когда ликвидируется область видимости, высвобождаются все созданные ею компоненты. Здесь это означает, что высвобождается граф объектов `meal`.

Следует заметить, что зависимости компонента всегда разрешаются на уровне или ниже области видимости компонента. Например, если Transient-зависимость нужно внедрить в Singleton-компонент, она будет исходить из корневого контейнера, даже если разрешение Singleton-компонента выполняется из вложенной области видимости. Это позволяет отследить Transient-зависимость из корневого контейнера

и не допустить ее ликвидации при ликвидации области видимости. Иначе Singleton-потребитель выйдет из строя, поскольку он продолжит существовать в корневом контейнере, находясь в зависимости от ликвидированного компонента.

ВНИМАНИЕ

При использовании MS.DI ожидается, что время жизни Transient-компонента будет совпадать с временем жизни того потребителя, в которого он внедрен. Поэтому MS.DI позволяет внедрять Transient-компоненты в Singleton-потребителей, хотя внедрение Scoped-экземпляров в Singleton-элементы блокируется[1]. Внедрение Transient-компонентов в Singleton-потребителей вполне может быть востребовано, но чаще всего такое поведение нежелательно, так как вы должны будете проверить, что эти Transient-компоненты не превратились по неосмотрительности в захваченные зависимости.

Ранее в этом разделе было показано, как компоненты конфигурируются под жизненные циклы Singleton или Transient. Конфигурирование компонента под жизненный цикл Scoped выполняется аналогично:

```
services.AddScoped<IIngredient, SauceBéarnaise>();
```

По аналогии с применением методов `AddTransient` и `AddSingleton`, чтобы указать, что время жизни компонента должно соответствовать той области видимости, которая создала экземпляр, можно воспользоваться методом `AddScoped`.

ВНИМАНИЕ

MS.DI отслеживает большинство компонентов, даже ликвидируемые компоненты с жизненным циклом Transient. Когда разрешение выполняется из корневого контейнера, а не из области видимости, могут возникнуть проблемы. При разрешении из корневого контейнера новые экземпляры по-прежнему создаются при каждом вызове метода GetService, но эти ликвидируемые Transient-экземпляры продолжают существовать, чтобы их можно было ликвидировать при ликвидации контейнера. Поскольку корневой контейнер не ликвидируется, пока не будет остановлено приложение, возникают утечки памяти, поэтому важно не забывать выполнять разрешение всех компонентов из области видимости и ликвидировать область после ее использования.

По своей природе Singleton-компоненты никогда не высвобождаются до истечения времени жизни самого контейнера. Если же контейнер вам больше не нужен, можно высвободить и эти компоненты. Это делается ликвидацией самого контейнера:

```
container.Dispose();
```

[1] Как говорилось в подразделе 15.1.1, это происходит, когда включена проверка области видимости.

На практике это не настолько важно, как ликвидация области видимости, поскольку время жизни контейнера тесно коррелирует с временем жизни поддерживаемого им приложения. Обычно контейнер сохраняется, пока выполняется приложение, поэтому он будет ликвидирован только по завершении работы приложения. В этом случае память будет восстановлена операционной системой.

На этом наш тур по управлению временем жизни в MS.DI завершается. Компоненты могут быть сконфигурированы со смешанными жизненными циклами, и это справедливо даже при регистрации сразу нескольких реализаций одной и той же абстракции. До сих пор контейнеру позволялось связывать зависимости исходя из предположения, что все компоненты будут использовать внедрение через конструктор. Но бывают и исключения. В следующем разделе рассмотрим работу с классами, экземпляры которых должны создаваться особым образом.

15.3. Регистрация сложных API

До сих пор мы рассматривали только возможности конфигурирования компонентов, применяющих внедрение через конструктор. Одним из многочисленных преимуществ подобного внедрения является то, что такие DI-контейнеры, как MS.DI, могут легко разобраться в составлении и создании всех классов в графе зависимостей. Но эта ясность утрачивается, когда API начинают хуже себя вести.

В этом разделе будет показано, как работать с элементарными аргументами конструкторов и статическими фабриками. Все это требует особого внимания. Начнем с рассмотрения классов, принимающих в качестве аргументов конструктора такие элементарные типы, как строки и целые числа.

15.3.1. Конфигурирование элементарных зависимостей

Пока в потребителей внедряются абстракции, не возникает никаких проблем. Но ситуация усложняется, когда конструктор зависит от элементарного типа, такого как строка, число или перечисление. Именно это и происходит с реализациями доступа к данным, принимающими в качестве параметра конструктора строку подключения. Однако сам вопрос гораздо шире и касается всех строковых и числовых типов.

Концептуально регистрация строки или числа в качестве компонента контейнера не всегда отвечает здравому смыслу. Задействуя ограничения обобщенного типа, MS.DI из своего обобщенного API даже блокирует регистрацию значимых типов наподобие чисел и перечислений. При использовании необобщенного API такая регистрация все же возможна. Рассмотрим следующий конструктор:

```
public ChiliConCarne(Spiciness spiciness)
```

В этом примере `Spiciness` является перечислением:

```
public enum Spiciness { Mild, Medium, Hot }
```

СОВЕТ

Общее правило гласит, что перечисления относятся к проблемному коду и должны быть переструктурированы в полиморфные классы[1]. Но для данного примера они являются вполне подходящей структурой.

Если нужно, чтобы все потребители `Spiciness` использовали одно и то же значение, то `Spiciness` и `ChiliConCarne` можно зарегистрировать независимо друг от друга:

```
services.AddSingleton(
    typeof(Spiciness), Spiciness.Medium);

services.AddTransient<ICourse, ChiliConCarne>();
```

Использование переопределения необобщенного метода AddSingleton, принимающего заранее созданный объект. В данном случае значение является перечислением

Автоматическое связывание ChiliConCarne с Spiciness

При последующем разрешении `ChiliConCarne` приобретает среднее (`Medium`) значение остроты `Spiciness`, как и все остальные компоненты с зависимостью от `Spiciness`. Если же нужен более строгий контроль взаимоотношений между `ChiliConCarne` и `Spiciness`, можно воспользоваться блоком кода, к чему мы ненадолго вернемся в подразделе 15.3.3.

Рассмотренный здесь вариант использует автоматическое связывание для предоставления компоненту конкретного значения. Но все же удобнее будет извлекать элементарные зависимости в граничные объекты.

15.3.2. Извлечение элементарных зависимостей в граничные объекты

В подразделе 10.3.3 рассматривался способ уменьшения последствий нарушения принципа открытости/закрытости, возникшего из-за применения `IProductService`, за счет введения граничных объектов. Но граничные объекты отлично подходят и для устранения неопределенности. Например, острота `Spiciness` нового блюда может быть описана более общим понятием вкуса (flavoring). `Flavoring` может включать другие свойства, такие как соленость, то есть и `Spiciness`, и соленость можно заключить в класс вкуса `Flavoring`:

```
public class Flavoring
{
    public readonly Spiciness Spiciness;
    public readonly bool ExtraSalty;

    public Flavoring(Spiciness spiciness, bool extraSalty)
    {
```

[1] *Фаулер М.* Рефакторинг. Улучшение существующего кода. — М.: Символ-Плюс, 2008. — С. 71.

```
        this.Spiciness = spiciness;
        this.ExtraSalty = extraSalty;
    }
}
```

Из подраздела 10.3.3 известно, что наличие у граничных объектов всего одного параметра — вполне нормальное явление. Целью является устранение неопределенности, и не только на техническом уровне. Имя такого граничного объекта может послужить описанием того, чем ваш код занят на функциональном уровне, что вполне изящно делается в классе `Flavoring`. Теперь с введением граничного объекта `Flavoring` появляется возможность, не допуская неопределенности, выполнить автоматическое связывание любой реализации `ICourse`, требующей указания какого-либо вкуса:

```
var flavoring = new Flavoring(Spiciness.Medium, extraSalty: true);
services.AddSingleton<Flavoring>(flavoring);

container.AddTransient<ICourse, ChiliConCarne>();
```

Этот код создает единственный экземпляр класса `Flavoring`, который становится объектом конфигурации для перемен блюд. Поскольку в наличии только один экземпляр `Flavoring`, его можно зарегистрировать в MS.DI, воспользовавшись переопределением метода `AddSingleton<T>`, принимающим заранее созданный экземпляр.

Следует отдавать предпочтение извлечению элементарных зависимостей в граничные объекты, отказываясь от ранее рассмотренных вариантов, поскольку граничные объекты устраняют неопределенность как на функциональном, так и на техническом уровне. Но это требует внесения изменений в конструктор компонента, что не всегда возможно. В таком случае приемлемым выбором может стать регистрация делегата.

15.3.3. Регистрация объектов с помощью блоков кода

Еще одним вариантом создания компонента с элементарным значением является использование одного из методов `Add...`, позволяющего предоставлять делегат, создающий компонент:

```
services.AddTransient<ICourse>(c => new ChiliConCarne(Spiciness.Hot));
```

Ранее метод `AddTransient` уже встречался при рассмотрении жизненных циклов в подразделе 15.1.2. Конструктор `ChiliConCarne` вызывается с указанием сильной остроты, `Hot` `Spiciness`, при каждом разрешении сервиса `ICourse`. Определение метода расширения `AddTransient<TService>` показано в следующем примере:

```
public static IServiceCollection AddTransient<TService>(
    this IServiceCollection services,
    Func<IServiceProvider, TService> implementationFactory)
    where TService : class;
```

Как видите, этот метод `AddTransient` принимает параметр, имеющий тип `Func<IServiceProvider, TService>`. С учетом предыдущей регистрации при разрешении `ICourse` MS.DI будет вызывать предоставленный делегат и снабжать его `IServiceProvider`, который принадлежит текущему `IServiceScope`. Благодаря этому ваш блок кода сможет разрешать экземпляры, происходящие из той же области видимости сервиса `IServiceScope`. Пример будет показан в следующем разделе.

Что же касается класса `ChiliConCarne`, то у вас есть выбор между автоматическим связыванием и использованием блока кода. Но на другие классы накладывается больше ограничений: их экземпляры не могут создаваться применением открытого конструктора. Вместо этого для создания экземпляров типа приходится задействовать какую-либо фабрику. Для DI-контейнеров это всегда проблематично, поскольку изначально они обходятся открытыми конструкторами. Рассмотрим пример конструктора для публичного класса `JunkFood`:

```
internal JunkFood(string name)
```

Хотя класс `JunkFood` может быть публичным, конструктор является внутренним. В следующем примере показано, что экземпляры `JunkFood` должны вместо этого создаваться путем применения статического класса `JunkFoodFactory`:

```
public static class JunkFoodFactory
{
    public static JunkFood Create(string name)
    {
        return new JunkFood(name);
    }
}
```

С позиции контейнера MS.DI мы имеем дело с проблемным API, поскольку в отношении статичных фабрик однозначных и устоявшихся соглашений нет. Здесь требуется помощь, получить которую можно за счет предоставления блока кода, выполняемого контейнером для создания экземпляра:

```
services.AddTransient<IMeal>(c => JunkFoodFactory.Create("chicken meal"));
```

На этот раз метод `AddTransient` используется для создания компонента путем вызова статической фабрики внутри блока кода. `JunkFoodFactory.Create` будет вызываться при каждом разрешении `IMeal`, возвращая соответствующий результат.

Когда все сводится к написанию кода для создания экземпляра, возникает вопрос: чем же это лучше вызова данного кода напрямую? За счет использования блока кода внутри вызова метода `AddTransient` извлекается такая польза:

- *создается отображение* `IMeal` *на* `JunkFood`. Это позволяет классам-потребителям сохранять слабую связанность;
- *сохраняется возможность конфигурирования жизненных циклов.* Хотя блок кода будет вызываться для создания экземпляра, при каждом запросе экземпляра этого может и не происходить. Вызов осуществляется по умолчанию, но, если изменить жизненный цикл экземпляра на Singleton, блок кода будет вызван только один раз, а результат — кэширован для повторного использования в дальнейшем.

В данном разделе был показан порядок применения MS.DI для работы с более сложными API. До сих пор примеры кода не отличались особой сложностью. Но ситуация быстро изменится, стоит только приступить к работе с несколькими компонентами, поэтому рассмотрим и это направление.

15.4. Работа с несколькими компонентами

Из подраздела 12.1.2 известно, что в DI-контейнерах делается ставка на определенность, а вот неопределенность создает для них существенные трудности. При внедрении через конструктор лучше воспользоваться единственным конструктором, чем несколькими переопределяемыми конструкторами, поскольку при этом создается четкое представление о том, какой конструктор следует применять в отсутствие выбора. То же самое верно и для отображения абстракций на конкретные типы. Если попытаться отобразить на одну и ту же абстракцию сразу несколько конкретных типов, возникнет неопределенность.

Несмотря на крайнюю нежелательность пребывания в неопределенности, зачастую все же приходится работать с несколькими реализациями одной и той же абстракции. Это может быть вызвано следующими обстоятельствами.

- Разные потребители используют разные конкретные типы.
- Зависимости представлены последовательностями.
- Применяются декораторы или компоновщики.

В данном разделе будут последовательно рассмотрены все эти случаи и то, как MS.DI справляется с каждым из них. Когда вопрос будет закрыт, у вас должно сложиться четкое представление о том, что можно сделать с MS.DI и где пролегают границы, когда в работе находятся сразу несколько реализаций одной и той же абстракции. Сначала изучим возможность получения более строгого контроля, чем при использовании автоматического связывания.

15.4.1. Выбор из нескольких кандидатов

При всех своих удобстве и эффективности автоматическое связывание не позволяет полностью контролировать ситуацию. Пока все абстракции четко отображаются на конкретные типы, не возникает никаких проблем. Но как только вводится сразу несколько реализаций одного и того же интерфейса, вскрывается вся неприглядность неопределенности. Вспомним, как MS.DI справляется с несколькими регистрациями одной и той же абстракции.

Конфигурирование нескольких реализаций одного и того же сервиса

Из подраздела 15.1.2 известно, что несколько реализаций одного и того же интерфейса могут быть зарегистрированы следующим образом:

```
services.AddTransient<IIngredient, SauceBéarnaise>();
services.AddTransient<IIngredient, Steak>();
```

В данном примере регистрация классов `Steak` и `SauceBéarnaise` выполняется в качестве сервиса `IIngredient`. Побеждает последняя регистрация, поэтому при разрешении `IIngredient` с `GetRequiredService<IIngredient>()` вы получите экземпляр `Steak`.

Можно также запросить у контейнера разрешение всех `IIngredient`-компонентов. В MS.DI для этого есть специальный метод `GetServices`. Пример его применения выглядит следующим образом:

```
IEnumerable<IIngredient> ingredients =
    scope.ServiceProvider.GetServices<IIngredient>();
```

Получение последовательности всех зарегистрированных ингредиентов

Когда у `GetServices` запрашивается `IEnumerable<IIngredient>`, он скрыто передает полномочия методу `GetRequiredService`. Вместо этого можно также затребовать у контейнера все компоненты `IIngredient`, воспользовавшись методом `GetRequiredService`:

```
IEnumerable<IIngredient> ingredients = scope.ServiceProvider
    .GetRequiredService<IEnumerable<IIngredient>>();
```

Заметьте, что здесь используется обычный метод `GetRequiredService`, но при этом запрашивается `IEnumerable<IIngredient>`. Контейнер интерпретирует это в качестве соглашения и дает вам все имеющиеся в нем компоненты `IIngredient`.

Зачастую при наличии нескольких реализаций конкретной абстракции есть и потребитель, зависящий от последовательности. И все же иногда компоненты должны работать с фиксированным набором или поднабором зависимостей одной и той же абстракции, что и будет рассмотрено далее.

Устранение неопределенности за счет использования блоков кода

Несмотря на всю пользу, получаемую от автоматического связывания, иногда возникает необходимость в переопределении обычного поведения для обеспечения более детального контроля над распределением зависимостей. Может потребоваться также избавление от API, страдающего от неопределенности. Рассмотрим в качестве примера следующий конструктор:

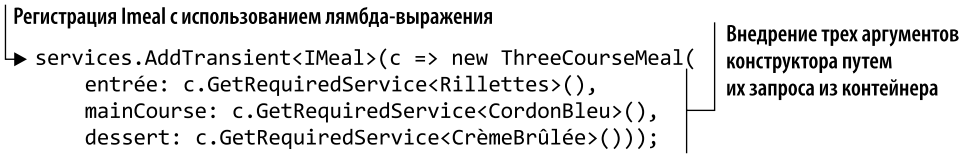
```
public ThreeCourseMeal(ICourse entrée, ICourse mainCourse, ICourse dessert)
```

В данном случае имеется три одинаково типизированные зависимости, каждая из которых представляет собственную концепцию. В большинстве случаев нужно отобразить каждую из зависимостей на отдельный тип.

Судя по ранее изложенному, по сравнению с функциональными возможностями Autofac и Simple Injector возможности MS.DI существенно скромнее. В то время как для устранения подобной неопределенности в Autofac предоставляются регистрации с использованием ключей, а в Simple Injector — условные регистрации, в MS.DI ничего подобного не имеется. То есть нет никаких встроенных функциональных

возможностей для решения данного вопроса. Чтобы связать неопределенный API с MS.DI, нужно вернуться к применению блока кода (листинг 15.4).

Листинг 15.4. Связывание ThreeCourseMeal путем разрешения перемены блюд в блоке кода

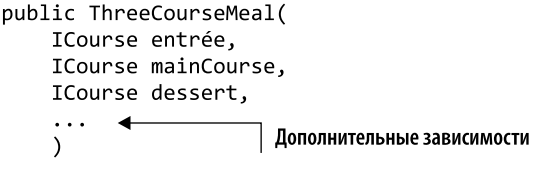



```
services.AddTransient<IMeal>(c => new ThreeCourseMeal(
    entrée: c.GetRequiredService<Rillettes>(),
    mainCourse: c.GetRequiredService<CordonBleu>(),
    dessert: c.GetRequiredService<CrèmeBrûlée>()));
```

В этой регистрации отменяется автоматическое связывание, вместо этого `ThreeCourseMeal` создается с помощью делегата. А вот сами три реализации `ICourse` сохраняют возможность автоматического связывания. Чтобы вернуть автоматическое связывание для `ThreeCourseMeal`, следует воспользоваться имеющимся в MS.DI классом `ActivatorUtilities`.

Устранение неопределенности за счет использования ActivatorUtilities

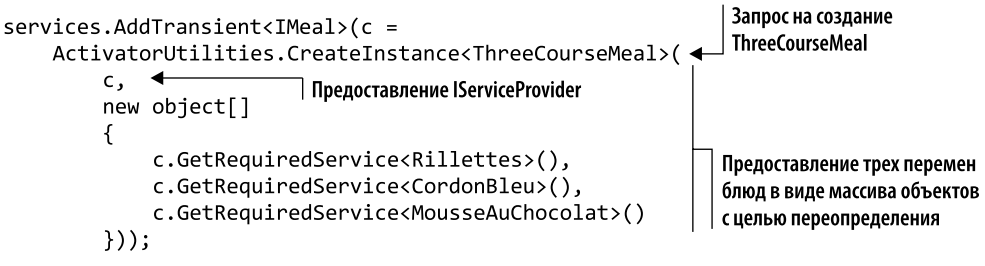
Отсутствие в этом примере автоматического связывания `ThreeCourseMeal` не представляет собой особой проблемы, поскольку в данном случае переопределяются все аргументы конструктора. Но ситуация может измениться, если в `ThreeCourseMeal` содержится больше зависимостей:




```
public ThreeCourseMeal(
    ICourse entrée,
    ICourse mainCourse,
    ICourse dessert,
    ...
    )
```

В MS.DI имеется вспомогательный класс `ActivatorUtilities`, позволяющий выполнять автоматическое связывание зависимостей класса, переопределяя другие зависимости путем явного предоставления их значений. Используя `ActivatorUtilities`, предыдущую регистрацию можно переписать (листинг 15.5).

Листинг 15.5. Связывание ThreeCourseMeal с помощью ActivatorUtilities




```
services.AddTransient<IMeal>(c =
    ActivatorUtilities.CreateInstance<ThreeCourseMeal>(
        c,
        new object[]
        {
            c.GetRequiredService<Rillettes>(),
            c.GetRequiredService<CordonBleu>(),
            c.GetRequiredService<MousseAuChocolat>()
        }));
```

В этом примере используется имеющийся в `ActivatorUtilities` метод `CreateInstance<T>`, определяемый следующим образом:

```
public static T CreateInstance<T>(
    IServiceProvider provider,
    params object[] parameters);
```

Метод `CreateInstance<T>` создает новый экземпляр предоставляемого `T`. Он выполняет последовательный перебор массива параметров и сравнивает каждый параметр с подходящим параметром конструктора. Затем выполняет разрешение оставшихся, не прошедших сравнение параметров конструктора с предоставленным `IServiceProvider`.

`ICourse` реализуют все три разрешенные перемены блюд, поэтому в вызове попрежнему сохраняется неопределенность. `CreateInstance<T>` разрешает эту неопределенность, применяя предоставленные параметры слева направо. Это означает следующее: поскольку `Rillettes` является первым элементом в массиве параметров, он будет применен к первому подходящему параметру конструктора `ThreeCourseMeal`. Им будет параметр `entrée`, имеющий тип `ICourse`.

ПРИМЕЧАНИЕ

Если количество параметров будет превышено, к примеру, за счет предоставления четвертого значения ICourse, CreateInstance<T> выдаст исключение, так как соответствие для лишнего параметра не будет найдено.

По сравнению с кодом листинга 15.4, у кода, показанного в листинге 15.5, есть большой недостаток. Код листинга 15.4 проверяется компилятором. Любая реструктуризация конструктора будет либо позволять коду сохранять работоспособность, либо давать сбой с сообщением об ошибке компиляции.

А вот для кода листинга 15.5 верно обратное. Если три параметра конструктора `ICourse` будут переставлены, код продолжит компилироваться, а `ActivatorUtilities` даже сможет сконструировать новый экземпляр `ThreeCourseMeal`. Но пока код листинга 15.5 не будет изменен в соответствии с этой перестановкой, перемены блюд будут внедряться в неверном порядке, что, скорее всего, вызовет неправильное поведение приложения. К сожалению, ни один из инструментов реструктуризации не подаст сигнал о необходимости соответствующего изменения регистрации.

Даже сходные регистрации Autofac и Simple Injector (см. листинги 13.7 и 14.9) лучше справляются с предотвращением ошибок. Хотя ни в одном из листингов код не является типобезопасным, поскольку оба примера кода рассчитаны на соответствие точным именам параметров, изменения, вносимые в `ThreeCourseMeal`, приведут как минимум к выдаче исключения при разрешении класса. Это все же лучше молчаливого сбоя, который может произойти при выполнении кода, показанного в листинге 15.5.

Переопределение автоматического связывания за счет явного отображения параметров на компоненты является универсальным решением. Там, где в Autofac

используются поименованные регистрации, а в Simple Injector — условные регистрации, в MS.DI выполняется переопределение параметров путем передачи самостоятельно разрешенных конкретных типов. Если придется управлять большим количеством типов, код может стать слишком хрупким. Чтобы избавиться от неопределенности, лучше всего разработать собственный API. Зачастую это улучшает конструкцию в целом.

В следующем разделе будет показано, как можно воспользоваться более определенным и гибким подходом, допускающим любое количество перемен блюд. Для этого нужно изучить порядок работы MS.DI с последовательностями.

15.4.2. Связывание последовательностей

В подразделе 6.1.1 рассматривалось внедрение через конструктор в качестве системы предупреждения о нарушении принципа единственной ответственности. Выяснилось, что вместо того, чтобы признать чрезмерное внедрение через конструктор слабостью данного паттерна, следует удовлетвориться тем, что при этом выявляются проблемы самой конструкции.

В ходе работы с DI-контейнерами в условиях неопределенности наблюдается такая же взаимосвязь. В общем-то DI-контейнеры не способны справиться с неопределенностью достойным образом. Их можно заставить с ней справиться, но все это может выглядеть весьма неуклюже. Зачастую это признак того, что конструкцию кода можно усовершенствовать.

В этом разделе рассмотрим пример, показывающий, как можно избавиться от неопределенности. Поговорим также о том, как MS.DI справляется с последовательностями.

Переход к более рациональной перемене блюд за счет устранения неопределенности

В подразделе 15.4.1 показано, как `ThreeCourseMeal` со свойственной ему неопределенностью заставил вас либо отказаться от автоматического связывания, либо воспользоваться весьма пространным вызовом `ActivatorUtilities`. Простое обобщение наводит на мысль о реализации `IMeal`, получающего произвольное число экземпляров `ICourse`, вместо строго трех перемен блюд, как было в классе `ThreeCourseMeal`:

```
public Meal(IEnumerable<ICourse> courses)
```

Заметьте, что вместо требования в конструкторе трех отдельных экземпляров `ICourse` одна-единственная зависимость от экземпляра `IEnumerable<ICourse>` позволяет предоставить классу `Meal` любое количество перемен блюд — от нуля до... полного изобилия! Тем самым решается вопрос с неопределенностью, поскольку теперь мы имеем дело только с одной зависимостью. Кроме того, за счет предоставления всего одного универсального класса, способного смоделировать различные типы трапез, от простого перекуса с одной переменой блюд до изысканного обеда на 12 перемен, улучшаются API и реализация компонента.

В данном разделе будут рассмотрены способы конфигурирования MS.DI для связывания экземпляров `Meal` с соответствующими зависимостями `ICourse`. Усвоив этот материал, вы сможете составить точное представление о доступных вариантах конфигурирования экземпляров с последовательностями зависимостей.

Автоматическое связывание последовательностей

MS.DI неплохо разбирается в последовательностях, поэтому, если нужно воспользоваться всеми зарегистрированными компонентами заданного сервиса, автоматическое связывание по-прежнему окажется работоспособным. В качестве примера сервис `IMeal` можно сконфигурировать с его переменами блюд следующим образом:

```
services.AddTransient<ICourse, Rillettes>();
services.AddTransient<ICourse, CordonBleu>();
services.AddTransient<ICourse, MousseAuChocolat>();

services.AddTransient<IMeal, Meal>();
```

Заметьте, что это абсолютно стандартное отображение абстракций на конкретные типы. MS.DI автоматически разбирается с конструктором `Meal` и определяет, что правильным действием станет разрешение всех компонентов `ICourse`. При разрешении `IMeal` будет получен экземпляр `Meal` с `ICourse`-компонентами `Rillettes`, `CordonBleu` и `MousseAuChocolat`.

MS.DI обрабатывает последовательности в автоматическом режиме и, пока не указано иное, делает то, чего от него ожидают: разрешает последовательность зависимостей для всех регистраций заданного типа. Дополнительные действия потребуются, только если понадобится явно выбрать некоторые компоненты из более широкого набора. Посмотрим, как это можно сделать.

Выбор ограниченного числа компонентов из широкого набора

Стратегия, по умолчанию используемая MS.DI для внедрения компонентов, зачастую представляет собой правильную политику, иногда требуется выбрать лишь некоторые из широкого набора зарегистрированных компонентов (рис. 15.4).

ПРИМЕЧАНИЕ

> Потребность во внедрении поднабора из полной коллекции не относится к типовым сценариям, но показывает порядок удовлетворения более сложных потребностей.

Когда ранее контейнеру MS.DI было позволено выполнить автоматическое связывание всех сконфигурированных экземпляров, это соответствовало ситуации, изображенной в правой части рисунка. Если же нужно зарегистрировать компонент так, как показано в левой его части, намеченные к использованию компоненты следует определить явно. Чтобы получить желаемый результат, можно воспользоваться методом `AddTransient`, принимающим делегат. На этот раз работа ведется с конструктором `Meal`, принимающим только один параметр (листинг 15.6).

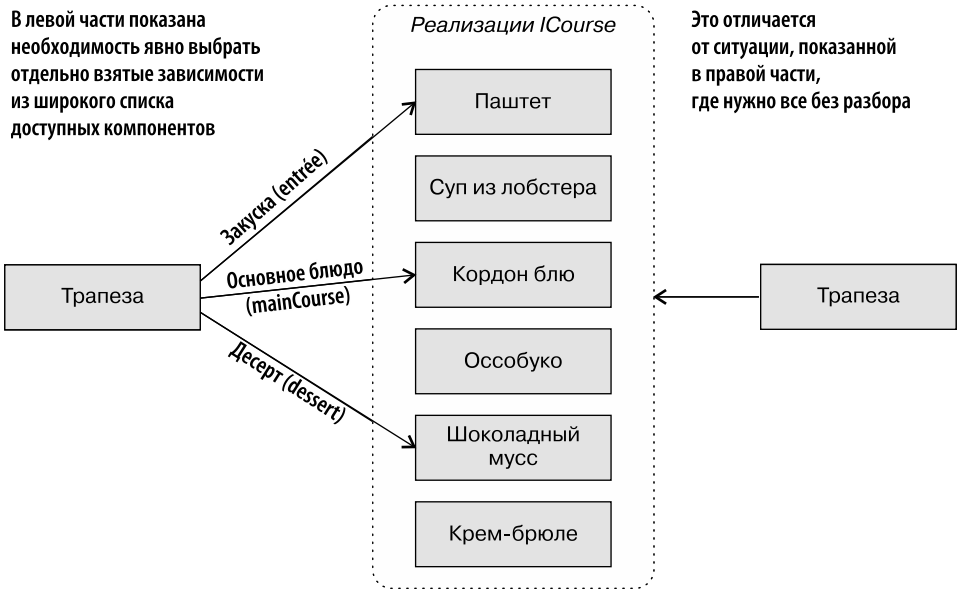


Рис. 15.4. Выбор компонентов из широкого набора зарегистрированных компонентов

Листинг 15.6. Внедрение поднабора ICourse в Meal

```
services.AddScoped<Rillettes>();
services.AddTransient<LobsterBisque>();
services.AddScoped<CordonBleu>();
services.AddScoped<OssoBuco>();
services.AddSingleton<MousseAuChocolat>();
services.AddTransient<CrèmeBrûlée>();
```

Регистрация всех блюд не по их интерфейсу, а по их конкретному типу. В данном случае используется несколько жизненных циклов

```
services.AddTransient<ICourse>(
    c => c.GetRequiredService<Rillettes>());
services.AddTransient<ICourse(
    c => c.GetRequiredService<LobsterBisque>());
services.AddTransient<ICourse>(
    c => c.GetRequiredService<CordonBleu>());
services.AddTransient<ICourse(
    c => c.GetRequiredService<OssoBuco>());
services.AddTransient<ICourse>(
    c => c.GetRequiredService<MousseAuChocolat>());
services.AddTransient<ICourse(
    c => c.GetRequiredService<CrèmeBrûlée>());
```

Регистрация всех блюд по их интерфейсу Icourse, что позволяет каждому из них быть разрешенным как IEnumerable<ICourse>. От рваных жизненных циклов удается избавиться за счет регистрации делегатов

```
services.AddTransient<IMeal>(c = new Meal(
    new ICourse[]
    {
        c.GetRequiredService<Rillettes>(),
        c.GetRequiredService<CordonBleu>(),
        c.GetRequiredService<MousseAuChocolat>()
    }));
```

Разрешение трех указанных перемен блюд по их конкретным типам и их внедрение в конструктор Meal

MS.DI разбирается в последовательностях. Этот контейнер автоматически все сделает правильно, если только не потребуется из всех сервисов заданного типа явно выбрать отдельно взятые компоненты. Автоматическое связывание работает не только с отдельными экземплярами, но и с последовательностями, и контейнер отображает последовательность на все сконфигурированные экземпляры соответствующего типа. Возможно, менее интуитивно понятным является рассматриваемое далее применение нескольких экземпляров одной и той же абстракции в паттерне проектирования «Декоратор».

15.4.3. Связывание декораторов

В подразделе 9.1.1 рассматривались преимущества, получаемые от использования паттерна проектирования «Декоратор» в процессе применения сквозной функциональности. Согласно определению, декораторы вводят в обращение несколько типов одной и той же абстракции. Имеются по крайней мере две реализации абстракции: сам декоратор и декорируемый тип. Если выстроить декораторы в цепочку, можно получить больше абстракций. Это еще один пример наличия нескольких регистраций одного и того же сервиса. В отличие от того, что было в предыдущих разделах, концептуально эти регистрации не равны — скорее всего, они являются зависимостями друг для друга.

Декорирование необобщенных абстракций

В MS.DI отсутствует встроенная поддержка декораторов, и это одна из областей, где ограничения MS.DI могут снижать производительность труда программиста. Все же мы покажем, как отчасти обойти эти ограничения.

Это упущение можно обойти, в очередной раз воспользовавшись классом `ActivatorUtilities`. Порядок его использования для применения `Breading` к `VealCutlet` показан в следующем примере:

```
services.AddTransient<IIngredient>(c =>
    ActivatorUtilities.CreateInstance<Breading>(
        c,
        ActivatorUtilities
            .CreateInstance<VealCutlet>(c)));
```

Регистрация блока кода, вызывающего CreateInstance для создания декоратора Breading Decorator с использованием автоматического связывания

Внедрение Breading с VealCutlet происходит за счет предоставления экземпляра VealCutlet массиву параметров. VealCutlet создается путем стандартного автоматического связывания

Из главы 9 известно, что кордон блю получается, если сделать в отбивной котлете надрез, начинить ее ветчиной, сыром и чесноком, а потом запанировать и поджарить.

В следующем примере показано, как декоратор `HamCheeseGarlic` добавляется между `VealCutlet` и декоратором `Breading`:

```
services.AddTransient<IIngredient>(c =>
    ActivatorUtilities.CreateInstance<Breading>(
        c,
        ActivatorUtilities
            .CreateInstance<HamCheeseGarlic>(
            c,
            ActivatorUtilities
                .CreateInstance<VealCutlet>(c))));
```

Добавление нового декоратора

За счет того, что `HamCheeseGarlic` становится зависимостью `Breading`, а `VealCutlet` — зависимостью `HamCheeseGarlic`, декоратор `HamCheeseGarlic` делается в графе объектов средним классом. В результате получается граф объектов, эквивалентный следующей версии, выполненной по технологии чистого DI:

```
new Breading(
    new HamCheeseGarlic(
        new VealCutlet()));
```

VealCutlet обернут в HamCheeseGarlic, который, в свою очередь, обернут в Breading

Нетрудно догадаться, что в MS.DI выстраивание декораторов в цепочку страдает сложностью и многословностью. Усугубим сложившуюся ситуацию и посмотрим, что произойдет при попытке применения декораторов к обобщенным абстракциям.

Декорирование обобщенных абстракций

В ходе изучения главы 10 было определено несколько обобщенных декораторов, подходящих для применения к любой реализации `ICommandService<TCommand>`. Далее в этой главе мы отставим в сторону все наши ингредиенты и перемены блюд и рассмотрим способы регистрации обобщенных декораторов с помощью MS.DI. В листинге 15.7 показано, как зарегистрировать все реализации `ICommandService<TCommand>` с тремя декораторами, представленными в разделе 10.3.

Листинг 15.7. Декорирование обобщенных абстракций, прошедших автоматическую регистрацию

```
Assembly assembly = typeof(AdjustInventoryService).Assembly;

var mappings =
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where !type.IsGenericType
    from i in type.GetInterfaces()
    where i.IsGenericType
    where i.GetGenericTypeDefinition()
        == typeof(ICommandService<>)
    select new { service = i, implementation = type };

foreach (var mapping in mappings)
{
    Type commandType =
        mapping.service.GetGenericArguments()[0];
```

Сканирование указанной сборки на наличие необобщенных реализаций ICommandService<TCommand>

Извлечение конкретного типа Tcommand из закрытой абстракции ICommandService<TCommand>

```
    Type secureDecoratoryType =
        typeof(SecureCommandServiceDecorator<>)
            .MakeGenericType(commandType);
    Type transactionDecoratorType =
        typeof(TransactionCommandServiceDecorator<>)
            .MakeGenericType(commandType);
    Type auditingDecoratorType =
        typeof(AuditingCommandServiceDecorator<>)
            .MakeGenericType(commandType);

    services.AddTransient(mapping.service, c =>
        ActivatorUtilities.CreateInstance(
            c,
            secureDecoratoryType,
            ActivatorUtilities.CreateInstance(
                c,
                transactionDecoratorType,
                ActivatorUtilities.CreateInstance(
                    c,
                    auditingDecoratorType,
                    ActivatorUtilities.CreateInstance(
                        c,
                        mapping.implementation)))));
}
```

Использование извлеченного commandType для создания закрытых обобщенных реализаций декораторов, требующих применения

Добавление регистрации делегата для закрытой абстракции ICommandService<TCommand>. Этот делегат несколько раз вызывает принадлежащий ActivatorUtilities метод CreateInstance для автоматического связывания всех декораторов и просканированных реализаций в качестве компонента, находящегося в самой внутренней области

Результат конфигурирования, выполненного кодом листинга 15.7, показан на рис. 15.5, который уже рассматривался в подразделе 10.3.4.

Если код в листинге 15.7 показался слишком сложным, то это, к сожалению, только начало. В нем множество недостатков, и обойти некоторые из них довольно трудно. Суть недостатков сводится к следующему.

Рис. 15.5. Оснащение реального сервиса команд аспектами ведения контрольного журнала, управления транзакциями и обеспечения безопасности

- Создание закрытых обобщенных типов декораторов может существенно усложниться, если один из имеющихся в декораторе аргументов обобщенного типа не полностью совпадет с аргументом абстракции[1].
- Невозможно добавлять открытые обобщенные реализации, применяющие декораторы, без принудительной явной регистрации каждой закрытой обобщенной абстракции.
- Усложняется условное использование декораторов, основанное, к примеру, на аргументах обобщенного типа.

[1] В качестве примера представьте себе CachingDecorator<TRequest, TResponse>, который реализует IHandler<TRequest, ReadOnlyCollection<TResponse>>.

- С применением альтернативного жизненного цикла становится трудно избежать рваных жизненных циклов при создании в реализации сразу нескольких интерфейсов.
- Возникают трудности с дифференциацией жизненных циклов — все декораторы в цепочке получают один и тот же жизненный цикл.

Можно попробовать поочередно преодолеть эти ограничения и предложить усовершенствованный вариант кода, показанного в листинге 15.7, но тогда, по сути, у вас получится разработка нового DI-контейнера поверх MS.DI, что нами не приветствуется. Это контрпродуктивно. Для такого сценария больше подойдут альтернативные варианты — использование Autofac и Simple Injector[1].

Хотя наиболее понятными на интуитивном уровне пользователями нескольких экземпляров одной и той же абстракции являются потребители, полагающиеся на последовательности зависимостей, хорошим примером могут послужить и декораторы. Есть и третий, возможно, несколько неожиданный случай выхода нескольких экземпляров на первый план — применение паттерна проектирования «Компоновщик» (Composite).

15.4.4. Связывание компоновщиков

Паттерн проектирования «Компоновщик» уже рассматривался в этой книге. Например, в подразделе 6.1.2 создан сервис `CompositeNotificationService` (см. листинг 6.4), в котором не только реализован сервис `INotificationService`, но и сформирована последовательность реализаций такого сервиса.

Связывание необобщенных компоновщиков

Посмотрим, как в MS.DI можно зарегистрировать компоновщики, подобные `CompositeNotificationService` из главы 6. В листинге 15.8 этот класс показан еще раз.

Листинг 15.8. Компоновщик CompositeNotificationService из главы 6

```
public class CompositeNotificationService : INotificationService
{
    private readonly IEnumerable<INotificationService> services;

    public CompositeNotificationService(
        IEnumerable<INotificationService> services)
    {
        this.services = services;
    }
```

[1] Как показано в листингах 13.10 и 14.11, и Autofac, и Simple Injector позволяют полностью реализовать данный сценарий всего в нескольких строчках кода.

```
    public void OrderApproved(Order order)
    {
        foreach (INotificationService service in this.services)
        {
            service.OrderApproved(order);
        }
    }
}
```

Регистрация компоновщика требует, чтобы он, будучи внедренным с последовательностью разрешенных экземпляров, был добавлен в качестве регистрации по умолчанию:

```
services.AddTransient<OrderApprovedReceiptSender>();
services.AddTransient<AccountingNotifier>();
services.AddTransient<OrderFulfillment>();

services.AddTransient<INotificationService>(c =>
    new CompositeNotificationService(
        new INotificationService[]
        {
            c.GetRequiredService<OrderApprovedReceiptSender>(),
            c.GetRequiredService<AccountingNotifier>(),
            c.GetRequiredService<OrderFulfillment>(),
        }));
```

В этом примере три реализации `INotificationService` регистрируются по их конкретному типу с использованием имеющегося в MS.DI API автоматического связывания. А вот `CompositeNotificationService` регистрируется с помощью делегата (листинг 15.9). Внутри делегата компоновщик обновляется в ручном режиме и внедряется с массивом экземпляров `INotificationService`. Ранее выполненные регистрации проходят разрешение за счет указания конкретных типов.

Поскольку количество уведомительных сервисов со временем, скорее всего, возрастет, нагрузку на корень композиции можно уменьшить, применив автоматическую регистрацию. Сканировать сборки придется самостоятельно, поскольку, как выяснилось ранее, функций для этого в MS.DI нет.

Листинг 15.9. Регистрация CompositeNotificationService

```
Assembly assembly = typeof(OrderFulfillment).Assembly;

Type[] types = (
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where typeof(INotificationService).IsAssignableFrom(type)
    select type)
    .ToArray();

foreach (Type type in types)
{
    services.AddTransient(type);
}
```

Материализация результатов запроса в массив

```
services.AddTransient<INotificationService>(c =>
    new CompositeNotificationService(
        types.Select(t =>
            (INotificationService)c.GetRequiredService(t))
        .ToArray()));
```

По сравнению с декоратором из листинга 15.7 код листинга 15.9 выглядит довольно просто. Сборка сканируется на наличие реализаций `INotificationService`, и каждый найденный тип добавляется к коллекции сервисов. Массив типов используется регистрацией `CompositeNotificationService`. Компоновщик внедряется с последовательностью экземпляров `INotificationService`, которая разрешается последовательным перебором элементов массива типов.

ПРИМЕЧАНИЕ

Получающиеся при выполнении кода листинга 15.9 типы материализуются в массив. Это не дает инструкции Select внутри регистрации компоновщика при каждом разрешении снова и снова перебирать все типы сборки OrderFulfillment, что легко ухудшит производительность приложения, если типов в сборке будет много.

Возможно, вы уже привыкли к неизбежным в ходе работы с MS.DI сложности и многословию, но это еще не все. Наш LINQ-запрос зарегистрирует любую необобщенную реализацию, создающую `INotificationService`. При попытке запуска на выполнение предыдущего кода MS.DI в зависимости от сборки, в которой находится компоновщик, может выдать исключение со следующим сообщением: Exception of type 'System.StackOverflowException' was thrown (Было выдано исключение типа 'System.StackOverflowException').

Да! Исключения, связанные с переполнением стека, воспринимаются болезненно, поскольку они прерывают процесс выполнения и плохо поддаются отладке. Кроме того, универсальное исключение не дает подробной информации о том, что стало причиной переполнения стека. Хотелось бы, чтобы контейнер MS.DI выдавал более информативное исключение, объясняющее суть цикла, как это делают Autofac и Simple Injector.

ПРИМЕЧАНИЕ

Работая над этой главой, мы обнаружили, что сообщения об исключениях, выданных MS.DI, носят общий характер или непонятны, из-за чего устранить неполадки труднее, чем в большинстве популярных DI-контейнеров. У многих хорошо проработанных DI-контейнеров сообщения о причинах выдачи исключений довольно четкие.

Это переполнение стека вызвано зацикленной зависимостью в `CompositeNotificationService`. Компоновщик выбран LINQ-запросом и разрешен в качестве части последовательности. В результате получилось, что компоновщик зависит сам от себя. Получился граф объектов, который по этой причине не может создать ни MS.DI, ни какой-то другой DI-контейнер. `CompositeNotificationService` стал

частью последовательности, потому что наш LINQ-запрос нашел все необобщенные реализации `INotificationService`, куда входит и компоновщик.

Обойти данную проблему можно несколькими способами. Самым простым решением будет перемещение компоновщика в другую сборку, например содержащую корень композиции. Тогда LINQ-запрос не сможет выбрать этот тип. Еще один вариант заключается в удалении `CompositeNotificationService` из списка за счет применения фильтра:

```
Type[] types = (
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where typeof(INotificationService)
        .IsAssignableFrom(type)
    where type != typeof(CompositeNotificationService)
    select type)
    .ToArray();
```

Установка фильтра на составитель

Но классы компоновщиков не единственные, которые могут требовать удаления. То же самое придется делать и с любым декоратором. Это не особо сложно, но, поскольку реализаций декораторов, как правило, бывает больше, стоит запросить информацию о типе, чтобы узнать, представляет тип декоратор или нет. Вот как можно отфильтровать декораторы:

```
Type[] types = (
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where typeof(INotificationService).IsAssignableFrom(type)
    where type != typeof(CompositeNotificationService)
    where type => !IsDecoratorFor<INotificationService>(type)
    select type)
    .ToArray();
```

Следующий код показывает метод `IsDecoratorFor`:

```
private static bool IsDecoratorFor<T>(Type type)
{
    return typeof(T).IsAssignableFrom(type) &&
        type.GetConstructors()[0].GetParameters()
            .Any(p => p.ParameterType == typeof(T));
}
```

Метод `IsDecoratorFor` предполагает, что у типа имеется только один конструктор. Тип считается декоратором, не только когда он реализует заданную абстракцию `T`, но и когда его конструктору также требуется `T`.

Связывание обобщенных компоновщиков

В подразделе 15.4.3 показан порядок регистрации обобщенных декораторов. В этом разделе рассмотрим способ регистрации компоновщиков для обобщенных абстракций.

В подразделе 6.1.3 класс `CompositeEventHandler<TEvent>` (из листинга 6.12) был указан в качестве реализации компоновщика последовательности реализаций `IEventHandler<TEvent>`. Посмотрим, можно ли зарегистрировать компоновщик

с заключенными в него реализациями обработчиков событий. Чтобы справиться с этим в MS.DI, придется проявить смекалку, поскольку нужно будет обойти несколько досадных ограничений.

Мы обнаружили, что самый простой способ сокрытия реализаций обработчиков событий за компоновщиком заключается в полном отказе от регистрации таких реализаций и ее замене перемещением конструкции обработчиков в компоновщик. Выглядит эта процедура неприглядно, но с задачей справляется. Чтобы скрыть обработчики за компоновщиком, придется переписать реализацию `CompositeEventHandler<TEvent>` из листинга 6.12, сделав ее такой, как показано в листинге 15.10.

Листинг 15.10. MS.DI-совместимая реализация CompositeEventHandler<TEvent>

```
public class CompositeSettings
{
    public Type[] AllHandlerTypes { get; set; }
}

public class CompositeEventHandler<TEvent>
    : IEventHandler<TEvent>
{
    private readonly IServiceProvider provider;
    private readonly CompositeSettings settings;

    public CompositeEventHandler(
        IServiceProvider provider,
        CompositeSettings settings)
    {
        this.provider = provider;
        this.settings = settings;
    }

    public void Handle(TEvent e)
    {
        foreach (var handler in this.GetHandlers())
        {
            handler.Handle(e);
        }
    }

    IEnumerable<IEventHandler<TEvent>> GetHandlers()
    {
        return
            from type in this.settings.AllHandlerTypes
            where typeof(IEventHandler<TEvent>)
                .IsAssignableFrom(type)
            select (IEventHandler<TEvent>)
                ActivatorUtilities.CreateInstance(
                    this.provider, type);
    }
}
```

Объект-параметр, позволяющий внедрять полный список обработчиков событий в составитель

Составитель зависит как от объекта-параметра, так и от IServiceProvider. А IServiceProvider позволяет ему выполнить разрешение обработчиков

Последовательный перебор элементов списка обработчиков и их поочередный вызов

Выбор только тех типов, в которых реализуется заданный интерфейс. В случае вызова, находящегося в составителе обработчика EventHandler<OrderApproved>, будут выбраны только те типы, в которых реализуется IEventHandler<OrderApproved>

Автоматическое связывание выбранного типа

По сравнению с исходной реализацией из листинга 6.12 эта реализация компоновщика намного сложнее. К тому же она сильно зависит от самого MS.DI, поскольку в ней задействуются имеющиеся в контейнере `IServiceProvider` и `ActivatorUtilities`. С учетом данной зависимости этот компоновщик, безусловно, принадлежит корню композиции, поскольку остальная часть приложения не должна замечать использования DI-контейнера.

Теперь компоновщик зависит не от последовательности `IEventHandler<TEvent>`, а от граничного объекта, в котором содержатся все типы обработчиков, в том числе те, которые не могут быть приведены к указанной закрытой обобщенной последовательности компоновщика `IEventHandler<TEvent>`. По этой причине компоновщик берет на себя ту часть работы, которую должен делать DI-контейнер. Он отфильтровывает все неподходящие типы путем вызова метода `typeof(IEventHandler<TEvent>).IsAssignableFrom(type)`. Вам остается лишь зарегистрировать компоновщик и просканировать все обработчики событий (листинг 15.11).

Листинг 15.11. Регистрация CompositeEventHandler<TEvent>

```
var handlerTypes =
    from type in assembly.GetTypes()
    where !type.IsAbstract
    where !type.IsGenericType
    let serviceTypes = type.GetInterfaces()
        .Where(i => i.IsGenericType &&
            i.GetGenericTypeDefinition()
                == typeof(IEventHandler<>))
    where serviceTypes.Any()
    select type;

services.AddSingleton(new CompositeSettings
{
    AllHandlerTypes = handlerTypes.ToArray()
});

services.AddTransient(
    typeof(IEventHandler<>),
    typeof(CompositeEventHandler<>));
```

Сканирование сборки на наличие всех необобщенных классов, реализующих IEventHandler<TEvent>

Регистрация объекта-параметра, позволяющая передавать список типов в составитель

Регистрация составителя

Вместе с раздутой реализацией компоновщика в этом последнем листинге показана реализация паттерна «Компоновщик» в сочетании с MS.DI.

СОВЕТ

Если потребуется применить декораторы к отдельно взятым обработчикам событий, можно использовать трюк с подмешиванием кода листинга 15.7 в код метода GetHandlers из листинга 15.10. Компоновщик при этом становится ответственным за создание графа объектов, включая декораторы.

Несмотря на то что мы успешно обошли некоторые ограничения, присущие MS.DI, в других случаях нам может и не повезти. Например, удача может отвернуться, если

последовательность элементов состоит как из необобщенных, так и из обобщенных реализаций, причем на последние действуют ограничения, присущие обобщенному типу, или когда декораторы должны быть условными.

Признаем, что это не лучшее решение. Мы предпочли сэкономить на объеме кода, чтобы показать, как применять MS.DI к паттернам, представленным в этой книге, не раскрыв всех особенностей этого контейнера. Именно поэтому в своей повседневной практике разработки программных средств отдаем предпочтение чистой технологии DI или одному из хорошо проработанных DI-контейнеров — Autofac или Simple Injector.

Неважно, выберете ли вы тот или иной DI-контейнер или предпочтете чистую технологию DI, мы надеемся, что эта книга довела до вас важное утверждение: внедрение зависимостей не зависит от конкретной технологии, например от какого-то особенного DI-контейнера. Приложение может и должно быть разработано с использованием подходящих к технологии DI паттернов и практических приемов, представленных здесь. Когда вы добьетесь в этом определенных успехов, выбор DI-контейнера станет играть более скромную роль. DI-контейнер — всего лишь инструмент, составляющий ваше приложение, но в идеале вы должны иметь возможность заменять один контейнер другим, не перезаписывая какую-либо часть приложения, кроме корня композиции.

Резюме

- DI-контейнер Microsoft.Extensions.DependencyInjection (MS.DI) имеет ограниченный набор функций. В нем нет полноценного API, справляющегося с автоматической регистрацией, декораторами и компоновщиками. Поэтому он хуже подходит для разработки приложений, создаваемых вокруг принципов и паттернов, представленных в этой книге.
- MS.DI обеспечивает строгое разделение вопросов конфигурирования и потребления контейнера. Конфигурирование компонентов выполняется с использованием экземпляра `ServiceCollection`, но он не может выполнять их разрешение. Когда конфигурирование экземпляра `ServiceCollection` завершено, он задействуется для создания `ServiceProvider`, который можно применить для разрешения компонентов.
- При использовании MS.DI непосредственное разрешение из корневого контейнера считается неприемлемым. Оно может привести к утечкам памяти или ошибкам одновременных вычислений. Разрешение всегда нужно выполнять из `IServiceScope`.
- MS.DI поддерживает три стандартных жизненных цикла: Transient, Singleton и Scoped.

Глоссарий

Здесь приводятся краткие определения некоторых терминов, паттернов и других понятий, рассматриваемых в книге. Каждое определение включает ссылку на главу или раздел, где оно раскрывается подробнее.

- **DI-контейнер** — библиотека программных модулей, предоставляющая функциональные возможности внедрения зависимостей и автоматизирующая многие задачи, связанные с составлением композиции объектов, перехватом и управлением временем жизни объектов. Это механизм, выполняющий разрешение графа объектов и управляющий им. См. главу 12.
- **SOLID** — акроним, который служит для обозначения пяти основополагающих принципов проектирования: принципа единственной ответственности (Single Responsibility Principle), принципа открытости/закрытости (Open/Closed Principle), принципа подстановки Лисков (Liskov Substitution Principle), принципа изоляции интерфейса (Interface Segregation Principle) и принципа инверсии зависимостей (Dependency Inversion Principle). См. главу 10.
- **Абстракция** — обобщающий термин, охватывающий как интерфейсы, так и абстрактные базовые классы. См. главу 1.
- **Абстракция с протечкой** — абстракция, у которой имеется определение, через которое просматриваются подробности реализации, что привязывает ее к реализации. См. подраздел 6.2.1.
- **Автоматическая регистрация** — возможность автоматической регистрации компонентов на основе конкретного соглашения в DI-контейнере путем сканирования одной или нескольких сборок на предмет реализации желаемых абстракций. См. подраздел 12.2.3.
- **Автоматическое связывание** — способность автоматически составить граф объектов из отображения между абстракциями и конкретными типами, используя информацию о типе, предоставленную компилятором и общеязыковой исполняющей средой (CLR). См. раздел 12.1.2.
- **Аспектно-ориентированное программирование** (Aspect-Oriented Programming (AOP)) — подход к созданию программных продуктов, нацеленный на сокращение

объема рутинного кода, требующегося для реализации сквозной функциональности и других паттернов создания программного кода. Эта цель достигается реализацией таких паттернов в одном месте и их применением к кодовой базе либо декларативно, либо на основе соглашений без внесения изменений в сам код. См. главу 10.

- **Внедрение через конструктор** — паттерн DI, где зависимости статически определены в виде списка параметров для конструктора класса. См. раздел 4.2.
- **Внедрение через метод** — паттерн DI, в котором зависимости внедряются в потребитель в виде параметров метода. См. раздел 4.3.
- **Внедрение через метод записи значения** — см. *Внедрение через свойство*.
- **Внедрение через свойство** — паттерн DI, в котором зависимости внедряются в потребитель через доступные для записи свойства. См. раздел 4.4.
- **Внешняя реализация по умолчанию** — реализация по умолчанию нестабильной зависимости, определение которой находится в модуле, отличном от модуля потребителя. См. подраздел 5.1.3.
- **Временная связанность** — проблемный код, возникающий, когда существует подразумеваемая связь между двумя и более компонентами класса, заставляющая клиентов вызывать один компонент перед другим. См. подраздел 4.3.2.
- **Время жизни зависимости** — см. *Время жизни объекта*.
- **Время жизни объекта** — в общем смысле этот термин охватывает понятия создания и высвобождения любого объекта. В контексте DI он относится к времени жизни зависимостей. См. главу 8.
- **Диктатор** — антипаттерн DI, где код полагается на применение нестабильной зависимости в любом месте, исключая корень композиции. Он нарушает принцип инверсии зависимостей. См. раздел 5.1.
- **Жизненный цикл** — формализованный способ описания предполагаемой продолжительности жизни зависимости. См. главу 8.
- **Жизненный цикл Scoped** — жизненный цикл, при котором имеется всего один экземпляр в пределах одной четко определенной области видимости или одного запроса и экземпляры не используются совместно сразу в нескольких областях видимости. См. подраздел 8.3.3.
- **Жизненный цикл Singleton** — жизненный цикл, при котором один и тот же экземпляр многократно используется всеми потребителями в пределах области видимости одного и того же компоновщика. См. подраздел 8.3.1.
- **Жизненный цикл Transient** — жизненный цикл, при котором все потребители получают собственный экземпляр зависимости. См. подраздел 8.3.2.
- **Зависимость** — в принципе, любая ссылка на другой модуль, имеющаяся у модуля. Когда модуль ссылается на другой модуль, он от него зависит. Неформально термин «зависимость» используется вместо более строгого термина «нестабильная зависимость». См. главу 1.

- **Захваченная зависимость** — зависимость, которая слишком долго и непреднамеренно остается жизнеспособной по причине того, что ее потребителю задано более продолжительное время жизни по сравнению с ожидаемым временем жизни зависимости. См. подраздел 8.4.1.
- **Инверсия управления** — понятие, означающее, что вместо непосредственного контроля над объектами контроль над их временем жизни разрешается осуществлять среде выполнения. См. главу 1.
- **Компоновка объектов** — составление приложений из отличных друг от друга модулей. См. главу 7.
- **Компоновщик** — обобщенное название любого объекта или метода, выполняющего компоновку зависимостей. См. главу 8.
- **Конфигурация в виде кода** — технология, позволяющая сохранять конфигурацию DI-контейнера в исходном коде. Каждое отображение между абстракцией и конкретной реализацией явно выражается непосредственно в коде. См. подраздел 12.2.2.
- **Корень композиции** — центральное место в приложении, когда приложение составлено из образующих его модулей. См. раздел 4.1.
- **Локальная реализация по умолчанию** — реализация абстракции по умолчанию, которая определена в той же сборке, что и потребитель. См. подраздел 4.2.2.
- **Локатор сервисов** — антипаттерн DI, предоставляющий компонентам приложения, находящимся вне корня композиции, доступ к неограниченному набору нестабильных зависимостей. См. раздел 5.2.
- **Нестабильная зависимость** — зависимость, вызывающая временами нежелательные побочные эффекты. Она может включать не существующие в данный момент модули или модули, предъявляющие неприемлемые требования к своей среде выполнения. Этот термин относится к зависимостям, рассматриваемым в рамках технологии DI. См. подраздел 1.3.2.
- **Ограниченная конструкция** — антипаттерн DI, вынуждающий все реализации заданной абстракции требовать от своих конструкторов наличия одинаковой сигнатуры. См. раздел 5.4.
- **Окружающий контекст** — антипаттерн DI, который за счет использования статических компонентов класса предоставляет коду приложения, находящемуся вне корня композиции, глобальный доступ к нестабильной зависимости или ее поведению. См. раздел 5.3.
- **Перехват** — возможность перехвата вызовов между двумя взаимодействующими компонентами, позволяющий обогатить или изменить поведение зависимости, не внося изменений в сами взаимодействующие компоненты. См. главу 9.
- **Принцип единственной ответственности** — принцип, который гласит, что у каждого класса должна быть только одна сфера ответственности. В акрониме SOLID он представлен буквой S. См. подраздел 2.1.3. См. также *SOLID*.

- **Принцип изоляции интерфейса** — принцип, который гласит, что никакого клиента нельзя принуждать быть зависимым от не используемых им методов. В акрониме SOLID он представлен буквой I. См. подраздел 6.2.1. См. также *SOLID*.
- **Принцип инверсии зависимостей** — принцип, который гласит, что имеющиеся в приложениях модули более высокого уровня не должны зависеть от модулей более низкого уровня, вместо этого модули обоих уровней должны зависеть от абстракций. В акрониме SOLID он представлен буквой D. См. подраздел 3.1.2. См. также *SOLID*.
- **Принцип открытости/закрытости** — принцип, утверждающий, что классы должны быть открыты для расширения, но закрыты для изменения. В акрониме SOLID он представлен буквой O. См. подраздел 4.4.2. См. также SOLID.
- **Принцип подстановки Лисков** — принцип проектирования программных продуктов, который гласит, что у потребителя должна быть возможность воспользоваться любой реализацией абстракции, не нарушая корректности системы. В акрониме SOLID он представлен буквой L. См. подраздел 10.2.3. См. также *SOLID*.
- **Разделение команд и запросов** — идея, заключающаяся в том, что каждый метод должен либо возвращать результат, но не изменять наблюдаемое состояние системы, либо изменять состояние, но не производить никакого значения. См. подраздел 10.3.3.
- **Сквозная функциональность** — аспект программы, затрагивающий основную часть приложения. Зачастую он является нефункциональным требованием. Типичными примерами являются регистрация событий, ведение контрольного журнала, контроль доступа и проверка допустимости. См. главу 9.
- **Стабильная зависимость** — зависимость, на которую можно ссылаться без каких-либо неблагоприятных последствий. Противоположность понятию «нестабильная зависимость». См. подраздел 1.3.1.
- **Сущность** — предметный объект с присущей ему долгосрочной идентичностью. См. подраздел 3.1.2.
- **Тестируемость** — степень допустимости применения к приложению автоматизированных модульных тестов. См. главу 1.
- **Управление временем жизни** — см. *Время жизни объекта*.
- **Чистая технология DI** — практика применения DI без использования DI-контейнера. См. часть III.
- **Шов** — место в коде приложения, где абстракции используются для разделения модулей. См. главу 1.

Список дополнительных источников

Книги

- *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования = Design Patterns: Elements of Reusable Object-Oriented Software. — СПб.: Питер, 2020.
- *Мартин Р. К.* Чистый код. Создание, анализ и рефакторинг = Clean Code: A Handbook of Agile Software Craftsmanship. — СПб.: Питер, 2019.
- *Мартин Р. К., Ньюкирк Дж. В., Косс Р. С.* Быстрая разработка программ. Принципы, примеры, практика = Agile Software Development, Principles, Patterns and Practices. — М.: Вильямс, 2004.
- *Фаулер М.* Рефакторинг. Улучшение существующего кода = Refactoring: Improving the Design of Existing Code. — М.: Символ-Плюс, 2008.
- *Фаулер М.* Шаблоны корпоративных приложений = Patterns of Enterprise Application Architecture. — М.: Диалектика, 2018.
- *Хант Э., Томас Д.* Программист-прагматик. Путь от подмастерья к мастеру = The Pragmatic Programmer: From Journeyman to Master. — М.: Лори, 2009.
- *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем = Domain-Driven Design: Tackling Complexity in the Heart of Software. — М.: Вильямс, 2011.
- *Boike D.* Learning NServiceBus, 2nd ed. — Packt Publishing, 2015.
- *Brown W. J. et al.* AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. — Wiley Computer Publishing, 1998.
- *Chatterjee A.* Building Apps for the Universal Windows Platform. — Apress, 2017.

- *Cwalina K., Abrams B.* Framework Design Guidelines: Conventions, Idioms and Patterns for Reusable .NET Libraries, 2nd Ed. — Addison-Wesley, 2009.
- *Feathers M. C.* Working Effectively with Legacy Code. — Prentice Hall, 2004.
- *Groves M. D.* AOP in .NET. — Manning, 2013.
- *Howard M., LeBlanc D.* Writing Secure Code, 2nd Ed. — Microsoft Press, 2003.
- *Lock A.* ASP.NET Core in Action. — Manning, 2018.
- *Martin R. C. et al.* Pattern Languages of Program Design 3. — Addison-Wesley, 1998.
- *Meszaros G.* xUnit Test Patterns: Refactoring Test Code. — Addison-Wesley, 2007.
- *Meyer B.* Object-Oriented Software Construction. — ISE Inc., 1988.
- *Nygard M. T.* Release It! Design and Deploy Production-Ready Software. — Pragmatic Bookshelf, 2007.
- *Osherove R.* The Art of Unit Testing, 2nd Ed. — Manning, 2013.
- *Smith J.* Entity Framework Core in Action. — Manning, 2018.

Онлайн-ресурсы

- *Atwood J.* The Problem with Logging (2008), https://blog.codinghorror.com/the-problem-with-logging/.
- *Deursen van S.* Meanwhile, on the Query Side of my Architecture (2011), https://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=92.
- *Deursen van S., Parker P.* What's Wrong with the ASP.NET Core DI Abstraction? (2016), https://simpleinjector.org/blog/2016/06/.
- *Deursen van S. et al.* Implementing Row Based Security (2014), https://github.com/dotnetjunkie/solidservices/issues/4.
- *Deursen van S. et al.* Logger Wrapper Best Practice (2011), https://stackoverflow.com/questions/5646820/logger-wrapper-best-practice.
- *Fowler M.* Domain Event (2005), https://martinfowler.com/eaaDev/DomainEvent.html.
- *Fowler M.* Event Sourcing (2005), https://martinfowler.com/eaaDev/EventSourcing.html.
- *Fowler M.* Introduce Parameter Object (1999), https://refactoring.com/catalog/introduceParameterObject.html.
- *Fowler M.* Inversion of Control Containers and the Dependency Injection Pattern (2004), https://martinfowler.com/articles/injection.html.
- *Fowler M.* Inversion Of Control (2005), https://martinfowler.com/bliki/InversionOfControl.html.
- *Gorman J.* Reused Abstractions Principle (RAP) (2010), http://www.codemanship.co.uk/parlezuml/blog/?postid=934.
- *Heintz J.* The Outbox Pattern (2014), http://gistlabs.com/2014/05/the-outbox/.
- *Lippert E.* Immutability in C# Part One: Kinds of Immutability (2007), https://blogs.msdn.microsoft.com/ericlippert/2007/11/13/immutabilityin-c-part-one-kinds-of-immutability/.

- *Munsch J. et al.* How to Explain Dependency Injection to a 5-year old (2009), https://stackoverflow.com/questions/1638919/.
- *Palermo J.* Constructor Over-injection Smell — follow up (2010), https://jeffreypalermo.com/2010/01/constructor-over-injection-smell-ndash-follow-up/.
- *Seemann M.* Interfaces are Not Abstractions (2010), https://blog.ploeh.dk/2010/12/02/Interfacesarenotabstractions/.
- *Seemann M.* Passive Attributes (2014), https://blog.ploeh.dk/2014/06/13/passive-attributes/.
- *Seemann M.* Pure DI (2014), https://blog.ploeh.dk/2014/06/10/pure-di/.
- *Seemann M.* Service Locator 2 Released (2007), https://blogs.msdn.microsoft.com/ploeh/2007/03/15/service-locator-2-released/.
- *Seemann M.* The Register Resolve Release Pattern (2010), https://blog.ploeh.dk/2010/09/29/TheRegisterResolveReleasepattern/.
- *Smith J.* Patterns: WPF Apps with the Model-View-ViewModel Design Pattern (2009), https://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

Другие ресурсы

- Autofac: https://autofac.org.
- Common Service Locator: https://github.com/unitycontainer/commonservicelocator.
- Common.Logging: https://github.com/net-commons/common-logging.
- JSON.NET: https://www.newtonsoft.com/json.
- log4net: https://logging.apache.org/log4net/.
- Microsoft.Extensions.DependencyInjection: https://github.com/aspnet/DependencyInjection.
- PostSharp: https://www.postsharp.net/.
- Simple Injector: https://simpleinjector.org.

Марк Симан, Стивен ван Дерсен

Внедрение зависимостей на платформе .NET

2-е издание

Перевел на русский *Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*М. Сагалович*
Литературные редакторы	*Н. Рощина, А. Тазеева*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 23.12.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 49,020. Тираж 700. Заказ 0000.

Владимир Хориков

ПРИНЦИПЫ ЮНИТ-ТЕСТИРОВАНИЯ

Юнит-тестирование — это процесс проверки отдельных модулей программы на корректность работы. Правильный подход к тестированию позволит максимизировать качество и скорость разработки проекта. Некачественные тесты, наоборот, могут нанести вред: нарушить работоспособность кода, увеличить количество ошибок, растянуть сроки и затраты. Грамотное внедрение юнит-тестирования — хорошее решение для развития проекта.

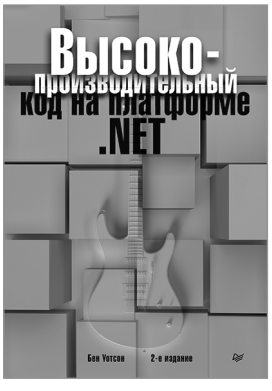
Научитесь разрабатывать тесты профессионального уровня, без ошибок автоматизировать процессы тестирования, а также интегрировать тестирование в жизненный цикл приложения. Со временем вы овладеете особым чутьем, присущим специалистам по тестированию. Как ни удивительно, практика написания хороших тестов способствует созданию более качественного кода.

В этой книге:

- Универсальные рекомендации по оценке тестов.
- Тестирование для выявления и исключения антипаттернов.
- Рефакторинг тестов вместе с рабочим кодом.
- Использование интеграционных тестов для проверки всей системы.

Бен Уотсон

ВЫСОКОПРОИЗВОДИТЕЛЬНЫЙ КОД НА ПЛАТФОРМЕ .NET. 2-Е ИЗДАНИЕ



Хотите выжать из вашего кода на .NET максимум производительности? Эта книга развеивает мифы о CLR, рассказывает, как писать код, который будет просто летать. Воспользуйтесь ценнейшим опытом специалиста, участвовавшего в разработке одной из крупнейших .NET-систем в мире.

В этом издании перечислены все достижения и улучшения, внесенные в .NET за последние несколько лет, в нем также значительно расширен охват инструментов, содержатся дополнительные темы и руководства.

Вот лишь некоторые из тем, рассматриваемых в книге:

- Различные способы анализа куч и выявления проблем, связанных с памятью.
- Профессиональное использование Visual Studio и других инструментов.
- Дополнительные сведения об эталонном тестировании.
- Новые варианты настройки сборки мусора.
- Приемы предварительной подготовки кода.
- Более подробный анализ LINQ.