ГЛАВА 2

Зависимости и разделение на уровни

По завершении этой главы вы сможете делать следующее:

- управлять сложными зависимостями на уровне отдельных методов и сборок;
- выявлять участки прикладного кода, где зависимости оказываются самыми сложными, и пользоваться инструментальными средствами для снижения сложности зависимостей;
- разбивать прикладной код на более мелкие, адаптивные фрагменты функциональных возможностей, способствующие его повторному использованию;
- применять шаблоны разделения на уровни там, где они наиболее полезны;
- ясно понимать, каким образом разрешаются зависимости, и устранять затруднения, к которым они приводят;
- скрывать реализации за простыми интерфейсами.

У каждого программного обеспечения имеются свои зависимости. Каждая зависимость является основной в пределах одной и той же кодовой базы, сторонней во внешней сборке или сплошной на платформе NET Framework корпорации Microsoft. Все эти три вида зависимостей используются в большинстве нетривиальных проектов.

Зависимость абстрагирует функциональные возможности от вызывающего кода. Это избавляет от необходимости особенно беспокоиться о том, *что* и тем более *как* делает зависимость. Но в то же время нужно непременно добиться того, чтобы управление всеми зависимостями осуществлялось правильно. Если управление цепочкой зависимостей организовано плохо, то разработчики принудительно вносят ненужные зависимости, тем самым завязывая прикладной код в запутанные узлы с ложными ссылками на сборки. Вам, возможно, приходилось слышать, что самым правильным считается код, который еще не написан. Аналогично самой управляемой считается такая зависимость, которая не существует.

Чтобы сохранить прикладной код адаптивным к изменениям, нужно научиться эффективно управлять зависимостями. Это относится ко всем уровням программного обеспечения: от архитектурных зависимостей между подсистемами до реализации зависимостей между отдельными методами. Приложение с неудачно построенной архитектурой способно замедлить выпуск рабочего программного обеспечения, а в худшем случае — даже полностью остановить его разработку.

Трудно переоценить важность выбора самого лучшего подхода к управлению зависимостями. Если пойти на компромисс по такому важному вопросу, то можно обнаружить временное увеличение в скорости работы, но в долгосрочной перспективе это способно привести к полному провалу проекта. Это слишком знакомая история, когда резкое повышение производительности в краткосрочной перспективе быстро

падает по мере увеличения объема кода и количества модулей. В конечном итоге прикладной код становится жестким и ненадежным, а продвижение его разработки резко замедляется. С точки зрения артефактов и количественных показателей по методике Scrum форма диаграммы сгорания спринта выравнивается, поскольку очки за историю не присуждаются. И если затруднение не разрешено, то диаграмма сгорания функционального средства следует заданному образцу, потому что ни одно из функциональных средств не завершено. В итоге возрастает даже количество программных ошибок. Когда структура зависимостей становится необъятной, изменения в одном модуле могут вызвать побочный эффект в другом, возможно, не связанным с ним напрямую модуле.

Соблюдая дисциплину и осознанный подход к делу, можно легко управлять зависимостями. Имеются установившиеся шаблоны, помогающие организовать приложение в краткие сроки таким образом, чтобы приспособить его к изменениям в долгосрочной перспективе. Разделение на уровни является одним из наиболее распространенных архитектурных шаблонов, и в этой главе обсуждаются разные варианты разделения на уровни в дополнение к другим способам управления зависимостями.

Определение зависимости

Что же такое зависимость? В целом, *зависимость* — это отношение двух разных сущностей, посредством которой одна из них не может выполнять некоторую функцию (или даже существовать) без другой. Подходящей аналогией такого отношения может служить *финансовая зависимость* одного человека от другого. Зачастую в юридических документах требуется заявлять о наличии любых иждивенцев, т.е. лиц, финансово зависящих от составителя документа. К таким лицам, как правило, относятся супруги или дети.

Если перенести приведенное выше определение на прикладной код, то роль сущностей зачастую выполняют сборки. Так, в сборке A может использоваться сборка B, и поэтому сборка A *зависит* от сборки B. Как правило, такое отношение между сборками устанавливается следующим образом: сборка A является *клиентом* сборки B, тогда как сборка B — *службой* сборки A. В итоге сборка A не может функционировать без сборки B. Следует, однако, иметь в виду, что сборка B не только *не* зависит от сборки A, но и *не должна* и *не может* от нее зависеть. Такое отношение клиента и службы наглядно показано на рис. 2.1.

Рис. 2.1. *В любом отношении зависимости зависимая сущность обозначается как клиент, а сущность, от которой она зависит, — как служба*

На протяжении всей данной книги рассматриваемый код обсуждается с точки зрения *клиента* и *службы*. И хотя некоторые службы размещаются удаленно, например, службы, созданные средствами Windows Communication Foundation (WCF), это совсем не обязательно для кода, обозначаемого как служба. Весь код относится к службе или клиенту в зависимости от точки зрения на сам код. Так, в любом классе, методе или сборке может быть вызван другой класс, метод или сборка, а следовательно, такой код относится к клиенту. Один и тот же класс, метод или сборка может быть также вызван из других методов, классов и сборок, а следовательно, такой код относится к службе.

Простой пример зависимости

Рассмотрим, каким образом зависимость действует в реальной ситуации. С этой целью обратимся к примеру очень простого консольного приложения, выводящего на экран традиционное для программирования сообщение `"Hello World!"` (Здравствуй, мир!). Данный пример намеренно сделан простым, чтобы ясно продемонстрировать саму суть трудностей, связанных с зависимостями.

Можете следовать приведенной ниже процедуре или извлечь готовое решение из информационного хранилища GitHub, как поясняется в приложениях А и Б к данной книге.

1. Откройте ИСР Microsoft Visual Studio и создайте новое консольное приложение, как показано на рис. 2.2. Можете назвать его **`SimpleDependency`**, хотя конкретное название не имеет особого значения.

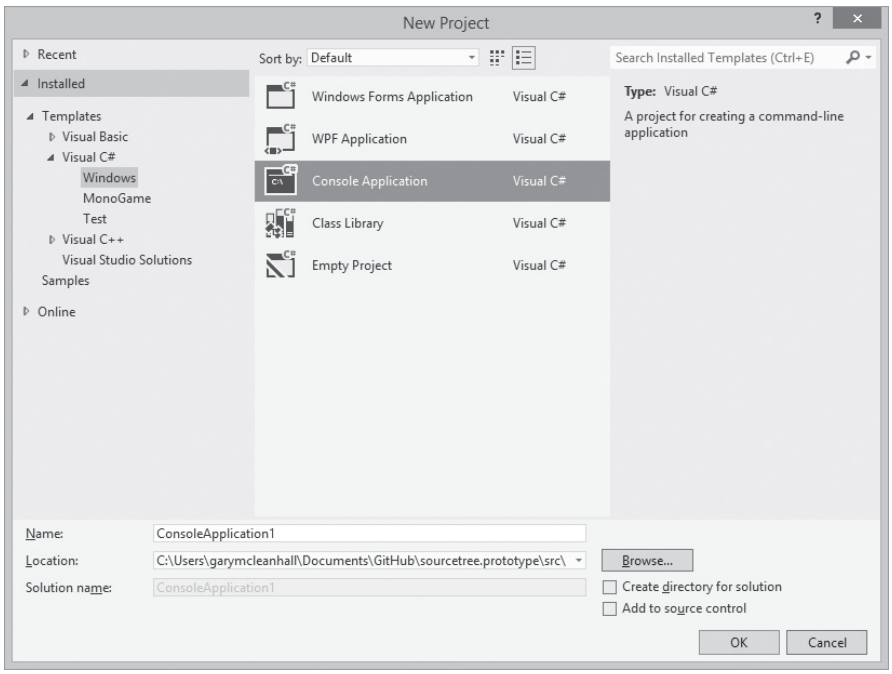

Рис. 2.2. *В диалоговом окне* New Project *(Новый проект) ИСР Visual Studio можно выбрать самые разные шаблоны проектов*

2. Введите в текущее решение второй проект — на этот раз библиотеку классов. Присвойте ему имя **`MessagePrinter`**.
3. Щелкните правой кнопкой мыши на узле *`References`* (Ссылки) текущего приложения и выберите команду Add Reference (Добавить ссылку) из контекстного меню.
4. Перейдите к панели Projects (Проекты) в диалоговом окне Add Reference и выберите проект библиотеки классов.

Итак, вы установили зависимость одной сборки от другой, как показано на рис. 2.3. Созданное вами консольное приложение зависит от библиотеки классов, тогда как сама библиотека классов не зависит от этого консольного приложения. Следовательно, консольное приложение является клиентом, а библиотека классов — службой. И хотя данное приложение пока еще мало что делает, постройте решение и перейдите к каталогу `bin` текущего проекта, где размещается исполняемый файл.

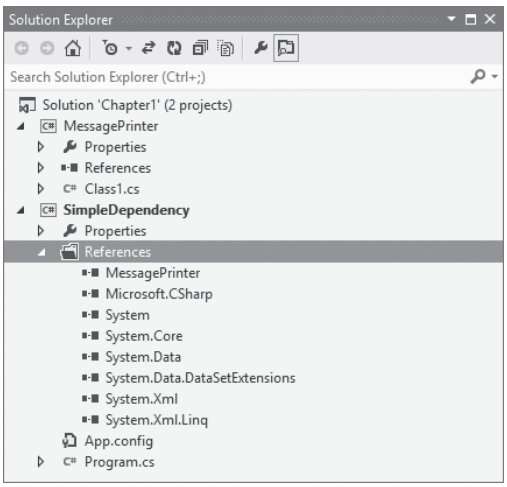

Рис. 2.3. *Сборки, на которые делаются ссылки в любом проекте, перечислены в его узле* `References`

Каталог `bin` содержит не только исполняемый файл `SimpleDependency.exe`, но и библиотечный файл `MessagePrinter.dll`, скопированный в этот каталог в процессе построения в Visual Studio, поскольку в исполняемом файле на него делается ссылка как на зависимость. Чтобы провести небольшой эксперимент, нужно сначала внести незначительные изменения в код рассматриваемого здесь приложения. Это консольное приложение пока еще ничего не делает, и поэтому оно инициализируется и закрывается, прежде чем пользователь сумеет отреагировать на него. Итак, откройте исходный файл `Program.cs` данного консольного приложения.

В листинге 2.1 полужирным выделена строка кода, введенная в тело метода `Main()`. Это точка входа в консольное приложение, которое пока еще ничего не делает и быстро завершается. Вводом вызова метода `Console.ReadKey()` гарантируется, что данное приложение будет ожидать нажатия пользователем клавиши, прежде чем завершить свою работу.

Листинг 2.1. Вызов метода `ReadKey()` препятствует немедленному завершению консольного приложения

```
namespace SimpleDependency
{
  class Program
  {
    static void Main()
    {
      Console.ReadKey();
    }
  }
}
```

Снова постройте решение и запустите данное приложение на выполнение. Как и предполагалось, оно отображает окно консоли, ожидает нажатия пользователем клавиши, а затем завершается. Введите точку прерывания в строке кода `Console.ReadKey()` и отладьте построенное решение в Visual Studio.

Когда выполнение консольного приложения остановится в точке прерывания, просмотрите сборки, загруженные в оперативную память для данного приложения. С этой целью выберите команду меню Debug⇨Windows⇨Modules (Отладка⇨Окна⇨Модули) или нажмите сначала комбинацию клавиш <Ctrl+D>, а затем клавишу <M>. На рис. 2.4 приведен перечень модулей, загруженных в оперативную память для данного приложения.

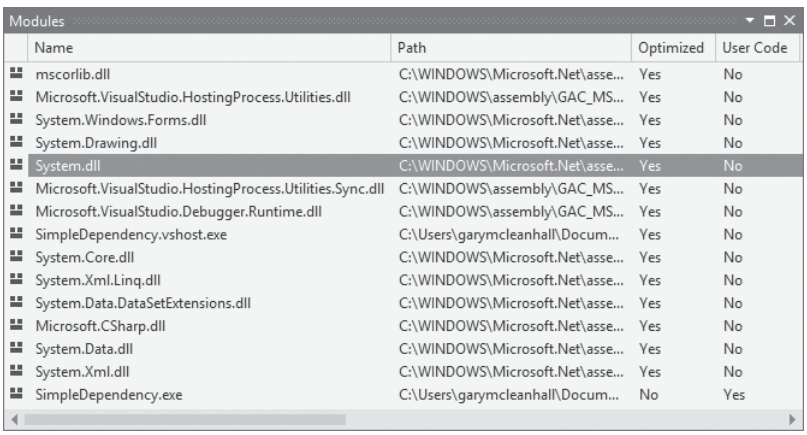

Рис. 2.4. *В режиме отладки в окне* Modules *отображаются все загруженные в настоящий момент сборки*

Обратите внимание на отсутствие всякого упоминания о созданной библиотеке классов. В связи с этим невольно возникает вопрос: должен ли быть загружен библиотечный файл `MessagePrinter.dll` для данного приложения? На самом деле не должен, и это вполне ожидаемое поведение. В данном приложении не используется никаких средств из сборки `MessagePrinter`, и поэтому исполняющая система .NET не загружает ее.

Чтобы окончательно убедиться в том, что зависимая сборка необязательна, снова перейдите к каталогу bin данного консольного приложения и удалите файл MessagePrinter.dll. Еще раз запустите данное приложение на выполнение. Оно будет благополучно выполнено без возникновения соответствующего исключения.

Повторите этот же эксперимент еще пару раз, чтобы окончательно выяснить, что же на самом деле происходит. С этой целью введите сначала директиву using MessagePrinter в самом начале исходного файла Program.cs. По этой директиве импортируется пространство имен MessagePrinter. Достаточно ли этого, чтобы общеязыковая исполняющая среда (Common Language Runtime — CLR) загрузила модуль? Недостаточно. Эта зависимость снова игнорируется и сборка не загружается потому, что оператор using, импортирующий пространство имен, служит лишь для синтаксического удобства с целью сократить объем кода, который требуется написать. Вместо того чтобы создавать целое пространство имен всякий раз, когда требуется воспользоваться типом данных из него, это пространство имен можно импортировать и ссылаться на его типы непосредственно. Оператор using формирует команды для выполнения в среде CLR.

Следующий тест построен на основе предыдущего теста, поэтому можете оставить оператор using на месте. Далее введите вызов конструктора класса MessagePrinting.Service выше вызова метода Console.ReadLine() в исходном файле Program.cs, как показано в листинге 2.2.

Листинг 2.2. Внедрение зависимости вызовом метода экземпляра

```
using System;
using MessagePrinter;

namespace SimpleDependency
{
  class Program
  {
    static void Main()
    {
      var service = new MessagePrintingService();
      service.PrintMessage();
      Console.ReadKey();
    }
  }
}
```

Теперь в окне Modules можно обнаружить, что сборка MessagePrinter.dll загружена. Ведь получить экземпляр класса MessagePrintingService без загрузки этой сборки в оперативную память нельзя. Можете убедиться в этом сами, удалив файл MessagePrinter.dll из каталога bin и снова запустив данное приложение на выполнение. На этот раз будет сгенерировано следующее исключение:

```
Unhandled Exception: System.IO.FileNotFoundException:
Could not load file or assembly 'MessagePrinter, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null' or one of its dependencies.
The system cannot find the file specified.
```

```
(Необработанное исключение: System.IO.FileNotFoundException:
Не удалось загрузить файл или сборку 'MessagePrinter, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null' или одну из ее зависимостей.
Система не может найти указанный файл.)
```

Каркасные зависимости

В предыдущем разделе была продемонстрирована так называемая *основная зависимость*. В этом случае консольное приложение и библиотека классов, от которой оно зависит, принадлежат одному и тому же решению, построенному в Visual Studio. Это означает, что зависимость должна быть постоянно доступной, поскольку проект с этой зависимостью может быть всегда перестроен из исходного кода по мере необходимости. Это также означает, что исходный код основных зависимостей можно видоизменить.

Оба проекта имеют другие зависимости в форме сборок на платформе .NET Framework. Они являются составной частью проекта, но предполагаются доступными. Каждая сборка обновляется до той версии платформы .NET Framework, на которой она была построена: 1, 1.1, 2, 3.5, 4, 4.5 и т.д. Одни сборки на платформе .NET Framework оказываются новыми для конкретной версии, и поэтому на них нельзя ссылаться в тех проектах, где используется более ранняя версия платформы .NET Framework. А другие сборки изменяются от одной версии платформы .NET Framework к другой, и поэтому они пригодны только для конкретной ее версии.

В простом проекте `SimpleDependency` имеется несколько ссылок на платформу .NET Framework (см. рис. 2.3). Многие из этих зависимостей внедряются во все проекты консольных приложений по умолчанию. Но в рассматриваемом здесь примере консольного приложения они не используются, и поэтому их можно благополучно удалить. В действительности все зависимости, кроме `System` и `System.Core`, в обоих проектах излишни, поэтому их можно удалить из списка ссылок. А приложение будет по-прежнему выполняться правильно. Удаляя ненужные каркасные ссылки, можно упростить наглядное представление зависимостей, требующихся в каждом проекте.

Каркасные сборки загружаются всегда

Следует иметь в виду, что в отличие от других зависимостей, ссылки на сборки .NET Framework всегда приводят к загрузке этих сборок. Даже если сборка не используется, она все равно загружается при запуске приложения на выполнение. Правда, если ссылка на одну и ту же сборку делается в нескольких проектах одного решения, то в оперативную память загружается только один экземпляр этой сборки, который становится общим для всех зависимых проектов.

Перечень ссылок по умолчанию

Ссылки по умолчанию разнятся в зависимости от типа проекта. У каждого типа проекта имеется свой шаблон, в котором перечисляются требующиеся ссылки. Именно таким образом в приложении Windows Forms можно ссылаться на сборку `System`.

Windows.Forms, тогда как в приложении Windows Presentation Foundation — на сборки WindowsBase, PresentationCore и PresentationFramework.

В листинге 2.3 демонстрируются ссылки для консольного приложения. Все шаблоны проектов в Visual Studio располагаются в корневом каталоге /Common7/IDE/ProjectTemplates/ установки Visual Studio и командируются по языкам программирования.

Листинг 2.3. Часть шаблона проекта в Visual Studio для ссылок на разные сборки по условию

```
<ItemGroup>
  <Reference Include="System"/>
  $if$ ($targetframeworkversion$ >= 3.5)
  <Reference Include="System.Core"/>
  <Reference Include="System.Xml.Linq"/>
  <Reference Include="System.Data.DataSetExtensions"/>
  $endif$
  $if$ ($targetframeworkversion$ >= 4.0)
  <Reference Include="Microsoft.CSharp"/>
  $endif$
  <Reference Include="System.Data"/>
  <Reference Include="System.Xml"/>
</ItemGroup>
```

В перечислении ссылок на файлы в листинге 2.3 имеется определенная логика, поскольку она может изменить порядок формирования настоящего экземпляра проекта. В частности, ссылки меняются в зависимости от версии платформы .NET Framework, применяемой для результирующего проекта. В данном случае ссылка на сборку Microsoft.CSharp делается только в том случае, если проект предназначается для версии 4, 4.5 или 4.5.1 платформы .NET Framework. И в этом есть свой смысл, поскольку данная сборка требуется только в том случае, если применяется ключевое слово dynamic, внедренное в версии .NET Framework 4.

Сторонние зависимости

Эта последняя разновидность зависимостей устанавливается в том случае, если применяются сборки, созданные сторонними разработчиками. Как правило, решение можно реализовать самостоятельно, установив основные зависимости, если какие-нибудь средства не предоставляются на платформе .NET Framework. Но в зависимости от масштабов требующегося решения для решения подобной задачи может потребоваться немало труда. Вместо этого можно воспользоваться уже готовыми решениями. Например, вряд ли стоит реализовывать собственный объектно-реляционный преобразователь (ORM — Object/Relational Mapper), поскольку для написания такого крупного фрагмента инфраструктурного кода может потребоваться не один месяц, а для доведения его до состояния полной готовности — не один год. Вместо этого лучше обратиться сначала к каркасу Entity Framework, входящему в состав платформы .NET Framework. Если этот каркас не удовлетворяет насущным потребностям, то вместо него можно обратиться к готовой библиотеке NHibernate объектно-реляционного преобразования, которая является зрелым и тщательно проверенным программным продуктом.

Главной причиной применения сторонней зависимости служит экономия затрат труда: вместо реализации некоторых функциональных средств или целой инфраструктуры достаточно внедрить готовые средства, уже созданные для решения поставленной задачи. Не следует, однако, забывать, что затраты труда на подобное внедрение могут оказаться значительными в зависимости от структуры стороннего кода и особенностей сопряжения с ним. Если преследуется цель постепенно повышать коммерческую ценность программного продукта с каждым его выпуском, как это зачастую делается по методике Scrum, то применение сторонних библиотек позволяет сосредоточить основные усилия на достижении данной цели.

Организация сторонних зависимостей

Простейший способ организовать зависимости, являющиеся внешними по отношению к проекту и платформе .NET Framework, — создать для решения данного проекта в Visual Studio папку `Dependencies` и ввести в нее файлы с расширением **`.dll`**. Когда же ссылки на сборки в этих файлах потребуется ввести в проекты текущего решения, достаточно открыть диалоговое окно Reference Manager (Диспетчер ссылок), приведенное на рис. 2.5, и найти в нем нужные файлы.

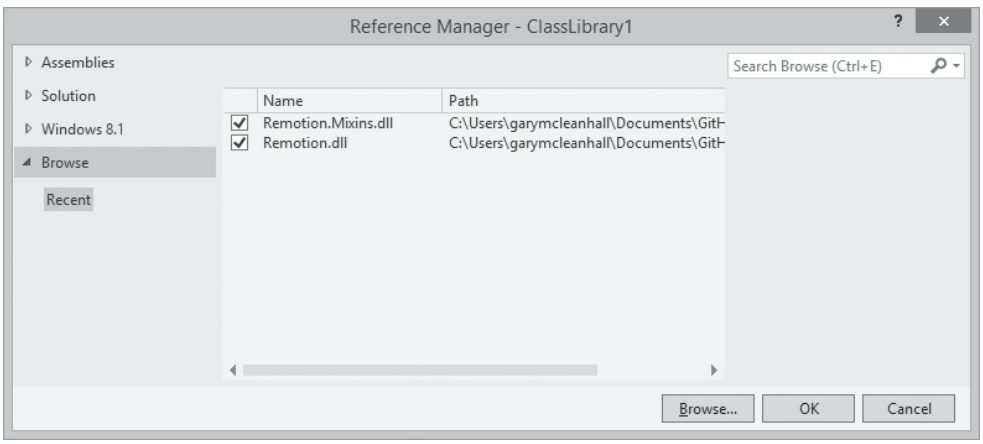

Рис. 2.5. *Сторонние ссылки могут храниться в папке `Dependencies` текущего решения в Visual Studio*

Еще одно преимущество такого подхода заключается в том, что все внешние зависимости хранятся в системе контроля версий исходного кода. Это дает другим разработчикам возможность получать зависимости, извлекая последнюю версию исходного кода из центрального информационного хранилища. В этом случае задача всех разработчиков намного упрощается, не требуя от них устанавливать или загружать нужные файлы самостоятельно.

Еще более совершенный способ организации сторонних зависимостей демонстрируется далее, в разделе “Управление зависимостями средствами NuGet”. А до тех пор вкратце поясним, что инструментальное средство NuGet управляет зависимостями в проекте автоматически, включая загрузку пакета, содержащего все соответствующие артефакты, обращение к сборкам и обновление версий библиотек.

Моделирование зависимостей в направленном графе

Граф представляет собой математическую конструкцию, состоящую из двух разных элементов: узлов и ребер. Ребро может находиться между двумя узлами и каким-то образом связывать их вместе. Любой узел может быть связан с каким угодно количеством других узлов в графе. Графы могут быть разнотипными в зависимости от состава их свойств. Например, граф, приведенный на рис. 2.6, имеет ненаправленные ребра. В частности, ребро между узлами A и C не направленно ни от узла A к узлу C, ни от узла C к узлу A, а только присутствует в графе. Такой граф называется *ненаправленным*.

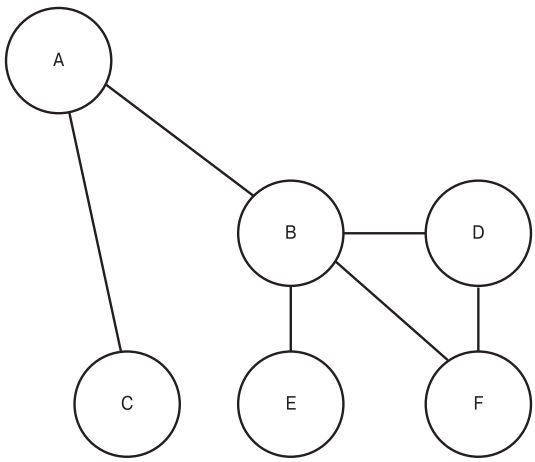

Рис. 2.6. *Граф состоит из узлов, соединяемых ребрами*

Если же ребра обозначены с одного конца стрелками, то по ним можно определить направление ребер. Так, на рис. 2.7 ребро направлено от узла A к узлу C, но не наоборот. Такой граф называется направленным, или *диграфом*.

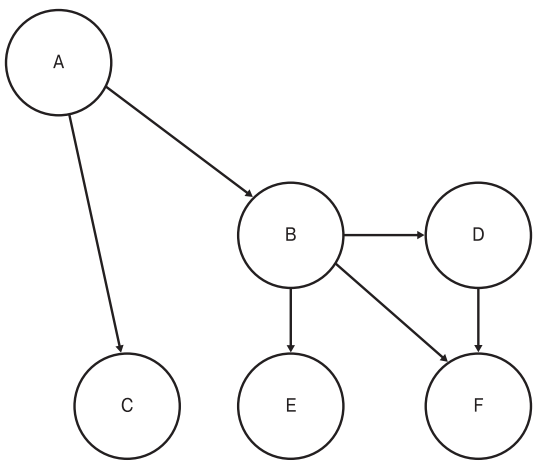

Рис. 2.7. *Ребра этого графа имеют конкретную направленность, и поэтому в нем присутствует ребро (A,B), но не ребро (B,A)*

Во многих областях разработки программного обеспечения графы применяются для построения моделей, но они особенно удобны для моделирования зависимостей в прикладном коде. Как упоминалось ранее, зависимости состоят из двух сущностей, связанных вместе в направлении от зависимого кода к зависимости. Такие сущности могут рассматриваться как узлы, связанные вместе направленными ребрами от зависимого узла к узлу зависимости. Если распространить эту взаимосвязь на остальные сущности, то в конечном итоге получится *граф зависимостей*.

Такая структура может применяться с разной степенью детализации, как показано на рис. 2.8. Узлы графа могут представлять отдельные классы в проекте, разные сборки или их команды, образующие подсистему. Во всех классах стрелками на ребрах между узлами графа обозначаются зависимости между отдельными компонентами, причем стрелки указывают направление от зависимого компонента к его зависимости.

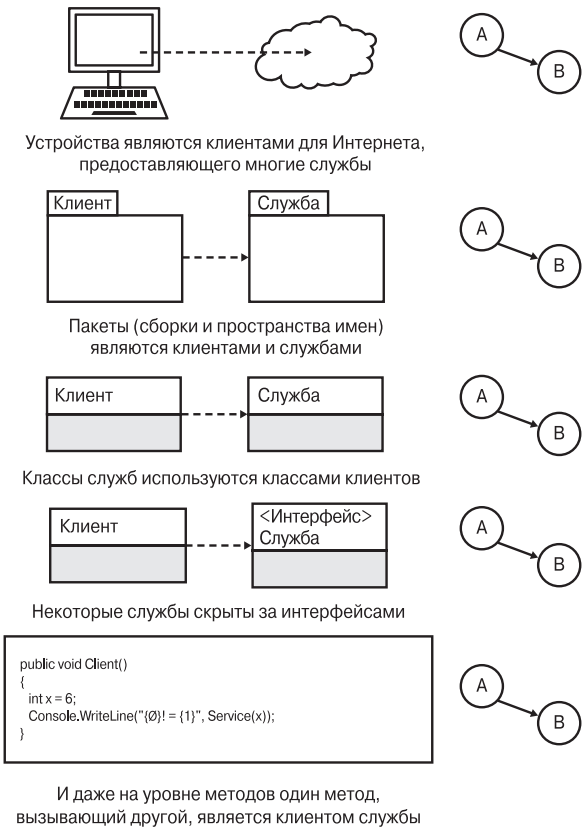

Рис. 2.8. *Зависимости на всех уровнях могут быть смоделированы в виде графов*

Для каждого узла на уровне грубой детализации имеется ряд узлов на уровне более точной детализации. Подсистемы состоят из сборок, сборки — из классов, а классы — из методов. Эти примеры наглядно показывают, как зависимость от одного метода может привести к целой подсистеме цепочечных зависимостей.

Тем не менее все эти примеры позволяют выяснить не *вид* отдельной зависимости (наследование, агрегирование, композицию или ассоциацию), а только наличие зависимости. Но и этого оказывается достаточно, поскольку для управления зависимостями требуется лишь знать двоичное отношение между двумя сущностями, т.е. *имеется* ли зависимость между ними или же она *отсутствует*.

Циклические зависимости

Как известно из теории графов, направленные графы могут образовывать циклы, позволяющие делать обход графа из одного узла и обратно возвращаться к нему, следуя по ребрам. Представленные до сих пор графы называются *ациклическими орграфами*, поскольку они не содержат циклы. А пример *циклического орграфа* приведен на рис. 2.9. Если начать обход этого графа с узла D, то можно последовать по одному ребру к узлу E, затем по другому ребру к узлу B и, наконец, вернуться по третьему ребру обратно к узлу D.

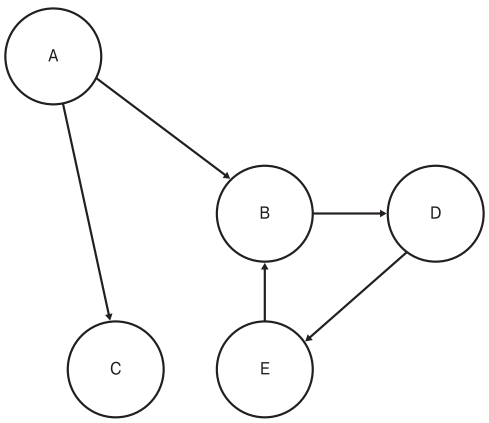

Рис. 2.9. *Этот граф содержит циклы*

Допустим, что узлы данного графа представляют собой сборки. Сборка D имеет косвенную зависимость от всего, что прямо или косвенно зависит от нее. Так, сборка D прямо зависит от сборки E, но косвенно от сборки B. Следовательно, сборка D зависит от *себя*.

На самом деле для сборок циклические зависимости недопустимы. Если попытаться установить их в Visual Studio, назначив ссылку из сборки E на сборку B, ИСР Visual Studio не позволит этого сделать, как показано на рис. 2.10.

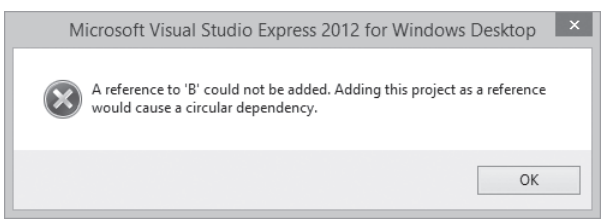

Рис. 2.10. *Установить циклическую зависимость в Visual Studio нельзя*

Таким образом, несмотря на академический характер моделирования зависимостей в виде графов, оно все же приносит очевидные выгоды при организации зависимостей. Циклические зависимости между сборками не являются каким-то отклонением от чистого идеала, а полностью запрещены, и поэтому их следует всячески избегать.

Петли являются частным случаем циклов в направленных графах. Если узел соединен ребром с *самим собой*, то такое ребро превращается в *петлю*. Пример направленного графа с петлей приведен на рис. 2.11.

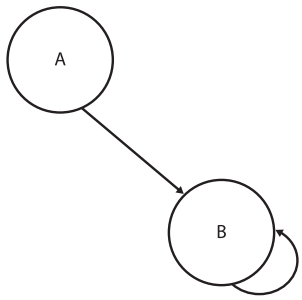

Рис. 2.11. *В этом графе узел B соединен петлей с самим собой*

На практике сборки всегда делаются самозависимыми в явном виде, хотя это наблюдение и не заслуживает особого внимания. Тем не менее на уровне методов петля явно свидетельствует о *рекурсии*, как демонстрируется в листинге 2.4.

Листинг 2.4. Петля самозависимости обозначает в направленном графе рекурсию на уровне методов

```
namespace Graphs
{
  public class RecursionLoop
  {
    public void A()
    {
      int x = 6;
      Console.WriteLine("{0}! = {1}", x, B(x));
    }

    public int B(int number)
    {
      if(number == 0)
      {
        return 1;
      }
      else
      {
        return number * B(number  1);
      }
    }
  }
}
```

Класс из листинга 2.4 служит функциональным эквивалентом графа зависимостей на рис. 2.11. Метод A вызывает метод B, а следовательно, можно утверждать, что метод A зависит от метода B. Но больший интерес представляет зависимость метода B от самого себя, поскольку это характерный пример рекурсивной функции, которая вызывает самое себя.

Управление зависимостями

Как пояснялось ранее, зависимости нужны, но требуют тщательного управления, чтобы не стать причиной серьезных осложнений на последующих стадиях разработки. Устранить эти осложнения после того, как они проявятся, будет совсем не просто. Поэтому лучше организовать правильное управление зависимостями с самого начала и не терять бдительность, чтобы возникли непредвиденные осложнения. Ведь неудачно организованное управление зависимостями может быстро перерасти из мелкого компромисса в проблему общеархитектурного характера.

В остальной части этой главы основное внимание уделяется более практическим вопросам непрерывного управления зависимостями. К их числу относится исключение антишаблонов, а еще важнее — разъяснение причин, по которым эти распространенные шаблоны становятся антишаблонами. С другой стороны, некоторые шаблоны желательны и должны применяться как непосредственные альтернативы упомянутым антишаблонам.

Шаблоны и антишаблоны

Разработка объектно-ориентированного программного обеспечения является относительно новой технической дисциплиной. За последние десятилетия некоторые повторяющиеся взаимодействия классов и интерфейсов были выявлены и запрограммированы в виде *шаблонов*.

Имеется немало шаблонов для разработки программного обеспечения, причем каждый из них предоставляет общее решение, которое может быть приспособлено для целей конкретной предметной области. Одни шаблоны могут быть использованы совместно для выработки изящных решений сложных задач. Безусловно, не все шаблоны применяются постоянно, и поэтому для выбора подходящих шаблонов требуется определенный опыт и практика.

А другие шаблоны не столько полезны, сколько вообще нежелательны. Поэтому их называют *антишаблонами*. Такие шаблоны наносят вред приспосабливаемости прикладного кода, а следовательно, их следует всячески избегать. Некоторые антишаблоны сначала ведут себя как шаблоны, а затем теряют свои качества по мере восприятия отрицательных побочных эффектов.

Реализации в сравнении с интерфейсами

Те разработчики, которым малознакомы принципы программирования по интерфейсам, нередко испытывают трудности, когда им требуется абстрагироваться от того, что стоит за интерфейсами. Во время компиляции любой клиент интерфейса не должен вообще ничего знать о том, какая именно реализация интерфейса применяется. Ведь такое знание может привести к неверному допущению, привязывающему клиента к конкретной реализации интерфейса.

Обратимся к типичному примеру класса, в котором запись требуется сохранить в постоянном хранилище. Для этого класс делегирует свои функции непосредственно интерфейсу, скрывающему подробности реализации механизма постоянного хранилища. Но было бы неверно делать какие-нибудь предположения о конкретной реализации интерфейса, применяемой во время выполнения. Например, попытка привести ссылку на интерфейс к любой реализации *вообще не* приветствуется.

Оператор `new` как признак недоброкачественного кода

Интерфейсы описывают, *что* нужно сделать, тогда как классы — *как* это сделать. И только классы включают в себя подробности реализации, а интерфейсам вообще ничего неизвестно об этом. Но поскольку конструкторы имеются только у классов, то подробности реализации скрываются именно в конструкторах. Из этого вытекает весьма любопытное следствие: появление ключевого слова `new` в прикладном коде может считаться, за некоторыми исключениями, признаком его *недоброкачественности*.

Код с душком

Когда говорят, что код с душком, это означает, что он потенциально недоброкачественный. Слово *потенциально* выбрано намеренно, поскольку два экземпляра кода с душком могут оказаться недоброкачественными в разной степени. В отличие от антишаблонов, которые в более общем смысле считаются вредными, недоброкачественный код совсем не обязательно может быть вредным. Код с душком лишь свидетельствует о каких-то неверных решениях, причины которых нужно искоренить, или же о техническом долге, который следует возместить безотлагательно.

Имеются самые разные категории кода с душком. Примером “неуместной близости” может служить применение ключевого слова `new` для непосредственно создания объекта. А поскольку конструкторы являются реализациями подробностей, то их применение может вызвать затребование в клиентском коде непреднамеренных (и нежелательных) зависимостей.

Код с душком, как и антишаблоны, устраняется путем реорганизации, чтобы сделать структуру кода более совершенной и адаптируемой. И хотя код может удовлетворять своим требованиям, его текущая структура все равно остается неоптимальной и впоследствии может вызвать трудности. Это, без сомнения, задача разработки, которая вряд ли принесет немедленно ощутимую коммерческую выгоду. Как и реорганизация кода, устранение недостатков, по-видимому, не повышает

коммерческую ценность программного продукта. Тем не менее технический долг может выйти из-под контроля и нарушить управление зависимостями, поставив под угрозу последующие усовершенствования и исправления в коде, подобно тому, как финансовый долг способен привести к постепенному росту сумм погашения процентов.

В листинге 2.5 демонстрируются два примера, где непосредственное получение экземпляра объекта с помощью ключевого слова `new` свидетельствует о недоброкачественности кода.

Листинг 2.5. Примеры того, как получение экземпляров объектов препятствует адаптивности кода

```
public class AccountController
{
    private readonly SecurityService securityService;

    public AccountController()
    {
        this.securityService = new SecurityService();
    }

    [HttpPost]
    public void ChangePassword(Guid userID, string newPassword)
    {
        var userRepository = new UserRepository();
        var user = userRepository.GetByID(userID);
        this.securityService.ChangeUsersPassword(user, newPassword);
    }
}
```

Класс `AccountController` является частью гипотетического приложения на платформе ASP.NET MVC. Не особенно вдаваясь в подробности его построения, рассмотрим выделенную полужирным строку кода, в которой объект создается неподходящим образом. Контроллер отвечает за предоставление пользователю возможности делать запросы учетной записи и выполнять соответствующие команды (в данном случае команды `ChangePassword`).

Рассматриваемому здесь коду присущ ряд недостатков, непосредственно связанных с двумя случаями применения оператора `new`. Эти недостатки перечислены ниже, а пути их устранения рассматриваются в последующих подразделах.

- Класс `AccountController` навсегда зависит от реализаций классов `SecurityService` и `UserRepository`.
- Любые зависимости, имеющиеся у классов `SecurityService` и `UserRepository`, теперь оказываются косвенными зависимостями от класса `AccountController`.
- Модульное тестирование класса `AccountController` сильно затруднено, поскольку оба других класса невозможно сымитировать обычными методами.

- Метод `SecurityService.ChangeUsersPassword()` требует от клиентов загрузки объектов типа `User`.

Неспособность усовершенствовать реализации

Если требуется изменить реализации класса `SecurityService`, то с этой целью следует внести изменения непосредственно в класс `AccountController`, чтобы ссылаться на новую его реализацию, или же ввести новые функциональные средства в существующий класс `SecurityService`. На протяжении всей данной книги поясняются причины, по которым ни одно из этих двух решений не годится. А до тех пор лишь подчеркнем, что главная цель — вообще не править классы `AccountController` или `SecurityService` после их создания.

Цепочка зависимостей

У класса `SecurityService`, вероятнее всего, имеются и свои зависимости. В то же время наличие конструктора по умолчанию явно означает, что у него нет никаких зависимостей. Но что, если код, приведенный в листинге 2.6, является реализацией конструктора класса `SecurityService`?

Листинг 2.6. Классу `SecurityService` присущ тот же недостаток, что и классу `AccountController`

```
public SecurityService()
{
    this.Session = SessionFactory.GetSession();
}
```

Данная служба фактически зависит от библиотеки NHibernate объектно-реляционного преобразования, применяемой для извлечения *сеанса*, который служит в NHibernate аналогией соединения с постоянным хранилищем вроде реляционной базы данных Microsoft SQL Server, Oracle или MySQL. Как было показано ранее, это означает, что класс `AccountController` также зависит, хотя и косвенно, от библиотеки NHibernate.

А что, если сигнатура конструктора класса `SecurityService` изменится, т.е. ему неожиданно потребуются клиенты для предоставления строки, связывающей с базой данных и требующейся для класса `Session`? Любой клиент, пользующийся классом `SecurityService`, в том числе и класс `AccountController`, должен быть обновлен, чтобы предоставить связывающую строку. И это изменение придется внести.

Недостаток тестируемости

Тестируемость — очень важное свойство прикладного кода. Она требует, чтобы прикладной код был разработан определенным образом. В противном случае его тестирование крайне затруднено. К сожалению, протестировать оба класса, `AccountController` и `SecurityService`, совсем не просто, поскольку нельзя заменить их зависимости поддельными их вариантами, не выполняющими никаких действий. Например, для тестирования класса `SecurityService` вряд ли требуется установление каких-нибудь соединений с базой данных. Ведь это бесполезно и медленно, не говоря уже о внесении крупной точки отказа в тест в том случае, если база

данных окажется недоступной. Эти классы можно, конечно, протестировать, заменив их зависимости на поддельные во время выполнения. Такие инструментальные средства, как Microsoft Moles и Typemock, способны обращаться к конструкторам и возвращать поддельные объекты. Но такой подход служит примером выявления последствий, а не причин.

Больше неуместной близости

В методе `AccountController.ChangePassword()` создается объект класса `UserRepository` с целью извлечь экземпляр класса `User`. И это делается только потому, что того требует метод `SecurityService.ChangeUsersPassword()`. Ведь без экземпляра класса `User` данный метод нельзя вызвать, что служит явным признаком неудачно разработанного метода интерфейса. Вместо того чтобы требовать от всех клиентов извлекать экземпляр класса `User`, класс `SecurityService` должен делать это сам. И в таком случае оба упомянутых метода будут выглядеть так, как показано в листинге 2.7.

Листинг 2.7. Усовершенствования, сделанные в классе `SecurityService` для всех его клиентов

```
[HttpPost]
public void ChangePassword(Guid userID, string newPassword)
{
    this.securityService.ChangeUsersPassword(userID, newPassword);
}
//...
public void ChangeUsersPassword(Guid userID, string newPassword)
{
    var userRepository = new UserRepository();
    var user = userRepository.GetByID(userID);
    user.ChangePassword(newPassword);
}
```

Это определенно усовершенствование для класса `AccountController`. Тем не менее в методе `ChangeUsersPassword()` по-прежнему получается экземпляр класса `UserRepository`.

Альтернативы построению объектов

Как же усовершенствовать классы `AccountController` и `SecurityService` или любой другой пример неподходящего построения объектов? Как сделать их очевидно правильными, чтобы устранить упомянутые выше недостатки? Для этого имеется несколько взаимодополняющих способов, из которых можно выбрать наиболее подходящий.

Перенос кода в интерфейс

Первое и самое важное изменение, которое следует сделать, состоит в том, чтобы скрыть реализацию класса `SecurityService` за интерфейсом. Благодаря этому класс `AccountController` становится зависимым только от интерфейса, а не от его реа-

лизации в классе SecurityService. В качестве первой реорганизации кода следует извлечь интерфейс из класса SecurityService, как показано в листинге 2.8.

Листинг 2.8. Извлечение интерфейса из класса SecurityService

```
public interface ISecurityService
{
    void ChangeUsersPassword(Guid userID, string newPassword);
}
//...
public class SecurityService : ISecurityService
{
    public ChangeUsersPassword(Guid userID, string newPassword)
    {
        //...
    }
}
```

Далее следует обновить клиент таким образом, чтобы он больше не зависел от класса SecurityService, но только от интерфейса ISecurityService. Эта реорганизация кода в классе AccountController приведена в листинге 2.9.

Листинг 2.9. Теперь в классе AccountController применяется интерфейс ISecurityService.

```
public class AccountController
{
    private readonly ISecurityService securityService;

    public AccountController()
    {
        this.securityService = new SecurityService();
    }

    [HttpPost]
    public void ChangePassword(Guid userID, string newPassword)
    {
        securityService.ChangeUsersPassword(user, newPassword);
    }
}
```

Этот пример еще не завершен, поскольку остается зависимость от реализации класса SecurityService через его конструктор. Экземпляр конкретного класса попрежнему получается в конструкторе классе AccountController. Чтобы полностью разделить оба класса, нужно дополнительно реорганизовать код путем *внедрения зависимостей*.

Внедрение зависимостей

Это довольно обширная тема, для полного раскрытия которой здесь недостаточно места. Ей посвящена отдельная глава 9, и на эту тему написаны целые книги. Правда, внедрять зависимости не так уж и сложно, как будет показано здесь на примере упомянутых выше классов. В листинге 2.10 демонстрируется еще одна реорганизация кода в конструкторе класса AccountController. Изменения, внесенные в этот

конструктор, выделены полужирным. Они весьма незначительны для данного класса, но совершенно меняют возможность управлять зависимостями. Вместо построения объекта самого класса `SecurityService` в классе `AccountController` теперь требуется другой класс, предоставляющий реализацию интерфейса `ISecurityService`. Помимо этого, в конструктор внедрено предусловие, препятствующее клиентам данного класса передавать его конструктору пустое значение `null` в качестве параметра `securityService`. Благодаря этому при использовании поля `securityService` в методе `ChangePassword()` гарантируется получение достоверного экземпляра и нигде больше не требуется проверка на пустое значение `null`.

Листинг 2.10. Внедрение зависимостей позволяет устранить зависимость от класса `SecurityService`

```
public class AccountController
{
    private readonly ISecurityService securityService;

    public AccountController(ISecurityService securityService)
    {
        if(securityService == null)
                  throw new ArgumentNullException("securityService");
        this.securityService = securityService;
    }

    [HttpPost]
    public void ChangePassword(Guid userID, string newPassword)
    {
       this.securityService.ChangeUsersPassword(user, newPassword);
    }
}
```

Следуя этому же образцу, внедрить зависимости нужно и в классе `SecurityService`. В листинге 2.11 показано, каким образом этот класс выглядит после реорганизации кода.

Листинге 2.11. Внедрение зависимостей является универсальным шаблоном, который можно применять повсеместно

```
public class SecurityService : ISecurityService
{
    private readonly IUserRepository userRepository;

    public SecurityService(IUserRepository userRepository)
    {
        if(userRepository == null) throw
               new ArgumentNullException("userRepository");
        this.userRepository = userRepository;
    }

    public ChangeUsersPassword()
    {
        var user = userRepository.GetByID(userID);
```

```
            user.ChangePassword(newPassword);
        }
}
```

Подобно тому, как класс `AccountController` осуществляет свою зависимость от достоверного экземпляра интерфейса `ISecurityService`, класс `SecurityService` осуществляет свою зависимость от достоверного экземпляра интерфейса `IUserRepository`, генерируя исключение, если при построении объекта его конструктор получает пустую (`null`) ссылку. Аналогичным образом полностью устраняется зависимость от класса `UserRepository` в пользу интерфейса `IUserRepository`.

Антишаблон "Антураж"

Антишаблон "Антураж" (Entourage) получил свое название благодаря тому, что он привносит с собой много лишнего, когда от него требуется сделать что-нибудь простое. Это можно сравнить со свитой прихлебателей и бездельников, сопровождающих звезды музыки и кино и составляющих их окружение. Название этого антишаблона как нельзя лучше описывает нежелательное управление зависимостями.

Антишаблон "Антураж" служит признаком типичной ошибки, которую совершают разработчики, поясняя программирование интерфейса. Вместо того чтобы предоставить полноценное решение, они зачастую ограничиваются кратким и недвусмысленным заявлением, что интерфейсы и их зависимости *не должны быть в одной и той же сборке*.

Блок-схема, составленная на унифицированном языке моделирования (Unified Modeling Language — UML) и представленная на рис. 2.12, наглядно показывает, каким образом рассматриваемый здесь пример класса `AccountController` организован на уровне пакета. Класс `AccountController` зависит от интерфейса `ISecurityService`, который реализуется в классе `SecurityService`. На этой блок-схеме показаны также пакеты, где располагается каждый компонент (на платформе .NET Framework пакетами являются сборки или проекты, разрабатываемые в Visual Studio). Это характерный пример антишаблона "Антураж", т.е. реализации интерфейса в той же самой сборке, что и сам интерфейс.

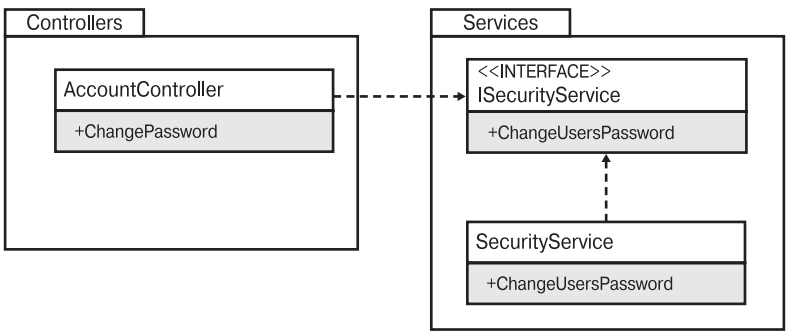

Рис. 2.12. *Сборка `AccountController` зависит от сборки `Services`*

Как пояснялось ранее, у класса `SecurityService` имеются некоторые зависимости от самого себя, а цепочка зависимостей приводит к косвенным зависимостям одних

клиентов от других. На блок-схеме, развернутой на уровне пакетов и приведенной на рис. 2.13, в полной мере показан проблемный характер антишаблона “Антураж”.

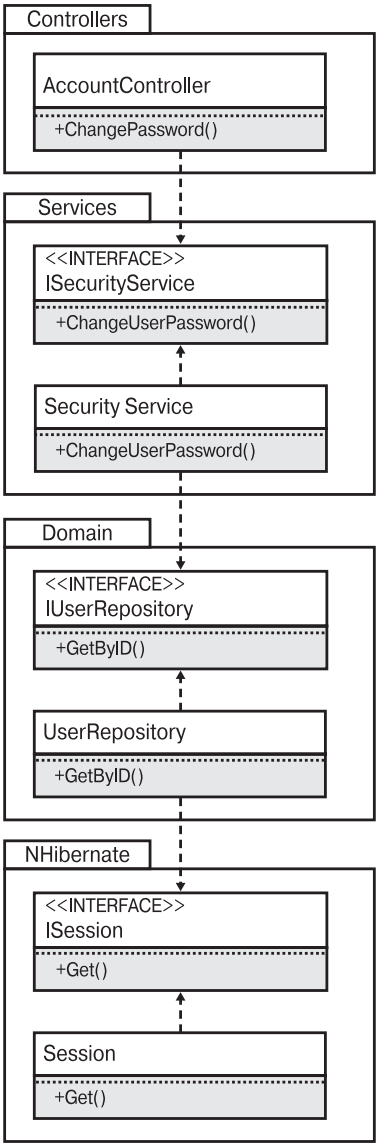

Рис. 2.13. *Класс* `AccountController` *по-прежнему зависит от слишком многих реализаций*

Если построить проект `Controllers` отдельно, то в каталоге `bin/` по-прежнему окажется библиотека NHibernate, что свидетельствует о его косвенной зависимости от данной библиотеки. Несмотря на то что ранее были предприняты эффективные меры, чтобы отделить класс `AccountController` от любых ненужных зависимостей, его связь с реализациями в каждой зависимой *сборке* все равно нельзя считать слабой.

Этот антишаблон приводит к двум затруднениям. Первое связано с дисциплиной программирования. В каждом пакете (`Services`, `Domain` и `NHibernate`) интерфейс требуется объявить открытым (`public`). Но то же самое требуется сделать и с конкретными классами, реализующими эти интерфейсы, чтобы они были доступны для построения объектов в любом месте, а не только в классах клиентов. Это означает, что недисциплинированному разработчику ничто не мешает сделать прямую ссылку на реализацию. При этом возникает сильное искушение пойти по кратчайшему пути и просто вызвать конструктор класса с помощью оператора `new`, чтобы получить его экземпляр.

Второе затруднение возникает в том случае, если попытаться создать новую реализацию класса `SecurityService`, в которой вместо зависимости от модели предметной области, сохраняемой средствами NHibernate, используется сторонняя шина служб (например, NServiceBus) для отправки командного сообщения обработчику. Если ввести такую реализацию в сборку `Services`, появится еще одна зависимость, чрезмерно раздувающая и делающая хрупкой кодовую базу, которую будет очень трудно приспособить к новым требованиям.

Как правило, реализации должны быть отделены от их интерфейсов. С этой целью они размещаются в отдельных сборках. И сделать это можно с помощью шаблона "Лестница".

Шаблон "Лестница"

Шаблон "Лестница" (Stairway) обеспечивает правильную организацию классов и интерфейсов. Размещая интерфейсы и их реализации в разных сборках, можно манипулировать ими по отдельности. А клиентам потребуется только одна ссылка на сборку интерфейса.

В связи с этим невольно возникает вопрос: "Сколько сборок придется отслеживать?" Ведь если разместить каждый интерфейс и класс в отдельной сборке, то в конечном итоге может получиться решение, состоящее из 200 проектов! Такие опасения напрасны, поскольку применение шаблона "Лестница" лишь ненамного увеличивает количество проектов, но в то же время дает возможность получить правильно организованное и простое для понимания решение. Вполне возможно, что после применения шаблона "Лестница" общее количество проектов уменьшится, если до этого они были неудачно организованы.

На рис. 2.14 приведен результат реорганизации кода из рассматриваемого здесь примера класса `AccountController` по шаблону "Лестница". Теперь каждая реализация, т.е. каждый класс, ссылается только на сборку с интерфейсом, от которого она зависит. Она не ссылается на сборку реализации — даже неявно. Кроме того, класс каждой реализации ссылается на сборку своего интерфейса. Это дает возможность получить выгодное во всех отношениях решение: интерфейсы без всяких зависимостей, клиентский код без всяких косвенных зависимостей, а также реализации, которые, следуя данному шаблону, зависят только от других сборок интерфейсов.

А теперь обсудим следующее преимущество, которое дает применение шаблона "Лестница": интерфейсы не должны иметь никаких внешних зависимостей. Этого правила следует придерживаться при всякой возможности. В интерфейсах не должно

быть методов или свойств, делающих доступными любые объекты данных или классы, определенные в ссылках на сторонние библиотеки. И хотя они могут и должны зависеть от классов и типов данных из других проектов в отдельном решении и общих для платформы .NET Framework библиотеках, ссылок на элементы инфраструктуры все же следует избегать. Сторонние библиотеки обычно применяются в инфраструктурных целях. Даже интерфейсы из таких библиотек, как Log4Net, NHibernate и MongoDB, имеют инфраструктурные зависимости, привязывающие интерфейсы разработчика к конкретной реализации. Дело в том, что эти библиотеки упакованы по антишаблону "Антураж", а не по шаблону "Лестница". Каждая из них предоставляет отдельную сборку, содержащую интерфейс, зависимость от которого *желательна*, а также реализацию, зависимость от которой *нежелательна*.

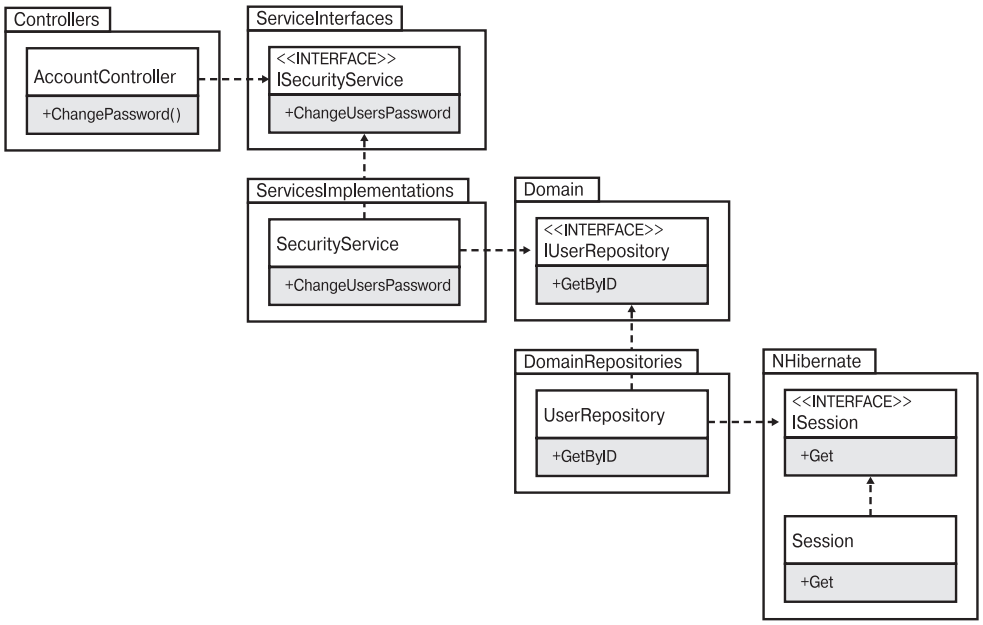

Рис. 2.14. *Шаблон "Лестница" вполне оправдывает свое название*

Чтобы обойти данное препятствие, можно ссылаться на свои интерфейсы для целей регистрации, сохраняемости данных в предметной области и хранения документов. С этой целью можно написать простой интерфейс, скрывающий стороннюю зависимость за основной зависимостью. А если в дальнейшем потребуется заменить стороннюю зависимость, то это можно будет сделать, написав новый адаптер к своему интерфейсу для новой библиотеки.

С прагматичной точки зрения это может показаться вообще неосуществимым. В тех случаях, когда для преобразования сторонней зависимости в основную требуются слишком большие затраты труда, команда разработчиков должна признать, что им придется сохранить ее, и тогда она станет повсеместной. Если же библиотека чрезмерно разрастется, то для ее замены потребуется немало труда и времени. В подобных случаях предпочтение обычно отдается *каркасам*, которые намного крупнее простых библиотек.

Разрешение зависимостей

Когда дело доходит до отладки зависимостей между сборками, то умение правильно организовать проекты и имеющиеся в них зависимости не всегда помогает. Ведь иногда сборки просто недоступны во время выполнения, и тогда возникает потребность выяснить причины их недоступности.

Сборки

Общеязыковая исполняющая среда (CLR), которая, по существу, служит виртуальной машиной для выполнения команд прикладной программы на платформе .NET Framework, является таким же программным продуктом, как и все остальные. Она запрограммирована на предсказуемое и логичное поведение при размещении в ней приложений. В связи с этим очень полезно иметь твердые знания и достаточный практический опыт разрешения сборок и устранения возникающих при этом ошибок. А недостаток знаний и опыта не позволяет быстро выяснить причины, по которым не удается обнаружить сборки.

Процесс разрешения

Процесс разрешения сборок является важным свойством среды CLR. Он восполняет пробел между вводом ссылки на сборку или проект и выполнением приложения с этой загруженной сборкой. Этот процесс состоит из нескольких стадий, и если что-нибудь пойдет не так во время данного процесса, то для выяснения причин потребуется нечто большее, чем приведенный ниже краткий его обзор.

На рис. 2.15 процесс разрешения сборок показан в виде блок-схемы. Эта блок-схема составлена на высоком уровне и не включает в себя многие подробности. Тем не менее она наглядно показывает основные стадии данного процесса, которые вкратце описаны ниже.

- В среде CLR применяется динамическая (JIT) модель для разрешения сборок. Как пояснялось ранее в этой главе, ссылки, содержащиеся в приложении, разрешаются не при запуске приложения на выполнение, а при первом использовании функционального средства из данной сборки, т.е. динамически, или в тот момент, когда оно требуется.
- Каждая сборка имеет свою идентичность, состоящую из ее имени, версии, культурной среды и маркера открытого ключа. Такие средства, как переадресация привязок, могут изменить эту идентичность, и поэтому определить ее не так-то просто, как может показаться на первый взгляд.
- Если идентичность сборки установлена, среда CLR в состоянии определить, была ли ранее предпринята попытка разрешить эту зависимость в течение текущего выполнения приложения, как показано в следующем фрагменте, взятом из файла проекта, разрабатываемого в Visual Studio:

```
<reference include="MyAssembly, Version=2.1.0.0, Culture=neutral,
   PublicKeyToken=17fac983cbea459c" />
```

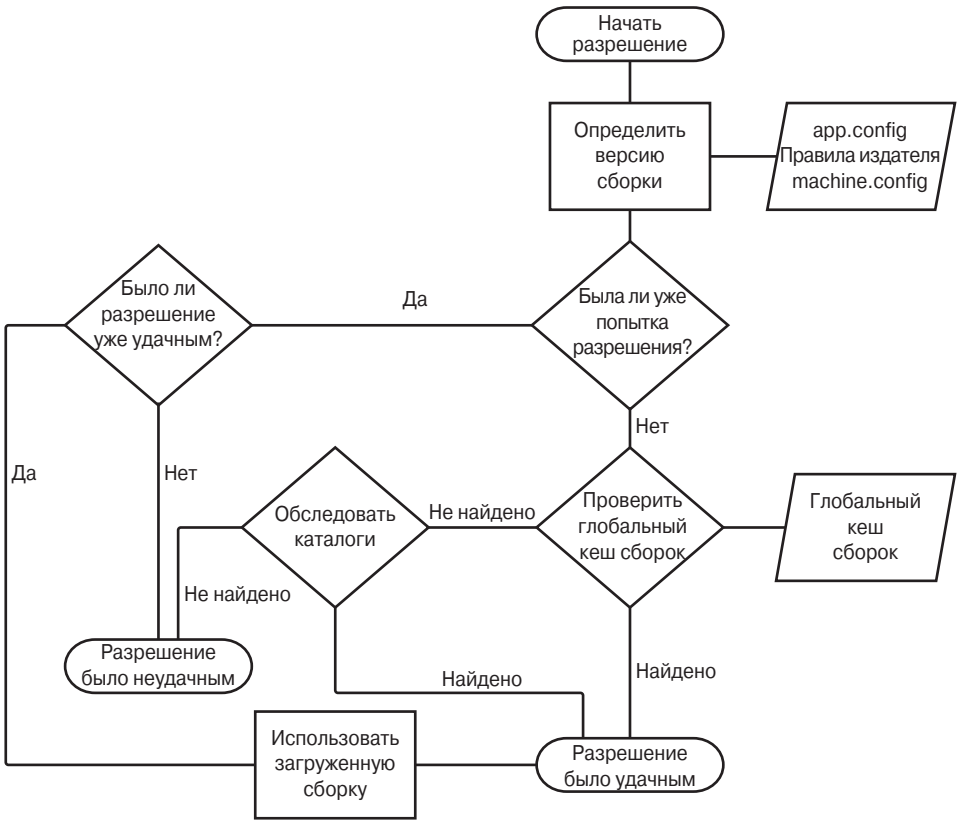

Рис. 2.15. *Краткий обзор процесса разрешения сборок*

- В зависимости от ответа на этот вопрос процесс разрешения в среде CLR разветвляется. Если попытка разрешить данную сборку предпринималась ранее, то данный процесс уже завершился удачно или неудачно. Если он завершился удачно, то загруженная сборка может быть использована в среде CLR, и на этом процесс разрешения завершается. В противном случае в среде CLR становится известно, что продолжать попытку разрешить данную сборку не следует, поскольку исход будет неудачным.
- С другой стороны, если это первая попытка разрешить сборку, то в среде CLR сначала проверяется глобальный кеш сборок (GAC) — машинное хранилище сборок, позволяющее выполнять несколько версий одной и той же сборки в одном приложении. Если сборка находится в GAC, процесс ее разрешения завершается удачно и найденная сборка загружается. Таким образом, наличию искомой сборки в GAC отдается предпочтение над ее присутствием в файловой системе, поскольку ее поиск начинается именно с кеша GAC.
- Если сборку не удается найти в GAC, то в среде CLR начинается обследование различных каталогов на наличие в них этой сборки. Поиск в каталогах осуществляется в зависимости от настроек в файле `app.config`. Если в этом файле

имеется элемент codeBase, то проверяется указанное в нем местонахождение сборки. И если сборка обнаруживается, то другие возможные места ее нахождения далее не проверяются. Но по умолчанию поиск начинается с корневого каталога приложения, которым, как правило, является папка /bin, связанная с точкой входа или веб-приложением. И если и там не удается найти сборку, то процесс ее разрешения завершается неудачно и в среде CLR генерируется исключение. Как правило, это приводит к прекращению работы приложения.

Файл регистрации Fusion

Это очень полезное средство для отладки неудачных попыток, предпринимаемых в среде CLR для привязки сборки во время выполнения. Вместо пошаговой отладки приложения в Visual Studio лучше активизировать Fusion и прочитать результаты из файла регистрации. Чтобы активизировать Fusion, следует внести приведенные ниже правки в системный реестр Windows.

```
HKLM\Software\Microsoft\Fusion\ForceLog 1
HKLM\Software\Microsoft\Fusion\LogPath C:\FusionLogs
```

Значение в поле ForceLog относится к типу DWORD, тогда как значение в поле LogPath — к строковому типу. В поле LogPath можно установить любой выбранный путь к файлу регистрации. В листинге 2.12 приведен пример регистрации неудачной попытки привязать сборку.

Листинг 2.12. Примерный результат, выводимый в файл регистрации Fusion при неудачной попытке привязать сборку

```
*** Assembly Binder Log Entry  (6/21/2013 @ 1:50:14 PM) ***

The operation failed.
Bind result: hr = 0x80070002. The system cannot find the file specified.

Assembly manager loaded from:
    C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
Running under executable
    C:\Program Files\1UPIndustries\Bins\v1.1.0.242\Bins.exe
 A detailed error log follows.

=== Prebind state information ===
LOG: User = DEV\gmclean
LOG: DisplayName = TaskbarDockUI.Xtensions.Bins.resources,
    Version=1.0.0.0, Culture=enUS, PublicKeyToken=null (Fullyspecified)
LOG: Appbase = file:///C:/Program Files/1UPIndustries/Bins/v1.1.0.242/
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = Bins.exe
Calling assembly : TaskbarDockUI.Xtensions.Bins, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=null.
===
LOG: This bind starts in default load context.
LOG: No application configuration file found.
LOG: Using host configuration file:
```

```
LOG: Using machine configuration file from
    C:\Windows\Microsoft.NET\Framework64\v4.0.30319\config\machine.config.
LOG: Policy not being applied to reference at this time
    (private, custom, partial, or locationbased assembly bind).
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/enUS/TaskbarDockUI.Xtensions.Bins.resources.DLL.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/enUS/TaskbarDockUI.Xtensions.Bins.resources/
   TaskbarDockUI.Xtensions.Bins.resources.DLL.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
    Bins/v1.1.0.242/enUS/TaskbarDockUI.Xtensions.Bins.resources.EXE.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/enUS/TaskbarDockUI.Xtensions.Bins.resources/
   TaskbarDockUI.Xtensions.Bins.resources.EXE.
LOG: All probing URLs attempted and failed.
```

После внесения упомянутых выше правок в системный реестр все попытки (удачные или неудачные) разрешить сборку в любом управляемом приложении будут записаны в файлы регистрации. Очевидно, что таких файлов будет немало. И это вроде бы неплохо, но когда их окажется слишком много, то причины неудачного завершения процесса разрешения придется искать как иголку в стоге сена. Правда, в Fusion имеется пользовательский интерфейс, немного упрощающий поиск нужного файла регистрации для конкретного приложения, а не во всей файловой системе (рис. 2.16).

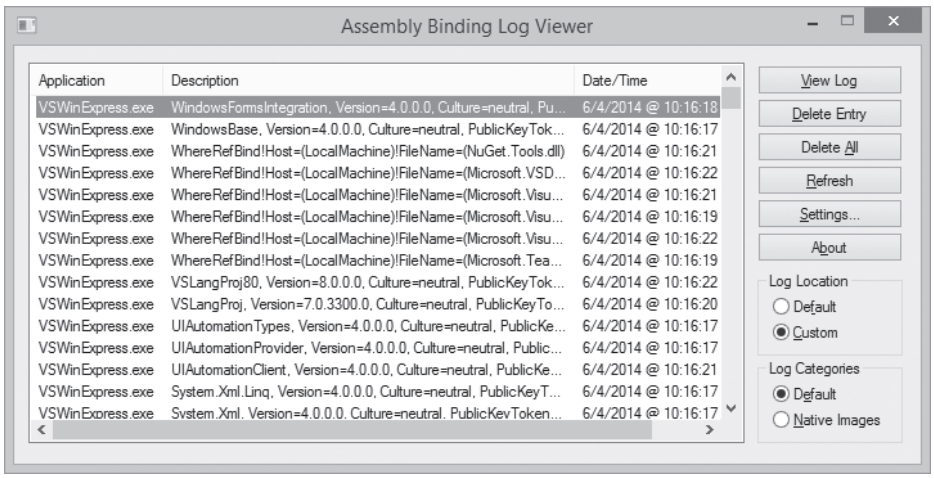

Рис. 2.16. *Для поиска нужного файла регистрации в Fusion имеется удобный пользовательский интерфейс*

Не всем зависимостям требуются ссылки на сборки. В качестве одной из альтернатив код службы можно развернуть как размещаемую на сервере службу. Для этого требуется межпроцессное или межсетевое взаимодействие, но в то же время сводится к минимуму количество ссылок на сборки, требующихся между клиентом и сервером, как поясняется в следующем подразделе.

Службы

В сравнении со сборками привязка клиента к размещаемой на сервере службе может быть намного более свободной, что выгодно, хотя и требует затрат. Клиент может знать много или, наоборот, мало о местоположении службы, что зависит от конкретных требований к приложению. Можно также варьировать порядок реализации службы таким образом, чтобы она удовлетворяла совсем немногим требованиям. Каждая из этих возможностей допускает разные компромиссы.

Известная конечная точка

Если местоположение службы известно клиентам во время компиляции, то на стороне клиента можно создать посредник этой службы. Такой посредник может быть создан, по крайней мере, двумя способами: автоматически в Visual Studio, где в проект вводится ссылка на службу, или вручную с помощью класса `ChannelFactory` на платформе .NET Framework.

Ввести ссылку на службу в Visual Studio очень просто. Для этого достаточно выбрать команду Add Service Reference из контекстного меню, щелкнув правой кнопкой мыши на выбранном проекте. В открывшемся диалоговом окне Add Service Reference (Ввод ссылки на службу) нужно лишь указать местоположение файла WSDL (Web Services Definition Language — язык определения веб-служб), предоставляющего метаданные, описывающие службу, ее типы данных и доступное поведение. После этого в Visual Studio будет автоматически сформирован ряд классов посредника данной службы, благодаря чему экономится немало времени. Для данной службы могут быть даже сформированы асинхронные версии методов, чтобы смягчить последствия блокировки. Но недостаток такого способа состоит в том, что теряется контроль над генерируемым кодом. Ведь код, генерируемый в Visual Studio, вряд ли будет соответствовать принятым внутренним нормам программирования, что может вызвать определенные трудности, если эти нормы достаточно строгие. Кроме того, автоматически формируемый посредник службы может не поддаваться модульному тестированию, поскольку он не предполагает формирование никаких интерфейсов, а только реализацию классов.

Альтернативой вводу ссылки на службу является создание посредника этой службы вручную. Такой способ лучше всего подходит для тех случаев, когда у клиента имеется доступ к интерфейсу службы и возможность повторного обращения к ней непосредственно по ссылке. В листинге 2.13 приведен пример создания посредника службы с помощью класса `ChannelFactory`.

Листинг 2.13. Класс `ChannelFactory` позволяет создать посредник службы

```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost/MyService");
var channelFactory = new ChannelFactory<IService>(binding, address);
var service = channelFactory.CreateChannel();
service.MyOperation();
service.Close();
channelFactory.Close();
```

Класс `ChannelFactory` является обобщенным, а его конструктору требуется интерфейс для создаваемого посредника службы. В приведенном выше коде применяются также конструкторы классов `BasicHttpBinding` и `EndpointAddress`, и поэтому класс `ChannelFactory` нужно снабдить полным адресом/привязкой/контрактом (ABC). В данном примере интерфейс `IService` такой же, как и реализуемый службой. В результате вызова метода `ChannelFactory.CreateChannel()` получается посредник, вызывающий эквивалентный метод из реализации службы на стороне сервера. А поскольку применяется один и тот же интерфейс, то он может потребоваться в классах на стороне клиента в качестве параметра их конструкторов, разрешаемого внедрением зависимости, и тогда эти классы сразу же станут тестируемыми. Кроме того, они избавляются от необходимости знать, что они вызывают удаленную службу.

Обнаружение службы

Иногда может быть известен тип привязки службы или ее контракт, но не адрес ее размещения. В таком случае для обнаружения службы можно воспользоваться компонентом Service Discovery, внедренным в Windows Communication Foundation (WCF) на платформе .NET Framework 4.

У компонента Service Discovery имеются две разновидности: управляемая и специальная. В управляемом режиме централизованная служба, называемая посредником обнаружения, хорошо известна клиентам, которые непосредственно посылают ей запросы для поиска других служб. Это менее интересный режим, поскольку в нем вносится единая точка отказа (SPOF). Если посредник обнаружения отсутствует, то клиентам не доступна ни одна из других служб, поскольку они не обнаруживаются.

Специальный режим избавляет от необходимости в посреднике обнаружения, поскольку вместо него используются многоадресные сетевые сообщения. В стандартной реализации этого режима применяется протокол UDP (User Datagram Protocol — пользовательский протокол данных), причем каждая обнаруживаемая служба принимает запросы по указанному IP-адресу[1]. Клиенты, по существу, посылают в сеть следующий запрос: имеется ли в ней служба, отвечающая критериям их запроса, например, типу контракта или привязки. И если одна из служб оказывается недоступной, то лишь она не может быть обнаружена, тогда как остальные службы ответят на запросы. В листинге 2.14 демонстрируется программный способ размещения обнаруживаемой службы, а в листинге 2.15 — способ ее размещения путем настройки.

Листинг 2.14. Программный способ размещения обнаруживаемой службы

```
class Program
{
    static void Main(string[] args)
    {
        using (ServiceHost serviceHost =
                    new ServiceHost(typeof(CalculatorService)))
```

[1] Используется IP-адрес **`239.255.255.250`** (по протоколу IPv4) или **`[FF02::C]`** (по протоколу IPv6) и порт номер **`3702`**. Эти параметры устанавливаются по стандарту WS-Discovery и не подлежат настройке.

```
        {
            serviceHost.Description.Behaviors.Add
                  (new ServiceDiscoveryBehavior());
            serviceHost.AddServiceEndpoint(typeof(ICalculator),
                  new BasicHttpBinding(),
                  new Uri("http://localhost:8090/CalculatorService"));
            serviceHost.AddServiceEndpoint(new UdpDiscoveryEndpoint());

            serviceHost.Open();
            Console.WriteLine("Discoverable Calculator Service is running...");
            Console.ReadKey();
        }
    }
}
```

Листинг 2.15. Способ размещения обнаруживаемой службы путем настройки

```
<system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="false"/>
        </behavior>
        <behavior name="calculatorServiceDiscovery">
          <serviceDiscovery />
        </behavior>
      </serviceBehaviors>
      <endpointBehaviors>
        <behavior name="calculatorHttpEndpointDiscovery">
          <endpointDiscovery enabled="true" />
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <protocolMapping>
        <add binding="basicHttpsBinding" scheme="https" />
    </protocolMapping>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
   multipleSiteBindingsEnabled="true" />
    <services>
      <service name="ConfigDiscoverableService.CalculatorService"
   behaviorConfiguration="calculatorServiceDiscovery">
        <endpoint address="CalculatorService.svc"
   behaviorConfiguration="calculatorHttpEndpointDiscovery"
contract="ServiceContract.ICalculator" binding="basicHttpBinding" />
        <endpoint kind="udpDiscoveryEndpoint" />
      </service>
    </services>
  </system.serviceModel>
```

Чтобы служба стала обнаруживаемой, достаточно ввести поведение типа `ServiceDiscoveryBehavior` и разместить конечную точку обнаружения типа `DiscoveryEndpoint`. В данном примере конечная точка типа `UdpDiscoveryEndpoint` используется для получения многоадресных сетевых сообщений от клиентов.

На заметку

Компонент Service Discovery в WCF соответствует стандарту WS-Discovery. Вследствие этого он пригоден для работы не только на платформе .NET Framework, но и на других платформах и языках программирования.

Клиенты пользуются классом `DiscoveryClient` в целях поиска обнаруживаемой службы, для чего требуется также конечная точка типа `DiscoveryEndpoint`. Затем вызывается метод `Find()` с настроенным экземпляром класса `FindCriteria`, содержащим свойство `Endpoints` с коллекцией экземпляров класса `EndpointDiscoveryMetadata`, — по одному на каждую совпадающую службу. В листинге 2.16 демонстрируются шаги, предпринимаемые для поиска обнаруживаемой службы.

Листинг 2.16. Компонент Service Discovery вполне пригоден для развязки прикладного кода

```
class Program
{
    private const int a = 11894;
    private const int b = 27834;

    static void Main(string[] args)
    {
        var foundEndpoints = FindEndpointsByContract<ICalculator>();

        if (!foundEndpoints.Any())
        {
            Console.WriteLine("No endpoints were found.");
        }
        else
        {
            var binding = new BasicHttpBinding();
            var channelFactory = new ChannelFactory<ICalculator>(binding);
            foreach (var endpointAddress in foundEndpoints)
            {
                var service = channelFactory.CreateChannel(endpointAddress);
                var additionResult = service.Add(a, b);
                Console.WriteLine("Service Found: {0}", endpointAddress.Uri);
                Console.WriteLine("{0} + {1} = {2}", a, b, additionResult);
            }
        }

        Console.ReadKey();
    }

    private static IEnumerable<EndpointAddress> FindEndpointsByContract
    <TServiceContract>()
    {
        var discoveryClient =
            new DiscoveryClient(new UdpDiscoveryEndpoint());
        var findResponse = discoveryClient.Find(new
```

```
    FindCriteria(typeof(TServiceContract)));
         return findResponse.Endpoints.Select(metadata => metadata.Address);
     }
}
```

Следует, однако, иметь в виду, что в протоколе UDP, в отличие от протокола TCP, доставка сообщения не гарантируется. Дейтаграммы могут теряться во время передачи, а следовательно, запрос может не достигнуть действующей службы, а посылаемый обратно ответ — клиента. Но в любом случае клиент посчитает, что служба недоступна для обработки запроса.

Совет

Если обнаруживаемая служба размещается на сервере IIS (Internet Information Services — информационные службы Интернета) или на сервере WAS (Windows Process Activation Service — служба активизации процессов в Windows), следует непременно воспользоваться функцией AutoStart (Автозапуск) на платформе Microsoft AppFabric. Обнаруживаемость службы зависит от ее доступности, а это означает, что служба должна действовать, чтобы получать запросы от клиентов. Функция AutoStart позволяет службе действовать при запуске приложения на сервере IIS. А без функции AutoStart служба не запускается до тех пор, пока не будет принят первый запрос.

Службы RESTful

Наиболее побудительной причиной для создания служб RESTful, где REST (REpresentational State Transfer, т.е. передача репрезентативного состояния) является очень малое бремя зависимостей, налагаемое на клиентов. Для этого требуется лишь HTTP-клиент, который обычно предоставляется каркасами и библиотекам многих языков программирования. Благодаря этому службы RESTful идеально подходят для разработки таких служб, которым требуется широкомасштабная поддержка на разных платформах. Например, у обеих веб-служб, Facebook и Twitter, имеются прикладные программные интерфейсы REST API для обработки различных запросов и команд. Этим гарантируется, что клиенты могут быть разработаны для обширного ряда платформ, в том числе Windows Phone 8, iPhone, Android, iPad, Windows 8, Linux и многих других. Реализовать единую службу, способную обслуживать всех этих клиентов, было бы много труднее без очень низких требований к зависимостям, допускаемых в архитектуре REST.

Прикладной программный интерфейс Web API на платформе ASP.NET служит для создания служб RESTful на платформе .NET Framework. Аналогично платформе ASP.NET MVC, он позволяет разработчикам создавать методы, непосредственно преобразуемые в веб-запросы. В прикладном программном интерфейсе Web API предоставляется базовый класс контроллера под названием `ApiController`. От этого класса контроллера наследуются или вводятся методы, называемые аналогично методам выдачи запросов по сетевому протоколу HTTP: `GET`, `POST`, `PUT`, `DELETE.HEAD`, `OPTIONS` и `PATCH`. Всякий раз, когда поступает HTTP-запрос, обозначаемый одним из этих имен, вызывается соответствующий метод. В листинге 2.17 демонстрируется пример службы, реализующей все эти методы.

Листинг 2.17. Почти все названия методов сетевого протокола HTTP поддерживаются в прикладном программном интерфейсе Web API на платформе ASP.NET

```
public class ValuesController : ApiController
{
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    public string Get(int id)
    {
        return "value";
    }

    public void Post([FromBody]string value)
    {
    }

    public void Put(int id, [FromBody]string value)
    {
    }

    public void Head()
    {
    }

    public void Options()
    {
    }

    public void Patch()
    {
    }

    public void Delete(int id)
    {
    }
}
```

В листинге 2.18 демонстрируется клиентский код для доступа к методам `GET` и `POST` данной службы средствами класса `HttpClient`. И хотя это всего лишь способ доступа к службам RESTful на платформе .NET Framework, он все-таки опирается только на саму платформу.

Листинг 2.18. Клиенты могут пользоваться классом `HttpClient` для доступа к службам RESTful

```
class Program
{
    static void Main(string[] args)
    {
        string uri = "http://localhost:7617/api/values";

        MakeGetRequest(uri);
```

```
        MakePostRequest(uri);

        Console.ReadKey();
    }

    private static async void MakeGetRequest(string uri)
    {
        var restClient = new HttpClient();
        var getRequest = await restClient.GetStringAsync(uri);

        Console.WriteLine(getRequest);
    }

    private static async void MakePostRequest(string uri)
    {
        var restClient = new HttpClient();
        var postRequest = await restClient.PostAsync(uri,
                new StringContent("Data to send to the server"));

            var responseContent =
                        await postRequest.Content.ReadAsStringAsync();
            Console.WriteLine(responseContent);
        }
    }
```

Чтобы лишний раз подчеркнуть то обстоятельство, что многие клиенты могут быть написаны с низкими в равной степени требованиями к зависимостям, в листинге 2.19 демонстрируется сценарий Windows PowerShell 3 для доступа к методам `GET` и `POST` запрашиваемой службы.

Листинг 2.19. Доступ к службе RESTful из сценария Windows PowerShell 3 весьма тривиален

```
$request = [System.Net.WebRequest]::Create("http://localhost:7617/api/
values")
$request.Method ="GET"
$request.ContentLength = 0

$response = $request.GetResponse()
$reader = newobject System.IO.StreamReader($response.GetResponseStream())
$responseContent = $reader.ReadToEnd()
WriteHost $responseContent
```

В приведенном выше коде объект класса `WebRequest` из платформы .NET Framework служит для вызова службы RESTful. Этот класс, в свою очередь, служит суперклассом для класса `HttpRequest`. А метод `Create()` является фабричным и возвращает экземпляр класса `HttpRequest`, поскольку в качестве параметра ему предоставлен URI с префиксом **`http://`**.

Управление зависимостями средствами NuGet

Управление зависимостями можно значительно упростить, применяя соответствующие инструментальные средства. Такие инструментальные средства отвечают за

прослеживание цепочек зависимостей, чтобы обеспечить наличие всех зависимых артефактов. Они также осуществляют контроль версий зависимостей, позволяя указать потребность положиться только на конкретные версии зависимостей, а все остальное сделают эти инструментальные средства.

NuGet — это утилита управления пакетами на платформе .NET Framework. С этой целью утилита NuGet ссылается на зависимость как на *пакет*, но действие этого инструментального средства не ограничивается только сборками. В состав пакетов NuGet входят также средства конфигурации, сценарии и изображения, т.е. практически все, что нужно. Одной из побудительных причин для применения такого диспетчера пакетов, как NuGet, служит то обстоятельство, что ему известны зависимости пакетов, и благодаря этому он может извлечь всю цепочку зависимостей, когда в проекте делается ссылка на пакет. В версии Visual Studio 2013 утилита управления пакетами NuGet полностью интегрирована в эту ИСР как стандартная.

Потребление пакетов

Утилита NuGet вводит новые команды в контекстное меню, доступное в окне Solution Explorer (Обозреватель решений) ИСР Visual Studio. Из этого меню можно открыть окно управления пакетами в NuGet и ввести ссылку на зависимость. В качестве примера введем зависимость в CorrugatedIron — драйвер клиентов на платформе .NET Framework для хранения пар "ключ–значение" в базе данных Riak NoSql. Окно управления пакетами в NuGet показано на рис. 2.17.

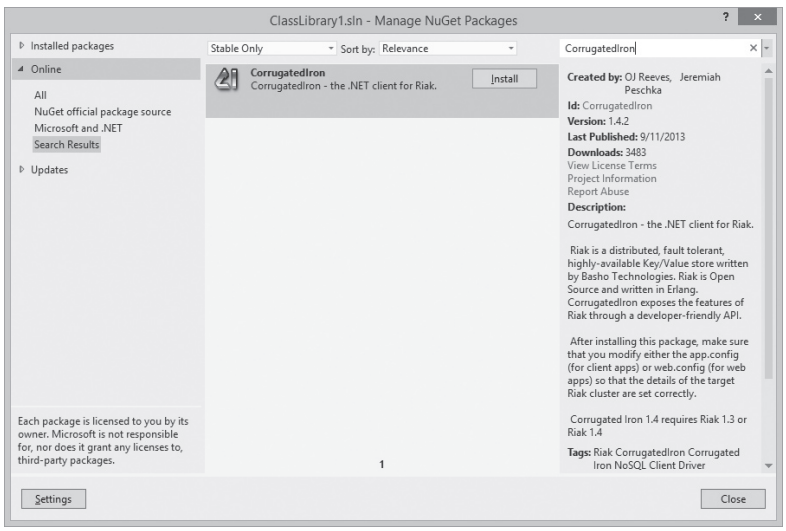

Рис. 2.17. *С пакетами в утилите NuGet связано немало полезных метаданных*

Всякий раз, когда пакет выбирается из списка, на информационной панели справа появляются некоторые метаданные о пакете. К ним относится однозначное имя пакета, его авторы, версия, дата последнего выпуска, описание и любые зависимости, имеющиеся у пакета. Эти зависимости весьма любопытны, поскольку они показывают, что еще должно быть установлено в результате ссылки на данный пакет,

помимо обязательных или поддерживаемых версий зависимостей. Например, пакету CorrugatedIron требуется, по крайней мере, версия 4.5.10 Newtonsoft.Json — сериализатора классов в формате JSON на платформе .NET Framework и хотя бы версия 2.0.0.602 протокола protobuf-net. Оба эти пакета могут иметь свои зависимости.

Если выбрать установку данного пакета, утилита NuGet сначала попытается загрузить все файлы и разместить их в папке `packages/`, входящей в корневой каталог текущего решения. Это дает возможность разместить всю данную папку в системе контроля версий аналогично упоминавшейся ранее папке `dependencies/`. Затем утилита NuGet ссылается на загруженные сборки в тот проект, где данная библиотека применяется. На рис. 2.18 показаны ссылки для проекта после ввода пакета Riak.

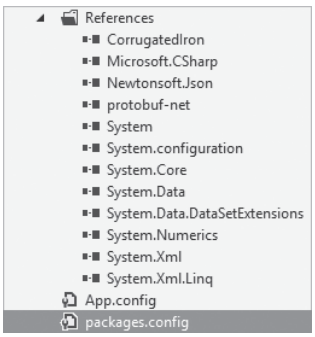

Рис. 2.18. *Ссылки в проекте на целевой пакет и все его зависимости*

Помимо ссылок, утилита NuGet также создает файл конфигурации `packages.config`, содержащий сведения о пакетах и их версиях, на которые делаются ссылки в проекте. Это удобно, когда дело касается обновления или удаления пакетов, на что утилита NuGet также способна.

Прежде чем воспользоваться пакетом Riak, требуется некоторая его исходная настройка. Поэтому утилита NuGet не только загружает и ссылается на многие сборки, но и вносит поправки в файл конфигурации `app.config`, чтобы ввести в него ряд значений по умолчанию, требующихся для правильной настройки пакета Riak. В листинге 2.20 демонстрируется текущее состояние файла конфигурации `app.config` после установки пакета Riak.

Листинг 2.20. Утилита ввела в файл конфигурации `app.config` новый раздел настроек `configSection` специально для пакета Riak.

```
<configuration>
  <configSections>
    <section name="riakConfig" type=
             "CorrugatedIron.Config.RiakClusterConfiguration,
             CorrugatedIron" />
      </configSections>
  <riakConfig nodePollTime="5000"
              defaultRetryWaitTime="200" defaultRetryCount="3">
    <nodes>
      <node name="dev1" hostAddress="riaktest"
              pbcPort="10017" restScheme="http"
```

```
            restPort="10018" poolSize="20" />
      <node name="dev2" hostAddress="riaktest"
            pbcPort="10027" restScheme="http"
            restPort="10028" poolSize="20" />
      <node name="dev3" hostAddress="riaktest"
            pbcPort="10037" restScheme="http"
            restPort="10038" poolSize="20" />
      <node name="dev4" hostAddress="riaktest"
            pbcPort="10047" restScheme="http"
            restPort="10048" poolSize="20" />
    </nodes>
  </riakConfig>
</configuration>
```

Очевидно, что это позволяет значительно экономить время, поскольку избавляет от необходимости обращаться на веб-сайт пакета Riak и загружать пакет CorrugatedIron или любые его зависимости. Все автоматически приведено в такое состояние, чтобы можно было сосредоточиться на самой разработке. А когда настанет момент перейти к следующей версии пакета CorrugatedIron, достаточно обратиться к утилите NuGet, чтобы автоматически обновить все зависящие от него пакеты во всем приложении.

Создание пакетов

Утилита NuGet позволяет также создавать пакеты. В частности, пакеты могут быть созданы для публикации в официальной галерее пакетов NuGet, чтобы ими могли воспользоваться другие разработчики, или же размещены каналы доступа к собственным пакетам для основных зависимостей. На рис. 2.19 показан обозреватель пакетов (Package Explorer), который может быть использован в утилите NuGet для создания собственных пакетов. В данном случае пакет CorrugatedIron настроен в качестве зависимости, чтобы стать также косвенно зависимым от пакетов Newtonsoft.Json и protobuf-net. Кроме того, введен библиотечный артефакт, специально предназначенный для платформы .NET Framework 4.5.1, а также текстовый файл, создаваемый в доступной по ссылке сборке и размещаемый по пути `My folder/NewFile.txt`. Имеется даже сценарий Windows PowerShell, запускаемый на выполнение во время установки. Благодаря гибкости сценария Windows PowerShell все это не требует почти никаких затрат труда.

В каждом пакете имеется XML-файл, отображаемый в окне установки. Он содержит метаданные, подробно описывающие пакет. В качестве примера в листинге 2.21 демонстрируется фрагмент метаданных из этого файла.

Листинг 2.21. Определение пакета в XML-файл, включая метаданные

```
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        <id>MyTestPackage</id>
        <version>1.0.0</version>
        <authors>Gary McLean Hall</authors>
        <requireLicenseAcceptance>false</requireLicenseAcceptance>
        <description>My package description.</description>
        <dependencies>
            <dependency id="CorrugatedIron" version="1.0.1" />
```

```
        </dependencies>
    </metadata>
</package>
```

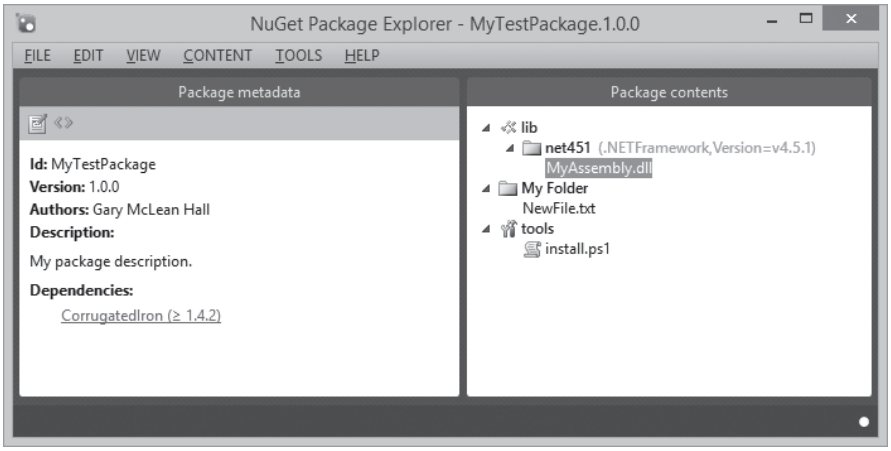

Рис. 2.19. *Обозреватель пакетов заметно упрощает построение собственных пакетов в утилите NuGet*

Утилита NuGet дает такие преимущества в производительности и организации труда, что было бы очень тяжело возвращаться к ручному управлению зависимостями в том случае, если для конкретной библиотеки отсутствует пакет. В действительности утилита NuGet позволяет управлять не только сторонними зависимостями. Когда решение становится достаточно крупным, рекомендуется разделить его на несколько частей, срезав по уровням, а затем упаковать сборки на каждом уровне, используя использованные выше уровни и утилиту NuGet. Благодаря этому решение остается компактным и удобным в обращении.

Chocolatey

Приложение Chocolatey лучше всего описать как инструментальное средство для управления пакетами аналогично утилите NuGet, но в данном случае пакетами служат приложения и прочие инструментальные средства, а не сборки. Те разработчики, у которых имеется некоторый опыт работы в ОС Linux, могут найти приложение Chocolatey похожим на команду `apt-get`, реализующую диспетчер пакетов в ОС Debian и Ubuntu. И в этом приложении реализованы многие преимущества управления пакетами, в том числе простота установки, использования и управления зависимостями.

Загрузить и установить Chocolatey можно по следующему сценарию:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command
    "iex ((new-object net.webclient).DownloadString(
    'https://chocolatey.org/install.ps1'))" &&
     SET PATH=%PATH%;%systemdrive%\chocolatey\bin
```

После установки Chocolatey можно воспользоваться режимом командной строки для последующей установки различных приложений и инструментальных средств. В процессе установки путь, указываемый в командной строке, обновляется

автоматически и включает в себя приложение `chocolatey.exe`. Как и в Git, в приложении Chocolatey поддерживаются подкоманды вроде `list` и `install`, но у них имеются соответствующие синонимы `clist` и `cinst`, применяемые для краткости. В листинге 2.22 демонстрируется сеанс работы в Chocolatey для поиска пакета FTP-клиента по имени Filezilla и последующей его установки.

Листинг 2.22. Сначала для приложения находятся требующиеся пакеты, а затем они устанавливаются

```
C:\dev> clist filezilla
ferventcoder.chocolatey.utilities 1.0.20130622
filezilla 3.7.1
filezilla.commandline 3.7.1
filezilla.server 0.9.41.20120523
jivkok.tools 1.1.0.2
kareemsultan.developer.toolkit 1.4
UAdevelopers.utils 1.9
C:\dev> cinst filezilla
Chocolatey (v0.9.8.20) is installing filezilla and dependencies.
By installing you accept the license for filezilla and each dependency
you are installing.

(Chocolatey (v0.9.8.20) устанавливает пакет filezilla и его зависимости.
Устанавливая пакет filezilla и каждую его зависимость, вы тем самым
принимаете условия лицензионного соглашения.)
 . . .
 This Finished installing 'filezilla' and dependencies - if errors not
shown in console, none detected. Check log for errors if unsure.

(На этом установка пакета filezilla и его зависимостей завершается —
если ошибки не выводятся на консоль, значит, они не обнаружены.
Чтобы убедиться в этом, проверьте журнал регистрации.)
```

Если приложение Chocolatey не сообщило об ошибках, значит, запрашиваемый пакет можно считать установленным. Следует, однако, иметь в виду, что приложение Chocolatey *должно* было изменить системную переменную окружения `PATH`, чтобы любые новые двоичные файлы могли исполняться из командной строки, хотя подобное изменение делается не всегда. Посредством Chocolatey становятся доступными многие пакеты, а удобство обращения с ними служит побудительной причиной для применения этого приложения.

Разделение на уровни

До сих пор в этой главе рассматривались главным образом вопросы управления зависимостями на уровне сборок. И это вполне естественный шаг на пути к правильной организации приложения, поскольку все классы и интерфейсы содержатся в сборках, а следовательно, главное внимание должно быть уделено упорядочению их взаимных ссылок. Если сборки организованы правильно, то они будут состоять из классов и интерфейсов, относящихся только к одной команде связанных вместе

функциональных возможностей. Но как обеспечить правильную организацию взаимосвязанных команд сборок?

Команды, состоящие из двух или более взаимосвязанных сборок, образуют *компоненты* разрабатываемой программной системы. Не менее, а возможно, и более важно, чтобы эти компоненты взаимодействовали вполне определенным и структурированным образом. Как показано на рис. 2.20, компоненты являются не физическими артефактами развертывания подобно динамически подключаемым библиотекам (DLL), а логическими командами сборок, разделяющих общую тему.

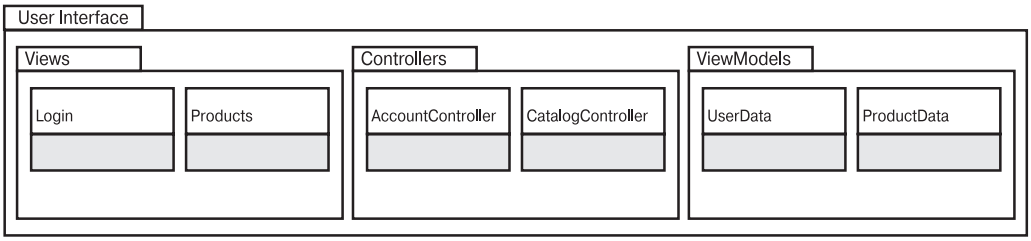

Рис. 2.20. *Командируя взаимосвязанные сборки, можно определить логические компоненты*

В блок-схеме на рис. 2.20 представлены следующие три сборки: `Views`, `Controllers` и `ViewModels`. Каждая сборка состоит из двух классов, которые имеют разное назначение и могут потребовать разные зависимости. И хотя эти классы и сборки относятся к конструкциям, предоставляемым платформой .NET Framework, пакет `User Interface` вполне логично командирует их вместе. Эти три сборки можно было бы разместить в папке `UserInteraces` одного решения, но ничто, кроме собственного усердия, не мешает засорить данный уровень другим, не относящимся к нему проектом.

Для управления зависимостями компоненты ничем не отличаются от других программных конструкций на более низких уровнях. Подобно методам, классам и сборкам, уровни можно рассматривать как еще один узел графа зависимостей, упоминавшегося ранее в этой главе. Следовательно, на них распространяются те же самые правила: сохранение орграфа ациклическим и обеспечение единственной ответственности.

Разделение на уровни является архитектурным шаблоном, побуждающим мысленно представлять программные компоненты в виде горизонтальных уровней функциональных возможностей, располагаемых друг над другом, образуя целое приложение. Компоненты разделяются на уровни, находящиеся друг над другом, а направление зависимости должно всегда указывать вниз. Это означает, что на нижнем уровне приложения отсутствуют зависимости,[2] причем каждый более высокий уровень зависит от уровня ниже него, а на самом верхнем уровне находится пользовательский интерфейс.

[2] Строго говоря, это не совсем так, поскольку вполне возможны некоторые зависимости. Но поскольку это самый нижний уровень, где отсутствует другой код с основной зависимостью, то такие зависимости, вероятнее всего, будут сторонними, инфраструктурными.

Если приложение относится к уровню службы, то на самом верхнем уровне окажется прикладной программный интерфейс API, через который клиенты могут взаимодействовать с системой.

Общие шаблоны

Имеется несколько общих шаблонов разделения на уровни, среди которых можно выбрать наиболее подходящий для любого проекта. Каждый из представленных здесь шаблонов следует использовать как общий ориентир, приспосабливаемый под конкретные требования и ограничения вырабатываемого решения. Шаблоны разделения на уровни отличаются лишь *количеством* применяемых уровней. В начале этого раздела рассматривается простая архитектура, состоящая только из двух уровней, а затем она дополняется третьим уровнем, вставляемым между ними, и, наконец, она экстраполируется на произвольное количество уровней.

Количество требующихся уровней тесно связано со сложностью решения, а сложность решения — со сложностью задачи. Следовательно, чем сложнее задача, тем сильнее побуждение вкладывать средства в многоуровневую архитектуру. Сложность в данном случае определяется многими факторами, в том числе временны*ми* ограничениями, накладываемыми на проект, требующейся его продолжительностью, частотой изменения требований к нему, а также тем значением, которое команда разработчиков придает шаблонам и практическим приемам.

Данная книга посвящена адаптации к изменениям в требованиях, и поэтому в ней пропагандируется принцип *делать все как можно проще* и усложнять реорганизацию *только тогда, когда это требуется*. Прежде всего, это позволяет выпустить программный продукт как можно скорее. В процессе разработки реакцию заказчика следует искать как можно раньше и чаще. Попытка выработать идеальное решение не принесет результатов, если идеальное представление о нем заказчика отличается от идеального представления команды разработчиков. Разработка многоуровневого решения отнимает больше времени, чем более простого двухуровневого решения, задерживая столь важную реакцию по цепи обратной связи.

Двухуровневая архитектура

Двухуровневая архитектура является простейшим шагом вперед по сравнению с отказом от разделения на уровни. Этот шаг полезен только в узком круге обстоятельств, но реализуется он очень быстро. Такая архитектура состоит только из двух уровней пользовательского интерфейса и доступа к данным. Но она не ограничивается лишь двумя *сборками*, а просто состоит из двух *команд* сборок: одной — для пользовательского интерфейса, другой — для доступа к данным.

На рис. 2.21 двухуровневая архитектура показана в виде пакетов в блок-схеме, составленной на языке UML, где каждый уровень содержит связанные с ним сборки. Единственная зависимость направлена сверху вниз от уровня пользовательского интерфейса к уровню доступа к данным. Это единственный способ направить зависимости в любой многоуровневой архитектуре.

Уровни и ярусы

Уровни отличаются от ярусов логической организацией и физическим развертыванием прикладного кода. Приложение можно логически разделить на три или четыре *уровня*, но физически развертывать его в одном *ярусе*. Количество ярусов в какой-то степени связано с количество машин, на которых развернуто приложение. Если развертывать его на одной машине, оно будет развернуто в одном ярусе. А если развертывать на двух машинах приложение, разделенное на два уровня, оно должно быть развернуто в двух ярусах.

С каждым ярусом, в котором развертывается приложение, приходится признавать, что тем самым пересекается граница сети, а это требует затрат времени. Если пересечение границы обработки на одной и той же машине требует немалых затрат, то еще бо́льших затрат потребует пересечение границы сети. Тем не менее разработка ярусами дает существенное преимущество, поскольку она позволяет масштабировать приложение. Так, если развертывается веб-приложение, состоящее из уровня пользовательского интерфейса, логического уровня и уровня доступа к базе данных, на одной машине, а следовательно, в одном ярусе, на эту машину возлагается большой объем работы, что по необходимости сокращает количество пользователей, которых способно поддерживать такое приложение. Разделяя разработку приложения на два яруса (с базой данных в одном ярусе, а пользовательским интерфейсом и логикой — в другом), можно фактически масштабировать уровень пользовательского интерфейса как по *горизонтали*, так и по *вертикали*.

Для масштабирования по вертикали достаточно нарастить вычислительные мощности компьютера дополнительными блоками оперативной памяти и процессоров. Но эти меры ограничиваются одной только машиной. А для масштабирования по горизонтали можно добавить новые машины, выполняющие точно такую же задачу. Так, на нескольких машинах может быть размещен код пользовательского интерфейса веб-приложения и распределитель нагрузки, который должен направлять клиентов на менее нагруженную машину в любой момент времени. Разумеется, это не панацея для поддержки большего количества пользователей, параллельно работающих с веб-приложением. Для этого придется уделить больше внимания кешированию и аутентификации пользователей, поскольку каждый запрос от пользователя может быть обработан на другой машине.

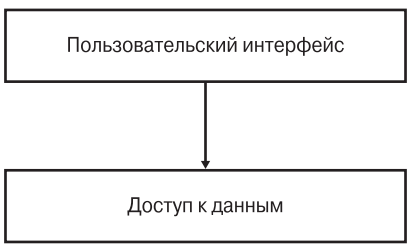

Рис. 2.21. *Двухуровневая архитектура приложения, состоящая из компонентов пользовательского интерфейса и доступа к данным*

Пользовательский интерфейс

Уровень пользовательского интерфейса отвечает за следующее.

- Обеспечение взаимодействия пользователя с приложением (например, через окна и элементы управления настольных приложений, страницы веб-приложений, командную строку или меню консольных приложений).
- Представление данных и прочих сведений пользователю.
- Получение требований от пользователя в форме запросов или команд.
- Проверка достоверности вводимых пользователем данных.

Реализация уровня пользовательского интерфейса может изменяться. Он может содержать клиент Windows Presentation Foundation (WPF), упакованный вместе с высококачественной графикой и анимацией, ряд веб-страниц для перемещения пользователя или простое консольное приложение с параметрами командной строки или простым меню для выбора пользователем исполняемой команды или запроса.

На заметку

В некоторых случаях пользовательский интерфейс можно было бы заменить рядом служб, оснащенных функциональными возможностями для клиентов. На самом деле это не пользовательский интерфейс, тем не менее, двухуровневая архитектура вполне очевидна, причем уровень пользовательского интерфейса заменяется уровнем прикладного пользовательского интерфейса API.

Уровень пользовательского интерфейса располагается выше уровня доступа к данным и позволяет им пользоваться. Но, как обсуждалось ранее в отношении ссылок на сборки, на уровне пользовательского интерфейса нельзя делать прямые ссылки на любые сборки реализации на уровне доступа к данным. Между сборками интерфейса и реализации на обоих уровнях должно быть строгое разграничение. Вследствие этого блок-схема разделения на уровни приобретает несколько иной вид, как показано на рис. 2.22.

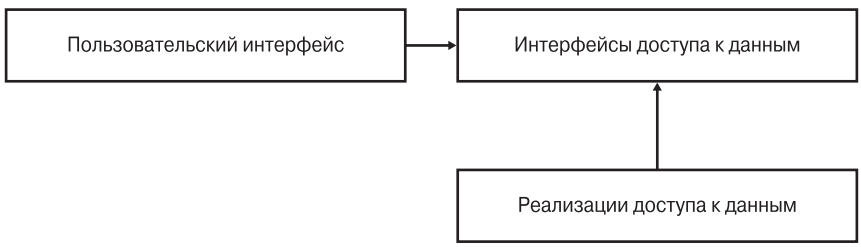

Рис. 2.22. *Два уровня разделяются на сборки реализации и сборки интерфейса*

Такое решение позволяет заменить антишаблон “Антураж” шаблоном “Лестница”, но на сей раз это делается на архитектурном уровне. Каждый уровень представляет собой сочетание абстракции функциональных возможностей, от которых зависит более высокий уровень, с реализацией этой абстракции. Если на более высоком уровне делается ссылка на часть реализации этого уровня, то такой уровень называется

протекающей абстракцией. Зависимости от реализации этого уровня начинают протекать на расположенные выше уровни и в конечном итоге становятся устранимыми.

Доступ к данным

На уровень доступа к данным возлагаются следующие виды ответственности.

- Обслуживание запросов данных.
- Сериализация и десериализация данных из объектных моделей в реляционные и обратно.

Реализация уровня доступа к данным может отличаться в такой же степени, как и реализация уровня пользовательского интерфейса. Как правило, этот уровень включает в себя постоянное хранилище данных вроде реляционной базы данных SQL Server, Oracle, MySQL, PostgreSQL или базы данных документов типа MongoDB, RavenDB, Riak. Помимо механизма сохранения данных, на этом уровне вполне возможно наличие одной или нескольких поддерживающих сборок для выполнения запросов или команд вставки, обновления, удаления путем вызова хранимых процедур или преобразования данных в формат реляционной базы данных средствами библиотеки Entity Framework или NHibernate.

Уровни доступа к данным должны скрываться за интерфейсами, не зависящими от выбора конкретной технологии. Как и во всех остальных интерфейсах, ссылка на стороннюю зависимость должна отсутствовать, а следовательно, клиенты отделены от выбора реализации.

Правильно построенный уровень доступа к данным допускает повторное применение во многих приложениях. Если в двух пользовательских интерфейсах требуются одни и те же данные, которые присутствуют в разных формах, то они могут разделять один и тот же общий уровень доступа к данным. Допустим, имеется приложение, работающее на платформах Windows 8 и Windows Phone 8. Обе эти платформы предъявляют разные требования к пользовательскому интерфейсу, но на каждой из них может быть использован один и тот же уровень доступа к данным.

Как и в любой другой архитектуре, применение только двух уровней вынуждает идти на некоторые компромиссы, которые требуют тщательного рассмотрения перед их принятием. Двухуровневая архитектура вполне подходит в следующих случаях.

- В приложении имеется немного (или вообще нет никакой) логики, кроме простой проверки достоверности данных. Такую логику нетрудно инкапсулировать на уровне доступа к данным или же на уровне пользовательского интерфейса.
- Приложение выполняет над данными, главным образом, операции создания, чтения, обновления, удаления (CRUD). Выполнять эти операции становится труднее с каждым дополнительным уровнем, размещаемым между пользовательским интерфейсом и самими данными.
- Мало времени. Если требуется разработать только прототип или начальный загрузчик, то, ограничив количество уровней, можно сэкономить немало времени и в то же время подтвердить обоснованность выбранной концепции. Если придерживаться такого положительного опыта проектирования, как шаблон “Лестница”, то вставить дополнительные уровни по мере надобности в дальнейшем будет много проще.

Но двухуровневой архитектуре присущи некоторые очевидные недостатки, и поэтому выбирать ее не рекомендуется в следующих случаях.

- В приложении имеется немало логики, или же эта логика подвержена изменениям. Любая логика, размещаемая на уровне пользовательского интерфейса или на уровне доступа к данным, формально засоряет этот уровень, уменьшая его гибкость и затрудняя его сопровождаемость.
- Приложение, очевидно, вырастает за пределы двух уровней после одного или двух спринтов. Любые уступки ради получения быстрой реакции на разработку не стоят вложенных средств, если такая архитектура просуществует в течение нескольких недель.

Тем не менее двухуровневая архитектура остается весьма жизнеспособной альтернативой. Ведь слишком многие разработчики, очарованные последними архитектурными тенденциями, просто упускают из виду более простые решения. А это приводит к тому, что реакция на разработку приложения, которое в противном случае можно было сделать простым, слишком запаздывает, что делает его ненадежным и затрудняет его сопровождение. Нередко простейший путь оказывается самым верным.

Трехуровневая архитектура

В такой архитектуре дополнительный уровень логики вводится между уровнями пользовательского интерфейс и доступа к данным. Это позволяет инкапсулировать на уровне логики более сложную обработку данных в приложении. Как и уровень доступа к данным, уровень логики может быть повторно использован во многих приложениях, а следовательно, его не требуется реализовывать многократно. Типичная трехуровневая архитектура представлена на рис. 2.23. Как и на уровне доступа к данным, на уровне логики клиентам могут быть предоставлены сборки реализации и сборки интерфейса, чтобы избежать протекающей абстракции.

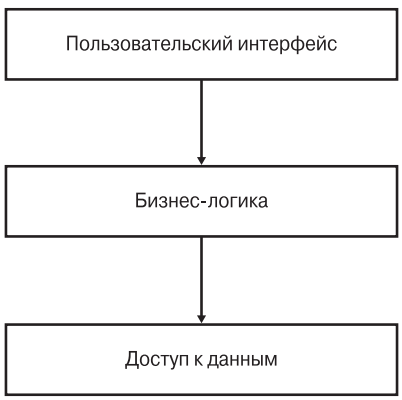

Рис. 2.23. *На третьем уровне располагается логика обработки данных или так называемая бизнес-логика приложения*

На заметку

Несмотря на то что трехуровневая архитектура весьма распространена в веб-приложениях, она, как правило, развертывается в двух ярусах. В одном ярусе располагается база данных, а в другом — практически все остальное: пользовательский интерфейс, логика и даже часть уровня доступа к данным.

Бизнес-логика

На уровень бизнес-логики возлагаются следующие виды ответственности.

- Обработка команд из уровня пользовательского интерфейса.
- Моделирование предметной области коммерческой деятельности для фиксации бизнес-процессов, правил и последовательности операций.

На уровне логики может располагаться командный процессор, который получает команды от пользователя через уровень пользовательского интерфейса и совместно с уровнем доступа к данным решает отдельную задачу или выполняет конкретное задание. Это может быть полностью разработанная модель предметной области, предназначенная для преобразования бизнесс-процессов в программном обеспечении. С этой целью уровень доступа к данным нередко включает в себя компонент объектно-реляционного преобразования, чтобы реализовать уровень логики непосредственно в классах, возможно, путем предметно-ориентированного проектирования (DDD). В модели предметной области не должно быть зависимостей, пронизывающих ее сверху вниз или характерных для конкретной технологии реализации. Например, сборки в модели предметной области не должны иметь зависимостей от библиотеки объектно-реляционного преобразования. Вместо этого должна быть создана отдельная, характерная для данной реализации сборка, предписывающая порядок объектно-реляционного преобразования для модели предметной области. Благодаря этому классы модели предметной области могут быть использованы без зависимости от объектно-реляционного преобразования, которое может быть заменено без всяких последствий для модели предметной области или ее клиентов. На рис. 2.24 приведена одна из возможных реализаций уровня логики, на котором применяется модель предметной области.

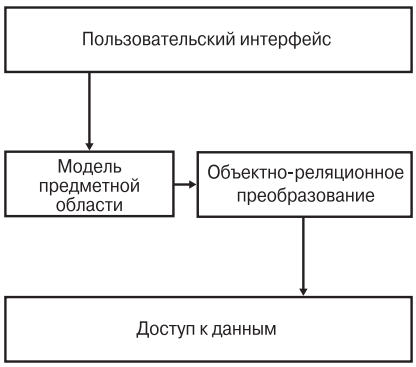

Рис. 2.24. *Образование уровня логики путем взаимодействия сборок в модели предметной области*

Уровень логики требуется ввести в том случае, если в приложении имеется сложная логика, например, бизнес-правила, призванные отражать реальные последовательности операций, входящих в обязанности отдельных работников. Даже если логика не особенно сложна, но часто изменяется, это обстоятельство служит веским аргументом в пользу внедрения отдельного уровня для инкапсуляции подобного поведения. Благодаря этому упрощаются уровни пользовательского интерфейса и доступа к данным, которые полностью сосредоточиваются только на своих целях.

Сквозные виды ответственности

Виды ответственности компонента нелегко ограничить одним уровнем. Такие функции, как ревизия, безопасность и кеширование, могут пронизывать все приложение, поскольку они применяются на *каждом* уровне. Например, отслеживание действий кода при каждом вызове и возврате из метода очень полезно для целей отладки, когда приложение развернуто, а отладчик Visual Studio нельзя подключить для пошаговой отладки кода. Вывод значений параметров можно организовать вручную по мере их передачи, а возвращаемых значений — при их возврате из различных методов, как демонстрируется в листинге 2.23.

Листинг 2.23. Применение сквозных видов ответственности вручную быстро заслоняет истинное назначение прикладного кода

```
public void OpenNewAccount(Guid ownerID, string accountName,
                           decimal openingBalance)
{
    log.WriteInfo("Creating new account for owner {0} with name
                  '{1}' and an opening balance of {2}",
                  ownerID, accountName, openingBalance");

    using(var transaction = session.BeginTransaction())
    {
        var user = userRepository.GetByID(ownerID);
        user.CreateAccount(accountName);
        var account = user.FindAccount(accountName);
        account.SetBalance(openingBalance);

        transaction.Commit();
    }
}
```

Это трудоемкий и чреватый ошибками процесс, немедленно засоряющий каждый метод неуместным шаблонным кодом. Вместо этого сквозные виды ответственности можно вынести в инкапсулированные функциональные возможности и применить их в прикладном коде намного менее агрессивным способом. Чаще всего функциональные возможности неагрессивно вводятся посредством аспектно-ориентированного программирования.

Аспекты

Аспектно-ориентированное программирование (АОП) — это применение сквозных видов ответственности (или аспектов) на многих уровнях прикладного кода.

На платформе .NET Framework имеется на выбор несколько библиотек АОП (например, NuGet), но в приведенных здесь примерах кода применяется библиотека PostSharp, имеющая бесплатную, хотя и функционально усеченную версию Express. В листинге 2.24 демонстрируется отслеживание прикладного кода, вынесенного в аспект PostSharp и применяемого в качестве атрибута в некоторых методах.

Листинг 2.24. Аспекты отлично подходят для реализации сквозных видов ответственности

```
[Logged]
[Transactional]
public void OpenNewAccount(Guid ownerID, string accountName,
                           decimal openingBalance)
{
    var user = userRepository.GetByID(ownerID);
    user.CreateAccount(accountName);
    var account = user.FindAccount(accountName);
    account.SetBalance(openingBalance);
}
```

Оба атрибута, декорирующих метод `OpenNewAccount()`, предоставляют те же самые функциональные возможности, что и в листинге 2.23, но в данном случае назначение метода яснее. В частности, атрибут `Logged` служит для записи сведений о вызове метода в журнал регистрации, включая значения параметров, тогда как атрибут `Transactional` заключает метод в оболочку транзакции базы данных, фиксируя транзакцию при удачном исходе или выполняя откат при неудачном исходе. Самое главное, что оба атрибута достаточно общие, чтобы применять их к *любому* методу, а следовательно, делать это неоднократно.

Асимметричное разделение на уровни

Все запросы приложения со стороны пользователя осуществляются через пользовательский интерфейс. Но путь, по которому после этого следует запрос, не всегда оказывается одинаковым. Разделение на уровни может быть асимметричным в зависимости от типа совершаемого запроса. Это обусловлено потребностью быть прагматичным и учитывать чрезмерность разделения на уровни для одних запросов и его недостаточность для других.

За последние годы широкое распространение нашел шаблон асимметричного разделения на уровни, называемый “Разделение ответственности команд и запросов” (Command/Query Responsibility Segregation — CQRS). Прежде чем обсуждать этот архитектурный шаблон, следует рассмотреть побудительные причины для разделения ответственности команд и запросов на уровне методов.

Разделение ответственности команд и запросов

В своей книге *Object-Oriented Software Construction* (издательство Prentice Hall, 1997 г.) Бертран Мейер (Bertrand Meyer) ввел принцип *разделения команд и запросов* (CQS), чтобы пояснить, что одновременно методы должны выполнять одно из двух: *команду* или *запрос*.

Команды являются императивными вызовами действия, требуя от метода *сделать* что-нибудь. Таким методам разрешается изменять состояние системы, но в то же время они должны возвращать значение. В листинге 2.25 сначала демонстрируется пример метода, способного выполнять команды по принципу CQS, а затем пример метода, неспособного на такое.

Листинг 2.25. Методы, способные и неспособные выполнять команды по принципу CQS

```
// Метод, способный выполнять команды по принципу CQS
Public void SaveUser(string name)
{
    session.Save(new User(name));
}
// Метод, неспособный выполнять команды по принципу CQS
public User SaveUser(string name)
{
    var user = new User(name);
    session.Save(user);
    return user;
}
```

Запросы служат требованиями к прикладному коду на *получение* данных. Методы, выполняющие запрос, возвращают данные вызывающим клиентам, но они должны также изменять состояние системы. В листинге 2.26 сначала демонстрируется пример метода запроса, действующего по принципу CQS, а затем пример метода запроса, не действующего по принципу CQS.

Листинг 2.26. Методы запроса, действующие и не действующие по принципу CQS

```
// Метод запроса, действующий по принципу CQS
Public IEnumerable<User> FindUserByID(Guid userID)
{
    return session.Get<User>(userID);
}
// Метод запроса, не действующий по принципу CQS
public IEnumerable<User> FindUserByID(Guid userID)
{
    var user = session.Get<User>(userID);
    user.LastAccessed = DateTime.Now;
    return user;
}
```

Таким образом, команды и запросы отличаются наличием возвращаемого значения. Если метод возвращает значение и действует по принципу CQS, вполне можно допустить, что он вообще не изменяет никакого состояния объекта. Преимущество такого подхода состоит в том, что вызовы методов запроса можно реорганизовать, зная, что они не оказывают никакого влияния на объект. Если метод не возвращает значение и действует по принципу CQS, это означает, что он действительно изменяет состояние объекта. Поэтому в отношении порядка таких вызовов следует проявлять бо́льшую осторожность.

Разделение ответственности команд и запросов

Создание шаблона “Разделение ответственности команд и запросов” (CQRS) приписывается Грег Янгу (Greg Young). Этот шаблон является приложением принципа разделения команд и запросов (CQS) на архитектурном уровне и служит примером асимметричного разделения на уровни. Команды и запросы следуют тем же правилам, что и по принципу CQS. Но кроме того, в шаблоне CQRS допускается, что команды и запросы могут быть направлены по разным путям посредством разделения на уровни, чтобы их можно было обработать наилучшим образом.

Простейшим примером применения шаблона CQRS может служить разработка трехуровневой архитектуры с моделью предметной области. В этом случае модель предметной области применяется лишь в командной части приложения, тогда как в запросной его части — еще более простая двухуровневая архитектура. Пример такой конструкции приведен на рис. 2.25.

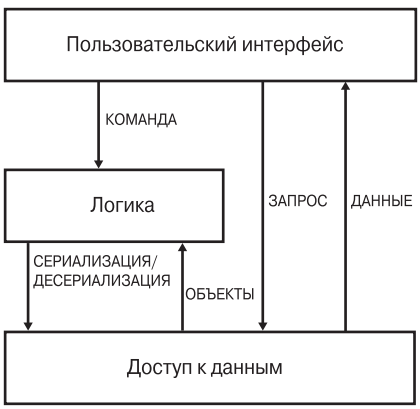

Рис. 2.25. *Модели предметной области следует применять только для обработки команд*

Запрашивание данных нередко должно осуществляться быстро и с минимальными гарантиями согласованности транзакций: фантомные или неповторяющиеся операции чтения могут стать приемлемым компромиссом повышению быстроты реагирования. Но для обработки команд нередко требуется согласованность транзакций, а следовательно, и разные уровни для команд и запросов. Иногда отличаться командами и запросами могут также уровни доступа к данным. Так, для обращения к базе данных, действующей по принципу ACID (атомарность, непротиворечивость, изоляция, долговечность), могут потребоваться команды, тогда как для обращения к простому хранилищу документов достаточно и запросов. Хранилище документов может обновляться асинхронно по событиям, наступающим на уровне команд, и благодаря этому достигается окончательная согласованность с операциями чтения по запросу.

Заключение

В этой главе было показано, что организация зависимостей вызывает серьезные трудности при создании приложений. Надежность в долгосрочной перспективе, приспосабливаемость, а возможно, и жизнеспособность проектов опирается на прочное основание управления зависимостями. Беспорядочность зависимостей возникает в том случае, если разработчики создают классы, ссылающиеся друг на друга без особых на то оснований. И это может стать серьезной помехой в обеспечении командой разработчиков коммерческой ценности программного продукта согласованным и предсказуемым образом.

Управление зависимостями должно быть организовано на всех уровнях: от отдельных классов и взаимодействующих методов до ссылок на сборки и компоненты архитектуры верхнего уровня. Разработчики должны постоянно следить за тем, чтобы ложные зависимости не просочились за пределы их методов, классов, сборок или уровней.

В какой-то степени материал этой главы закладывает прочное основание для остальной части данной книги, где постепенно раскрываются особенности написания сопровождаемого адаптивного кода. Если сборки имеют беспорядочные зависимости, а на уровнях не скрываются их архитектурные зависимости, то тестируемость и удобочитаемость прикладного кода затрудняется независимо от попыток следовать другим шаблонам, методикам и практическим приемам.