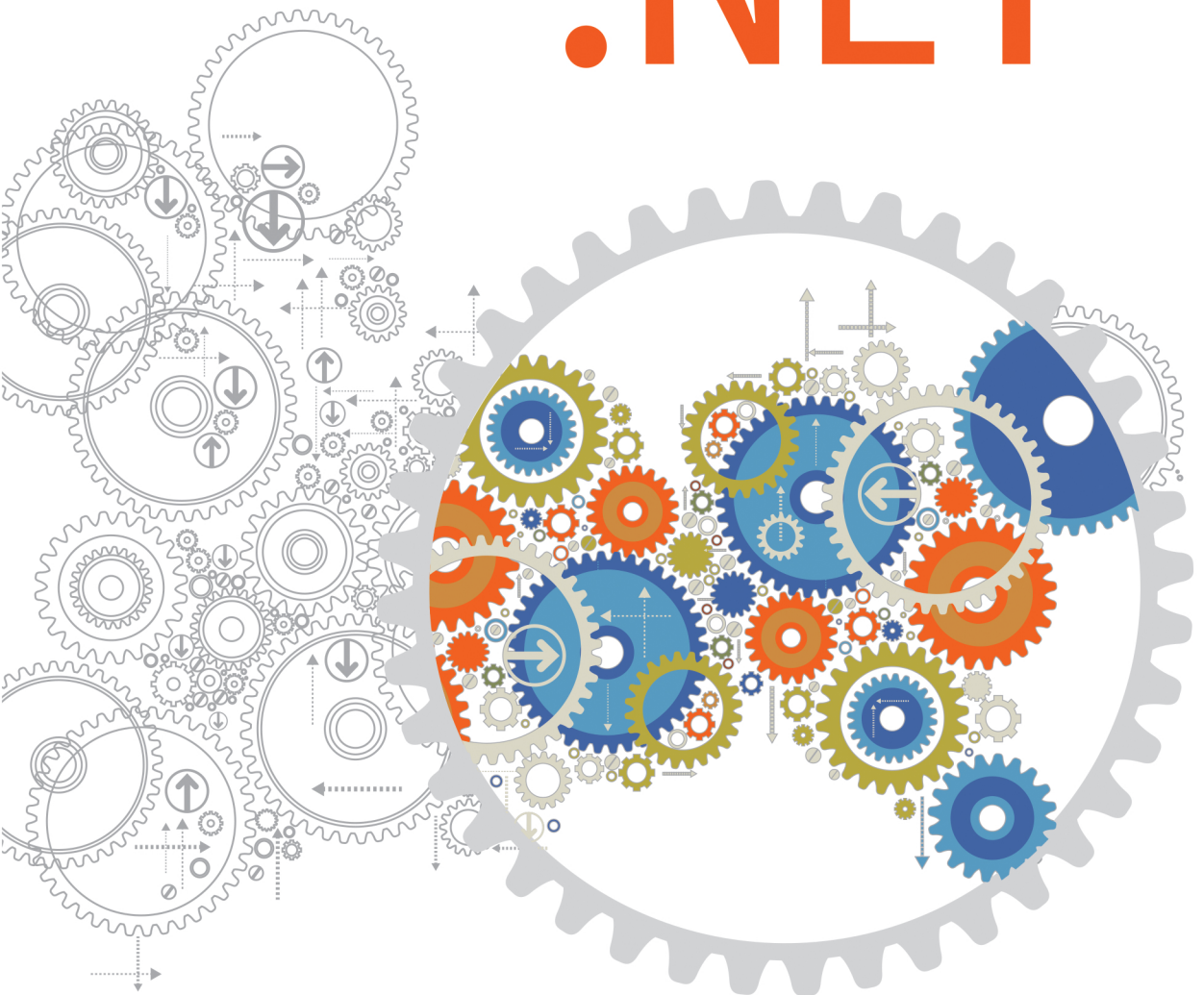
ПАТТЕРНЫ проектирования на платформе .NET



Сергей Тепляков

ПИТЕР®

ББК 32.973.2-018-02
УДК 004.42
Т34

Тепляков С.

Т34 Паттерны проектирования на платформе .NET. — СПб.: Питер, 2015. — 320 с.: ил.
ISBN 978-5-496-01649-0

Паттерны проектирования остаются важным инструментом в арсенале разработчика, поскольку они опираются на фундаментальные принципы проектирования. Тем не менее, появление новых конструкций в современных языках программирования делает одни паттерны более важными, а значимость других сводит к минимуму.

Цель данной книги — показать, как изменились паттерны проектирования за это время, как на них повлияло современное увлечение функциональным программированием, и объяснить, каким образом они используются в современных .NET-приложениях. В издании вы найдете подробное описание классических паттернов проектирования с особенностями их реализации на платформе .NET, а также примеры их использования в .NET Framework. Вы также изучите принципы проектирования, известные под аббревиатурой SOLID, и научитесь применять их при разработке собственных приложений.

Книга предназначена для профессиональных программистов, которые хотят изучить особенности классических принципов и паттернов программирования с примерами на языке C# и понять их роль в разработке современных приложений на платформе .NET.

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

ББК 32.973.2-018-02
УДК 004.42

ISBN 978-5-496-01649-0

Краткое содержание

Часть II. Порождающие паттерны

Часть III. Структурные паттерны

Часть IV. Принципы проектирования

Оглавление

Часть III. Структурные паттерны

Часть IV. Принципы проектирования

Об авторе

Сергей Тепляков занимается разработкой программного обеспечения более десяти лет. За это время он прошел путь от младшего разработчика встроенных систем до архитектора одной из ведущих аутсорсинговых компаний Европы, а потом перебрался в подразделение разработки (DevDiv) компании Microsoft.

Сергей — автор довольно популярного в Рунете блога, посвященного программированию, — Programming Stuff[1], в котором опубликовано несколько сотен статей самой разной тематики. С 2011 года Сергей был обладателем титула Microsoft C# MVP, которого он лишился при переходе в Microsoft в конце 2014 года.

Основной интерес автора лежит в области проектирования систем, прагматичного использования принципов и паттернов проектирования, а также совмещения объектно-ориентированного и функционального программирования.

Связаться с Сергеем можно по электронной почте Sergey.Teplyakov@gmail.com.

---

[1] SergeyTeplyakov.blogspot.ru.

Кому адресована эта книга

Книга предназначена профессиональным разработчикам, которым интересны вопросы проектирования. Наибольшую пользу книга принесет программистам, у которых за плечами несколько лет опыта работы с языком C#, базовые знания об объектно-ориентированном программировании и о паттернах проектирования.

Менее опытным разработчикам я бы посоветовал прочитать эту книгу дважды: сейчас и через несколько лет, когда взгляд на разработку изменится под влиянием нового опыта. Процесс познания в целом и изучение вопросов проектирования в частности является итеративным. Практический опыт и набитые шишки позволяют посмотреть на такие «теоретические» и, казалось бы, малоинтересные вопросы, как паттерны, с совершенно иной точки зрения.

Опытным разработчикам книга будет полезна в качестве средства обобщения знаний. Классические паттерны здесь рассматриваются со всех возможных точек зрения, так что, вполне возможно, автору удалось показать их с тех сторон, о которых вы не задумывались.

В качестве целевой платформы используются .NET Framework и язык C#. Часть материалов довольно сильно завязана на особенности платформы и языка программирования. Если вашим основным языком программирования является C++ или Java, то книга все равно будет полезна, поскольку существенная ее часть посвящена вопросам проектирования, слабо зависящим от языка программирования.

Как читать эту книгу

Данная книга не является учебником, который предполагает четкую последовательность чтения. Все главы о паттернах проектирования являются автономными, их можно читать независимо друг от друга. Последняя часть книги посвящена принципам проектирования и незначительно пересекается с другими ее частями.

Книга может быть использована в качестве справочника для углубления знаний об определенном принципе или паттерне. Я бы все же советовал вначале прочитать книгу от начала до конца, чтобы сформировать общее представление или освежить знания в области проектирования. Это позволит возвратиться к конкретной главе позже, когда станет очевидной необходимость углубленных знаний по конкретному паттерну для более успешного его применения на практике.

Отзывы

Ни одна книга не является идеальной, и эта, по всей видимости, не станет исключением. В книге могут быть неточности, которые пропустили я и команда редакторов. А может быть, чтение определенной главы вызовет у вас определенные мысли, которыми вы захотите поделиться с автором этой книги.

Другими словами, если вам что-то понравилось, не понравилось или вас просто заинтересовал определенный аспект проектирования, пишите об этом на адрес электронной почты Sergey.Teplyakov@gmail.com.

Благодарности

Написание книги — это сложный и трудоемкий процесс, который отнял у меня, а значит и у моей семьи, большое количество времени и сил. Я не знаю, что бы я делал без своих любимых девочек — супруги Юли и дочки Анжелики. Без них у меня просто не было бы желания становиться лучше и заниматься подобными вещами. Хочется сказать спасибо родителям и брату, которые всегда верили в меня и поддерживали.

У меня никогда не было живого ментора, но были люди, которые оказали влияние на формирование моих взглядов в области проектирования. Они не знают о моем существовании, но без них я бы стал совсем другим специалистом. Это Гради Буч и Бертран Мейер, Кент Бек и Мартин Фаулер, Крэг Ларман и Эрик Эванс, ну и, конечно же, «банда четырех» — Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес.

За пять лет в киевском офисе компании «Люксофт» я познакомился с огромным количеством интереснейших специалистов. Мы обсуждали вопросы проектирования, спорили об особенностях объектно-ориентированного и функционального программирования, доказывали превосходство одного языка над другим, а иногда обсуждали научные подходы к наиболее эффективному остужению манной каши. Тут не будет имен, вы уж извините. Просто знайте, что я пока не встречал более сильного и интересного коллектива!

Особую благодарность хочется выразить команде рецензентов, чья конструктивная критика помогла сделать книгу лучше. Это Сергей Усок, Леша Давидич, Вячеслав Иванов, Саша Шер, Женя Чепурных, Саша Березовский, Саша Гриценко и Миша Барабаш. Ребята, огромное спасибо!

От издательства

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

Мы будем рады узнать ваше мнение!

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

Предисловие

GoF-паттерны на платформе .NET

У нашей индустрии есть одна интересная черта: когда появляется новый инструмент или технология, часто ими настолько увлекаются, что начинают забывать про старые проверенные вещи. Возможно, это связано с желанием найти в конце концов серебряную пулю и надеждой на то, что благодаря новинке хорошая система родится сама собой, а не как обычно — благодаря затраченным усилиям и опыту команды разработчиков.

Когда в начале 1990-х на арене разработки программного обеспечения появились паттерны проектирования, многие стали мечтать о том, что с их помощью даже бизнес-пользователи и 1С-программисты смогут собирать приложение из готовых «кирпичиков». Довольно скоро стало понятно, что планы были чересчур оптимистичными, и разработчики начали искать другие подходы. Так появилось «программирование через конфигурацию», пламенно воспетое Хантом и Томасом в их «Программисте-прагматике»[1].

Затем появились IoC- или DI-контейнеры (IoC — Inversion of Control (инверсия управления), DI — Dependency Injection (внедрение зависимости)) и начался новый этап создания слабосвязанных приложений. У многих возникла привычка выделять интерфейсы[2] не задумываясь, а количество зависимостей у класса

---

[1] *Хант Э.* Программист-прагматик. Путь от подмастерья к мастеру // Э. Хант, Д. Томас. — СПб.: Питер, 2007. — 288 с.

[2] Речь идет об интерфейсах в таких языках программирования, как C# или Java.

начало переваливать за 5–6. В результате разбираться в приложении стало еще сложнее, поскольку прямые связи между классами начали заменяться косвенными.

Поиск идеального инструмента, языка, принципа или методологии разработки — это святой Грааль в разработке ПО. Все хотят найти идеальный инструмент, позволяющий справиться со сложностью современных систем и навести порядок в том хаосе, который творится в мире программирования. Но, может быть, вместо того, чтобы каждый раз хвататься за что-то новое как за спасительную соломинку, стоит понять, что за этой соломинкой скрыто? Ведь если присмотреться, то новый инструмент очень часто оказывается лишь новой оберткой, в которую завернуты старые идеи.

Отношение к паттернам проектирования

Большинство разработчиков ПО склоняются к мысли, что паттерны проектирования — вещь интересная, но далеко не всегда полезная. Почему так вышло? Когда молодой разработчик сталкивается с новым инструментом, он изо всех сил старается воспользоваться им по максимуму. В результате инструмент проходит определенные стадии развития, которые для паттернов проектирования выглядят так.

1. Ух ты, ух ты, ух ты! Я узнал, что такое паттерны! Класс! Когда и где я смогу ими воспользоваться?
2. Ух ты! Я отрефакторил[1] старый код и вместо десяти строк кода воспользовался семью паттернами! Вот как здорово!
3. Ух ты. Ну, паттерны — это классная штука, но через пару месяцев сделанный мной рефакторинг не кажется таким уж полезным. Что-то я и сам начал путаться со всеми этими абстрактными фасадированными декораторами, завернутыми в синглтон.
4. Паттерны — это хорошо, но нужно отталкиваться не от них, а от решаемой задачи и, уже исходя из проблемы, выбирать подходящие решения. Паттерны — хорошо, но своя голова лучше!

---

[1] Речь идет о рефакторинге кода — изменении структуры программы без изменения ее поведения. Подробнее об этом можно почитать в Сети или в книге Мартина Фаулера «Рефакторинг. Улучшение существующего кода».

Есть разработчики, которые успешно прошли все четыре стадии и достигли «просветления», набивая шишки в разных проектах и на себе оценивая последствия использования тех или иных паттернов. Но ведь есть и те, кто пришел на проект, в котором царствовала вторая стадия использования паттернов, и увидел решения простых задач невероятно изощренным способом.

Каким будет ваше отношение к паттернам при виде классов вроде `AbstractSingletonProxyFactoryBean`[1] и приложений «Hello, World», таких как в листинге П.1?

Листинг П.1. Чрезмерное использование паттернов проектирования

```
public class HelloWorld
{
  public static void Main(String[] args)
  {
    MessageBody mb = new MessageBody();
    mb.Configure("Hello World!");
    AbstractStrategyFactory asf = DefaultFactory.Instance;
    MessageStrategy strategy = asf.CreateStrategy(mb);
    mb.Send(strategy);
  }
}
```

Подливает масла в огонь наше чересчур наивное отношение к паттернам проектирования, которое отлично описал Джошуа Кериевски в своей книге «Рефакторинг с использованием шаблонов»: *«К сожалению, когда программисты смотрят на единственную диаграмму, сопровождающую каждый шаблон в книге Design Patterns*[2]*, они часто приходят к выводу, что приведенная диаграмма и есть способ реализации шаблона. Они бы гораздо лучше разобрались в ситуации, если бы внимательно прочитали самое интересное — примечание к реализации. Многие*

---

[1] Пример кода взят из обсуждения паттернов проектирования на shashdot: http://developers.slashdot.org/comments.pl?sid=33602&cid=3636102.

[2] Речь идет о самой знаменитой книге по паттернам проектирования «Приемы объектно-ориентированного проектирования. Паттерны проектирования», написанной «бандой четырех» — Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом.

программисты берут в руки книгу Design Patterns, всматриваются в структурную диаграмму шаблона и начинают кодировать. Полученный код в точности отражает диаграмму, а не реализацию шаблона, наиболее полно соответствующую решаемой задаче».

Большинство экспертов в области разработки ПО склоняются к мысли, что главные инструменты разработчика — его голова и собственный опыт. Если в умной книге нарисована диаграмма классов, это не значит, что нужно бросать все и пробовать пристроить ее в свой проект. Чтобы использовать паттерн проектирования по максимуму, нужно вникнуть в его суть, понять, какую проблему он призван решить и каким образом он это делает. Понимание целей паттерна и контекста его использования позволит варьировать реализацию паттерна и лучше адаптировать его под свои нужды.

Фреймворки паттернов

Описанные четыре стадии изучения паттернов характерны для молодых специалистов, но не меньшие проблемы ждут команду, если о паттернах внезапно узнает ее авторитетный член[1]. Что делает опытный специалист, когда знакомится с новыми принципами проектирования? Правильно, он старается их обобщить и поделиться новым опытом с остальными бойцами. В результате появляются библиотеки, или фреймворки, паттернов проектирования.

Я не говорю, что это абсолютно неверная идея, но в большинстве случаев такой подход противоречит самому понятию паттернов проектирования. Есть редкие исключения, такие как библиотека Loki[2] Андрея Александреску, которая представляет собой набор базовых решений для упрощения реализации основных паттернов проектирования в конкретном языке программирования. Но чаще всего выгода от повторного использования таких библиотек будет очень мала, качество последних будет невысоким, а решения, полученные на их основе, окажутся чрезмерно сложными.

---

[1] Да, сегодня такая ситуация может показаться маловероятной, но она возможна. Многие матерые гуру набили шишки, когда решали практические задачи, не особенно задумываясь о теории. И если такой авторитет решит воспользоваться паттернами, то ждите их во всех будущих проектах.

[2] *Александреску А.* Современное проектирование на C++. — М.: Вильямс, 2004. — 336 с. Библиотеку loki можно найти здесь: loki-lib.sourceforge.net.

Гибкость vs. конкретность

У большинства паттернов проектирования есть каноническая реализация, а есть упрощенные/усложненные вариации. Как вы, наверное, знаете, у гибкости есть своя цена, которая может быть оправданна в одном случае и не иметь смысла в другом. Хотя при описании паттернов проектирования дается контекст, в котором его применение будет наиболее актуальным, лишь разработчик конкретного приложения может сказать, где проходит грань между сложностью и гибкостью, подходящая для конкретного случая.

Большинство паттернов проектирования предназначены для получения расширяемости системы в определенной плоскости. Причем эта плоскость может быть полезной для одного приложения и вредной — для другого. Наличие иерархии наследования может добавлять сложности простому приложению, но в случае библиотеки нередко делает решение чересчур сложным.

Наследование — чрезвычайно полезный инструмент для расширения функционала, но оно приводит к сильной связи между базовым классом и наследниками. Для одних паттернов наследование является неотъемлемой частью реализации, для других — обузой, которая сделает решение громоздким. Канонические примеры большинства паттернов, приведенные «бандой четырех», влючают в себя наследование. Но это не значит, что вы должны слепо использовать его.

Наследование должно применяться осознанно и лишь тогда, когда обеспечиваемая им гибкость действительно необходима.

Для чего нужна еще одна книга о паттернах

Паттерны не привязаны к платформе, но их типовая реализация несколько различается от языка к языку иногда из-за технических различий, иногда — из-за культурных. Я не хочу здесь поднимать вопрос о полезности паттернов проектирования. Это все равно, что поднимать вопрос о пользе исключений или многопоточности в .NET-приложениях: хотите вы того или нет, но вам без них не обойтись. Так и с паттернами. Код любого .NET-приложения просто пропитан паттернами в явном или неявном виде, и игнорирование этого факта вряд ли принесет вам пользу.

В этой книге я хочу вернуться к стандартным паттернам проектирования, посмотреть, что с ними случилось за последние 20 лет, и показать, в каком виде они применяются в современных .NET-приложениях и самом .NET Framework.

Я собираюсь рассмотреть использование паттернов проектирования на примере простого приложения импорта лог-файлов для последующего полнотекстового поиска (full text search). Подавляющее число паттернов естественным образом укладывается на эту задачу, и они в том или ином виде были использованы в полноценной версии приложения под названием Application Insights, над которым я сейчас работаю в Microsoft.

Часть I

Паттерны поведения

Глава 1

Паттерн «Стратегия» (Strategy)

Назначение: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Другими словами: стратегия инкапсулирует определенное поведение с возможностью его подмены.

Мотивация

Паттерн «Стратегия» является настолько распространенным и общепринятым, что многие его используют постоянно, даже не задумываясь о том, что это хитроумный паттерн проектирования, расписанный когда-то «бандой четырех».

Каждый второй раз, когда мы пользуемся наследованием, мы используем стратегию; каждый раз, когда абстрагируемся от некоторого процесса, поведения или алгоритма за базовым классом или интерфейсом, мы используем стратегию. Сортировка, анализ данных, валидация, разбор данных, сериализация, кодирование/декодирование, получение конфигурации — все эти концепции могут и должны быть выражены в виде стратегий или политик (policy).

Стратегия является фундаментальным паттерном, поскольку она проявляется в большинстве других классических паттернов проектирования, которые поддер-

живают специализацию за счет наследования. Абстрактная фабрика — это стратегия создания семейства объектов; фабричный метод — стратегия создания одного объекта; строитель — стратегия построения объекта; итератор — стратегия перебора элементов и т. д.[1]

Давайте в качестве примера рассмотрим задачу импорта лог-файлов для последующего полнотекстового поиска. Главной задачей данного приложения является чтение лог-файлов из различных источников (рис. 1.1), приведение их к некоторому каноническому виду и сохранение в каком-то хранилище, например Elasticsearch или SQL Server.

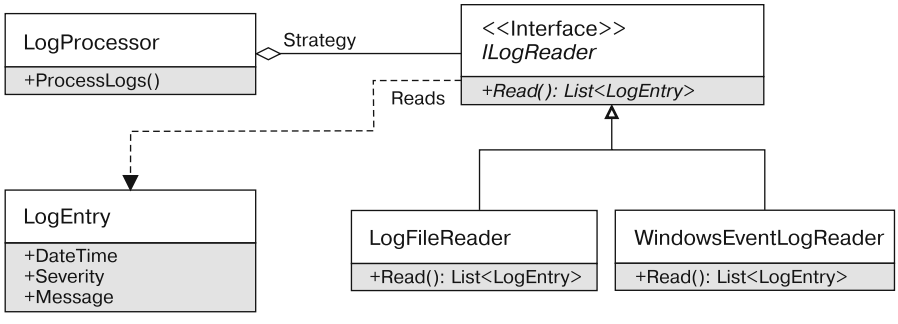


Рис. 1.1. Диаграмма классов импорта логов

`LogProcessor` отвечает за импорт лог-файлов и должен работать с любой разновидностью логов: файлами (`LogFileReader`), логами Windows (`WindowsEventLogReader`) и т. д. Для этого процесс чтения логов выделяется в виде интерфейса или базового класса `ILogReader`, а класс `LogProcessor` знает лишь о нем и не зависит от конкретной реализации.

Мотивация использования паттерна «Стратегия»: выделение поведения или алгоритма с возможностью его замены во время исполнения.

Классическая диаграмма классов паттерна «Стратегия» приведена на рис. 1.2.

Участники:

- `Strategy (ILogReader)` — определяет интерфейс алгоритма;
- `Context (LogProcessor)` — является клиентом стратегии;
- `ConcreteStrategyA`, `ConcreteStrategyB` (`LogFileReader`, `WindowsEventLogReader`) — являются конкретными реализациями стратегии.

[1] Подробнее все эти паттерны проектирования будут рассмотрены в последующих главах.

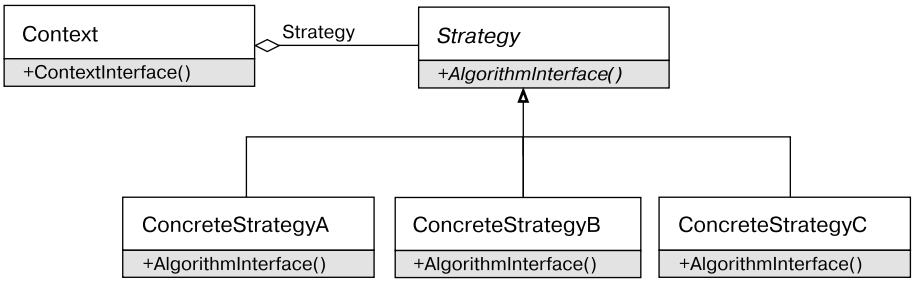


Рис. 1.2. Диаграмма классов паттерна Стратегия

Обратите внимание на то, что классический паттерн «Стратегия» весьма абстрактен.

- Паттерн «Стратегия» не определяет, как стратегия получит данные, необходимые для выполнения своей работы. Они могут передаваться в аргументах метода `AlgorithmInterface`, или стратегия может получать ссылку на сам контекст и получать требуемые данные самостоятельно.
- Паттерн «Стратегия» не определяет, каким образом контекст получает экземпляр стратегии. Контекст может получать ее в аргументах конструктора, через метод, свойство или у третьей стороны.

Варианты реализации в .NET

В общем случае паттерн «Стратегия» не определяет, какое количество операций будет у «выделенного поведения или алгоритма». Это может быть одна операция (метод `Sort` интерфейса `ISortable`) или семейство операций (`Encode/Decode` интерфейса `IMessageProcessor`).

При этом если операция лишь одна, то вместо выделения и передачи интерфейса в современных .NET-приложениях очень часто используются делегаты. Так, в нашем случае вместо передачи интерфейса `ILogReader` класс `LogProcessor` мог бы принимать делегат вида `Func<List<LogEntry>>`, который соответствует сигнатуре единственного метода стратегии (листинг 1.1).

Листинг 1.1. Класс LogProcessor

```
class LogProcessor
{
    private readonly Func<List<LogEntry>> _logImporter;
    public LogProcessor(Func<List<LogEntry>> logImporter)
    {
```

```
        _logImporter = logImporter;
    }

    public void ProcessLogs()
    {
        foreach(var logEntry in _logImporter.Invoke())
        {
            SaveLogEntry(logEntry);
        }
    }
    // Остальные методы пропущены...
}
```

Использование функциональных стратегий является единственной платформенно-зависимой особенностью паттерна «Стратегия» на платформе .NET. Да и то эта особенность обусловлена не столько самой платформой, сколько возрастающей популярностью техник функционального программирования.

ПРИМЕЧАНИЕ

В некоторых командах возникают попытки обобщить паттерны проектирования и использовать их повторно в виде библиотечного кода. В результате появляются интерфейсы IStrategy и IContext, вокруг которых строится решение в коде приложения. Есть лишь несколько паттернов проектирования, повторное использование которых возможно на уровне библиотеки: «Синглтон», «Наблюдатель», «Команда». В общем же случае попытка обобщить паттерны приводит к переусложненным решениям и показывает непонимание фундаментальных принципов паттернов проектирования.

Обсуждение паттерна «Стратегия»

По определению применение стратегии обусловлено двумя причинами:

- необходимостью инкапсуляции поведения или алгоритма;
- необходимостью замены поведения или алгоритма во время исполнения.

Любой нормально спроектированный класс уже инкапсулирует в себе поведение или алгоритм, но не любой класс с некоторым поведением является или должен быть стратегией. Стратегия нужна тогда, когда не просто требуется спрятать алгоритм, а важно иметь возможность заменить его во время исполнения!

Другими словами, стратегия обеспечивает точку расширения системы в определенной плоскости: класс-контекст принимает экземпляр стратегии и не знает, какой вариант стратегии он собирается использовать.

Выделять интерфейс или нет

ПРИМЕЧАНИЕ

Выделение интерфейсов является довольно острым вопросом в современной разработке ПО и актуально не только в контексте стратегий. Поэтому все последующие размышления применимы к любым иерархиям наследования.

Сейчас существует два противоположных лагеря в мире объектно-ориентированного программирования: ярые сторонники и ярые противники выделения интерфейсов. Когда возникает вопрос о необходимости выделения интерфейса и добавления наследования, мне нравится думать об этом как о необходимости выделения стратегии. Это не всегда точно, но может быть хорошей лакмусовой бумажкой.

Нужно ли выделять интерфейс `IValidator` для проверки корректности ввода пользователя? Нужны ли нам интерфейсы `IFactory` или `IAbstractFactory`, или подойдет один конкретный класс? Ответы на эти вопросы зависят от того, нужна ли нам стратегия (или политика) валидации или создания объектов. Хотим ли мы заменять эту стратегию во время исполнения или можем использовать конкретную реализацию и внести в нее изменение в случае необходимости?

У выделения интерфейса и передачи его в качестве зависимости есть еще несколько особенностей.

Передача интерфейса `ILogReader` классу `LogProcessor` увеличивает гибкость, но в то же время повышает сложность. Теперь клиентам класса `LogProcessor` нужно решить, какую реализацию использовать, или переложить эту ответственность на вызывающий код.

Важно понимать, нужен ли дополнительный уровень абстракции именно сейчас. Может быть, на текущем этапе достаточно использовать напрямую класс `LogFile-Importer`, а выделить стратегию импорта тогда, когда в этом действительно появится необходимость.

Интерфейс vs. делегат

Поскольку некоторые стратегии содержат лишь один метод, очень часто вместо классической стратегии на основе наследования можно использовать стратегию на

основе делегатов. Иногда эти подходы совмещаются, что позволяет использовать наиболее удобный вариант.

Классическим примером такой ситуации является стратегия сортировки, представленная интерфейсами `IComparable<T>` и делегатом `Comparison<T>` (листинг 1.2).

Листинг 1.2. Примеры стратегий сортировки

```
class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public override string ToString()
    {
      returnstring.Format("Id = {0}, Name = {1}", Id, Name);
    }
}

class EmployeeByIdComparer : IComparer<Employee>
{
    public int Compare(Employee x, Employee y)
    {
      return x.Id.CompareTo(y.Id);
    }
}

public static void SortLists()
{
    var list = new List<Employee>();

    // Используем "функтор"
    list.Sort(new EmployeeByIdComparer());

    // Используем делегат
    list.Sort((x, y) => x.Name.CompareTo(y.Name));
}
```

По сравнению с использованием лямбда-выражений реализация интерфейса требует больше кода и приводит к переключению контекста при чтении. При использовании метода `List.Sort` у нас есть оба варианта, но в некоторых случаях классы могут принимать лишь стратегию на основе интерфейса и не принимать стратегию на основе делегатов, как в случае с классами `SortedList` или `SortedSet` (листинг 1.3).

Листинг 1.3. Конструирование объекта SortedSet

```
var comparer = new EmployeeByIdComparer();

// Конструктор принимает IComparable
var set = new SortedSet<Employee>(comparer);

// Нет конструктора, принимающего делегат Comparison<T>
```

В этом случае можно создать небольшой адаптерный фабричный класс, который будет принимать делегат `Comparison<T>` и возвращать интерфейс `IComparable<T>` (листинг 1.4).

Листинг 1.4. Фабричный класс для создания экземпляров IComparer

```
class ComparerFactory
{
    public static IComparer<T> Create<T>(Comparison<T> comparer)
    {
        return new DelegateComparer<T>(comparer);
    }
    private class DelegateComparer<T> : IComparer<T>
    {
        private readonly Comparison<T> _comparer;

        public DelegateComparer(Comparison<T> comparer)
        {
            _comparer = comparer;
        }
```

```
        public int Compare(T x, T y)
        {
            return _comparer(x, y);
        }
    }
}
```

Теперь можно использовать этот фабричный класс следующим образом (листинг 1.5).

Листинг 1.5. Пример использования класса ComparerFactory

```
var comparer = ComparerFactory.Create<Employee>(
    (x, y) => x.Id.CompareTo(x.Id));
var set = new SortedSet<Employee>(comparer);
```

ПРИМЕЧАНИЕ

Можно пойти еще дальше и вместо метода императивного подхода на основе делегата Comparison‹T› получить более декларативное решение, аналогичное тому, что используется в методе Enumerable.OrderyBy: на основе селектора свойств для сравнения.

Применимость

Применимость стратегии полностью определяется ее назначением: паттерн «Стратегия» нужно использовать для моделирования семейства алгоритмов и операций, когда есть необходимость замены одного поведения другим во время исполнения.

Не следует использовать стратегию на всякий случай. Наследование добавляет гибкости, однако увеличивает сложность. Любой класс уже отделяет своих клиентов от деталей реализации и позволяет изменять эти детали, не затрагивая клиентов. Наличие полиморфизма усложняет чтение кода, а «дырявые абстракции» и нарушения принципа замещения Лисков[1] существенно усложняют поддержку и сопровождение такого кода.

Гибкость не бывает бесплатной, поэтому выделять стратегии стоит тогда, когда действительно нужна замена поведения во время исполнения.

[1] Принцип замещения Лисков будет рассмотрен в части IV книги.

Примеры в .NET Framework

Стратегия является невероятно распространенным паттерном в .NET Framework.

- LINQ (Language Integrated Query) — это набор методов расширения, принимающих стратегии фильтрации, получения проекции и т. д. Коллекции принимают стратегии сравнения элементов, а значит, любой класс, который принимает `IComparer<T>` или `IEqualityComparer<T>`, использует паттерн «Стратегия».
- WCF просто переполнен стратегиями: `IErrorHandler` — стратегия обработки коммуникационных ошибок; `IChannelInitializer` — стратегия инициализации канала; `IDispatchMessageFormatter` — стратегия форматирования сообщений; `MessageFilter` — стратегия фильтрации сообщений и т. д. Обилие стратегий наблюдается также в Windows Forms, WPF, ASP.NET и других фреймворках.

Любая библиотека просто переполнена стратегиями, поскольку они представляют собой универсальный механизм расширения требуемого пользователем функционала и адаптации поведения под не известные заранее требования.

Глава 2

Паттерн «Шаблонный метод» (Template Method)

Назначение: шаблонный метод определяет основу алгоритма и позволяет подклассам переопределять некоторые шаги алгоритма, не изменяя его структуры в целом.

Другими словами: шаблонный метод — это каркас, в который наследники могут подставить реализации недостающих элементов.

Мотивация

На заре становления объектно-ориентированного программирования (ООП) наследование считалось ключевым механизмом для расширения и повторного использования кода. Однако со временем многим разработчикам стало очевидно, что наследование — не такой уж простой инструмент, использование которого создает сильную связанность (tight coupling) между базовым классом и его наследником. Эта связанность приводит к сложности понимания иерархии классов, а отсутствие формализации отношений между базовым классом и наследниками не позволяет четко понять, как именно разделены обязанности между классами `Base` и `Derived`, что можно делать наследнику, а что — нет.

Наследование подтипов подразумевает возможность подмены объектов базового класса объектами классов наследников. Такое наследование моделирует отношение «ЯВЛЯЕТСЯ», и поведение наследника должно соответствовать принципу наименьшего удивления: все, что корректно работало с базовыми классами, должно работать и с наследниками.

Более формальные отношения между родителями и потомками описываются с помощью предусловий и постусловий. Эта техника лежит в основе принципа подстановки Лисков, который мы рассмотрим в одной из глав книги. Другой способ заключается в использовании паттерна «Шаблонный метод», который позволяет более четко определить «контракт» между базовым классом и потомками.

Давайте вернемся к теме разработки приложения для импорта логов для полнотекстового поиска. Процесс чтения лога состоит из нескольких этапов.

1. Прочитать новые записи с места последнего чтения.
2. Разобрать их и вернуть вызывающему коду.

Можно в каждой реализации продублировать эту логику или же описать основные шаги алгоритма в базовом классе и отложить реализацию конкретных шагов до момента реализации наследников (листинг 2.1).

Листинг 2.1. Класс LogReader

```
public abstract class LogReader
{
    private int _currentPosition;

    // Метод ReadLogEntry невиртуальный: определяет алгоритм импорта
    public IEnumerable<LogEntry> ReadLogEntry()
    {
        return ReadEntries(ref _currentPosition).Select(ParseLogEntry);
    }

    protected abstract IEnumerable<string> ReadEntries(ref int position);

    protected abstract LogEntry ParseLogEntry(string stringEntry);
}
```

Теперь все реализации читателей логов будут вынуждены следовать согласованному протоколу. Классу `LogFileReader` достаточно будет реализовать методы

`ReadEntries` и `ParseLogEntry` и не думать о порядке их вызова или о необходимости вызова базовой реализации виртуального метода.

Шаблонный метод позволяет создать небольшой каркас (framework) для решения определенной задачи, когда базовый класс описывает основные шаги решения, заставляя наследников предоставить недостающие куски головоломки.

Классическая диаграмма классов паттерна «Шаблонный метод» приведена на рис. 2.1.

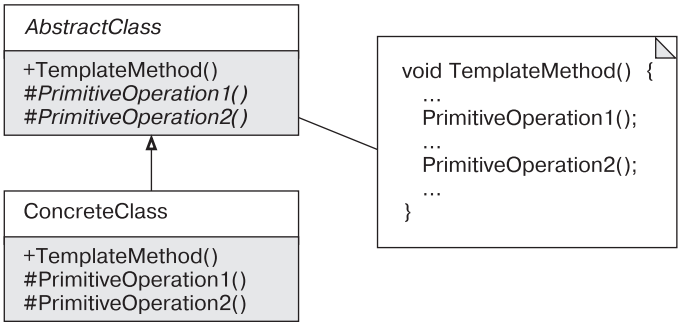


Рис. 2.1. Диаграмма классов паттерна «Шаблонный метод»

Участники:

- `AbstractClass (LogReader)` — определяет невиртуальный метод `TemplateMethod (ReadLogEntry)`, который вызывает внутри примитивные операции `PrimitiveOperation1()`, `PrimitiveOperation2()` и т. д. (`ReadEntries` и `ParseLogEntry`);
- `ConcreteClass (LogFileReader)` — реализует примитивные шаги алгоритма.

Шаблонный метод — это один из классических паттернов, который и по сей день используется в каноническом виде во многих приложениях.

Варианты реализации в .NET

Локальный шаблонный метод на основе делегатов

Классический вариант паттерна «Шаблонный метод» подразумевает, что каркас алгоритма описывается в базовом классе, а переменные шаги алгоритма задаются

наследниками путем переопределения абстрактных или виртуальных методов. Но в некоторых случаях схожие операции с единым каркасом исполнения и переменными составляющими бывают в рамках одного класса. Использование наследования является слишком тяжеловесным решением, поэтому в таких случаях применяется подход, при котором переменный шаг алгоритма задается делегатом.

Данный подход устраняет дублирование кода и довольно часто применяется в современных .NET-приложениях. Шаблонный метод на основе делегатов постоянно используется при работе с WCF-сервисами, поскольку протокол работы с прокси-объектами довольно сложен и отличается лишь конкретным методом сервиса (листинг 2.2).

Листинг 2.2. Использование шаблонного метода с WCF

```
// Интерфейс сервиса сохранения записей
interface ILogSaver
{
    void UploadLogEntries(IEnumerable<LogEntry> logEntries);
    void UploadExceptions(IEnumerable<ExceptionLogEntry> exceptions);
}

// Прокси-класс инкапсулирует особенности работы
// с WCF-инфраструктурой
class LogSaverProxy : ILogSaver
{
    class LogSaverClient : ClientBase<ILogSaver>
    {
        public ILogSaver LogSaver
        {
            get { return Channel; }
        }
    }

    public void UploadLogEntries(IEnumerable<LogEntry> logEntries)
    {
        UseProxyClient(c => c.UploadLogEntries(logEntries));
    }
```

```
    public void UploadExceptions(
        IEnumerable<ExceptionLogEntry> exceptions)
    {
        UseProxyClient(c => c.UploadExceptions(exceptions));
    }

    private void UseProxyClient(Action<ILogSaver> accessor)
    {
        var client = new LogSaverClient();

        try
        {
            accessor(client.LogSaver);
            client.Close();
        }
        catch (CommunicationException e)
        {
            client.Abort();
            throw new OperationFailedException(e);
        }
    }
}
```

Теперь для добавления нового метода сервиса достаточно добавить лишь одну строку кода, при этом протокол работы с сервисом будет соблюден.

ПРИМЕЧАНИЕ

Подробнее о том, почему с WCF-прокси нужно работать именно таким образом, можно прочитать в разделе Closing the proxy and using statement книги Джувала Лови Programming WCF Services.

Подход на основе делегатов может не только применяться для определения локальных действий внутри класса, но и передаваться извне другому объекту в аргументах конструктора. В этом случае грань между шаблонным методом и стратегией стирается практически полностью, разница остается лишь на логическом уровне: стратегия, даже представленная в виде делегата, чаще всего

подразумевает законченное действие, в то время как переменный шаг шаблонного метода обычно является более контекстно зависимой операцией[1].

Шаблонный метод на основе методов расширения

В языке C# существует возможность добавления операции существующим типам с помощью методов расширения (Extension Methods). Обычно методы расширения используются для расширения кода, который находится вне нашего контроля: библиотечных классов, перечислений, классов сторонних производителей. Но эту же возможность можно использовать и для своего собственного кода, что позволит выделить функциональность во вспомогательные классы, разгружая при этом основные.

Для удобства диагностики было бы разумно иметь возможность получения строкового представления прочитанных записей. Прочитанные записи сами по себе формируют небольшую иерархию наследования (рис. 2.2).

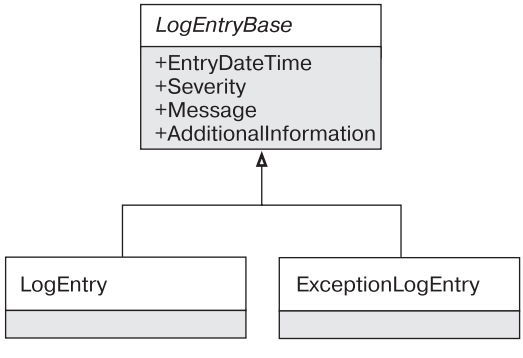


Рис. 2.2. Иерархия классов LogEntry

Мы могли бы переопределить метод `ToString()` или же вынести эту ответственность в методы расширения (листинг 2.3).

Листинг 2.3. Шаблонный метод на основе методов расширения

```
public abstract class LogEntryBase
{
    public DateTime EntryDateTime { get; internal set; }
```

[1] Более подробно о связи шаблонного метода и стратегии поговорим в соответствующем разделе данной главы.

```
    public Severity Severity { get; internal set; }
    public string Message { get; internal set; }

    // ExceptionLogEntry будет возвращать информацию об исключении
    public string AdditionalInformation { get; internal set; }
}

public static class LogEntryBaseEx
{
    public static string GetText(this LogEntryBase logEntry)
    {
        var sb = new StringBuilder();

        sb.AppendFormat("[{0}] ", logEntry.EntryDateTime)
            .AppendFormat("[{0}] ", logEntry.Severity)
            .AppendLine(logEntry.Message)
            .AppendLine(logEntry.AdditionalInformation);

        return sb.ToString();
    }
}
```

У этого подхода есть свои преимущества и недостатки.

Недостатки

- Переменные шаги алгоритма должны определяться открытыми методами.
- Может потребоваться приведение к конкретным типам наследников, если не вся информация доступна через призму базового класса (нарушение принципа «открыт/закрыт»).

Достоинства

- Упрощается исходный класс (следование принципу «открыт/закрыт»).
- Класс и его методы расширения могут находиться в разных пространствах имен. Это позволит клиентам самостоятельно решать, нужно им импортировать метод расширения или нет (следование принципу разделения интерфейса).

- Существует возможность разных реализаций метода в зависимости от контекста и потребностей. Метод `GetText` может иметь одну реализацию в серверной части, а другую — в клиентской.

Обсуждение паттерна «Шаблонный метод»

Изменение уровня абстракции

Шаблонный метод может применяться классом-наследником повторно при реализации переменного шага алгоритма, объявленного в базовом классе.

В случае импорта лог-файлов можно выделить отдельный абстрактный базовый класс `LogFileReaderBase`, который будет содержать логику чтения файла. Наследнику останется лишь переопределить операцию разбора прочитанной строки (листинг 2.4).

Листинг 2.4. Базовый класс LogFileReaderBase

```
public abstract class LogFileReaderBase : LogImporter, IDisposable
{
    private readonly Lazy<Stream> _stream;

    protected LogFileReaderBase(string fileName)
    {
        _stream = new Lazy<Stream>(
            () => new FileStream(fileName, FileMode.Open));
    }

    public void Dispose()
    {
        if (_stream.IsValueCreated)
        {
            _stream.Value.Close();
        }
    }
```

```
    protected override sealed IEnumerable<string> ReadEntries(
        ref int position)
    {
        Contract.Assert(_stream.Value.CanSeek);

        if (_stream.Value.Position != position)
            _stream.Value.Seek(position, SeekOrigin.Begin);

        return ReadLineByLine(_stream.Value, ref position);
    }

    protected override abstract LogEntry ParseLogEntry(
        string stringEntry);

    private IEnumerable<string> ReadLineByLine(
        Stream stream, ref int position)
    {
        // Построчное чтение из потока ввода/вывода
    }
}
```

Бывает, что шаблонный метод «скользит» по иерархии наследования. Исходный переменный шаг алгоритма может оказаться довольно сложной операцией, что потребует ее дальнейшей декомпозиции. В результате может появиться еще один абстрактный класс, который еще сильнее разобьет исходные шаги алгоритма на еще более простые и конкретные этапы.

Стратегия vs. шаблонный метод

В некоторых случаях переменный шаг алгоритма является довольно самостоятельной операцией, которую лучше спрятать в отдельной абстракции.

Например, в предыдущем примере всю ответственность за разбор строки можно выделить в отдельный аспект — стратегию разбора записи. При этом данная стратегия может как передаваться на базовом уровне, так и являться деталью реализации конкретного импортера. Вполне возможно, что стратегия разбора будет применяться для всех файловых импортеров, но не применяться в других случаях (листинг 2.5).

Листинг 2.5. Класс LogFileReaderBase, принимающий стратегию ILogParser

```
public interface ILogParser
{
    LogEntry ParseLogEntry(string stringEntry);
}

public abstract class LogFileReaderBase : LogImporter
{
    protected(string fileName, ILogParser logParser)
    {
        // ...
    }

    protected override LogEntry ParseLogEntry(string stringEntry)
    {
        return _logParser.ParseLogEntry(stringEntry);
    }

    // ...
}
```

ПРИМЕЧАНИЕ

В предыдущей главе уже говорилось о том, что стратегия обеспечивает гибкость, однако ведет к увеличению сложности. Более разумно вначале спрятать логику разбора строки в классе LogParser, который может принимать формат разбираемой строки. Только если такой гибкости окажется недостаточно, стоит выделять интерфейс ILogParser.

Шаблонный метод и обеспечение тестируемости

Типичным подходом для обеспечения тестируемости является использование интерфейсов. Определенное поведение, завязанное на внешнее окружение, выделяется в отдельный интерфейс, и затем интерфейс передается текущему классу. Теперь с помощью объектов-подделок (или моков, от mock) можно эмулировать внешнее окружение и покрыть класс тестами в изоляции.

Вместо выделения интерфейса можно воспользоваться разновидностью паттерна «Шаблонный метод» под названием «Выделение и переопределение зависимости» (Extract and Override)[1]. Суть техники заключается в выделении изменчивого поведения в виртуальный метод, поведение которого затем можно переопределить в тесте.

Так, класс `LogFileReader` можно сделать тестируемым путем вынесения открытия файла в виртуальный метод. Затем в тестах можно будет переопределить этот виртуальный метод и вернуть экземпляр `MemoryStream` или своего собственного класса, унаследованного от `Stream` (листинг 2.6).

Листинг 2.6. Использование шаблонного метода в тестах

```
public abstract class LogFileReaderBase : LogImporter, IDisposable
{
    protected LogFileReaderBase(string fileName)
    {
        _stream = new Lazy<Stream>(() => OpenFileStream(fileName)));
    }

    protected virtual Stream OpenFileStream(string fileName)
    {
        return new FileStream(fileName, FileMode.Open);
    }
}

// В тестах
class FakeLogFileReader : LogFileReaderBase
{
    private readonly MemoryStream _mockStream;

    public FakeLogFileReader(MemoryStream mockStream)
        : base(string.Empty)
    {
        _mockStream = mockStream;
    }
```

[1] Впервые данный прием был описан Майклом Физерсом в его книге Working Effectively With Legacy Code, а затем в книге Роя Ошерова The Art of Unit Testing.

```
    protected override Stream OpenFileStream(string fileName)
    {
        return _mockStream;
    }
}

[Test]
public void TestFakedMemoryStreamProvidedOneElement()
{
    // Arrange
    LogFileReaderBase cut = new FakeLogFileReader(
        GetMemoryStreamWithOneElement());

    // Act
    var logEntries = cut.ReadLogEntry();

    // Assert
    Assert.That(logEntries.Count(), Is.EqualTo(1));
}
```

ВНИМАНИЕ

! Желание протестировать некоторый код в изоляции может привести к ощущению ложной безопасности: код протестирован, а значит, в нем нет ошибок. В юнит-тестах практически невозможно учесть все тонкости, которые могут возникнуть при работе с реальными файлами. При работе с внешним окружением, помимо юнит-тестов, обязательно должны присутствовать интеграционные тесты, которые проверят граничные условия в сложных ситуациях.

Шаблонный метод и контракты

Шаблонный метод определяет контракт между базовым классом и наследниками, поэтому очень важно, чтобы этот контракт был максимально понятным. Подходящие сигнатуры методов и комментарии могут дать понять разработчику класса-наследника, что ему можно делать, а что — нет. Более подходящим способом формализации отношений между классами является использование принципов проектирования по контракту.

С помощью предусловий и постусловий можно более четко показать желаемое поведение и ограничения методов наследников. Сделать это можно с помощью библиотеки Code Contracts (листинг 2.7).

Листинг 2.7. Пример использования контрактов с шаблонным методом

```
[ContractClass(typeof (LogImporterContract))]
public abstract class LogImporter
{
    protected abstract IEnumerable<string> ReadEntries(ref int position);

    protected abstract LogEntry ParseLogEntry(string stringEntry);
}

[ExcludeFromCodeCoverage, ContractClassFor(typeof (LogImporter))]
public abstract class LogImporterContract : LogImporter
{
    protected override IEnumerable<string> ReadEntries(ref int position)
    {
        Contract.Ensures(Contract.Result<IEnumerable<string>>() != null);
        Contract.Ensures(
            Contract.ValueAtReturn(out position) >=
            Contract.OldValue(position));

        throw new System.NotImplementedException();
    }

    protected override LogEntry ParseLogEntry(string stringEntry)
    {
        Contract.Requires(stringEntry != null);
        Contract.Ensures(Contract.Result<LogEntry>() != null);

        throw new System.NotImplementedException();
    }
}
```

Предусловия и постусловия в библиотеке Code Contracts задаются с помощью методов класса `Contract`, таких как `Requires`, `Ensures`, `Assume` и др. Поскольку утверждения задаются с помощью методов, то контракты абстрактных методов и интерфейсов задаются в специальном классе, помеченном атрибутом `ContractClassFor`.

С помощью предусловий и постусловий разработчик базового класса может четко указать, что методы `ReadEntries` и `ParseLogEntry` не могут возвращать `null`. А также что значение аргумента `position` не должно уменьшиться после вызова метода `ReadEntries`.

ПРИМЕЧАНИЕ

Полноценное описание принципов контрактного программирования выходит за рамки данной книги. Лучшим источником по этой теме является книга Бертрана Мейера «Объектно-ориентированное конструирование программных систем», в которой контрактное программирование используется в качестве основного инструмента объектно-ориентированного проектирования. С контрактным программированием на платформе .NET можно познакомиться в моих статьях по этой теме по адресу http://bit.ly/DesignByContractArticles, а также в официальной документации библиотеки Code Contracts.

Применимость

Практически всегда, когда у вас в голове появляется мысль о повторном использовании кода с помощью наследования, стоит подумать, как можно выразить отношения между базовым классом и его наследником максимально четко. При этом нужно помнить о своих клиентах — как о внешних классах, так и о наследниках: насколько просто создать наследника класса, какие методы нужно переопределить, что в них можно делать, а что — нельзя? Когда и от каких классов иерархии наследования нужно наследовать? Насколько легко добавить еще одного наследника, не вдаваясь в детали реализации базового класса?

Формализация отношений между базовым классом и наследником с помощью контрактов и шаблонного метода сделает жизнь разработчиков наследников проще и понятнее. Шаблонный метод задает каркас, который четко говорит пользователю, что он может сделать и в каком контексте.

Примеры в .NET Framework

Примеров использования паттерна «Шаблонный метод» в .NET Framework очень много. По большому счету, любой абстрактный класс, который содержит защищенный абстрактный метод, является примером паттерна «Шаблонный метод».

WCF просто пропитан этим паттерном. Одним из примеров является класс `CommunicationObject`, методы которого `Open`, `Close`, `Abort` и др. «запечатаны» (sealed), но при этом вызывают виртуальные или абстрактные методы `OnClosed`, `OnAbort` и т. д.

Другими примерами этого паттерна в составе WCF могут служить `ChannelBase`, `ChannelFactoryBase`, `MessageHeader`, `ServiceHostBase`, `BodyWriter`, которые определяют каркас алгоритма и позволяют наследникам задавать лишь некоторые шаги.

Еще примеры:

- `SafeHandle` (и его наследники) с абстрактным методом `ReleaseHandle`;
- класс `TaskScheduler` с его внутренним `QueueTask` и открытым `TryExecuteTaskInline`;
- класс `HashAlgorithm` с его `HashCore`, класс `DbCommandBuilder` и многие другие.

Паттерн освобождения ресурсов (Dispose Pattern)

На платформе .NET проблемой управления памятью занимается сборщик мусора, но при этом остаются вопросы со своевременной очисткой ресурсов. В языке C++ для этого используются деструкторы, а в управляемых языках, таких как C#, — метод `Dispose` интерфейса `IDisposable`, который и дал название соответствующему паттерну проектирования.

Паттерн освобождения ресурсов (Dispose Pattern) на платформе .NET весьма тяжеловесен. Почему вообще появилась необходимость в управлении ресурсами в системе с автоматической сборкой мусора? Потому что, помимо памяти, приложение может владеть и другими важными ресурсами, такими как дескрипторы файлов и потоков, мьютексы, семафоры и т. п. Время сборки мусора зависит от многих факторов и не может (и не должно) контролироваться приложением. С ресурсами дело обстоит иначе: разработчик должен приложить все усилия, чтобы время владения ими было минимальным.

В основе паттерна освобождения ресурсов лежат понятия управляемых и неуправляемых ресурсов, которые в рамках «управляемой» платформы кажутся нелогичными. Неуправляемые ресурсы представляют собой объекты, о которых совершенно ничего не известно среде исполнения (CLR, Common Language Runtime). Хорошим примером могут служить «сырые дескрипторы», представленные значением `IntPtr`. Как только «сырой объект» оборачивается в управляемый класс, такой ресурс становится управляемым.

Управление ресурсами в .NET основывается на интерфейсе `IDisposable`, метод `Dispose` которого вызывается пользовательским кодом, и на финализаторе

(finalizers), который вызывается во время сборки мусора. Разница между финализатором и методом `Dispose` состоит в том, что первый вызывается сборщиком мусора в неизвестный момент времени, при этом порядок вызова финализаторов для разных объектов не определен[1]. Второй вызывается пользовательским кодом, после чего ссылка на «мертвый» объект продолжает существовать.

Полноценная реализация паттерна управления ресурсами для «незапечатанных» (non-sealed) классов довольно тяжеловесна. Разработчик класса должен добавить дополнительный виртуальный метод `Dispose(bool)`, ответственный за освобождение управляемых и неуправляемых ресурсов в зависимости от переданного аргумента (листинг 2.8).

Листинг 2.8. Пример реализации паттерна освобождения ресурсов

```
public class ComplexResourceHolder : IDisposable
{
    // Буфер из неуправляемого кода (неуправляемый ресурс)
    private IntPtr _buffer;
    // Дескриптор события ОС (управляемый ресурс)
    private SafeHandle _handle;

    public ComplexResourceHolder()
    {
        // Захватываем ресурсы
        _buffer = AllocateBuffer();
        _handle = new SafeWaitHandle(IntPtr.Zero, true);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Неуправляемые ресурсы освобождаются в любом случае
        ReleaseBuffer(_buffer);

        // Вызываем из метода Dispose, освобождаем управляемые ресурсы
```

[1] Строго говоря, это не совсем так. Очередь сборки мусора двухприоритетная. Сначала вызываются «обычные» объекты, после которых вызываются «критические» объекты, классы которых унаследованы от CriticalFinalizerObject. Эта особенность позволяет «обычным» пользовательским типам обращаться в своих финализаторах к полям типа Thread, ReaderWriterLock или SafeHandle.

```
        if (disposing)
        {
            if (_handle != null)
                _handle.Dispose();
        }
    }

    ~ComplexResourceHolder()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    // Методы AllocateBuffer и ReleaseBuffer
}
```

В основе данного паттерна лежит предположение, что любой класс с ресурсами может содержать *одновременно* управляемые и неуправляемые ресурсы. При этом основная работа по освобождению ресурсов делается в методе `Dispose(bool)`, который вызывается из метода `Dispose` и финализатора, а булев аргумент позволяет понять, откуда вызывается этот метод.

Поскольку при вызове метода `Dispose` объект еще полностью «жив», то в этом случае можно освобождать управляемые и неуправляемые ресурсы. При вызове финализатора нет никаких гарантий, что управляемые ресурсы, сохраненные в полях текущего объекта, еще не были освобождены своими собственными финализаторами, поэтому нам остается освободить лишь неуправляемые ресурсы.

У этого паттерна есть два очень важных недостатка. Во-первых, паттерн очень известен, а во-вторых, он является избыточным в 99,999 % случаев.

Главный вопрос, который нужно себе задать: а будет ли мой класс или его наследник содержать одновременно управляемые и неуправляемые ресурсы? Какой смысл в одном классе держать поля типа `FileStream` и `IntPtr`? Разве это не

будет нарушать здравый смысл и принципы проектирования, такие как принцип единственной обязанности?

Работать с неуправляемыми ресурсами довольно сложно, поэтому первое, что нужно сделать, — завернуть его в управляемую оболочку, например в `SmartHandle`. Но в реализации этого класса паттерн управления ресурсами тоже не понадобится, поскольку класс `SmartHandle` будет содержать лишь неуправляемый ресурс и никогда не будет хранить в нем ничего больше.

Даже если кому-то потребуется смешать два вида ресурсов в одном классе, то вместо малопонятного метода `Dispose(bool)` намного полезнее воспользоваться паттерном «Шаблонный метод» и четко разделить эти шаги друг от друга. Для этого процесс очистки ресурсов следует разбить на две составляющие: очистку неуправляемых ресурсов с помощью метода `DisposeNativeResources` и очистку управляемых ресурсов `DisposeManagedResources`.

Листинг 2.9. Использование шаблонного метода для управления ресурсами

```
public class ProperComplexResourceHolder : IDisposable
{
    // Поля и конструктор класса аналогичны

    protected virtual void DisposeNativeResources()
    {
        ReleaseBuffer(_buffer);
    }

    protected virtual void DisposeManagedResources()
    {
        if (_handle != null)
            _handle.Dispose();
    }

    ~ProperComplexResourceHolder()
    {
        DisposeNativeResources();
    }

    public void Dispose()
    {
```

```
        DisposeNativeResources();
        DisposeManagedResources();
        GC.SuppressFinalize(this);
    }

    // Методы AllocateBuffer и ReleaseBuffer
}
```

В этом случае метод `Dispose` стал немного сложнее, но при разработке повторно используемого кода больше внимания нужно уделить не количеству кода в базовом классе, а легкости и однозначности реализации наследников. Автору любого производного класса будет четко понятно, что можно делать в переопределенном методе, а что — нет.

ПРИМЕЧАНИЕ

Подробнее о паттерне Dispose, разнице между управляемыми и неуправляемыми ресурсами можно почитать в статье Dispose Pattern по адресу bit.ly/Dispose-PatternDotNet. Самым полным описанием этого паттерна является статья Джо Даффи DG Update: Dispose, Finalization and Resource Management по адресу http://joeduffyblog.com/2005/04/08/dg-update-dispose-finalization-and-resource-management/.

SafeHandle и шаблонный метод

Несмотря на то что предложенная версия паттерна освобождения ресурсов не является общепринятой, существуют классы .NET Framework, которые используют полноценный шаблонный метод для управления ресурсами. Один из таких представителей — класс `SafeHandle`.

Так, помимо виртуального метода `Dispose(bool)`, в классе `SafeHandle` определен абстрактный метод `ReleaseHandle`, предназначенный непосредственно для освобождения неуправляемого дескриптора, и абстрактное свойство `IsInvalid`, которое должно сказать, валиден ли текущий дескриптор.

В результате, если вы захотите реализовать свой управляемый дескриптор, это можно сделать путем создания наследника класса `SafeHandle` (листинг 2.10) и путем переопределения «недостающих шагов алгоритма шаблонного метода».

Листинг 2.10. Пример создания наследника класса SafeHandle

```
// Класс HativeHelper в коде отсутствует
class CustomSafeHandler : SafeHandle
{
    private readonly IntPtr _nativeHandle =
```

```
        NativeHelper.InvalidHandler;

    [ReliabilityContract(
        Consistency.WillNotCorruptState, Cer.MayFail)]
    public CustomSafeHandler()
       : base(NativeHelper.InvalidHandler, true)
    {
        _nativeHandle = NativeHelper.AcquireHandle();
    }

    public override bool IsInvalid
    {
        [SecurityCritical]
        get
        {
            return
                _nativeHandle == NativeHelper.InvalidHandler;
        }
    }

    protected override bool ReleaseHandle()
    {
        NativeHelper.ReleaseHandle(DangerousGetHandle());
        return true;
    }
}
```

Метод `ReleaseHandle` будет вызван при вызове метода `Dispose` и при вызове финализатора, но лишь в том случае, если дескриптор окажется валидным (свойство `IsInvalid` вернет `false`).

Предложенный вариант паттерна освобождения ресурсов не является общепринятым. В случае повторно используемого кода я предпочту общепринятую версию паттерна. Однако в случае внутреннего кода я настоятельно рекомендую обдумать и обсудить с коллегами возможность применения упрощенной версии паттерна на основе шаблонного метода.

Глава 3

Паттерн «Посредник» (Mediator)

Назначение: определяет объект, инкапсулирующий способ взаимодействия множества объектов.

Другими словами: посредник — это клей, связывающий несколько независимых классов между собой. Он избавляет классы от необходимости ссылаться друг на друга, позволяя тем самым их независимо изменять и анализировать.

Мотивация

Самая простая реализация импорта логов может выглядеть так. Класс `LogFileReader` читает лог-файлы и вызывает методы класса `LogSaver`, который сохраняет лог-файлы для последующего полнотекстового поиска[1] (рис. 3.1).

---

[1] Рассмотренный здесь пример очень напоминает вариант, данный Робертом Мартином в статье «Принцип инверсии зависимостей», которую можно найти по адресу http://www.objectmentor.com/resources/articles/dip.pdf. Разница лишь в том, что вместо читателя и писателя логов Роберт использует классы Лампы (Lamp) и Кнопки (Button). Хотя задачи очень похожи, Роберт предлагает несколько иное решение, основанное на выделении интерфейса и инверсии зависимостей. С моей точки зрения, данная задача идеально решается именно с помощью паттернов «Наблюдатель» и «Посредник» и не требует выделения никаких дополнительных интерфейсов.

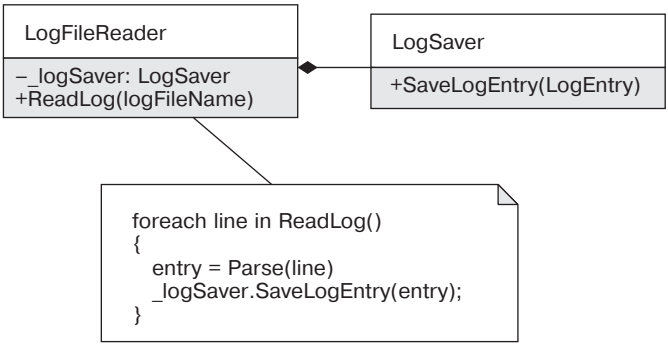


Рис. 3.1. Дизайн примитивного процесса импорта лог-файлов

При таком подходе класс `LogFileReader` жестко связан с `LogSaver`, что не позволяет использовать класс `LogFileReader` повторно в другом контексте. Также это усложняет анализ и развитие класса `LogFileReader`, поскольку требует понимания работы его зависимостей.

Основная проблема этого дизайна в том, что он плохо моделирует задачу. Классы чтения и сохранения логов являются независимыми шагами импорта лог-файлов и не должны знать друг о друге. Решение о «перекладывании» логов из одного источника в другой должно решаться не на уровне этих классов, а выше. И дело здесь не столько в отсутствии гибкости текущего решения, сколько в ненужной сложности и плохом разделении ответственности. В системе можно четко выделить три аспекта: чтение логов, сохранение логов и связующее звено, которое знает, что нужно перекладывать логи именно таким образом.

Самое простое решение заключается в выделении еще одного класса, `LogImporter`, который будет знать о двух других классах и заниматься импортом логов из одного источника в другой (рис. 3.2).

Класс `LogImporter` выступает в роли посредника: он связывает воедино несколько низкоуровневых классов для обеспечения нового высокоуровневого поведения. Такой подход обеспечивает гибкость в развитии системы, хотя и не вводит полиморфного поведения. Посредник в этом случае выступает барьером, который гасит изменения в одной части системы, не давая им распространяться на другие части! Любые изменения в классе `LogFileReader` приведут к модификации `LogImporter`, но не потребуют изменений класса `LogFileSaver` или его клиентов.

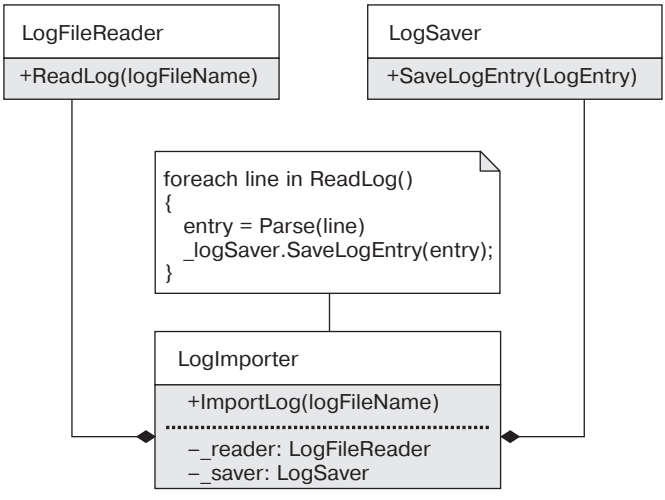


Рис. 3.2. Диаграмма классов импорта лог-файлов

ПРИМЕЧАНИЕ

Эрик Эванс в своей знаменитой книге Domain-Driven Design. Tackling Complexity in the Heart of Software четко разделяет понятия гибкого (flexible) и податливого (supple) дизайна. Гибкость обычно обеспечивается за счет дополнительных уровней абстракции (например, полиморфизма и иерархий наследования), что неизбежно приводит к увеличению сложности. Податливый же дизайн представляет собой самый простой способ решения поставленной задачи. Он не обеспечивает возможности адаптации системы к новым требования на лету, во время исполнения. Изменение поведения потребует внесения изменений в код приложения, но поскольку дизайн прост и податлив, то сделать это будет просто. Такой подход экономит силы и позволяет избежать одной из типичных ловушек современного проектирования — проблемы преждевременного обобщения (premature generalization).

Как и большинство других классических паттернов, описание паттерна «Посредник» от «банды четырех» предполагает наличие наследования. Классическая диаграмма классов паттерна «Посредник» приведена на рис. 3.3.

Обычно взаимодействующие компоненты не содержат общего предка (если не считать класса `System.Object`) и совсем не обязательно знают о классе-посреднике. Иерархия посредников также применяется довольно редко.

Участники:

- `Mediator (LogFileImporter)` — определяет интерфейс посредника. На практике базовый класс посредника выделяется редко, поэтому класс `Mediator` обычно содержит всю логику взаимодействия;

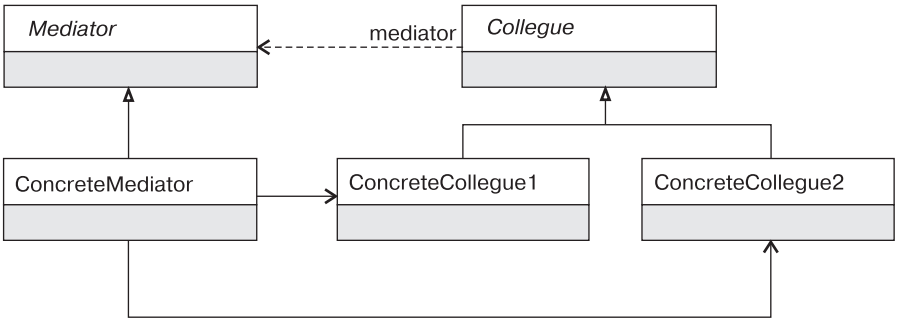


Рис. 3.3. Диаграмма классов паттерна «Посредник»

- `ConcreteCollegue1`, `ConcreteCollegue2` (`LogFileReader`, `LogFileSaver`) — классы одного уровня абстракции, которые взаимодействуют друг с другом косвенным образом через посредника.

Обсуждение паттерна «Посредник»

Основная суть борьбы со сложностью заключается в изоляции информации, чтобы у разработчика была возможность сосредоточиться на одной проблеме и игнорировать другие. Именно по этой причине мы стараемся объединять логически связанные куски системы в отдельные модули (отсюда сильная связность — high cohesion), максимально изолируя эти модули друг от друга (отсюда слабая связанность — low coupling).

При этом любая сложная система вырастает на основе более простых компонентов. Мы получаем иерархичную систему, объединяя независимые строительные блоки в более высокоуровневые абстракции. Любой класс или модуль строится на основе автономных классов/модулей более низкого уровня, обеспечивая передачу управления между ними. Любой такой класс/модуль играет роль посредника.

«Посредник» — это один из самых распространенных паттернов проектирования. Они десятками используются в любом приложении, хотя в именах классов это практически никогда не отражается.

Если же подходить к вариантам реализации взаимодействия, то можно выделить две разновидности паттерна «Посредник»: посредник может быть явным и о нем могут знать объединяемые компоненты или же он может быть невидимым для этих компонентов и объединять их между собой без их ведома.

Явный и неявный посредник

В классической реализации паттерна «Посредник» независимые классы не знают друг о друге, но знают о существовании посредника и все взаимодействие происходит через него явным образом.

Такой подход довольно широко распространен и активно применяется в паттернах «Поставщик»/«Потребитель» (Producer/Consumer), «Агрегатор событий» (Event Aggregator) или в других случаях, когда классы знают о существовании общей шины взаимодействия. При этом посредник может содержать ссылки на взаимодействующие классы и маршрутизировать вызовы явно. Или же он может быть наблюдаемым объектом и предоставлять набор событий, на которые будут подписываться коллеги для взаимодействия между собой.

В то же время классы низкого уровня могут и не знать о существовании посредника. Классы `LogFileReader` и `LogFileSaver` не знают о существовании посредника `LogImporter`. Это делает их более автономными, а дизайн — более естественным. Если же одному из участников понадобится активно управлять процессом обмена сообщениями (то есть использовать push-модель взаимодействия[1]), то достаточно сделать его наблюдаемым. В этом случае участник останется автономным и не будет знать о посреднике (рис. 3.4).

Класс `LogFileReader` вполне может быть активным, следить за содержимым логфайла и сохранять новые фрагменты по мере их поступления. В этом случае наиболее естественное решение заключается в использовании событий или интерфейса `IObservable` для активного уведомления посредника о прочитанных записях.

ПРИМЕЧАНИЕ

Наличие в дизайне системы наблюдателей говорит о наличии посредников. Паттерн «Наблюдатель» будет рассмотрен в одной из последующих глав.

Именно такой подход используется в Windows Forms и других UI-платформах. Форма выступает в роли посредника: следит за событиями компонентов формы и передает управление другим компонентам или бизнес-объектам.

При наличии формы (`CustomForm`) с двумя элементами управления — `TextBox` и `Button` — мы можем добавить логику разрешения кнопки сохранения при вводе пользователем данных в `TextBox` (листинг 3.1).

[1] Push- и pull-модели взаимодействия более подробно будут рассмотрены в главе 5.

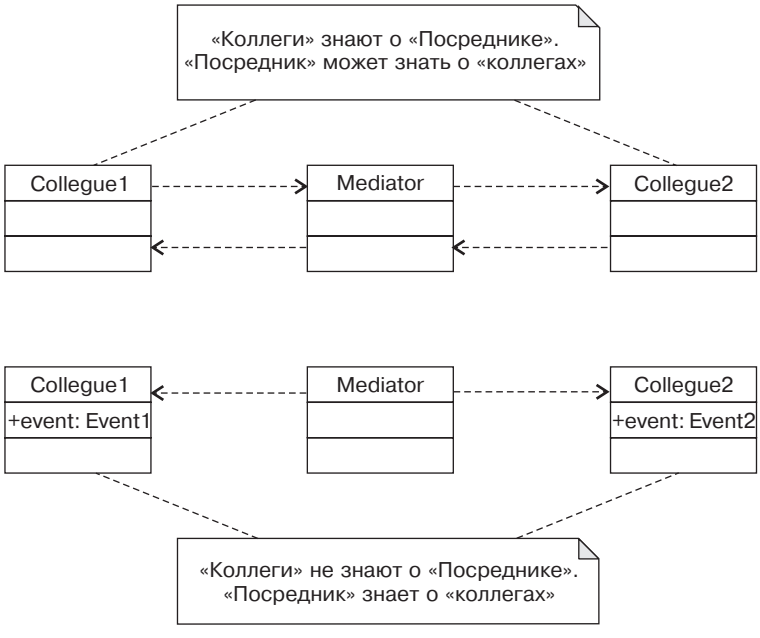


Рис. 3.4. Явная и неявная реализация посредника

Листинг 3.1. Пример паттерна «Наблюдатель» в Windows Forms

```
public CustomForm()
{
    InitializeComponent();
    buttonSave.Enabled = false;

    textBoxName.TextChanged += (s,ea) =>
    {
        buttonSave.Enabled = true;
    };
}
```

Явные и неявные связи

Одной из главных целей многих паттернов проектирования является получение слабосвязанного дизайна. Это позволяет развивать части системы независимо, бороться со сложностью и выпускать новые версии систем со скоростью света.

Однако, как и все в дизайне, слабая связанность имеет свою цену, особенно когда с ней начинают перегибать палку. Дизайн системы, в которой никто ни о ком не знает, так же утопичен, как и дизайн, в котором каждый класс знает обо всех остальных.

Слабая связанность всегда должна идти рука об руку с сильной связностью. Хороший дизайн разрезает систему на модули, которые совместно решают некую задачу. При этом внутри модуля может быть довольно много связей, и вполне нормально, если эти связи будут явными. А это значит, что слабую связанность нужно обеспечивать на границах модуля, а не внутри него. При этом польза отсутствия явных связей между классами может быть обманчивой.

Вместо явного и конкретного посредника (класса `LogImporter`), который контролирует логическое взаимодействие классов `LogFileReader` и `LogSaver`, можно было бы использовать универсальный посредник — агрегатор событий (Event Aggregator) (рис. 3.5).

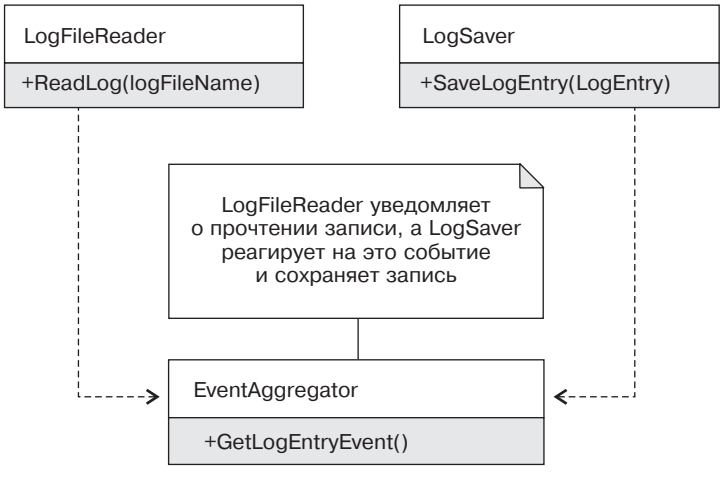


Рис. 3.5. Импорт логов на основе агрегатора событий

В этом случае класс `LogFileReader` публикует событие о чтении/записи логфайла и не будет знать, кто его обрабатывает. `LogSaver` слушает это событие и сохраняет записи, не зная, откуда они были прочитаны.

Обобщение простого решения, в котором нет никакой необходимости, убило не один проект. Смысл посредника в том, что он инкапсулирует в себе процесс взаимодействия объектов. Агрегатор событий тоже может выполнять эту роль, но при этом логика взаимодействия начинает расплываться.

Агрегаторы событий не устраняют связи между классами, они лишь делают их неявными. Когда в системе взаимодействуют десятки объектов, возможно, это и оправданно. Но значительно лучше начинать с явного решения и обобщать его лишь тогда, когда стало понятно, в чем заключается обобщение и что оно действительно нужно.

Слабосвязанный дизайн не означает, что класс `А` не имеет ссылки на класс `Б`. Слабосвязанный дизайн говорит о том, что класс `А` может успешно функционировать, развиваться и изменяться, никак не завися от класса `Б`. Наличие же неявных связей через события, интерфейсы или глобальные переменные не устраняет связей между этими классами, он лишь прячет их и делает неявными.

Если предметная область говорит о наличии связи между понятием `А` и понятием `Б`, то в дизайне системы лучше всего отразить эти отношения явным образом.

Тестировать или не тестировать? Вот в чем вопрос!

При наличии в системах довольно большого количества посредников может возникнуть логичный вопрос: стоит ли подвергать их юнит-тестам? У сторонников TDD (Test-Driven Development) ответ на этот вопрос есть, но что делать всем остальным?

С одной стороны, любая сложная логика, которая важна с точки зрения работы приложения, должна быть подвергнута тестам. С другой — в случае простого посредника вполне достаточно будет проверить его логику интеграционными тестами и не тратить время на модульное тестирование.

Здесь, как и во многих других случаях, следует попытаться найти компромисс между трудозатратами на разработку и поддержку тестов и выгодой от их наличия. Любая критически важная логика должна быть подвергнута тестам. Точка. Но должны ли посредники содержать критически важную логику? Вполне возможно, что сам факт наличия сложной логики в посреднике говорит о том, что он делает слишком многое. Если же его посредническая логика сложная и важная, то стоит подумать о том, нельзя ли ее упростить.

Тестирование поведения осуществляется с помощью моков (mocks) — специального вида тестовых подделок, которые поддерживаются большинством современных тестовых фреймворков. Благодаря им обычно не составляет особого труда написать набор тестов, которые будут проверять, что в определенных условиях тестируемый класс (CUT — Class Under Test) вызывает те или иные методы своих зависимостей. Однако с такими тестами нужно быть осторожными и уделять особое внимание тому, чтобы проверять лишь ключевую логику (например, лишь факт вызова ме-

тода зависимости без проверки точных аргументов вызова), а также стараться избегать проверок деталей реализации.

ПРИМЕЧАНИЕ

Подробнее о разнице между тестированием состояния и тестированием поведения можно прочитать в моей статье «Моки и стабы» (bit.ly/StubsVsMocks) или в статье Stubs are not Mocks Мартина Фаулера (http://martinfowler.com/articles/mocks-ArentStubs.html).

Архитектурные посредники

Паттерн «Посредник» может применяться на разных уровнях приложения. Существуют классы-посредники, компоненты-посредники, есть целые модули или слои приложения, выполняющие эту роль.

Очень часто слой приложения играет роль посредника: объединяет вместе доменные и сервисные слои и обеспечивает поведение, уникальное для конкретного приложения.

Например, модули импорта/экспорта логов могут использоваться по-разному в разных приложениях. Можно создать консольное приложение, которое будет импортировать логи лишь определенных типов и запускаться по расписанию. Можно создать сервис, который будет принимать логи по сети для последующего сохранения (рис. 3.6).

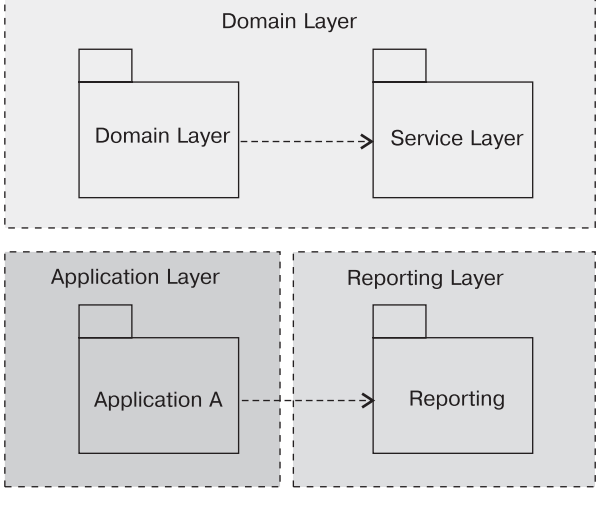


Рис. 3.6. Посредник на уровне слоев приложения

Основная логика связывания компонентов в готовое приложение обычно находится в точке входа приложения или недалеко от нее — в функции `Main`, `ServiceMain` или `global.asax`. Такая точка называется корнем композиции (Composition Root) и уникальным образом позволяет отличить одно приложение от другого. В ней может находиться логика конфигурирования IoC-контейнера или же создание явного класса-посредника.

Применимость

Паттерн «Посредник» идеально подходит для объединения нескольких автономных классов или компонентов. Каждый раз, когда вы задаетесь вопросом, как же изолировать классы `A` и `Б`, чтобы они могли жить независимо, подумайте об использовании посредника.

При этом есть несколько вариантов решения: классы `A` и `Б` могут знать о посреднике или же один из классов может быть наблюдаемым. Выбор того или иного варианта зависит от «толщины» интерфейса взаимодействия между этими классами. Если протокол общения прост, то использование наблюдателя будет вполне оправданным, если же он довольно сложен, то проще использовать посредник явно.

Когда третий лишний

Как и любой другой паттерн, «Посредник» нужно использовать с умом. Этот паттерн изолирует изменения в одной части системы, не давая им распространяться на другие части. В некоторых случаях это упрощает внесение изменений, а в некоторых, наоборот, усложняет.

Не нужно разделять с помощью посредника тесно связанные вещи. Если процессы импорта и экспорта всегда изменяются совместно, то, возможно, они должны знать друг о друге. Наличие посредника между классами, которые всегда изменяются совместно, лишь усложнит внесение изменений: вместо изменения двух классов придется изменять три.

Примеры в .NET Framework

«Посредник» — это паттерн, который не проявляется в открытом интерфейсе модуля или библиотеки, поэтому примеры использования нужно искать в реализа-

ции .NET Framework или пользовательском коде. Тем не менее примеров использования паттерна «Посредник» в .NET-приложениях довольно много.

- В Windows Forms любая форма по своей сути представляет собой посредник, объединяющий элементы управления между собой: форма подписывается на события одного элемента управления и в случае изменения состояния уведомляет бизнес-объекты или другие элементы управления.
- Класс `EventAggregator`, активно используемый в WPF и за его пределами, и является примером глобального посредника для связи разных независимых компонентов между собой.
- В паттернах MVC (Model — View — Controller, «Модель — представление — контроллер») и MVP (Model — View — Presenter, «Модель — представление — презентер»), Controller и Presenter выступают в роли посредника между представлением и моделью[1].
- Паттерн Producer/Consumer («Поставщик»/«Потребитель») является еще одним примером паттерна «Посредник». В этом паттерне потребитель и поставщик не знают друг о друге и общаются между собой за счет общей очереди, которая является посредником. В случае .NET Framework таким посредником может служить BlockingCollection.

[1] Немного подробнее MVx-паттерны будут рассмотрены в заключительной части книги, в главе 21, посвященной принципу инверсии зависимостей.

Глава 4

Паттерн «Итератор» (Iterator)

Назначение: представляет доступ ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Мотивация

Практически любое приложение в той или иной мере работает с коллекциями данных. Мы постоянно используем векторы, списки, деревья и хеш-таблицы. В некоторых случаях для обработки данных коллекции используется специфический интерфейс конкретных коллекций, но в большинстве случаев внутренний доступ осуществляется за счет специального абстрактного слоя — итераторов.

Итераторы предоставляют абстрактный интерфейс для доступа к содержимому составных объектов, не раскрывая клиентам их внутреннюю структуру. В результате получается четкое разделение ответственностей: клиенты получают возможность работать с разными коллекциями унифицированным образом, а классы коллекций становятся проще за счет того, что ответственность за перебор ее элементов возлагается на отдельную сущность.

Итераторы настолько укоренились в большинстве языков и платформ, что их поддержка появилась даже на уровне языков программирования: `foreach` в C#, `range-for` в C++ 11, `for` в Java 5+, и даже в консервативном Eiffel появилась

аналогичная конструкция — `across`. Более того, многие языки поддерживают не только потребление итераторов с помощью циклов `foreach` и им подобных, но и их создание за счет блоков итераторов (Iterator Block в C# и VB) или так называемых конструкторов последовательностей (Sequence Comprehension), доступных в F#, Scala, Phyton и многих других языках.

Итераторы отлично подходят для чтения данных из некоторого источника. Так, например, вместо класса `LogFileReader` мы могли бы использовать класс `LogFileSource`, который бы реализовывал интерфейс `IEnumerable<LogEntry>` (листинг 4.1).

Листинг 4.1. Код класса LogFileSource

```
public class LogFileSource : IEnumerable<LogEntry>
{
    private readonly string _logFileName;

    public LogFileSource(string logFileName)
    {
        _logFileName = logFileName;
    }

    public IEnumerator<LogEntry> GetEnumerator()
    {
        foreach (var line in File.ReadAllLines(_logFileName))
        {
            yield return LogEntry.Parse(line);
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Паттерн «Итератор» — это один из немногих паттернов, который пришел в .NET Framework из книги «банды четырех» практически в неизменном виде. Если взять

исходную диаграмму классов из книги *Design Patterns*, заменить `Aggregate` на `IEnumerable<T>`, `Iterator` на `IEnumerator<T>` и немного изменить методы класса `Iterator`, то мы получим очень похожую картину (рис. 4.1, 4.2).

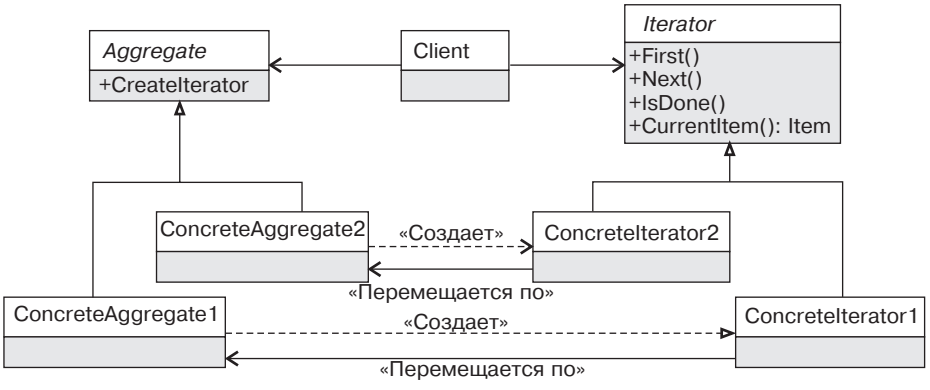


Рис. 4.1. Классическая диаграмма паттерна «Итератор»

Участники:

- `Iterator (IEnumerator<T>)` — определяет интерфейс итератора;
- `Aggregate (IEnumerable<T>)` — составной объект, по которому может перемещаться итератор;
- `ConcreteAggregate1 (List<T>)` — конкретная реализация агрегата;
- `ConcreteIterator (List.Enumerator<T>)` — конкретная реализация итератора для определенного агрегата.

Обсуждение

Итераторы в .NET являются *однонаправленными итераторами только для чтения*. При этом для получения итератора используется метод `GetEnumerator` интерфейса `IEnumerable`, который каждый раз возвращает новый экземпляр итератора.

Интерфейс `IEnumerator` также довольно прост:

- `MoveNext` — переход на следующий элемент агрегата. Возвращает `false`, если достигнут конец последовательности;
- `Current` — возвращает текущий элемент;
- `Reset` — возвращает итератор к началу агрегата. Реализуется не всегда.

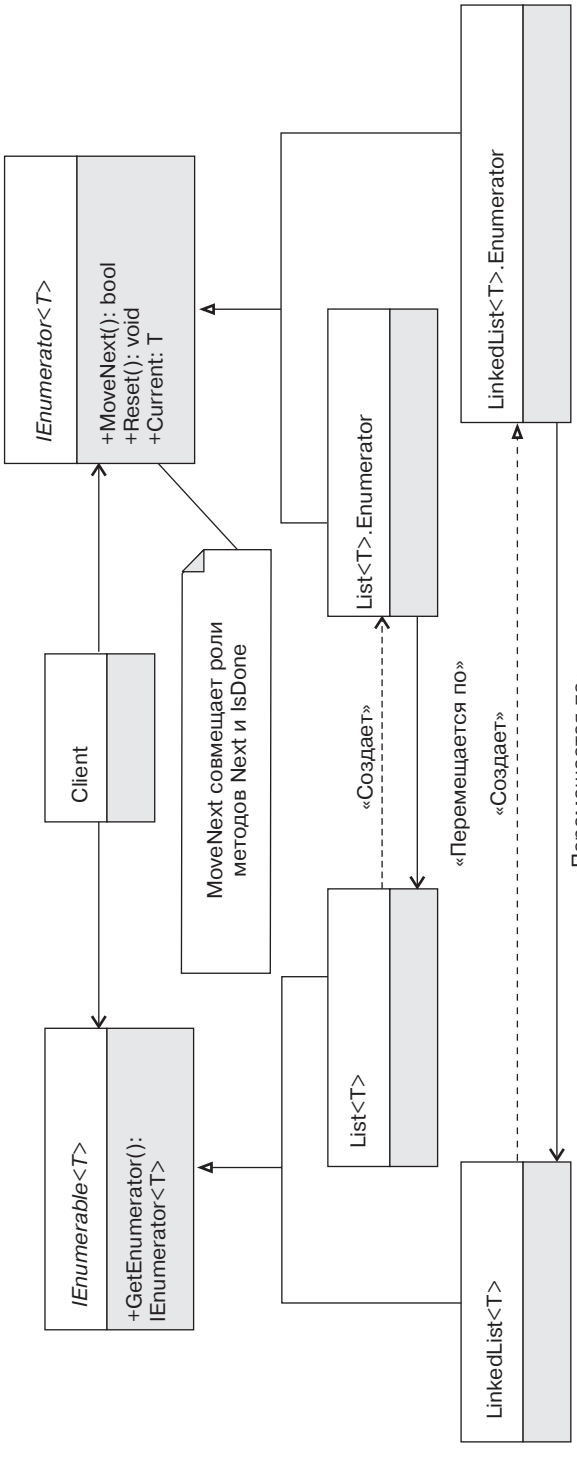


Рис. 4.2. Паттерн Итератор в .NET Framework

Сразу после создания итератор указывает на –1-й элемент, поэтому для перехода к первому элементу нужно вызвать `MoveNext` как минимум один раз.

Итераторы в .NET могут показаться довольно примитивными, особенно по сравнению с итераторами в C++. Стандарт C++ определяет несколько разных типов итераторов: `Input` ← `Forward` ← `Bidirectional` ← `RandomAccess`. В C++ есть даже `OutputIterator`, то есть итератор для вывода данных.

Платформа .NET поддерживает два типа итераторов: необобщенные и обобщенные. Первые появились в первой версии платформы .NET, а вторые были добавлены со второй версии вместе с обобщениями (generics) и блоками итераторов (iterator blocks) (рис. 4.3).

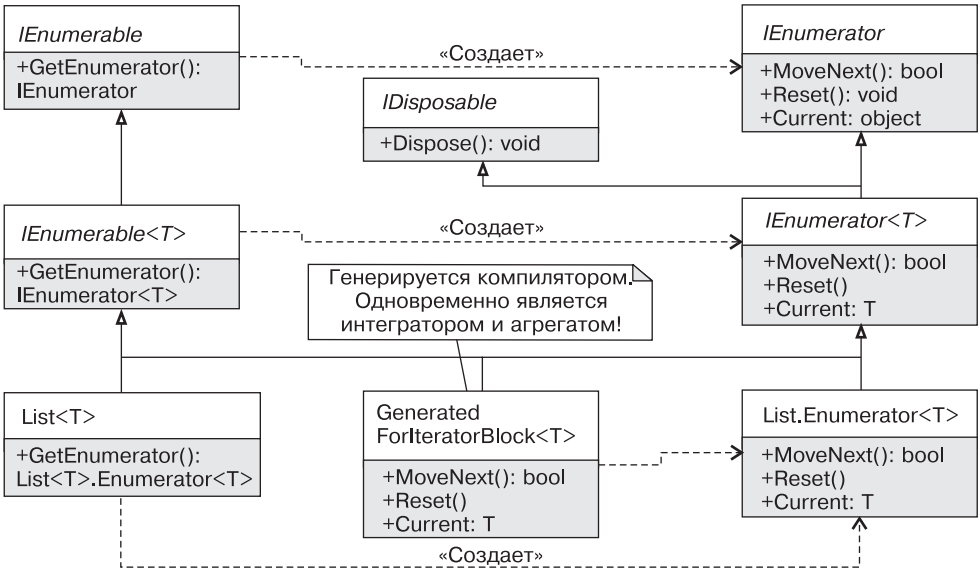


Рис. 4.3. Обобщенные и необобщенные итераторы

Особенности итераторов в C#/.NET

Контракт итераторов. У любого класса или интерфейса есть контракт. Иногда контракт бывает явным, и тогда он описывается с помощью соответствующих инструментов, таких как Code Contracts. Иногда он является неявным, тогда для понимания принципов использования, предусловий, постусловий и инвариантов нужно обратиться к документации, внимательно прочитать секцию remarks, а часто еще и вникнуть в детали реализации.

При установке библиотеки Code Contracts в нашем распоряжении появляются контракты всех типов BCL. Однако встроенные контракты интерфейсов `IEnumerable/IEnumerator` не слишком хороши, поэтому давайте выведем их самостоятельно.

Нас будут интересовать контракты лишь метода `MoveNext` и свойства `Current` (листинг 4.2).

Листинг 4.2. Неформальный контракт итераторов в .NET

```
public T Current
{
    get
    {
        Contract.Requires(!Disposed, "Iterator should not be disposed.");
        Contract.Requires(IteratorPointsToCorrectValue,
            "MoveNext() should be called and return 'true'.");

        Contract.Ensures(true, "Returns current value from the
        Aggregate.");

        // Возвращаем произвольное значение, чтобы удовлетворить компилятор
        return default(T);
    }
}

public bool MoveNext()
{
    Contract.Requires(!Disposed, "Iterator should not be disposed.");
    Contract.Requires(Valid, "Iterator should  be valid.");

    // Если итератор еще не дошел до конца агрегата,
    // он будет перемещен на следующий элемент
    Contract.Ensures(Finished() ||
                InternalIndex == Contract.OldValue(InternalIndex) + 1);
    Contract.Ensures(Contract.Result<bool>() ==
```

```
                    (InternalIndex < InternalLength));
    return default(bool);
}
```

ПРИМЕЧАНИЕ

Обратите внимание на то, что это не настоящие контракты итератора, это лишь мое представление того, какими они могли бы быть! Настоящий контракт итератора, определенный в mscorlib.Contracts.dll, не содержит всех этих вещей.

Контракт свойства Current

- **Предусловие:**
  - итератор не должен быть освобожден с помощью вызова `Dispose`;
  - итератор должен указывать на корректное значение: пользовательский код должен вызвать метод `MoveNext`, который должен вернуть `true`.
- **Постусловие:** свойство `Current` вернет значение, на которое «указывает» итератор. Это условие нельзя выразить в форме предиката, потому используется `Contract.Ensures(true, "")`. Итератор не налагает ограничений, вернет это свойство `null` или нет.

Контракт метода MoveNext

- **Предусловие:**
  - итератор не должен быть освобожден с помощью вызова `Dispose`;
  - итератор должен быть валидным (коллекция не должна быть изменена после получения текущего итератора).
- **Постусловие:** если итератор не дошел до конца коллекции, то он перейдет на следующий элемент и метод вернет `true`, в противном случае метод вернет `false`.

Тут есть несколько интересных моментов. Во-первых, предусловие свойства `Current` слабее предусловия метода `MoveNext`. На самом деле у свойства `Current` вообще нет предусловий: мы можем обратиться к свойству `Current` после вызова `Dispose` и до вызова `MoveNext` и не получим исключений! Я же добавил эти требования в контракт, поскольку никто в здравом уме не должен обращаться к свойству `Current` без выполнения этих условий.

И еще один момент, связанный с методом `MoveNext`: вам никто не запрещает вызывать `MoveNext` на завершенном итераторе. В этом случае метод `MoveNext`

просто вернет `false`! Вот это более валидное требование, поскольку оно позволяет заново проходить по завершенному итератору, а также свободно использовать итератор пустой коллекции (который можно рассматривать как завершенный).

Блоки итераторов. Процесс создания итераторов вручную является довольно утомительным занятием, которое включает управление состоянием и перемещением по элементам коллекции при вызове `MoveNext`. С помощью блока итераторов реализовать итератор существенно проще[1] (листинг 4.3).

Листинг 4.3. Пример итератора массива

```
public static IEnumerator<int> CustomArrayIterator(this int[] array)
{
    foreach (var n in array) { yield return n; }
}
```

С помощью блока итераторов можно создавать итераторы для своих коллекций, существующих коллекций или вообще для внешних ресурсов, таких как файлы. Для этого достаточно открыть файл в начале метода и возвращать прочитанные блоки данных с помощью `yield return`. Кроме того, мы можем генерировать данные бесконечно, о чем поговорим в разделе «Итераторы или генераторы».

ПРИМЕЧАНИЕ

Теперь должно быть понятно, почему метод Reset в контракте итератора является необязательным. Представьте себе, что итератор возвращает данные, пришедшие по сети. Как в этом случае мы сможем реализовать метод Reset?

Блок итераторов преобразуется компилятором языка C# в конечный автомат с несколькими состояниями, соответствующими начальному положению итератора (когда он указывает на –1-й элемент), конечному положению (когда итератор прошел все элементы) и «среднему» положению, при котором он указывает на определенный элемент. При этом блок итераторов представляет собой некую форму сопрограмм (corouting)[2], которые продолжают исполнение с предыдущего места благодаря методу `MoveNext`.

---

[1] Пример создания итератора вручную рассмотрен в моей статье «Итераторы в C#. Часть 1», расположенной по адресу http://bit.ly/IteratorPartOne.

[2] Подробнее о сопрограммах можно прочитать в Сети, например https://ru.wikipedia.org/wiki/Сопрограмма.

«Ленивость» итераторов

Итераторы, полученные с помощью блока итераторов, являются «ленивыми»: их тело исполняется не в момент вызова метода, а при переборе элементов с помощью метода `MoveNext`. Это приводит к некоторым особенностям обработки ошибок, ведь даже валидация аргументов метода, возвращающего итератор, будет производиться уже в момент «потребления» итератора (листинг 4.4).

Листинг 4.4. Пример блока итераторов

```
public static IEnumerable<string> ReadFromFile(string path)
{
    if (path == null) throw new ArgumentNullException("path");
    foreach(string line in File.ReadLines(path))
    {
        yield return line;
    }
}

// Где будет ошибка?
var result = ReadFromFile(null); //1
foreach (var l in result)
{
    Console.WriteLine(l); //2
}
```

На этом же принципе построена большая часть методов LINQ (Language Integrated Query), что позволяет получать сложные запросы без лишних накладных расходов.

Подробнее об итераторах в языке C#, а также о деталях реализации блоков итераторов смотрите в статье «Итераторы в C#», доступной по адресу http://bit.ly/IteratorsInCSharp. Подробнее о предусловиях в блоке итераторов и асинхронных методах можно прочитать в моей статье «Когда предусловия не являются предусловиями».

Использование итераторов в цикле foreach

Цикл `foreach` является универсальным инструментом для обработки коллекций/последовательностей. Способ его преобразования компилятором зависит от типа

перебираемой коллекции (обобщенная/необобщенная) и представляет собой простой цикл `while`. Пример обхода необобщенной коллекции выглядит следующим образом (листинг 4.5).

Листинг 4.5. Внутренняя реализация цикла foreach

```
public static void ForEachIEnumerable(IEnumerable sequence)
{
    // foreach(var e in sequence) {Console.WriteLine(e);}
    IEnumerator enumerator = sequence.GetEnumerator();
    object current = null;
    try
    {
        while (enumerator.MoveNext())
        {
            current = enumerator.Current;
            Console.WriteLine(current);
        }
    }
    finally
    {
        IDisposable disposable = enumerator as IDisposable;
        if (disposable != null)
        {
            disposable.Dispose();
        }
    }
}
```

ПРИМЕЧАНИЕ

Для поддержки цикла foreach не обязательно наличие интерфейса IEnumerable/IEnumerable<T>. Достаточно, чтобы класс коллекции содержал метод GetEnumerator, который будет возвращать тип с методом bool MoveNext() и свойством Current.

Подробнее об этом можно почитать в статье «Duck typing, или Так ли прост foreach».

Также стоит обратить внимание на то, что реализация блока `foreach` изменилась в C# 5.0, начиная с которого переменная `current` внесена во внутреннюю

область видимости (подробности в статье «Замыкания на переменных цикла в C# 5.0»[1]).

Любой типизированный итератор реализует интерфейс `IDisposable`, поскольку сам интерфейс `IEnumerator<T>` наследует `IDisposable`. Причина этого в том, что итераторы, полученные с помощью блока итераторов, могут содержать ресурсы, которые освобождаются в блоке `finally`, вызов которого как раз и осуществляется путем вызова `Dispose` итератора. Но все дело в том, что блок итераторов может возвращать не только типизированный итератор, но и его предшественников `IEnumerable/IEnumerator`, которые не реализуют интерфейс `IDisposable`.

Итераторы или генераторы

Блок итераторов может использоваться для создания итераторов, то есть для обхода некоторого агрегата в памяти (коллекции) или за ее пределами (итератор содержимого файла). Но, помимо этого, блок итераторов может использоваться для создания генераторов.

Пример простого бесконечного генератора чисел Фибоначчи приведен в листинге 4.6.

Листинг 4.6. Использование блока итераторов для генерации бесконечной последовательности

```
public static IEnumerable<int> GenerateFibonacci()
{
    int prev = 0;
    int current = 1;

    while (true)
    {
        yield return current;

        int tmp = current;
        current = prev + current;
        prev = tmp;
    }
}
```

[1] Данная статья доступна по адресу http://bit.ly/ClosingOverLoopVariable.

В этом плане блок итераторов в C# напоминает более общие концепции из других языков программирования под названием «списковое включение» (list comprehension), которые предназначены для создания последовательностей и коллекций.

Валидность итераторов

В некоторых языках, таких как C++, понятие инвалидации итераторов (когда итератор коллекции становится недействительным) определено в спецификации языка, в разделе, посвященном конкретной коллекции. Так, например, не для всех контейнеров операция добавления элемента делает итератор недействительным: добавление элемента в двусвязный список вполне допустимо, а добавление элемента в вектор — нет.

Подобные правила, хотя и не столь формальные, существуют и для коллекций .NET Framework. Точнее, есть лишь одно правило, и оно не привязано к конкретному типу коллекции: при изменении коллекции все ранее полученные итераторы становятся недействительными. Так, в обоих случаях в листинге 4.7 будет сгенерирован `InvalidOperationException`.

Листинг 4.7. Примеры инвалидации итераторов в .NET

```
var list = new List<int> { 42, 12 };
var listIter = list.GetEnumerator();
listIter.MoveNext();

list.RemoveAt(1); // Удаляем 2-й элемент
Console.WriteLine(listIter.Current); // OK
listIter.MoveNext(); // InvalidOperationException

var linked = new LinkedList<int>();
linked.AddLast(42);
var linkedIter = linked.GetEnumerator();
linkedIter.MoveNext();

linked.AddLast(12);
Console.WriteLine(linkedIter.Current); // OK
linkedIter.MoveNext(); // InvalidOperationException
```

Это поведение коренным образом отличается от правил коллекций языка C++, поскольку в случае `std::list` обе приведенные операции были бы допустимыми.

Итераторы и структуры

Итераторы всех коллекций .NET Framework являются изменяемыми структурами. Это, с одной стороны, избавляет от дополнительного выделения памяти в управляемой куче при проходе по коллекции, а с другой стороны, может привести к неожиданному результату. Попробуйте предугадать поведение следующего кода (листинг 4.8), а потом запустите его, чтобы проверить свою догадку.

Листинг 4.8. Неочевидное поведение итераторов

```
var x = new {Items = new List<int> {1, 2, 3}.GetEnumerator()};
while (x.Items.MoveNext())
{
    Console.WriteLine(x.Items);
}
```

Но, несмотря на потенциальную опасность, итераторы любой широко используемой коллекции должны быть структурой. Более того, в некоторых случаях есть правила, запрещающие использовать коллекции с классами-итераторами. Хорошим примером является правило участия в проекте Roslyn, которое запрещает использовать классы-итераторы в критических участках кода! (Roslyn. How to Contribute, раздел Coding Conventions.)

ПРИМЕЧАНИЕ

Подробнее о проблемах с изменяемыми значимыми типами читайте в заметках «О вреде изменяемых значимых типов» и «О вреде изменяемых значимых типов. Часть 2». Еще один пример проблемы изменяемых итераторов рассмотрен в заметке «Observable.Generate и перечисление списков»[1].

Push-based-итераторы

Мало кто обратил внимание на то, что в книге «банды четырех» определены два вида итераторов, *внешний* и *внутренний*, в зависимости от того, кто управляет итерацией — клиент или сам итератор.

Внешний итератор — это классический (pull-based) итератор, когда процессом обхода явно управляет клиент путем вызова метода `Next` или ему подобного.

[1] Статья доступна по ссылке http://bit.ly/ObservableGenerate.

Внутренний итератор — это push-based-итератор, которому передается метод обратного вызова, и он сам уведомляет клиента о «посещении» следующего элемента.

Несложно догадаться, что ранее мы рассмотрели внешний итератор, а внутренний итератор в .NET представлен библиотекой Reactive Extensions и парой интерфейсов `IObserver<T>/IObservable<T>`. Да, эта пара интерфейсов больше напоминает наблюдатель, а не итератор, но пример все расставит по местам (листинг 4.9).

Листинг 4.9. Пример использования IObservable

```
var list = new List<int> {1, 2, 3};
IObservable<int> observable = list.ToObservable();
observable.Subscribe(
    onNext: n => Console.WriteLine("Processing: {0}", n),
    onCompleted: () => Console.WriteLine("Sequece finished"));
```

Данный пример не имеет особого смысла, но, например, преобразование в «наблюдаемую» коллекцию объекта `SqlDataReader`, который также реализует `IEnumerable`, вполне имело бы смысл[1].

Применимость

В соответствии со своим определением итератор применяется для доступа к содержимому составных объектов, типичным примером которых являются коллекции. Но стоит ли делать итерируемыми бизнес-объекты?

Какой подход более разумен: использовать `LogFileReader` с методом `IEnumerable<LogEntry Read()` или класс `LogFileSource`, который будет реализовывать `IEnumerable<LogEntry>`?

Для меня первый вариант является предпочтительным, поскольку более четко отражает производимые действия. `LogFileSource` прячет информацию о том, что происходит чтение записей из файла, и то, когда выполняется это действие. Происходит ли чтение файла в конструкторе? Проверяется ли наличие файла

[1] Подробнее ознакомиться с реактивными расширениями можно в серии статей Ли Кэмпбелла Introduction to Rx. В контексте задачи импорта логов реактивные последовательности будут рассмотрены в главе 5, при описании паттерна «Наблюдатель».

в конструкторе, а чтение выполняется при первом вызове метода `MoveNext`? Всех этих вопросов можно избежать при использовании класса `LogFileReader`.

ПРИМЕЧАНИЕ

Недавно Эрик Липперт дал похожий совет на StackOverflow.com в ответе на вопрос: Why not inherit from List, поясняя, должен ли класс FootballTeam наследовать от List‹Player›. Футбольная команда *не является* списком игроков, поэтому класс FootballTeam не должен наследовать List‹Player›. В этом случае гораздо лучше подходит отношение ИМЕЕТ, а значит, команда должна содержать список игроков.

Примеры в .NET Framework

В .NET Framework итераторы представлены парами интерфейсов:

- `IEnumerable/IEnumerator` — для работы с необобщенными коллекциями (составными объектами);
- `IEnumerable<T>/IEnumerator<T>` — для работы с обобщенными коллекциями (составными объектами);
- `IObservable<T>/IObserver<T>` — для работы с реактивными (или push-based) коллекциями.

Использование итераторов в языке C# осуществляется с помощью цикла `foreach`, а создание итераторов упрощается за счет блоков итераторов.

Глава 5

Паттерн «Наблюдатель» (Observer)

Назначение: определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Другими словами: наблюдатель уведомляет все заинтересованные стороны о произошедшем событии или об изменении своего состояния.

Общие сведения

Существует два способа общения между двумя программными элементами. `Компонент 1` может обратиться к `Компоненту 2` для получения каких-то данных или выполнения некоторой операции. В этом случае `Компонент 2` выполняет определенную работу, когда его об этом попросят.

В некоторых случаях `Компонент 2` является активным, содержит собственный поток исполнения или каким-то другим способом следит за своим состоянием. В этом случае `Компонент 2` может уведомить `Компонент 1` о некотором событии.

Первая модель взаимодействия называется pull-моделью, а вторая — push-моделью (рис. 5.1).

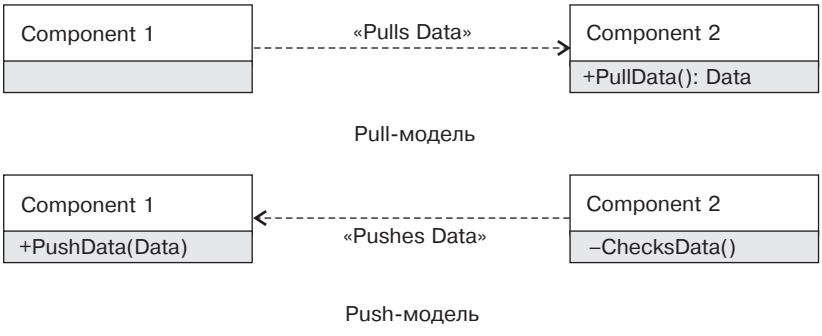


Рис. 5.1. Pull- и push-модель взаимодействия

Push-модель взаимодействия появилась задолго до распространения ООП и паттернов проектирования. В мире структурного программирования push-модель реализуется с помощью методов обратного вызова (callbacks). В мире функционального программирования, которое тоже появилось задолго до ООП, push-модель представлена в виде реактивной модели программирования. А в мире ООП push-модель реализуется с помощью паттерна «Наблюдатель».

Мотивация

В системе для импорта лог-файлов приложение может следить за набором файлов и загружать их содержимое по мере появления в логах новых записей. За чтение записей из лог-файла отвечает `LogFileReader`.

Возможны две реализации данного класса. Класс `LogFileReader` может быть пассивным и вычитывать новый фрагмент лог-файла при вызове метода `Read` (pull-модель). Или же класс `LogFileReader` может быть активным и сам вычитывать новые фрагменты лог-файла по мере его обновления (push-модель).

Возможны два варианта реализации push-модели. Класс `LogFileReader` может знать о классе `LogSaver`, который будет обрабатывать и сохранять новые фрагменты лог-файла в некотором хранилище. Или же `LogFileReader` может быть наблюдаемым и уведомлять любых заинтересованных подписчиков о прочитанных новых фрагментах, например, с помощью событий.

В первом случае мы получаем жесткую связь между классами `LogFileReader` и `LogSaver`, а во втором — слабосвязанный дизайн, в котором процессы чтения и сохранения логов могут развиваться независимо.

Мотивация использования паттерна «Наблюдатель»: уменьшить связанность класса с его зависимостями путем уничтожения связи инициатора некоторого события с его обработчиками.

Классическая диаграмма классов паттерна «Наблюдатель» приведена на рис. 5.2.

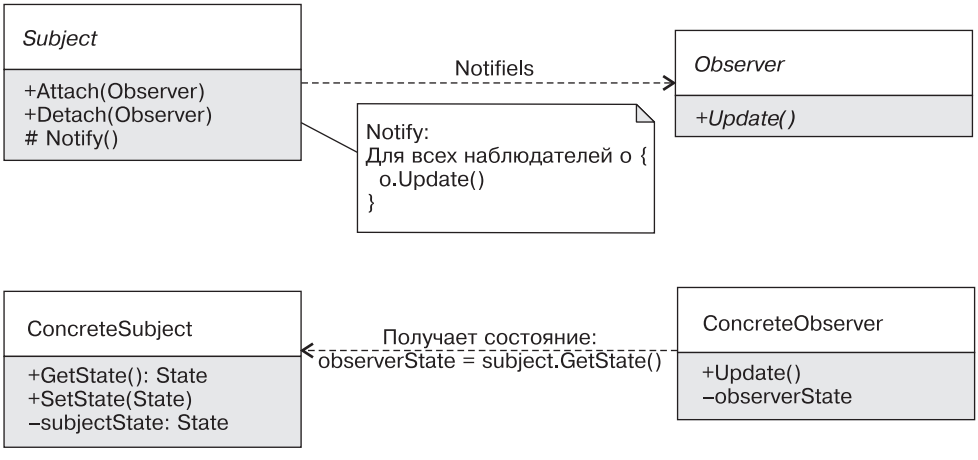


Рис. 5.2. Классическая диаграмма паттерна «Наблюдатель»

Участники:

- `Observer` — определяет интерфейс наблюдателя;
- `Subject` (наблюдаемый объект) — определяет методы подключения и отключения наблюдателей;
- `ConcreteObserver` — реализует интерфейс наблюдателя;
- `ConcreteSubject` — конкретный тип наблюдаемого объекта.

На платформе .NET практически невозможно встретить классическую реализацию паттерна «Наблюдатель».

Наличие делегатов и событий, а также интерфейсов `IObserver/IObservable` и библиотеки реактивных расширений делает использование наследования излишним.

ПРИМЕЧАНИЕ

Язык C# и платформа .NET очень многое позаимствовали из Java, но наблюдатели были реализованы по-своему. В Java используется каноническая версия паттерна: базовый класс object содержит метод notify/notifyAll и выступает одновременно в роли базового класса наблюдаемых объектов (BaseSubject) и базового класса наблюдателей (BaseObserver).

Варианты реализации

Существует несколько вариантов реализации паттерна «Наблюдатель» на платформе .NET:

- с помощью делегатов (методов обратного вызова);
- с помощью событий (events);
- с помощью специализированных интерфейсов-наблюдателей;
- с помощью интерфейсов `IObserver/IObservable`.

Методы обратного вызова

Самая простая форма наблюдателя на платформе .NET реализуется с помощью делегатов. Для этого достаточно, чтобы класс потребовал делегат в аргументах конструктора и уведомлял вызывающий код с его помощью (листинг 5.1).

Листинг 5.1. Реализация класса LogFileReader на основе делегатов

```
public class LogFileReader : IDisposable
{
    private readonly string _logFileName;
    private readonly Action<string> _logEntrySubscriber;
    private readonly static Timespan CheckFileInterval =
        Timespan.FromSeconds(5);
    private readonly Timer _timer;

    public LogFileReader(string logFileName,
        Action<string> logEntrySubscriber)
    {
        Contract.Requries(File.Exists(logFileName));

        _logEntrySubscriber = logEntrySubscriber;
        _timer = new Timer(() =>
            CheckFile(), CheckFileInterval, CheckFileInterval);
    }

    public void Dispose()
    {
```

```
        _timer.Dispose();
    }

    private void CheckFile()
    {
        foreach(var logEntry in ReadNewLogEntries())
        {
            _logEntrySubscriber(logEntry);
        }
    }

    private IEnumerable<string> ReadNewLogEntries()
    {
        // ...
        // Читаем новые записи из файла,
        // которые появились с момента последнего чтения
    }
}
```

Это самая простая и ограниченная реализация паттерна «Наблюдатель». Делегаты на платформе .NET могут содержать цепочку вызовов, но подобный код моделирует отношение 1:1 между классом `LogFileReader` и его наблюдателем. Зачастую этого вполне достаточно, к тому же это позволяет гарантировать наличие наблюдателя, что может быть полезным, когда наблюдаемому объекту потребуется получить назад некоторые результаты.

События

Паттерн «Наблюдатель» является настолько распространенным, что многие языки программирования поддерживают его из коробки. В языках платформы .NET паттерн «Наблюдатель» (или паттерн «Издатель» — «Подписчик») реализуется с помощью событий.

События представляют собой умную оболочку над делегатами, которая позволяет клиентам лишь подписываться на события или отказываться от подписки, а владельцу события — еще и инициировать событие для уведомления всех подписчиков (листинг 5.2).

Листинг 5.2. Реализация класса LogFileReader на основе событий

```
public class LogEntryEventArgs : EventArgs
{
    public string LogEntry {get; internal set;}
}

public class LogFileReader : IDisposable
{
    private readonly string _logFileName;

    public LogFileReader(string logFileName)
    {
        //...
    }

    public event EventHandler<LogEntryEventArgs> OnNewLogEntry;

    private void CheckFile()
    {
        foreach(var logEntry in ReadNewLogEntries())
        {
            RaiseNewLogEntry(logEntry);
        }
    }

    private void RaiseNewLogEntry(string logEntry)
    {
        var handler = OnNewLogEntry;
        if (handler != null)
            handler(this, new LogEntryEventArgs(logEntry));
    }
}
```

Данный вариант очень похож на вариант с делегатами — с одной важной разницей. Интерфейс класса `LogFileReader` позволяет подписаться на событие получения

новых записей лог-файлов любому числу подписчиков. При этом нет гарантии, что эти подписчики вообще будут.

Строго типизированный наблюдатель

В некоторых случаях группу событий или делегатов удобно объединить в одном интерфейсе. Иногда это говорит о наличии скрытой абстракции, но иногда бывает просто удобно оперировать интерфейсом, а не набором событий (листинг 5.3).

Листинг 5.3. Пример строго типизированного наблюдателя

```
public interface ILogFileReaderObserver
{
    void NewLogEntry(string logEntry);
    void FileWasRolled(string oldLogFile, string newLogFile);
}

public class LogFileReader : IDisposable
{
    private readonly ILogFileReaderObserver _observer;
    private readonly string _logFileName;

    public LogFileReader(string logFileName,
        ILogFileReaderObserver observer)
    {
        _logFileName = logFileName;
        _observer = observer;
    }

    // Добавлена дополнительная логика, которая определяет,
    // что логер перестал писать в текущий лог-файл и переключился
    // на новый.

    private void DetectThatNewFileWasCreated()
    {
        // Метод вызывается по таймеру
```

```
        if (NewLogFileWasCreated())
            _observer.FileWasRolled(_logFileName, GetNewLogFileName());
    }
}
```

Данный вариант очень похож на классическую версию паттерна «Наблюдатель» с той лишь разницей, что обычно наблюдатель является единственным. Если нужно множество «подписчиков», то проще воспользоваться версией с событиями.

IObserver/IObservable

Все перечисленные ранее варианты реализации паттерна «Наблюдатель» содержат одно ограничение: они плохо объединяются для получения более высокоуровневого поведения (not composable). Над событиями или делегатами невозможно выполнять операции, которые можно выполнять над последовательностями. В случае pull-based-реализации клиенты класса `LogFileReader` могут использовать LINQ (Language Integrated Query) для манипулирования прочитанными сообщениями.

Например, с помощью LINQ довольно легко обрабатывать критические сообщения особым образом, сохраняя их пачками по десять элементов:

```
// Сохраняем только критические ошибки группами по десять элементов

var messages =
    logFileReader.Read()
     .Select(ParseLogMessage)
     .Where(m => m.Severity == Critical);

foreach(var criticalMessages in messages.Buffer(10))
{
    BulkSaveMessages(criticalMessages);
}
```

Метод расширения `Buffer` отсутствует в классе `Enumerable`, но его довольно легко реализовать самостоятельно.

С 4-й версии в .NET Framework появилась пара интерфейсов `IObserver/IObservable` с набором методов расширений, известных под названием реактивных

расширений (Rx, Reactive Extensions). Интерфейс `IObservable` моделирует реактивные последовательности и позволяет работать с наблюдаемыми последовательностями через привычный LINQ-синтаксис.

Реактивные расширения представляют собой push-последовательности, и для их использования класс `LogFileReader` нужно изменить следующим образом:

```
public class LogFileReader : IDisposable
{
    private readonly string _fileName = fileName;
    private readonly Subject<string> _logEntriesSubject =
        new Subject<string>();

    public LogFileReader(string fileName)
    {
        _fileName = fileName;
    }

    public void Dispose()
    {
        // Закрываем файл
        CloseFile();
        // Уведомляем подписчиков, что событий больше не будет
        _logEntriesSubject.OnComplete();
    }

    public IObservable<string> NewMessages
    {
        get { return _logEntriesSubject; }
    }

    private void CheckFile()
    {
        foreach(var logEntry in ReadNewLogEntries())
        {
```

```
            _logEntriesSubject.OnNext(logEntry);
        }
    }
}
```

Что позволяет использовать с наблюдателями привычный LINQ-синтаксис:

```
var messagesObservable =
    logFileReader.NewMessages
     .Select(ParseLogMessages)
     .Where(m => m.Severity == Critical);


messagesObservable
  .Buffer(10)
  .Subscribe(IList<LogEntry> criticalMessages => BulkSaveMessages(critical
Messages));
```

ПРИМЕЧАНИЕ

Реактивные последовательности[1] являются своеобразной смесью паттернов «Итератор» и «Наблюдатель». IObservable можно рассматривать как «вывернутые наизнанку» последовательности, когда процессом итерирования управляет не вызывающая сторона, как в случае с IEnumerable, а сама последовательность. Стандартные последовательности являются pull-based, поскольку процессом получения новых элементов управляет вызывающая сторона, а реактивные последовательности — push-based, и наблюдатели получают новые элементы по мере их поступления. Подробнее паттерн «Итератор» рассмотрен в главе 4.

Интерфейсы `IObserver/IObservable` не являются наблюдателями общего назначения и предполагают наличие определенного протокола между наблюдаемым объектом и его подписчиками. Интерфейс `IObservable` предполагает, что однородные события будут периодически повторяться. Наблюдаемый объект может уведомить о новом событии (`OnNext`), о том, что в процессе события произошла ошибка (`OnError`), или о том, что цепочка событий завершена (`OnComplete`). Нет особого смысла использовать `IObservable` для уведомления о переконфигурации приложения, поскольку композиция подобных событий вряд ли возможна.

[1] Подробное описание реактивных расширений выходит за рамки данной книги. Для более подробного знакомства подойдет книга Introduction to Rx Ли Кэмпбелла.

Обсуждение паттерна «Наблюдатель»

Выбор варианта реализации «Наблюдателя»

Проще всего дело обстоит с `IObserver/IObservable`. Этот вариант подходит, когда события возникают периодически и их можно рассматривать в виде push-based-последовательности, над которой удобно производить трансформации с помощью LINQ-запросов. Поток сетевых сообщений от клиента или сервера, координаты устройства — все это подходящие задачи для использования реактивных последовательностей.

К тому же наличие классов `Subject<T>` и специальных методов-адаптеров (`Observable.FromEvent`) позволяет легко получить реактивную последовательность из других реализаций наблюдателей. Выбор между остальными тремя реализациями более интересный.

Делегаты

Реализация паттерна «Наблюдатель» на основе методов обратного вызова пришла из мира структурного программирования и не похожа на классическую реализацию, описанную «бандой четырех». Тем не менее эта реализация весьма распространена и успешно решает задачу данного паттерна: получить слабосвязанный дизайн путем разделения инициатора события и его обработчиков.

Данная реализация моделирует отношение 1:1 между наблюдаемым объектом и наблюдателем. Обязательное наличие наблюдателя позволяет установить двустороннюю связь: наблюдаемый объект может не только уведомлять об изменении своего состояния, но и требовать некоторого результата. Результатом может быть объект класса `Task`, поскольку обработка события может быть длительной, или некоторое значение, необходимое наблюдаемому объекту. Это может быть внешняя валидация, взаимодействие с пользователем или что-то еще.

ПРИМЕЧАНИЕ

Делегаты могут использоваться для реализации паттернов «Наблюдатель» или «Стратегия».

Когда использовать?

- Наблюдатель должен быть обязательно.
- Наблюдаемый объект не просто уведомляет наблюдателя, но и ожидает некоторого результата.

Когда не использовать?

- Когда число делегатов начинает расти и передавать их все через аргументы конструктора становится неудобно. В этом случае лучше использовать именованную зависимость или выделить интерфейс наблюдателя.
- Для повторно используемых компонентов. В этом случае лучше использовать события.
- Когда через делегат передается поток событий, которым будет удобнее манипулировать с помощью `IObservable`.

События

События являются самым распространенным вариантом реализации паттерна «Наблюдатель» на платформе .NET. С их помощью происходит уведомление о событиях пользовательского интерфейса, приеме данных по сети и т. п. Главная особенность событий в том, что класс не может гарантировать наличие наблюдателей-подписчиков, а значит, не может потребовать от них никаких результатов. В некоторых случаях используются изменяемые аргументы, например для отмены некоторого действия (в событиях `Form.OnClosing` или `TaskScheduler.UnobservedException`), но это уже примеры паттерна «Цепочка обязанностей», а не наблюдателя.

Когда использовать?

- Для повторно используемых компонентов.
- Для уведомления множества наблюдателей, когда не ожидаешь от них каких-либо ответных действий.
- Для реализации pull-модели получения данных наблюдателем.

Когда не использовать?

- Когда наблюдаемому объекту нужно получить от наблюдателей некоторый результат.

Наблюдатель в виде специализированного интерфейса

Интерфейс наблюдателя обычно является первым этапом для выделения именованной зависимости. На некотором этапе разработки становится понятно, что

у класса должна быть зависимость, но с именем пока определиться сложно. Поэтому, как в случае с классом `LogFileReader`, выделяется интерфейс наблюдателя (`ILogFileReaderObserver`), который затем может переродиться в полноценную зависимость (например, в `ILogFileProcessor`).

Когда использовать?

- В качестве временной именованной зависимости для группировки набора событий.

Когда не использовать?

- В открытом API — в повторно используемом коде или на стыке модулей.

Сколько информации передавать наблюдателю

Существует две модели передачи нужной информации наблюдателям: push и pull[1]. В случае push-модели наблюдаемый объект передает всю нужную информацию о произошедшем событии в аргументах: в свойствах объекта `EventArgs`, в аргументах делегата или метода. В случае pull-модели наблюдаемый объект лишь уведомляет о произошедшем событии, а всю нужную информацию для его обработки наблюдатель самостоятельно получает у наблюдаемого объекта.

Хорошим примером компромиссного решения между push- и pull-вариантами реализации наблюдателей являются события в .NET.

Стандартное событие в .NET объявляется с помощью обобщенного делегата `EventHandler<T>`, который содержит два аргумента: `object sender` и `TEventArgs args`:

```
public class LogEntryEventArgs : EventArgs
{
    public string LogEntry {get; internal set;}
}
```

[1] Push/pull-модель взаимодействия в этом обсуждении встречается дважды. Паттерн «Наблюдатель» представляет собой push-модель взаимодействия между объектами, поскольку наблюдаемый объект самостоятельно «проталкивает» информацию о произошедшем событии. Но затем возникает вопрос о количестве передаваемой информации. И здесь push/pull-модели вступают в игру вновь, но уже в ином контексте — в контексте количества передаваемой информации.

```
public class LogFileReader : IDisposable
{
    private readonly string _logFileName;

    public string LogFileName { get { return _logFileName; } }

    public event EventHandler<LogEntryEventArgs> OnNewLogEntry;

    // Остальные методы пропущены ...
}

// Код наблюдателя
public class LogForwarder
{
    public LogForwarder(LogFileReader logFileReader)
    {
        logFileReader.OnNewLogEntry += HandleNewLogEntry;
    }

    private void HandleNewLogEntry(object sender, LogEntryEventArgs ea)
    {
        var logEntry = ea.LogEntry;
        var logFile = ((LogFileReader)sender).LogFileName;

        // Обрабатываем logEntry с учетом имени файла
    }
}
```

ПРИМЕЧАНИЕ

Технически никто не запрещает использовать в качестве событий любые другие делегаты, а не только EventHandler. Но стандарты кодирования платформы .NET предполагают использовать именно EventHandler‹T›.

В данном случае наблюдаемый объект «выталкивает» (push) наблюдателям лишь минимально необходимую информацию, позволяя им, если потребуется, самостоятельно «вытянуть» (pull) дополнительные данные.

Рекомендации по выбору модели взаимодействия. Компромисс между push- и pull-моделями взаимодействия представляет собой классическую проблему выбора между простым и гибким решением. Push-модель проще, поскольку вся информация, требующаяся для обработки события, передается наблюдателю в виде аргументов. Pull-модель гибче, поскольку позволяет передавать наблюдателям лишь минимальный объем информации и дает возможность наблюдателям самим решать, что им нужно для обработки события.

На практике pull-модель взаимодействия бывает полезной в случае пользовательского интерфейса, когда UI-компонент не знает, что пользователю может понадобиться для обработки определенного события. В случае бизнес-логики push-модель является более предпочтительной, поскольку простота решения в этом случае полезнее гибкости.

Наблюдатели и утечки памяти

Долгоживущие наблюдаемые объекты являются наиболее распространенной причиной утечки памяти в .NET-приложениях. Поскольку наблюдаемый объект содержит неявную ссылку на все наблюдатели, то, пока он жив, будут жить и они:

```
public class Singleton
{
    private static readonly Singleton _instance = new Singleton();

    public static Singleton Instance { get { return _instance; } }

    public event EventHandler Event;
}

class MemoryLeak
{
    public MemoryLeak()
    {
        Singleton.Instance.Event +=
            (s, e) => Console.WriteLine("Hello, Memory Leak!");
    }
}
```

В этом случае создание объекта `MemoryLeak` гарантированно приведет к утечке памяти, поскольку вновь созданный экземпляр навсегда останется в списке подписчиков события `Singleton.MyEvent`.

Существует несколько решений данной проблемы.

1. Стоит по возможности избегать долгоживущих объектов вообще, а долгоживущих объектов с событиями — в особенности.
3. Наблюдатели могут реализовать интерфейс `IDisposable` и отписываться от событий в методе `Dispose`.
4. Можно воспользоваться слабыми событиями (Weak Event Pattern). Специальной реализацией событий, в которой используются слабые ссылки (Weak References) для управления наблюдателями. В этом случае слабое событие не будет являться корневой ссылкой и не станет препятствовать сборке мусора, когда на наблюдатель не останется других ссылок.

Применимость

Наблюдатель является универсальным механизмом уменьшения связанности в приложении.

При проектировании некоторого класса у разработчика всегда есть несколько вариантов реализации. Класс `A` может знать о существовании класса `B` и уведомлять его о произошедшем событии. Или же класс `A` может быть наблюдаемым и уведомлять о некотором событии всех заинтересованных подписчиков.

Использование наблюдателей уменьшает связанность между классами/модулями и упрощает повторное использование. Не менее важно и то, что наблюдатель четко показывает выходной интерфейс класса (то, что нужно классу для его успешной работы) и позволяет проще думать о нем в изоляции.

Благодаря слабой связанности наблюдаемый объект и наблюдатели могут располагаться на разных уровнях абстракции или слоях приложения. Например, слой пользовательского интерфейса знает о модели, но модель не должна ничего знать о пользовательском интерфейсе. Косвенная связь моделей с верхним уровнем реализуется с помощью наблюдателей: модель уведомляет всех подписчиков об изменении состояния, а пользовательский интерфейс обновляет свое состояние или другим способом реагирует на события.

Наблюдатели очень часто используются в качестве составных частей более сложных паттернов. Семейство паттернов MVx (Model — View — Controller/Presenter — View Model) освобождает модель от необходимости представления именно с помощью наблюдателей. Паттерн «Посредник» очень часто реализуется с помощью наблюдателя и связывает две независимые части системы воедино.

ПРИМЕЧАНИЕ

Подробнее о паттернах MVx и слоях приложения вы узнаете из главы 21.

Примеры в .NET Framework

- Наблюдатели в форме событий. В .NET Framework насчитываются тысячи классов, содержащих события.
- Наблюдатели в форме делегатов. Наблюдатели в форме делегатов часто используются в качестве методов обратного вызова для выполнения дополнительной инициализации (`AppDomainSetup.AppDomainInitializer`, `HttpConfiguration.Initializer`) или в качестве точек расширения (фильтры и селекторы в WPF/Windows Forms/WCF).
- Наблюдатели в форме интерфейсов. Хорошим примером именованного наблюдателя является API для работы с Event Hub — масштабируемой системой обмена сообщениями. Интерфейс `IEventProcessor` содержит методы `CloseAsync`, `OpenAsync` и `ProcessEventsAsync`.

Глава 6

Паттерн «Посетитель» (Visitor)

Назначение: описывает операцию, выполняемую с каждым объектом из некоторой иерархии классов. Паттерн «Посетитель» позволяет определить новую операцию, не изменяя классов этих объектов.

Мотивация

Объектно-ориентированное программирование предполагает единство данных и операций. Обычно классы представляют некоторые операции, скрывая структуры данных, над которыми эти операции производятся. Но не всегда удобно или возможно смешивать их в одном месте.

Структуры данных некоторых предметных областей могут быть довольно сложными, а операции над ними настолько разнообразными, что совмещать эти два мира нет никакого смысла. Например, в мире финансовых инструментов гораздо логичнее отделить данные ценных бумаг от выполняемых над ними операций. Деревья выражений — это еще один классический пример того, где происходит такое разделение. Но даже в простой задаче, такой как экспорт лог-файлов, можно найти места, где такой подход будет более разумным.

Различные виды прочитанных записей формируют простую иерархию наследования (рис. 6.1).

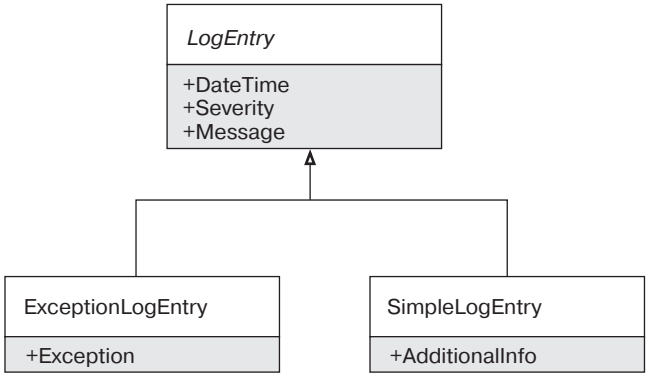


Рис. 6.1. Иерархия классов LogEntry

Разные типы записей могут обрабатываться по-разному. Например, сообщения с исключениями могут сохраняться в другую таблицу базы данных, разные типы сообщений могут иметь разную логику группировки для пакетного сохранения и т. п. Эту логику нельзя поместить прямо в иерархию сообщений, поскольку логика сохранения может изменяться независимо от самих записей. Кроме того, наличие логики сохранения прямо в классе `LogEntry` сделает эту иерархию слишком тяжеловесной. Мы не хотим нарушать принципы «открыт/закрыт» и принцип единой обязанности, а значит, должны вынести подобные аспекты поведения в отдельные классы.

Добавление подобной бизнес-логики вне иерархии чревато дублированием и хрупкостью. Один из вариантов реализации выглядит так (листинг 6.1).

Листинг 6.1. Простой вариант перебора типов LogEntry

```
public class DatabaseLogSaver
{
    public void SaveLogEntry(LogEntry logEntry)
    {
        var exception = logEntry as ExceptionLogEntry;
        if (exception != null)
        {
            SaveException(exception);
        }
```

```
        else
        {
            var simpleLogEntry = logEntry as SimpleLogEntry;
            if (simpleLogEntry != null)
                SaveSimpleLogEntry(simpleLogEntry);

            throw new InvalidOperationException("Unknown log entry type");
        }
    }

    private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}
    private void SaveException(ExcetpionLogEntry exceptionLogEntry) {...}
}
```

Поскольку иерархия `LogEntry` является легковесной, то можно ожидать наличия подобного кода в других местах приложения. Решение заключается в выделении логики «посещения» иерархии классов с помощью паттерна «Посетитель».

Для этого в базовый класс `LogEntry` добавляется абстрактный метод `Accept`, который принимает `ILogEntryVisitor`, а каждый конкретный класс иерархии просто вызывает метод `Visit` (листинг 6.2).

Листинг 6.2. Пример интерфейса посетителя

```
public interface ILogEntryVisitor
{
    void Visit(ExceptionLogEntry exceptionLogEntry);
    void Visit(SimpleLogEntry simpleLogEntry);
}

public abstract class LogEntry
{
    public abstract void Accept(ILogEntryVisitor logEntryVisitor);
    // Остальные члены остались без изменения
}
```

```
public class ExceptionLogEntry : LogEntry
{
    public override void Accept(ILogEntryVisitor logEntryVisitor)
    {
        // Благодаря перегрузке методов выбирается метод
        // Visit(ExceptionLogEntry)
        logEntryVisitor.Visit(this);
    }
}
```

Теперь, если кому-то понадобится добавить операцию над иерархией записей, достаточно будет реализовать интерфейс `ILogEntryVisitor` (листинг 6.3).

Листинг 6.3. Пример использования посетителя

```
public class DatabaseLogSaver : ILogEntryVisitor
{
    public void SaveLogEntry(LogEntry logEntry)
    {
        logEntry.Accept(this);
    }

    void ILogEntryVisitor.Visit(ExceptionLogEntry exceptionLogEntry)
    {
        SaveException(exceptionLogEntry);
    }

    void ILogEntryVisitor.Visit(SimpleLogEntry simpleLogEntry)
    {
        SaveSimpleLogEntry(simpleLogEntry);
    }

    private void SaveException(ExceptionLogEntry logEntry) {...}
    private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}
}
```

Паттерн Посетитель предназначен для добавления новых операций над иерархией типов без ее изменения.

Классическая диаграмма классов паттерна Посетитель представлена на рис. 6.2.

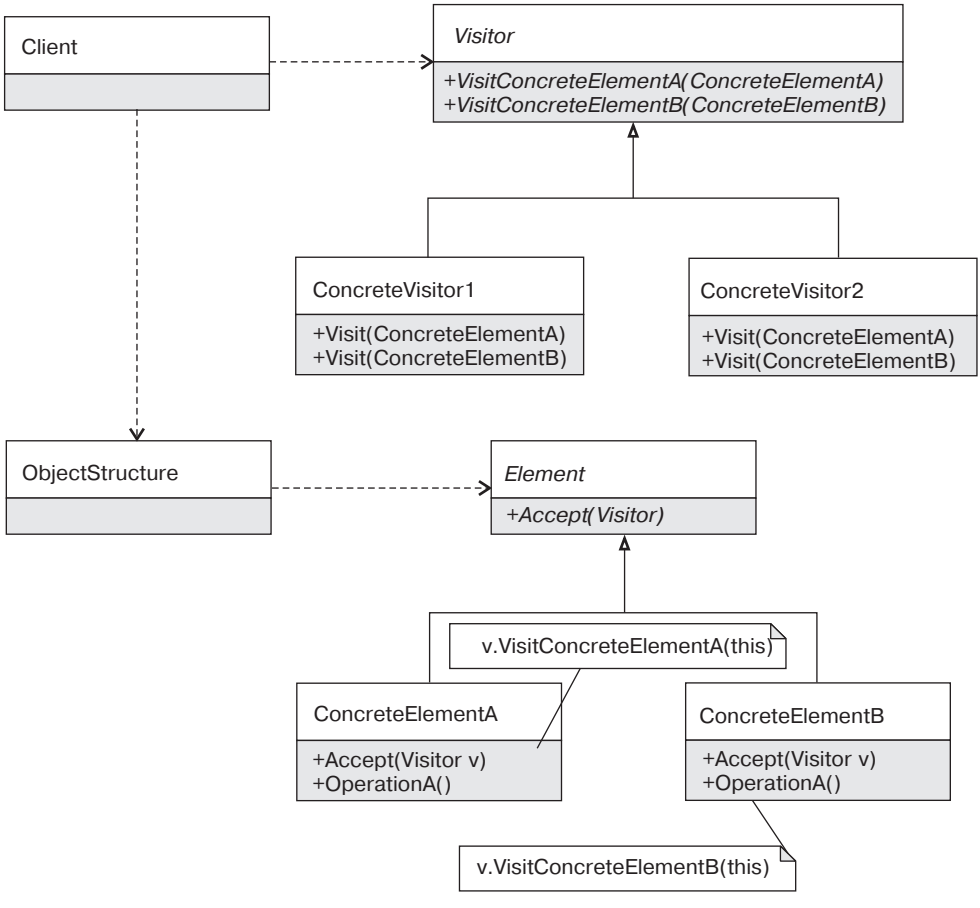


Рис. 6.2. Классическая диаграмма классов паттерна Посетитель

Участники:

- `Visitor (ILogEntryVisitor)` — определяет интерфейс посетителя;
- `Element (LogEntry)` — базовый класс иерархии, для которой нужно добавить новую операцию;
- `Client (DatabaseLogSaver)` — использует посетитель для обработки иерархии элементов.

Обсуждение

Базовый класс иерархии задает семейство операций, поведение которых определяется наследниками. В иерархии фигур класс `Shape` может определять набор допустимых операций, а конкретные классы будут реализовывать все операции для конкретного вида фигур. Класс `Rectangle` будет отвечать за все поведение прямоугольника: рисование, вычисление площади и т. п. Полиморфное использование базовых классов позволяет сделать систему расширяемой, поскольку добавить новый тип в иерархию наследования довольно просто.

В мире функционального программирования данные и функции отделены друг от друга. При этом одна функция может решать одну задачу над множеством типов. В случае иерархии фигур у нас может одна функция `Draw` для рисования всех типов фигур, одна функция `GetArea` для вычисления площади всех типов фигур и т. д.

В объектно-ориентированном решении легко добавлять новый тип иерархии, но сложно добавлять новую операцию. Паттерн «Посетитель» позволяет решить эту проблему. Посетитель позволяет клиентскому коду исследовать иерархию типов и выполнять различные операции в зависимости от конкретного типа объекта.

При этом паттерн «Посетитель» усложняет добавление новых типов в иерархию наследования. Добавление нового типа требует изменения интерфейса `IVisitor` и ломает все его реализации. Это значит, что паттерн «Посетитель» идеально подходит для расширения функциональности стабильных иерархий наследования с переменным числом операций.

ПРИМЕЧАНИЕ

Подробнее расширяемость и роль паттерна «Посетитель» рассмотрена в главе 18.

Классическая реализация паттерна «Посетитель» является довольно сложной. Посещаемая иерархия должна решать дополнительную задачу: реализовывать метод `Accept`. Сами посетители должны использовать наследование и реализовывать интерфейс `IVisitor`, что делает поток исполнения не вполне очевидным.

Функциональная vs. Объектная версия

Когда количество конкретных типов иерархии наследования невелико, интерфейс посетителя можно заменить списком делегатов. Для этого метод `Accept` можно

переименовать в `Match`, который будет принимать несколько делегатов для обработки конкретных типов иерархии.

Метод `Match`[1] для классов иерархии `LogEntry` будет выглядеть так, как показано в листинге 6.4.

Листинг 6.4. Функциональная версия паттерна «Посетитель»

```
public abstract class LogEntry
{
    public void Match(
        Action<ExceptionLogEntry> exceptionEntryMatch,
        Action<SimpleLogEntry> simpleEntryMatch)
    {
        var exceptionLogEntry = this as ExceptionLogEntry;
        if (exceptionLogEntry != null)
        {
            exceptionEntryMatch(exceptionLogEntry);
            return;
        }

        var simpleLogEntry = this as SimpleLogEntry;
        if (simpleLogEntry != null)
        {
            simpleEntryMatch(simpleLogEntry);
            return;
        }

        throw new InvalidOperationException("Unknown LogEntry type");
    }
}
```

Теперь вместо создания специализированного класса посетителя для каждого случая можно просто использовать метод `Match` прямо в коде анализатора (листинг 6.5).

---

[1] Разумно добавить еще одну перегруженную версию метода T Match<T>(), которая будет принимать Func<EceptionLogEntry, T> и Func<SimpleLogEntry, T>.

Листинг 6.5. Пример использования «функционального» посетителя

```
public class DatabaseLogSaver
{
    public void SaveLogEntry(LogEntry logEntry)
    {
        logEntry.Match(
            ex => SaveException(ex),
            simple => SaveSimpleLogEntry(simple));
    }

    private void SaveSimpleLogEntry(SimpleLogEntry logEntry) {...}
    private void SaveException(ExceptionLogEntry exceptionLogEntry) {...}
}
```

ПРИМЕЧАНИЕ

Может показаться, что реализация метода Match на основе приведения типов грубо нарушает принцип «открыт/закрыт». Если перебор типов находится лишь в одном месте в коде, то такой код следует принципу единственного выбора и не нарушает принципа «открыт/закрыт». Подробнее этот вопрос будет рассмотрен в главе 18.

Данный вид посетителя напоминает стандартную технику функционального программирования под названием *«сопоставление с образцом»*[1] (pattern matching). Выбор между функциональной и объектно-ориентированной версией паттерна «Посетитель» такой же, как и выбор между функциональным и объектно-ориентированным паттерном «Стратегия». В некоторых случаях удобнее создавать именованный класс, реализующий сложную стратегию сохранения или шифро-

[1] Сопоставление с образцом является очень мощной конструкцией функциональных языков программирования и позволяет сопоставлять не только типы, но и диапазоны значений, а также деконструировать кортежи и записи. В общем случае сопоставление с образцом можно рассматривать как оператор switch «на стероидах». На данный момент язык C# не поддерживает сопоставление с образцом из коробки. Поэтому разработчику приходится дублировать код или использовать довольно изощренные решения наподобие того, что описал Барт де Смет в статье Pattern Matching in C# — Part 0. Разработчики языка C# рассматривают возможность добавления полноценного сопоставления с образцом в одну из следующих версий языка. Функциональные языки программирования поддерживают сопоставление с образцом из коробки, а также возможность создания «вариантов» (алгебраических типов) более удобным способом.

вания данных, но когда речь заходит о сравнении или сортировке объектов, то стратегия на основе лямбда-выражений будет предпочтительной. В данном случае компромисс аналогичен: для большой иерархии типов придется использовать классический посетитель и именованные классы. Для небольших иерархий наследования и посетителей, которые лишь перенаправляют работу другим методам, вариант с лямбда-выражениями будет проще и понятнее, поскольку читателю не нужно переключать контекст на реализацию посетителя.

В случае использования самописного сопоставления с образцом мы можем добавить несколько перегруженных методов `Match`, которые будут принимать не все возможные типы иерархии, а лишь некоторые наиболее часто используемые. При этом метод `Match` может находиться в базовом классе иерархии наследования, а может быть реализован в виде метода расширения в классе `LogEntryEx`.

Двойная диспетчеризация

Выбор виртуального метода осуществляется на основе типа аргумента во время исполнения. Этот выбор называют одиночной диспетчеризацией. Метод `Accept` является методом с двойной диспетчеризацией: выбор выполняемого метода определяется типами посещаемого объекта и посетителя.

Паттерн «Посетитель» реализует двойную диспетчеризацию, поскольку выбор метода определяется на основе типов двух объектов: посещаемого объекта и посетителя (листинг 6.6).

Листинг 6.6. Пример двойной диспетчеризации

```
LogEntry logEntry = new ExceptionLogEntry();
ILogEntryVisitor visitor = new Visitor1();

// Вызывается Visitor1.Visit(ExcpetionLogEntry)
logEntry.Accept(visitor);

logEntry = new SimpleLogEntry();
visitor = new Visitor2();

// Вызывается Visitor2.Visit(SimpleLogEntry)
logEntry.Accept(visitor);
```

Интерфейс vs. абстрактный класс посетителя

Обычно посетитель определяется интерфейсом `IVisitor`. Такой подход налагает меньше ограничений на клиентов, но делает их более хрупкими. Каждый раз при добавлении типа в иерархию интерфейс посетителя обновляется и в нем появляется новый метод `Visit(YetAnotherType)`.

Использование абстрактного базового класса `VisitorBase` позволяет клиентам посещать лишь нужные типы иерархии, переопределяя лишь нужные методы `Visit`. Также это делает клиентов менее хрупкими, поскольку добавление нового виртуального метода `Visit` не нарушает работу существующих клиентов.

Однако базовый класс налагает более жесткие ограничения на клиентов, поскольку языки платформы .NET не поддерживают множественного наследования классов. Поэтому обычно эти подходы совмещаются и одновременно используется интерфейс `IVisitor` с базовым классом `VisitorBase`.

Проблема же с множественным наследованием обычно решается путем отделения посетителей от основной логики (листинг 6.7).

Листинг 6.7. Использование вложенного класса посетителя

```
public abstract class LogEntryVisitorBase : ILogEntryVisitor
{
    public virtual void Visit(ExceptionLogEntry exceptionLogEntry)
    {}

    public virtual void Visit(SimpleLogEntry simpleLogEntry)
    {}
}

public class DatabaseExceptionLogEntrySaver : LogSaverBase
{
    public void SaveLogEntry(LogEntry logEntry)
    {
        logEntry.Accept(new ExceptionLogEntryVisitor(this));
    }

    private void SaveException(ExceptionLogEntry exceptionLogEntry) {...}
```

```
    private class ExceptionLogEntryVisitor : LogEntryVisitorBase
    {
        private readonly DatabaseExceptionLogEntrySaver _parent;

        public ExceptionLogEntryVisitor(
            DatabaseExceptionLogEntrySaver parent)
        {
            _parent = parent;
        }

        // «Посещаем» лишь ExceptionLogEntry
        public override void Visit(ExceptionLogEntry exceptionLogEntry)
        {
            _parent.SaveException(exceptionLogEntry);
        }
    }
}
```

В данном случае `DatabaseExceptionLogEntrySaver` не может сам наследовать от `LogEntryVisitorBase`, поскольку является частью другой иерархии типов. Обойти это ограничение можно с помощью внутреннего типа, который будет наследником от `LogEntryVisitorBase` и переопределит лишь один виртуальный метод для обработки записей с исключениями.

Базовые классы посетителей полезны также при использовании древовидных структур, таких как деревья выражений. В этом случае базовый класс может содержать логику навигации по составной структуре данных.

Применимость

Использовать посетитель нужно лишь тогда, когда появляется необходимость разделить иерархию типов и набор выполняемых операций. Паттерн «Посетитель» позволит легко разбирать составную иерархическую структуру и обрабатывать разные типы узлов особым образом.

- Использовать паттерн «Посетитель» нужно тогда, когда набор типов иерархии стабилен, а набор операций — нет.

- Классический вариант паттерна лучше всего подходит для больших составных иерархий и когда заранее не известно, какие типы будут посещаться чаще других.
- Функциональный вариант посетителя всегда можно построить на основе классической реализации, когда станет известно, что многим клиентам нужно посещать лишь небольшое число типов иерархии. Вариант на основе делегатов в самостоятельном виде подходит лишь для небольшой иерархии типов. По сути, он является простой реализацией сопоставления с образцом и подходит для работы с простыми иерархиями типов, которые моделируют размеченные объединения. Также этот вариант может быть добавлен с помощью методов расширения для существующих иерархий.

Примеры в .NET Framework

- `ExpressionTreeVisitor` используется для работы с деревьями выражений (Expression Trees) в .NET Framework. Данный посетитель используется для навигации и преобразования деревьев выражений при реализации специализированных LINQ-провайдеров, а также для решения других задач.
- `Roslyn` содержит множество посетителей. `CSharpSyntaxVisitor` предназначен для работы с синтаксическим деревом, `SymbolVisitor<TResult>` — для работы с символами и др.
- `DbExpressionVisitor` используется в Entity Framework для SQL-выражений.

Глава 7

Другие паттерны поведения

В предыдущих главах были рассмотрены шесть ключевых паттернов поведения. Каждый из них применяется ежедневно в подавляющем числе проектов и оказывает значительное влияние на дизайн приложения.

Без **«Стратегии»** не было бы возможности заменить алгоритм во время исполнения. Без **«Шаблонного метода»** было бы сложно использовать повторно базовые классы. **«Посредник»** прячет способ взаимодействия более простых объектов и делает их менее зависимыми друг от друга. Без **«Итераторов»** было бы невозможно писать обобщенные алгоритмы, работающие с множеством коллекций. **«Наблюдатель»** позволяет полностью упразднить явную связь между классом и подписчиками. **«Посетитель»** делает возможным функциональный дизайн и позволяет добавлять новое поведение в иерархии классов без их модификации.

Более детальное рассмотрение лишь нескольких паттернов не значит, что другие паттерны поведения неважны, просто они либо реже используются, либо не оказывают столь сильного влияния на дизайн типичного приложения.

Паттерн «Команда»

Назначение: инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Паттерн «Команда» позволяет спрятать действие в объекте и отвязать источник этого действия от места его исполнения. Классический пример — проектирование пользовательского интерфейса. Пункт меню не должен знать, что происходит при его активизации пользователем, он должен знать лишь о некотором действии, которое нужно выполнить при нажатии кнопки.

Классическая диаграмма паттерна «Команда» представляет собой интерфейс `ICommand` с методом `Execute`, который может принимать необязательный контекст исполнения (рис. 7.1).

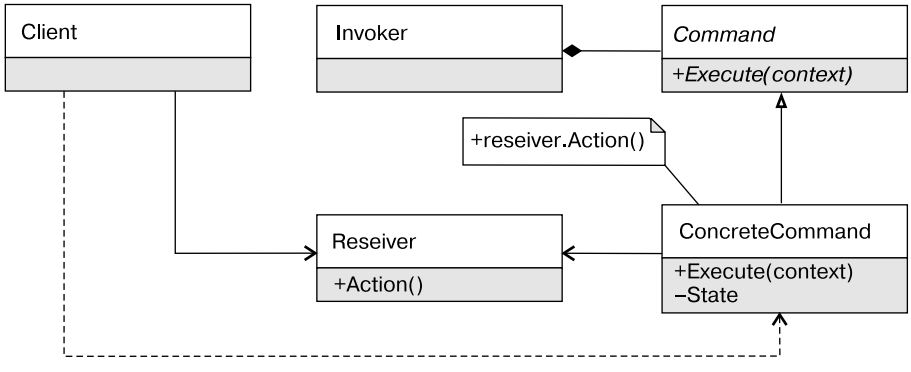


Рис. 7.1. Диаграмма классов паттерна «Команда»

`Invoker` (пункт меню) во время создания получает команду, которая затем делегирует свою работу объекту класса `Receiver` (объекту `ViewModel`).

Существует две основные реализации команды.

- *Самостоятельная команда* может выполнять простую операцию самостоятельно. Например, команда `SimplifyBinaryConditionCommand` может самостоятельно упростить выражение из двух аргументов.
- *Делегирующая команда* не выполняет операцию самостоятельно, но знает, кто это может сделать. Классическая диаграмма паттерна «Команда» описывает именно этот вид реализации.

Помимо объектно-ориентированной версии, в языке C# очень распространено использование функциональной версии паттерна «Команда» на основе делегатов. С помощью анонимных методов легко получить делегат, который будет захватывать внешний контекст и выполнять требуемое действие. Подобный вид команд применяется очень часто, и мало кто задумывается о том, что при этом используется один из классических паттернов проектирования (листинг 7.1).

Листинг 7.1. Пример паттерна «Команда»

```
public class LogExporterViewModel
{
    public Action GetExportLogsCommand()
    {
        return () => _logSaver.Save(LogEntries);
    }

    public IEnumerable<LogEntry> LogEntries {get; private set;}
}
```

Популярность команд на основе делегатов привела к появлению особой реализации интерфейса `ICommand` — `RelayCommand`, которая принимает в качестве аргумента конструктор `Func<T>` или `Action`[1].

Примеры в .NET Framework:

- `ICommand` в WPF, на основе которых строится привязка операций к событиям пользовательского интерфейса;
- `IDbCommand` в ADO.NET инкапсулирует операцию, исполняемую на стороне СУБД;
- Объект класса `Task<T>` принимает делегат `Func<T>`, который можно рассматривать в виде команды, которая будет исполнена в будущем для получения результата задачи.

Паттерн «Состояние»

Назначение: позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Некоторые объекты предметной области могут выполнять разные наборы операций в зависимости от состояния. Объект класса `Socket` может выполнить операцию `Connect`, только если соединение еще не установлено, а объект потока (класса

[1] Подробнее о командах в WPF и паттерне MVVM можно почитать в классической статье Джоша Смита «Приложения WPF с шаблоном проектирования модель — представление — модель представления» (MSDN Magazine, 2009. — Февраль).

Thread) может быть запущен, только если он не был запущен ранее. Подобные классы моделируют конечный автомат с набором допустимых переходов.

Паттерн «Состояние» предполагает выделение базового класса или интерфейса для всех допустимых операций и наследника для каждого возможного состояния (рис. 7.2).

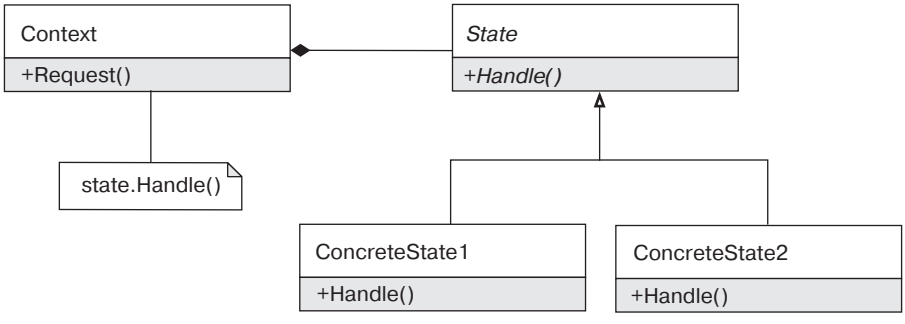


Рис. 7.2. Диаграмма классов паттерна «Состояние»

Контекст (Socket) делегирует операции по переходу между состояниями объектам State, что позволяет перейти из состояния ConcreteState1 (NotConnected) в новое состояние ConcreteState2 (Connected) при вызове метода Handle (метода Connect).

Паттерн «Состояние» в полноценном виде довольно редко применяется на практике. Его применимость определяется сложностью конечного автомата. Например, для сложной логики анализа состояния тревожной кнопки охранной системы такой подход оправдан, поскольку переход из одного состояния в другое определяется десятком условий. Для более простых случаев, таких как управление соединением с удаленным сервером, обычно достаточно спрятать конечный автомат в отдельном классе, но не выделять отдельный класс на каждое состояние. В еще более простых случаях, таких как управление состоянием класса Thread, разумнее всего поместить логику конечного автомата прямо в класс Thread и не выносить ее отдельно.

Примеры использования. В .NET Framework конечные автоматы применяются в огромном числе классов, но классической реализации паттерна «Стратегия» среди открытых (public) типов .NET Framework я не нашел.

- CommunicationObject реализует конечный автомат перехода между состояниями WCF клиента: Created, Opening, Opened, Closing, Closed и Faulted.

- `Task` реализует конечный автомат перехода между состояниями задачи: `Created`, `WaitingForActivation`, `WaitingToRun`, `Running`, `RunToCompletion`, `Canceled`, `Faulted`.

Паттерн «Цепочка обязанностей»

Назначение: позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

«Цепочка обязанностей» является довольно распространенным паттерном в .NET Framework, хотя не все знают, что часто пользуются им. Цепочка обязанностей — это любое событие, аргументы которого позволяют уведомить инициатора, что событие обработано с помощью метода `Handle()` или путем установки свойства `Handled` в `True`.

В случае закрытия формы в Windows Forms генерируется событие `Closing`, подписчики которого могут отложить закрытие окна в случае необходимости (листинг 7.2).

Листинг 7.2. Пример паттерна «Цепочка обязанностей»

```
public DialogForm()
{
    this.Closing += HandleClosing;
}

private void HandleClosing(object sender, CancelEventArgs ea)
{
    if (UserDontWantToCloseTheWindow())
    {
        ea.Cancel = true;
    }
}
```

Поскольку делегаты в .NET могут содержать более одного обработчика, то на их основе цепочка обязанностей строится простым и элегантным образом. Для этого событие возбуждается не с помощью метода `Invoke`, а вручную для каждого подписчика (листинг 7.3).

Листинг 7.3. Псевдокод реализации цепочки обязанностей

```
// Form.cs
private void ClosingForm()
{
    if (Closing != null)
    {
        var eventArgs = new CancelEventArgs(cancel: false);
        var invocationList = Closing.GetInvocationList();
        foreach (EventHandler<CancelEventArgs> handler in invocationList)
        {
            handler(this, eventArgs);
            if (eventArgs.Cancel)
            {
                // Отменяем закрытие формы
                return;
            }
        }
    }

    CloseForm();
}
```

Диаграмма классов паттерна «Цепочка обязанностей» приведена на рис. 7.3.

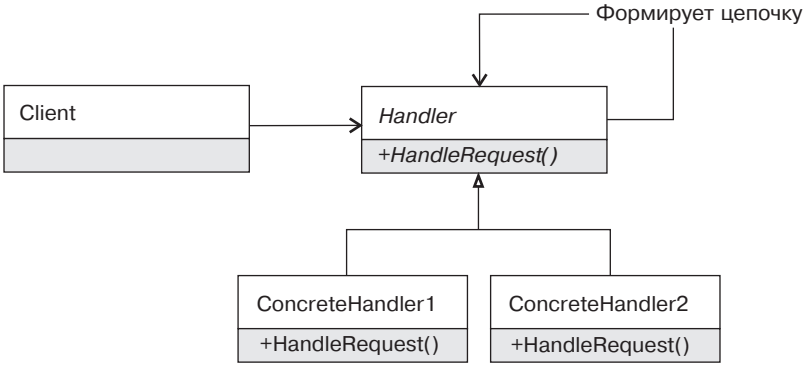


Рис. 7.3. Диаграмма классов паттерна «Цепочка обязанностей»

Примеры в .NET Framework:

- событие `Closing` в Windows Forms с возможностью отмены закрытия формы путем установки свойства `Cancel` аргумента `CancelEventArgs` в `True`;
- событие `TaskScheduler.UnobservedException`, обработчик которого может уведомить планировщик, что необработанное исключение задачи не является критическим, путем вызова метода `UnobservedTaskExceptionEventArgs.SetObserved`. Для аналогичных целей используются события `Contract.ContractFailed` и `Application.DispatcherUnhandledException`.

Часть II

Порождающие паттерны

- Глава 8. Паттерн «Синглтон» (Singleton)
- Глава 9. Паттерн «Абстрактная фабрика» (Abstract Factory)
- Глава 10. Паттерн «Фабричный метод» (Factory Method)
- Глава 11. Паттерн «Строитель» (Builder)

В большинстве объектно-ориентированных языков программирования за конструирование объекта отвечает конструктор. В самом простом случае клиент знает, какого типа объект ему требуется, и создает его путем вызова соответствующего конструктора. Но в некоторых случаях тип объекта может быть неизвестен вызывающему коду или процесс конструирования может быть настолько сложным, что использование конструктора будет неудобным или невозможным. Порождающие паттерны предназначены для решения типовых проблем создания объектов.

В некоторых случаях разработчик хочет гарантировать, что будет создан лишь один экземпляр некоторого класса. Иногда такая потребность диктуется спецификацией системы, в которой сказано, что в системе должен быть лишь один считыватель мыслей пользователя. Но гораздо чаще такой подход применяется из-за того, что он обеспечивает глобальную точку доступа к некоторому объекту. Это позволяет любому объекту получить доступ к любой точке системы, что избавляет разработчика от необходимости продумывать обязанности классов и выделять их зависимости.

Класс, который гарантирует создание лишь одного экземпляра и предоставляет глобальную точку доступа к нему, известен как паттерн «Синглтон». Это самый знаменитый паттерн проектирования, недостатки которого ставят под сомнение его полезность. Использование глобальных объектов с изменяемым состоянием вызывает эффект бабочки, когда поведение системы начинает зависеть от порядка вызова методов в разных модулях и практически не поддается тестированию и сопровождению.

Но не все порождающие поведения столь сомнительны.

Возрастающая сложность системы часто приводит к необходимости изоляции процесса создания объектов. Бывает полезно скрыть от вызывающего кода конкретный тип создаваемого объекта, чтобы иметь возможность изменить его в будущем. В других случаях тип создаваемого объекта зависит от аргументов метода, что также делает невозможным использование конструкторов. А иногда процесс конструирования должен контролироваться наследником, что приводит к появлению стратегии создания. Так мы приходим ко второму по популярности паттерну проектирования — к фабрикам.

Существует несколько разновидностей фабрик. *«Абстрактная фабрика»* предназначена для создания семейства объектов и позволяет заменять это семейство путем использования нужного подкласса фабрики. Статический *«Фабричный метод»* представляет собой статический метод, который возвращает экземпляр кон-

кретного или полиморфного класса в зависимости от аргументов метода или конфигурации. Классический фабричный метод является стратегией конструирования объектов и обеспечивает гибкость за счет полиморфного использования.

Фабрики прекрасно справляются с инкапсуляцией процесса создания и обеспечивают гибкость за счет наследования. Но иногда клиентам требуется гибкость иного рода. Процесс создания некоторых объектов состоит из множества этапов, и лишь потребителю известно, какие этапы обязательны, а какие — нет. Паттерн *«Строитель»* позволяет клиентам собирать сложный объект по кусочкам, не вдаваясь в подробности того, как из этих кусочков получается окончательный результат. «Строитель» отлично подходит для создания тестируемых классов и тестовых данных, но активно применяется и в логике приложения.

В этой части книги мы очень подробно рассмотрим порождающие паттерны. Эти паттерны весьма просты по своей природе, но обладают множеством нюансов с точки зрения реализации и влияния на дизайн приложения.

Глава 8

Паттерн «Синглтон» (Singleton)

Назначение: гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему.

Другими словами: синглтон эмулирует глобальные переменные в объектно-ориентированных языках программирования.

Мотивация

Практически в любом приложении возникает необходимость в глобальных переменных или объектах с ограниченным числом экземпляров. Даже в таком простом приложении, как импорт логов, может возникнуть необходимость в логировании. И самый простой способ решить эту задачу — создать глобальный объект, который будет доступен из любой точки приложения.

По своему определению синглтон гарантирует, что у некоего класса есть лишь один экземпляр. В некоторых случаях анализ предметной области строго требует, чтобы класс существовал лишь в одном экземпляре. Однако на практике паттерн «Синглтон» обычно используется для обеспечения доступа к какому-либо ресурсу, который требуется разным частям приложения.

Диаграмма классической реализации паттерна «Синглтон» приведена на рис. 8.1.

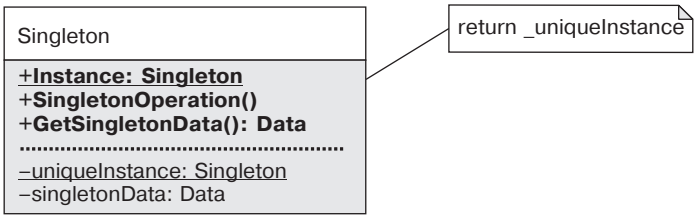


Рис. 8.1. Диаграмма классической реализации паттерна «Синглтон»

Варианты реализации в .NET

В оригинальном описании паттерна «Синглтон» «бандой четырех» на его реализацию не накладывались никакие ограничения, однако на практике (в частности, для платформы .NET) любая реализация должна отвечать двум требованиям:

- в многопоточной среде должна обеспечиваться возможность доступа к синглтону;
- должна обеспечиваться «ленивость»[1] создания синглтона.

Потокобезопасность является необходимым свойством, поскольку представить себе реальное однопоточное .NET-приложение довольно сложно. «Ленивость» же является скорее желательным свойством реализации.

ПРИМЕЧАНИЕ

Все приведенные далее реализации потокобезопасны с точки зрения количества экземпляров, когда при первом одновременном доступе к синглтону из разных потоков мы не получим лишних экземпляров. Для обеспечения же потокобезопасного использования сам объект синглтона (то есть все его экземплярные методы) должен быть потокобезопасным. Это одна из причин того, почему настоятельно не рекомендуется использовать синглтоны с изменяемым состоянием, обеспечить потокобезопасность которых становится сложнее.

Реализация на основе Lazy of T

Это самая простая реализация (листинг 8.1), отвечающая исходным требованиям (потокобезопасности и «ленивости»).

[1] «Ленивость» создания подразумевает, что экземпляр потенциально дорогостоящего класса будет создан лишь перед первым использованием.

Листинг 8.1. Реализация на основе Lazy<T>

```
public sealed class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    LazySingleton() {}

    public static LazySingleton Instance { get { return _instance.Value; } }
}
```

Она настолько проста, что устраняет необходимость в обобщенных решениях типа `Singleton<T>`, поскольку требует минимального количества усилий.

Достоинства: простота + потокобезопасность + «ленивость»!

Недостаток: доступна только в .NET 4.0+.

Блокировка с двойной проверкой

Эта реализация также отвечает критериям потокобезопасности и «ленивости» (листинг 8.2), но она существенно сложнее предыдущей.

Листинг 8.2. Реализация на основе блокировки с двойной проверкой

```
public sealed class DoubleCheckedLock
{
    // Поле должно быть volatile!
    private static volatile DoubleCheckedLock _instance;
    private static readonly object _syncRoot = new object();

    DoubleCheckedLock()
    {}

    public static DoubleCheckedLock Instance
    {
        get
        {
```

```
            if (_instance == null)
            {
                lock (_syncRoot)
                {
                    if (_instance == null)
                    {
                        _instance = new DoubleCheckedLock();
                    }
                }
            }
            return _instance;
        }
    }
}
```

Для того чтобы решение было корректным с точки зрения многопоточности, необходимо, чтобы поле `_instance` было помечено ключевым словом `volatile`! Без этого вполне возможна ситуация, когда другой поток, обращающийся к свойству `Instance`, получит доступ к частично валидному экземпляру синглтона, конструирование которого еще не завершено.

Причина такого поведения заключается в следующем. Создание экземпляра с помощью `_instance = new DoubleCheckedLock();` не является атомарной операцией, а состоит (упрощенно) из следующих этапов.

1. Выделение памяти в управляемой куче.
2. Конструирование объекта по указанному адресу (вызов конструктора).
3. Инициализация поля `_instance`.

Если поле `_instance` не помечено ключевым словом `volatile`, то компилятор имеет право изменить порядок этих операций. И если он поменяет 2-й и 3-й шаги местами для одного потока, то другой поток вполне сможет «увидеть», что поле `_instance` уже проинициализировано, и начнет использовать еще не сконструированный объект[1] (рис. 8.2).

---

[1] Подробнее о порядке конструирования объектов можно прочитать в разделе Publication via Volatile Field статьи Игоря Островского The C# Memory Model in Theory and Practice (MSDN Magazine, 2012. — Декабрь).

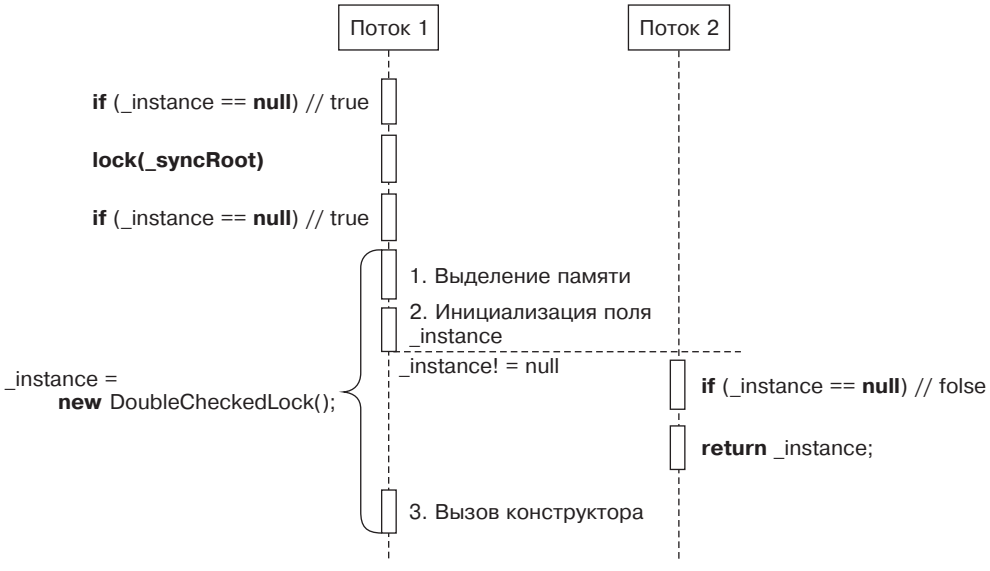


Рис. 8.2. Конкурентный порядок конструирования экземпляра

Достоинства:

- потокобезопасность + «ленивость»;
- распространенность решения;
- доступна под любой версией .NET Framework.

Недостатки:

- сложность + потенциальная хрупкость;
- большой объем кода.

Реализация на основе инициализатора статического поля

Почти «ленивая» реализация показана в листинге 8.3.

Листинг 8.3. Реализация на основе статического поля

```
public sealed class FieldInitSingleton
{
    // Как вариант, можно перенести инициализацию синглтона прямо
    // в статический конструктор
```

```
    private static readonly FieldInitSingleton _instance =
        new FieldInitSingleton();

    FieldInitSingleton() {}

    // Добавление явного статического конструктора приказывает компилятору
    // не помечать тип атрибутом beforefieldinit
    static FieldInitSingleton() {}

    public static FieldInitSingleton Instance { get { return _instance; } }
}
```

Полностью «ленивая» реализация показана в листинге 8.4.

Листинг 8.4. Реализация на основе внутреннего класса

```
public sealed class LazyFieldInitSingleton
{
    private LazyFieldInitSingleton() {}

    public static LazyFieldInitSingleton Instance
    {
        get { return SingletonHolder._instance; }
    }

    // Именно вложенный класс делает реализацию полностью «ленивой»
    private static class SingletonHolder
    {
        public static readonly LazyFieldInitSingleton _instance =
            new LazyFieldInitSingleton();

        // Пустой статический конструктор уже не нужен, если мы будем
        // обращаться к полю _instance лишь из свойства Instance
        // класса LazyFieldSingleton
    }
}
```

Использование открытого статического поля показано в листинге 8.5. В некоторых случаях вместо свойства можно воспользоваться неизменяемым статическим полем (`readonly field`).

Листинг 8.5. Реализация на основе статического поля

```
class FieldBasedSingleton
{
    public static readonly FieldBasedSingleton Instance =
            new FieldBasedSingleton();
}
```

У всех представленных реализаций есть несколько важных особенностей.

1. Обработка исключений. Поскольку инициализация синглтона происходит в статическом конструкторе, то в случае генерации исключения все клиенты получат его «завернутым» в `TypeInitializationException`. И, в отличие от предыдущих реализаций, попытка инициализации синглтона будет лишь одна.
2. Время создания синглтона. Если «забыть» пустой статический конструктор, то время инициализации синглтона станет недетерминированным. Если у типа не определен статический конструктор явно, то компилятор помечает тип атрибутом `beforeFieldInit`, что позволит вызвать сгенерированный статический конструктор отложенным (relaxed) образом задолго до первого обращения к синглтону.

   Так, в примере в листинге 8.6 (при запуске в релизе и без подключенного отладчика) синглтон будет проинициализирован еще до вызова метода `Main`, даже если условие не будет выполняться во время исполнения.

Листинг 8.6. Пример инициализации синглтона до вызова метода

```
static void Main(string[] args)
{
    Console.WriteLine("Starting Main...");
    if (args.Length == 1)
    {
        var s = SingletonWithoutStaticCtor.Instance;
    }
    Console.ReadLine();
}
```

ПРИМЕЧАНИЕ

Причина такого поведения заключается в том, что это дает среде исполнения дополнительные возможности оптимизации. Более детальное описание тонкостей работы статических конструкторов, а также проблемы, к которым это может привести, выходят за рамки данной книги. Об этих особенностях можно почитать в моих статьях, ссылки на которые приведены в конце главы.

3. «Ленивость». Эта реализация не полностью «ленива». Инициализация такого синглтона происходит во время вызова статического конструктора, который может быть вызван не только при использовании синглтона, но и при обращении к статическому члену этого класса.

Достоинство: относительная простота реализации.

Недостатки:

- особенности генерации исключений;
- возможные проблемы с «ленивостью» (без вложенного класса);
- проблемы с временем инициализации при отсутствии статического конструктора.

Какую реализацию выбрать? Помочь сделать это может рис. 8.3.

Обсуждение паттерна «Синглтон»

В ответ на вопрос «Какие паттерны проектирования вы знаете?» семь человек из десяти первым назовут синглтон, двое назовут его вторым, после фабрики, а оставшийся скажет, что не знает никаких паттернов проектирования. Это самый обсуждаемый и, наверное, самый коварный паттерн проектирования, у которого есть масса особенностей реализации как с технической точки зрения, так и с точки зрения дизайна.

Singleton vs. Ambient Context

В оригинальном описании паттерна «Синглтон» «бандой четырех» дается ряд особенностей реализации, на которые не всегда обращают внимание. Одна из них звучит так: синглтон допускает уточнение операций и представления. От класса `Singleton` можно порождать подклассы, и приложение легко можно сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время исполнения.

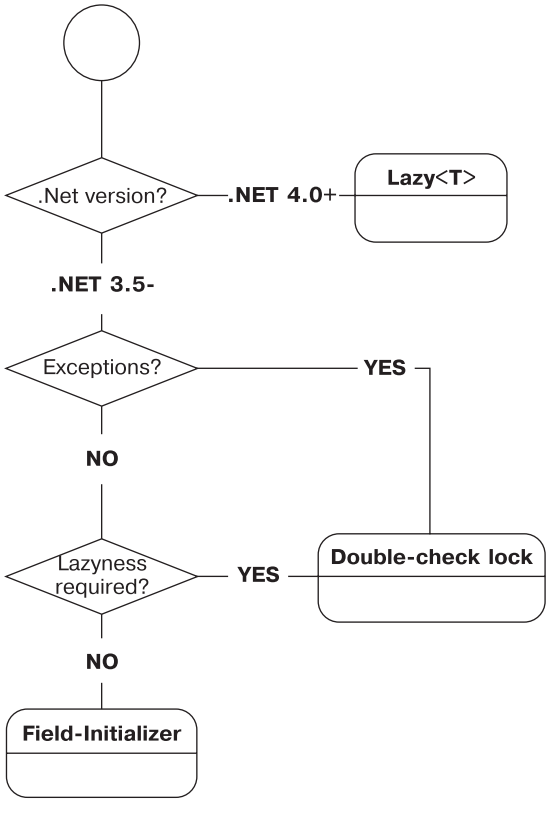


Рис. 8.3. Выбор реализации паттерна «Синглтон»

Главными недостатками синглтонов считаются невозможность юнит-тестирования классов, которые пользуются услугами синглтона, и низкая гибкость. Но если воспользоваться этой оговоркой и дать возможность приложению устанавливать нужный экземпляр синглтона, то многие недостатки исчезнут сами собой. По сути, использование данного аспекта приводит к вариации синглтона, называемой Ambient Context[1] (листинг 8.7).

Основная суть такой модификации состоит в том, что статическое свойство `Instance` вместо возврата конкретного класса возвращает экземпляр абстрактного класса или интерфейса. Также появляется `setter` свойства, который позволяет установить нужный экземпляр синглтона при старте приложения, во время смены контекста или во время инициализации юнит-тестов. В результате решение

[1] Ambient Context очень хорошо описан в книге Марка Сиимана The Dependency Injection in .NET. Я также описывал этот паттерн в статье «Инверсия зависимостей на практике» в разделе Ambient Context: http://bit.ly/AmbientContext.

будет более гибким и тестируемым, чего так не хватает классической реализации паттерна.

Листинг 8.7. Пример паттерна Ambient Context

```
public interface ILogger
{
    void Write();
}

internal class DefaultLogger : ILogger
{
    public void Write() {}
}

public class GlobalLogger
{
    private static ILogger _logger = new DefaultLogger();
    // Классы этой сборки (или друзья) смогут задать
    // нужный экземпляр логера
    public static ILogger Logger
    {
        get { return _logger; }
        internal set { _logger = value; }
    }
}
```

Иногда достаточно, чтобы клиенты синглтона могли сконфигурировать и установить глобальное значение, даже если оно и не представлено интерфейсом или абстрактным классом.

Вместо интерфейса свойство `Instance` может возвращать экземпляр абстрактного класса или даже экземпляр «незапечатанного» (non-sealed) конкретного класса, поведение которого может быть переопределено клиентами. В этом случае конструктор класса `SingletonBase` должен быть не закрытым, как обычно, а защищенным.

Синглтон решает две задачи: гарантирует наличие одного экземпляра класса и обеспечивает глобальную точку доступа. Данный вариант паттерна «Синглтон»

не гарантирует наличия одного экземпляра, а лишь обеспечивает глобальную точку доступа к некоторой зависимости. Ambient Context хорошо подходит для использования зависимости (обычно инфраструктурной) разными слоями приложения, делая эти слои менее зависимыми друг от друга. При этом определенная гибкость обеспечивается за счет возможности установить глобальное состояние при старте приложения.

Singleton vs. Static Class

Альтернативой паттерну «Синглтон» в объектно-ориентированном мире является использование класса с исключительно статическими членами. Синглтон явно обладает большей гибкостью, но статическими функциями проще пользоваться. Какой же из двух подходов выбрать?

Можно предложить следующее эмпирическое правило: при отсутствии состояния и наличии небольшого числа операций статические методы являются более подходящим решением. Если же глобальный объект обладает состоянием, то реализация на основе паттерна «Синглтон» будет проще.

Существует компромиссное решение: статический класс с небольшим набором методов может выполнять роль фасада[1] над реализацией на основе синглтона. `ThreadPool.QueueUserWorkItem` является хорошим примером такого подхода.

Особенности и недостатки

«Синглтон» — это самый критикуемый паттерн, описанный «бандой четырех», главный недостаток которого кроется в его определении: *синглтон гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа*. Это определение состоит из двух частей, и каждая из них несет в себе потенциальную опасность.

Проблема № 1: «Синглтон гарантирует, что у класса есть только один экземпляр…»

Хотя в классическом описании паттерна говорится, что синглтон прячет от пользователя количество экземпляров и мы всегда сможем добавить создание еще нескольких экземпляров, на практике сделать это оказывается сложно. Поскольку приложение завязано на определенное статическое свойство (например, `Instance`), то

[1] Паттерн «Фасад» будет рассмотрен в части III.

попытка добавить еще один экземпляр путем добавления нового статического свойства (`AnotherInstance`?) будет выглядеть нелепо, а попытка создать параметризованный метод типа `GetInstance(name)` поломает весь существующий код.

На самом деле бизнес-логика очень редко накладывает жесткие ограничения на количество экземпляров класса. Обычно это наши с вами уловки и попытки оправдать ошибки дизайна: легче связать несколько кусков системы с помощью синглтонов, вместо того чтобы изменить дизайн и передать классам лишь нужные зависимости.

Часто в приложении применяются несколько синглтонов, которые используют друг друга по цепочке (рис. 8.4).

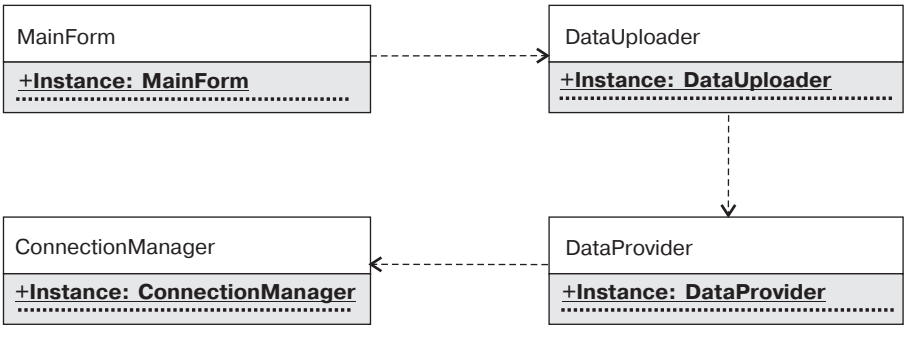


Рис. 8.4. Типичная диаграмма приложения, использующего несколько синглтонов

Данный дизайн легко упростить, заменив его композицией и созданием корневого объекта (`DataUploader`) в корне приложения — в классе `Program`.

Проблема № 2: «… и предоставляет глобальную точку доступа»

Главная же проблема паттерна «Синглтон» заключается в том, что синглтон по своему определению является глобальной переменной со всеми ее недостатками.

- **Необходимость конструктора по умолчанию.** В большинстве случаев создание экземпляра синглтона происходит «ленивым» образом, а это значит, что классу синглтона требуется конструктор по умолчанию. Это, в свою очередь, приводит к тому, что синглтону нельзя передать требуемые зависимости и он будет использовать другие глобальные объекты. В результате легко прийти к дизайну приложения, состоящего из набора глобальных объектов.
- **Неявные зависимости.** Самый простой способ определить сложность класса — проанализировать список полей и аргументов конструктора. Если полей

много и/или конструктор принимает слишком большое число аргументов (больше четырех?!), то это свидетельствует: класс сложный и с его дизайном что-то не так. Но что, если класс не содержит полей и не принимает никаких зависимостей через конструктор, но использует несколько синглтонов? Тогда, чтобы понять его сложность, придется проанализировать все закрытые методы.

- **Состояние.** Синглтон с изменчивым состоянием является источником очень коварных ошибок. Внесение изменений в одну часть системы может изменить работу произвольного числа модулей, у которых были определенные предположения относительно состояния синглтона. Синглтон может обладать невидимым состоянием, например кэшированием, но полноценной изменяемости нужно избегать всеми силами.

Применимость: паттерн или антипаттерн

Уже ни для кого не секрет, что количество недостатков у синглтона таково, что можно считать его не столько паттерном, сколько антипаттерном. Бездумное и бесконтрольное его использование однозначно приведет к проблемам сопровождения, но это не значит, что у него нет сферы применения.

- **Синглтон без видимого состояния.** Нет ничего смертельного в использовании синглтона, через который можно получить доступ к стабильной справочной информации или некоторым утилитам.
- **Настраиваемый контекст.** Аналогично нет ничего смертельного в протаскивании инфраструктурных зависимостей в виде Ambient Context, то есть в использовании синглтона, возвращающего абстрактный класс или интерфейс, который можно установить в начале приложения или при инициализации юнит-теста.
- **Минимальная область использования.** Ограничьте использование синглтона минимальным числом классов/модулей. Чем меньше у синглтона прямых пользователей, тем легче будет от него избавиться и перейти на более продуманную модель управления зависимостями. Помните, что чем больше у классов пользователей, тем сложнее его изменить. Если уж вы вынуждены использовать синглтон, возвращающий бизнес-объект, то пусть лишь несколько высокоуровневых классов-медиаторов используют синглтоны напрямую и передают его экземпляр в качестве зависимостей классам более низкого уровня.
- **Сделайте использование синглтона явным.** Если передать зависимость через аргументы конструктора не удается, то сделайте использование синглтона яв-

ным. Вместо обращения к синглтону из нескольких методов сделайте статическую переменную и проинициализируйте ее экземпляром синглтона (листинг 8.8).

Листинг 8.8. Предпочтительное использование синглтонов

```
// Внутри класса, который использует синглтон
// Теперь тот факт, что мы используем синглтон, становится явным
private static IRepository _repository = Repository.Instance;
```

В этом случае, по крайней мере, всем будет очевидно, что с дизайном что-то не так и нужно его менять.

Примеры в .NET Framework

- Классические синглтоны. Большинство классических реализаций паттерна «Синглтон» в составе .NET Framework являются внутренними типами: `System.ServiceModel.Dispatcher.PeerValidationBehavior.Instance`, `System.Net.NetworkingPerfCounters.Instance`, `System.Threading.TimerQueue.Instance` и т. д. Довольно редкими примерами использования синглтонов в открытой части библиотек являются `SystemClock.Instance` из библиотеки NodaTime и `SqlClientFactory.Instance` в .NET Framework.
- Фасады в виде статических методов. Примерами такого подхода могут служить `ThreadPool.QueueUserWorkItem`, `log4net.LogManager.GetLogger` и т. д.
- Примеры конфигурируемых синглтонов (так называемые Ambient Context). А вот примеров облегченных синглтонов на удивление много: `AppDomain.CurrentDomain`, `Thread.CurrentThread`, `SynchronizationContext.Current`, `TaskSceduler.Defult`, `Form.ActiveForm`, `HttpContext.Current`, `OperationContext.Current` и т. д.

Глобальный контекст часто применяется и в других библиотеках: ASP.NET MVC — `ControllerBuilder.Current`, Entity Framework — `Database.SetInitializer`.

Приведенные примеры показывают, что синглтоны в чистом виде в .NET Framework и вообще в библиотеках практически не применяются — они присутствуют в виде деталей реализации, но очень редко «торчат наружу» в виде открытых классов. Это вполне логично, поскольку чистыми синглтонами тяжело пользоваться и они

делают дизайн чрезмерно жестким. Поэтому классические синглтоны обычно прячутся за фасадными классами (типа `ThreadPool.QueueUserWorkItem`) или же используется изменяемая версия синглтона (в виде Ambient Context) для протаскивания некоторых зависимостей через разные уровни фреймворка или приложения.

Дополнительные ссылки

- *Skeet Jon*. Implementing the Singleton Pattern in C#. — http://csharpindepth.com/articles/general/singleton.aspx.
- Programming Stuff: «О синглтонах и статических конструкторах». — http://bit.ly/SingletonsAndStaticCtor.
- Programming Stuff: «О времени вызова статических конструкторов». — http://bit.ly/StaticCtors.
- Programming Stuff: Цикл статей об управлении зависимостями. — http://bit.ly/DependencyManagement.
- *Островский И*. The C# Memory Model in Theory and Practice. — MSDN Magazine, 2012. — Декабрь.

Глава 9

Паттерн «Абстрактная фабрика» (Abstract Factory)

Фабрика — это второй по популярности паттерн после паттерна «Синглтон». Существуют две классические разновидности фабрик: «Абстрактная фабрика» и «Фабричный метод», предназначенные для инкапсуляции создания объекта или семейства объектов. На практике очень часто отходят от классических реализаций этих паттернов и называют фабрикой любой класс, инкапсулирующий в себе создание объектов.

В данной главе будет рассмотрен паттерн «Абстрактная фабрика» и его особенности, а в следующей — «Фабричный метод».

Назначение: абстрактная фабрика предоставляет интерфейс для создания семейства взаимосвязанных или родственных объектов (dependent or related objects), не специфицируя их конкретных классов.

Другими словами: абстрактная фабрика представляет собой стратегию создания семейства взаимосвязанных или родственных объектов.

Мотивация

Выразительность наследования обеспечивается за счет полиморфизма. Использование интерфейсов или базовых классов позволяет абстрагироваться от конкретной реализации, что делает решение простым и расширяемым. Однако где-то в приложении должна быть точка, в которой создаются объекты и известен их конкретный тип.

В некоторых случаях решение о конкретных типах можно откладывать до последнего, вплоть до корня приложения (Application Root). В этом случае конструирование конкретных типов происходит в методе `Main` (или аналогичном методе в зависимости от типа приложения), и затем созданные объекты передаются для последующей обработки. Однако в некоторых случаях создание объекта должно происходить раньше — в коде приложения.

Давайте вернемся к задаче сохранения прочитанных лог-файлов в хранилище для последующего полнотекстового поиска. Многие реляционные базы данных, например SQL Server, поддерживают полнотекстовый поиск по текстовым полям. Чтобы не завязываться на конкретную систему управления базами данных (СУБД), реализация класса `LogSaver` может использовать класс `DbProviderFactory` библиотеки ADO.NET (листинг 9.1).

Листинг 9.1. Пример использования абстрактной фабрики

```
public class LogSaver
{
    private readonly DbProviderFactory _factory;

    public LogSaver(DbProviderFactory factory)
    {
        _factory = factory;
    }

    public void Save(IEnumerable<LogEntry> logEntries)
    {
        using (var connection = _factory.CreateConnection())
        {
```

```
            SetConnectionString(connection);
            using (var command = _factory.CreateCommand())
            {
                SetCommandArguments(logEntries);
                command.ExecuteNonQuery();
            }
        }
    }

    private void SetConnectionString(DbConnection connection)
    {}

    private void SetCommandArguments(IEnumerable<LogEntry> logEntry)
    {}
}
```

Теперь вызывающий код может передать нужный экезмпляр `DbProviderFactory`, например `SqlClientFactory`, для работы с SQL Server, а затем передать `NpgsqlConnectionFactory`, чтобы перейти на `PostgreSql`.

Класс `DbProviderFactory` в данном случае предназначен для создания семейства взаимосвязанных объектов, таких как подключение к базе данных (`DbConnection`), команд (`DbCommand`), адаптеров (`DbDataAdapter`) и др., и является примером паттерна «Абстрактная фабрика».

Классическая диаграмма паттерна «Абстрактная фабрика» приведена на рис. 9.1.

Участники:

- `AbstractFactory` (`DbProviderFactory`) — объявляет интерфейс (с возможной базовой реализацией) для создания семейства продуктов;
- `AbstractProductA`, `AbstractProductB` (`DbCommand`, `DbConnection`) — семейство продуктов, которые будут использоваться клиентом для выполнения своих задач;
- `ProductA1`, `ProductB1` (`SqlConnection`, `NpgsqlConnection`) — конкретные типы продуктов;
- `Client` (`LogSaver`) — клиент фабрики, который получает конкретные продукты для реализации своего поведения.

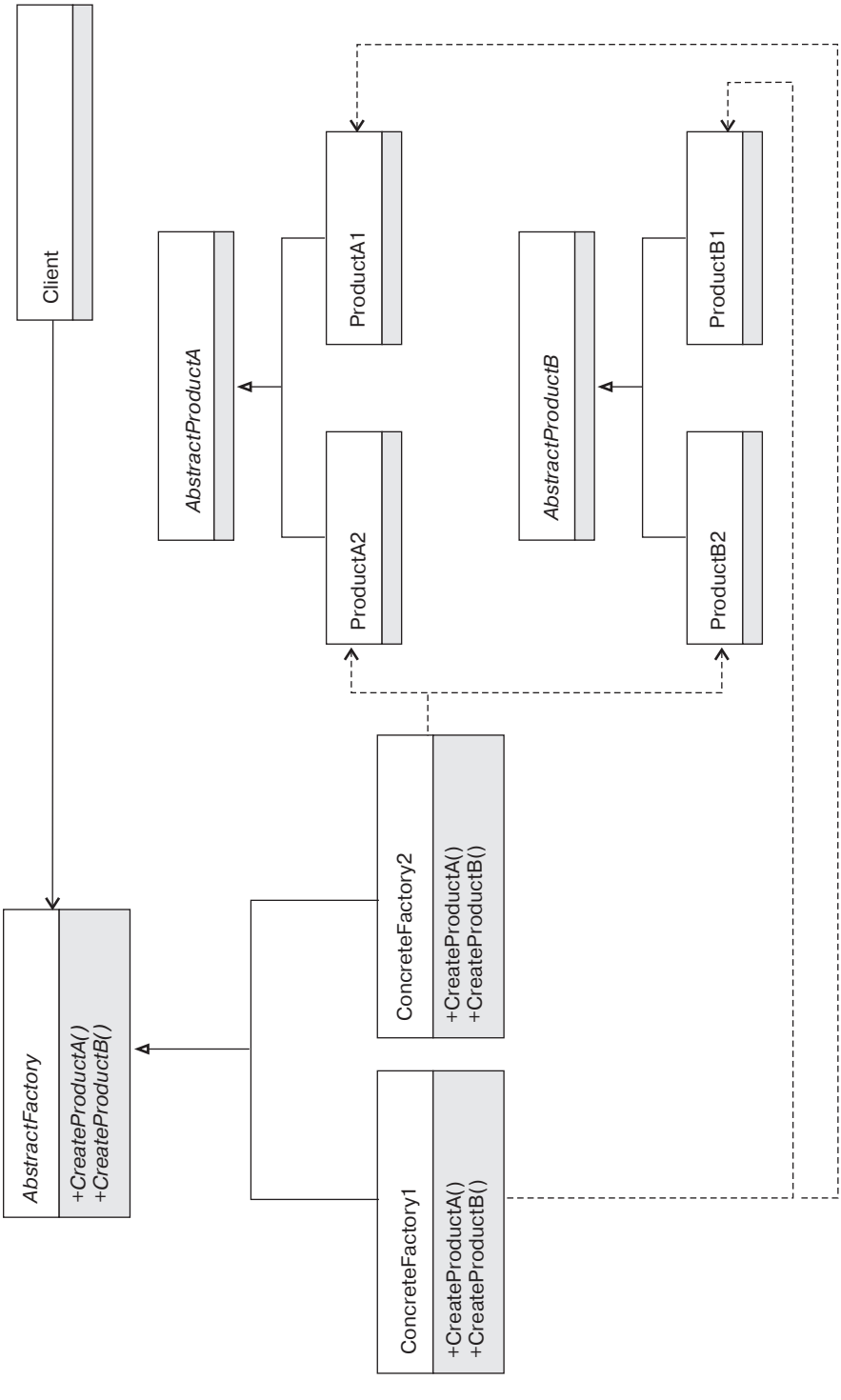


Рис. 9.1. Классическая диаграмма паттерна «Абстрактная фабрика»

Обсуждение паттерна «Абстрактная фабрика»

Основная особенность абстрактной фабрики заключается в том, что она предназначена для создания семейства объектов, что сильно сужает ее применимость. Но в некоторых предметных областях или инфраструктурном коде периодически возникают задачи, которые решаются набором классов: сериализаторы/десериализаторы, классы для сжатия/распаковки, шифрования/дешифрования и т. п. Приложение должно использовать согласованные типы объектов, и абстрактная фабрика идеально подходит для решения этой задачи. Интерфейс абстрактной фабрики объявляет набор фабричных методов, а конкретная реализация обеспечивает создание этого семейства объектов.

Абстрактная фабрика представляет собой стратегию[1] создания семейства объектов. Но при этом остается один вопрос: а кто создает конкретный экземпляр абстрактной фабрики? Еще одна фабрика?

Проблема курицы и яйца

Использование конкретного типа. В случае простых приложений конкретный экземпляр абстрактной фабрики можно создать в корне приложения. Например, можно создать конкретное приложение импорта файлов вашего приложения для сохранения их в SQL Server. В этом случае можно создать экземпляр `SqlClientFactory` и передать его классу `LogSaver` (листинг 9.2).

Листинг 9.2. Использование абстрактной фабрики в точке входа приложения

```
public static void Main(string[] args)
{
    var saver = new LogSaver(SqlClientFactory.Instance);
    var loader = new LogFileLoader(args[0]);
    saver.Save(loader.Load());
}
```

Фабрика фабрик. Вместо использования конкретного типа абстрактной фабрики можно выделить отдельный класс, задачей которого будет получение экземпляра фабрики. В этом случае мы имеем дело с фабрикой фабрик и наша главная задача — не попасть в бесконечную рекурсию. В случае с абстрактной фабрикой

[1] Паттерн «Стратегия» был рассмотрен в части I.

DbProviderFactory такой класс уже существует: DbProviderFactories.GetFactory[1] (листинг 9.3).

Листинг 9.3. Использование фабричного метода

```
public static void Main(string[] args)
{
    var saver = new LogSaver(GetDbProviderFactory());
    var loader = new LogFileLoader(args[0]);
    saver.Save(loader.Load());
}

private static DbProviderFactory GetDbProviderFactory()
{
    const string factoryName = "System.Data.SqlClient";
    return DbProviderFactories.GetFactory(factoryName);
}
```

Теперь можно пойти еще дальше и читать имя фабрики из конфигурационного файла приложения или обращаться к другому источнику для получения информации о требуемом типе фабрики.

Обобщенная абстрактная фабрика

С помощью абстрактной фабрики легко добавить новый подвид семейства объектов. Например, в случае с ADO.NET добавить еще одну реализацию DbProviderFactory относительно несложно. Но добавить новый фабричный метод для создания нового продукта довольно сложно, поскольку добавление нового абстрактного метода сломает все существующие реализации фабрики.

Это свойство абстрактной фабрики было известно ее авторам, поэтому одной из разновидностей реализации паттерна «Абстрактная фабрика» является обобщенная фабрика, которая позволяет создавать произвольные типы объектов (листинг 9.4).

Листинг 9.4. Пример обобщенной абстрактной фабрики

```
class GenericAbstractFactory
{
    public object Make(string id) { ... }
```

[1] Фабричный метод более подробно будет рассмотрен в следующей главе.

```
    public IProduct MakeProduct(string id) { ... }
    public T MakeGeneric<T>(string id) where T : IProduct { ... }
}
```

В данном случае абстрактная фабрика становится конфигурируемой, с ее помощью можно создавать объект любого или заданного типа (в данном случае `IProduct`). Эта реализация абстрактной фабрики нашла свое воплощение в виде DI-контейнеров (Dependency Injection Containers или IoC — Inversion of Control Containers), таких как Unity, StructureMap[1] и др. Контейнер позволяет создать экземпляр нужного типа и найти экземпляры всех его зависимостей.

Использование обобщенных фабрик и контейнеров обеспечивает высокую гибкость приложения и не требует наследования, поскольку набор создаваемых типов задается в процессе конфигурирования фабрики. Но это решение может привести к переусложненному и хрупкому дизайну, поскольку в приложении появляются неявные связи между точкой конфигурирования фабрики и точкой ее использования.

Применимость паттерна «Абстрактная фабрика»

Абстрактная фабрика представляет собой слой для полиморфного создания семейства объектов. Ее использование подразумевает обязательное наличие двух составляющих: семейства объектов и возможности замены создаваемого семейства объектов во время исполнения.

Наличие наследования обеспечивает гибкость, но в то же время приводит к проблеме курицы и яйца, рассмотренной ранее. Иногда необходимость полноценной абстрактной фабрики очевидна с самого начала, но обычно лучше начать с наиболее простого решения и добавить гибкость лишь в случае необходимости.

Я предпочитаю идти таким путем.

1. Использую конкретные классы напрямую, пока не появляется необходимость в полиморфном поведении.

---

[1] Подробное описание контейнеров выходит за рамки этой книги. Хорошее описание DI-контейнеров можно найти в книге Марка Сиимана Dependency Injection in .NET. Более подробную информацию о проблемах непосредственного использования контейнеров можно найти в моей статье «DI-паттерны. Service Locator» (http://bit.ly/ServiceLocatorPattern).

2. Прячу процесс создания объекта или семейства объектов за конкретными фабричными методами. Выделяю конкретный класс фабрики с набором неполиморфных фабричных методов.
3. Перехожу к абстрактной фабрике, лишь когда появляется необходимость подмены процесса создания объектов во время исполнения.

Примеры в .NET Framework

- Класс `DbProviderFactory` из ADO.NET с фабричными методами `CreateCommand(): DbCommand`, `CreateConnection(): DbConnection` и др.
- Класс `CodeDomProvider` с фабричными методами `CreateGenerator(): ICodeGenerator`, `CreateCompiler(): ICodeCompiler`, `CreateParser(): ICodeParser`.
- Класс `SymetricAlgorithm` с фабричными методами `CreateEncryptor(): ICryptoTransform` и `CreateDecryptor(): ICryptoTransform`.

Глава 10

Паттерн «Фабричный метод» (Factory Method)

Когда при обсуждении дизайна упоминается фабрика, то в подавляющем большинстве случаев имеется в виду одна из разновидностей паттерна «Фабричный метод».

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Мотивация

Как мы уже увидели в предыдущей главе, иерархии классов обеспечивают гибкость за счет полиморфного использования, но привносят дополнительную сложность. Абстрактная фабрика решает задачу полиморфного создания семейства объектов, но очень часто возникает более простая задача — создания одного экземпляра иерархии наследования.

В задаче импорта лог-файлов класс `LogFileReader` может самостоятельно открывать файл, анализировать его содержимое, создавать нужный экземпляр иерархии `LogEntry` (`ExceptionLogEntry` для исключений и `SimpleLogEntry` для обычных записей) и возвращать его вызывающему коду. Это может быть неплохим решением для первой итерации, но его нельзя назвать удачным с точки зрения долгосрочной перспективы.

Существует как минимум два изменения, которые сделают решение лучше расширяемым и тестируемым. Можно выделить класс `LogReaderBase`, который будет оперировать потоками ввода-вывода (экземплярами `Stream`) вместо файлов, а также создать отдельный класс `LogEntryParser`, ответственный за создание экземпляров `LogEntry` (листинг 10.1).

Листинг 10.1. Мотивация использования фабричного метода

```
public static class LogEntryParser
{
    public static LogEntry Parse(string data)
    {
        // Анализирует содержание data и создает нужный
        // экземпляр: ExceptionLogEntry или SimpleLogEntry
    }
}

public abstract class LogReaderBase
{
    public IEnumerable<LogEntry> Read()
    {
        using (var stream = OpenLogSource())
        {
            using (var reader = new StreamReader(stream))
            {
                string line = null;
                while ((line = reader.ReadLine()) != null)
                {
                    yield return LogEntryParser.Parse(line);
                }
            }
        }
    }

    protected abstract Stream OpenLogSource();
}
```

Теперь клиенты класса `LogReaderBase`, такие как `LogImporter`, могут работать с различными источниками лог-файлов независимо от их местоположения. Также данную реализацию легко протестировать в юнит-тестах путем создания класса `MemoryStreamLogReader`, который будет возвращать записи, помещенные туда во время инициализации теста.

В этом коде паттерн «Фабричный метод» используется дважды. Метод `OpenLogSource` является классическим фабричным методом, когда базовый класс определяет абстрактный метод для создания объекта, а наследник его реализует. Класс `LogEntryParser` является разновидностью фабричного метода, который создает нужный экземпляр иерархии в зависимости от переданных аргументов.

Цель любой фабрики — оградить клиентов от подробностей создания экземпляров класса или иерархии классов.

Диаграмма паттерна «Фабричный метод»

На практике встречаются три вида паттерна «Фабричный метод»:

- классическая реализация на основе шаблонного метода;
- статический фабричный метод;
- полиморфный фабричный метод.

Классическая реализация

Диаграмма классического паттерна «Фабричный метод» приведена на рис. 10.1.

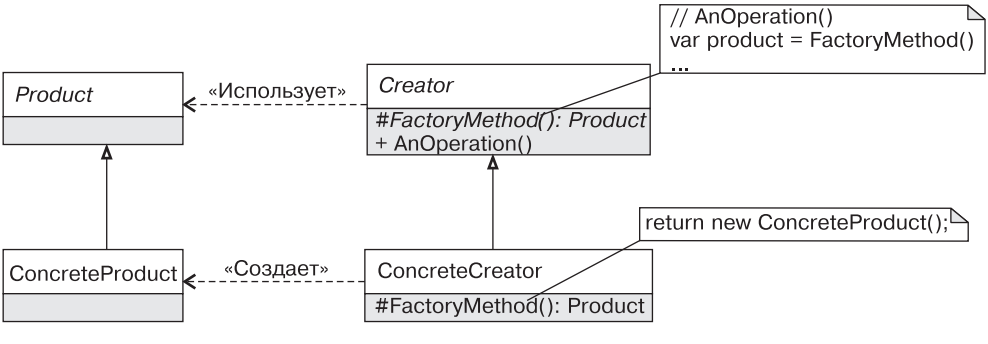


Рис. 10.1. Диаграмма классического паттерна «Фабричный метод»

Классический «Фабричный метод» является частным случаем паттерна «Шаблонный метод», переменный шаг которого отвечает за создание нужного типа объекта.

Участники:

- `Creator` (`LogReaderBase`) — объявляет абстрактный или виртуальный метод создания продукта. Использует фабричный метод в своей реализации;
- `ConcreteCreator` (`LogFileReader`) — реализует фабричный метод, который возвращает `ConcreteProduct` (`FileStream`);
- `Product` (`Stream`) — определяет интерфейс продуктов, создаваемых фабричным методом;
- `ConcreteProduct` (`FileStream`) — определяет конкретный вид продуктов.

Статический фабричный метод

Самой простой версией фабричного метода является статический метод, который создает экземпляр нужного типа в зависимости от переданных аргументов (рис. 10.2).

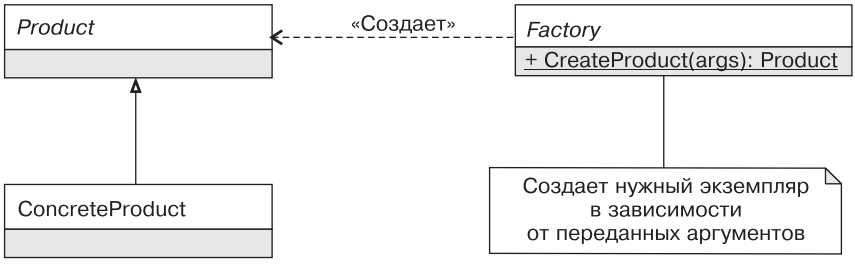


Рис. 10.2. Диаграмма статического фабричного метода

Основное отличие статического метода от классического фабричного в том, что тип создаваемого фабрикой объекта определяется не типом наследника, а аргументами, переданными методу `Create`.

Полиморфный фабричный метод

Полиморфный фабричный метод определяет интерфейс фабрики, а за создание конкретного экземпляра продукта отвечает конкретная фабрика (рис. 10.3).

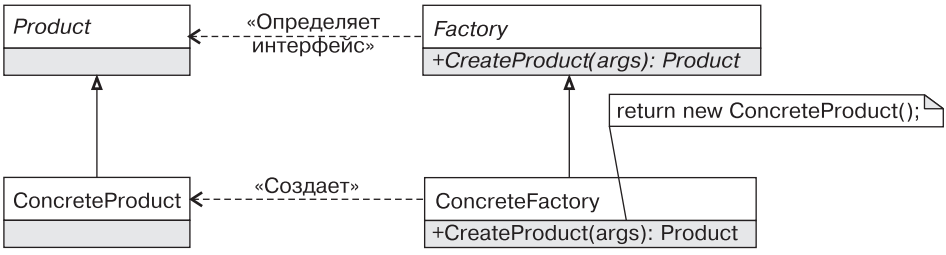


Рис. 10.3. Диаграмма полиморфного фабричного метода

Варианты реализации

В .NET существует несколько классических идиом, которые используются для реализации фабрик.

Использование делегатов в статической фабрике

Типичная реализация статического фабричного метода содержит оператор `switch`, в котором происходит выбор создаваемого типа. Вместо этого можно воспользоваться словарем, ключом которого будет некоторый идентификатор (строка, перечисление и т. п.), а значением — делегат, который будет создавать объект нужного класса.

Этот подход может быть использован, например, для создания различных классов импорта данных, зависящих от расширения файла (листинг 10.2).

Листинг 10.2. Статический фабричный метод

```
static class ImporterFactory
{
    private static readonly Dictionary<string, Func<Importer>> _map =
        new Dictionary<string, Func<Importer>>();

    static ImporterFactory()
    {
        _map[".json"] = () => new JsonImporter();
        _map[".xls"] = () => new XlsImporter();
        _map[".xlsx"] = () => new XlsImporter();
    }
```

```
    public static Importer Create(string fileName)
    {
        var extension = Path.GetExtension(fileName);

        var creator = GetCreator(extension);
        if (creator == null)
            throw new UnsupportedImporterTypeException(extension);

        return creator();
    }

    private static Func<Importer> GetCreator(string extension)
    {
        Func<Importer> creator;
        _map.TryGetValue(extension, out creator);
        return creator;
    }
}
```

В статическом конструкторе фабрики все доступные типы регистрируются в словаре `_map`, который затем используется для создания нужного класса в методе `Create`.

Обобщенные фабрики

Существует проблема, с которой сталкиваются практически все разработчики независимо от используемого языка программирования: как гарантировать вызов виртуального метода при конструировании любого объекта определенной иерархии типов? Вызов виртуального метода в конструкторе базового класса не подходит: в языке C# это может привести к непредсказуемому поведению, так как будет вызван метод наследника, конструктор которого еще не отработал. Можно воспользоваться приемами аспектно-ориентированного программирования, а можно воспользоваться фабрикой, которая вызовет виртуальный метод уже после создания экземпляра.

ПРИМЕЧАНИЕ

Использование фабрик для выполнения обязательных действий после создания объекта является довольно распространенным подходом. Подробнее об этом можно почитать в статье Тала Коена Better Construction with Factories (Journal of Object Technology, 2002. — http://www.jot.fm/issues/issue_2007_07/article3.pdf).

Для решения этой задачи на языке C# нам понадобятся обобщения (generics), отражение (reflection) и немного магии для корректного пробрасывания исключений (листинг 10.3).

Листинг 10.3. Обобщенный фабричный метод

```
public abstract class Product
{
    protected internal abstract void PostConstruction();
}

public class ConcreteProduct : Product
{
    // Внутренний конструктор не позволит клиентам иерархии
    // создавать объекты напрямую.
    internal ConcreteProduct() {}

    protected internal override void PostConstruction()
    {
        Console.WriteLine("ConcreteProduct: post construction");
    }
}

// Единственно законный способ создания объектов семейства Product
public static class ProductFactory
{
    public static T Create<T>() where T : Product, new()
    {
        try
        {
            var t = new T();
```

```
            // Вызываем постобработку
            t.PostConstruction();

            return t;
        }
        catch (TargetInvocationException e)
        {
            // «разворачиваем» исключение и бросаем исходное
            var edi = ExceptionDispatchInfo.Capture(e.InnerException);
            edi.Throw();
            // эта точка недостижима, но компилятор об этом не знает!
            return default(T);
        }
    }
}
```

Пример использования приведен в листинге 10.4.

Листинг 10.4. Пример использования обобщенного фабричного метода

```
var p1 = ProductFactory.Create<ConcreteProduct>();
var p2 = ProductFactory.Create<AnotherProduct>();
```

Обратите внимание на реализацию метода `Create` и перехват `TargetInvocationException`. Поскольку конструкция вида `new T()` использует отражение для создания экземпляра типа `T`, то в случае возникновения исключения в конструкторе типа `T` исходное исключение будет «завернуто» в `TargetInvocationException`. Чтобы упростить работу с нашей фабрикой, можно «развернуть» это исключение в методе `Create` и пробросить исходное исключение с сохранением стека вызовов с помощью `ExceptionDispatchInfo`[1].

ПРИМЕЧАНИЕ

Оборачивание исключения в TargetInvocationException при вызове new T() является одним из примеров «дырявых абстракций» (leaky abstractions). Более подробное изложение причин такого поведения и особенностей поведения класса ExceptionDispatchInfo выходит за рамки данной книги. Подробнее об этом можно прочитать в моей статье «Повторная генерация исключений», доступной по адресу http://bit.ly/ExceptionRethrowing.

[1] Тип ExceptionDispatchInfo появился лишь в .NET 4.5.

Обсуждение паттерна «Фабричный метод»

У каждой реализации фабричного метода есть свои особенности.

- **Классический фабричный метод** является частным случаем шаблонного метода. Это значит, что фабричный метод привязан к текущей иерархии типов и не может быть использован повторно в другом контексте.
- **Полиморфный фабричный метод** является стратегией создания экземпляров некоторого семейства типов, что позволяет использовать одну фабрику в разных контекстах. Тип создаваемого объекта определяется типом фабрики и обычно не зависит от аргументов фабричного метода.
- **Статический фабричный метод** является самой простой формой фабричного метода. Статический метод создания позволяет обойти ограничения конструкторов. Например, тип создаваемого объекта может зависеть от аргументов метода, экземпляр может возвращаться из кэша, а не создаваться заново или же фабричный метод может быть асинхронным.

Сокрытие наследования

Фабричный метод скрывает от своих клиентов детали конструирования объектов. Это бывает полезно, когда процесс создания экземпляра сложен, состоит из нескольких этапов или когда мы хотим скрыть от клиентов настоящий тип создаваемых объектов.

Давайте вернемся к рассмотренному ранее примеру с классами `LogReaderBase` и `LogFileReader`. Сам факт иерархии наследования довольно просто сделать деталью реализации. Для этого достаточно переименовать класс `LogReaderBase` в `LogReader` и добавить в него два статических фабричных метода, `FromFile` и `FromStream` (рис. 10.4).

ПРИМЕЧАНИЕ

В данном случае иерархия классов LogReader содержит два вида фабричных методов: классический фабричный метод OpenLogSource для создания нужного экземпляра Stream и два статических фабричных метода — FromStream и FromFile.

Данный подход обеспечивает высокую адаптивность решения, поскольку позволяет модифицировать иерархию наследования, не затрагивая существующих клиентов.

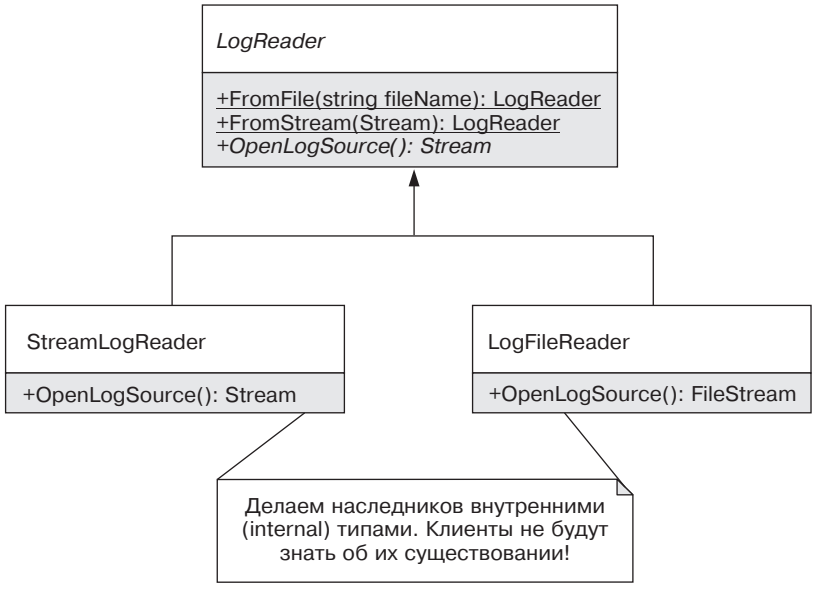


Рис. 10.4. Сокрытие иерархии наследования за фабричными методами

Устранение наследования

Рассмотренный ранее подход скрывает наследников от клиентов фабрики, что позволяет избавиться от иерархии наследования, если в ней отпадает необходимость. В случае простых иерархий наследования, как в нашем случае, виртуальные методы можно легко заменить делегатами. Вместо трех классов, `LogReader`, `LogFileReader` и `StreamLogReader`, достаточно оставить лишь первый, а виртуальный метод `OpenLogStream` реализовать с помощью делегата `Func<Stream>` (листинг 10.5).

Листинг 10.5. Реализация класса LogReader на основе делегатов

```
public class LogReader
{
    private readonly Func<Stream> _streamFactory;

    private LogReader(Func<Stream> streamFactory)
    {
        _streamFactory = streamFactory;
    }
```

```
    public static LogReader FromFile(string fileName)
    {
        Func<Stream> factory =
            () => new FileStream(fileName, FileMode.Open);
        return new LogReader(factory);
    }

    public static LogReader FromStream(Stream stream)
    {
        Func<Stream> factory = () => stream;
        return new LogReader(factory);
    }

    public IEnumerable<LogEntry> Read()
    {
        using (var stream = OpenLogSource())
        {
            using (var reader = new StreamReader(stream))
            {
                string line = null;
                while ((line = reader.ReadLine()) != null)
                {
                    yield return LogEntryParser.Parse(line);
                }
            }
        }
    }

    private Stream OpenLogSource()
    {
        return _streamFactory();
    }
}
```

Использование Func в качестве фабрики

В предыдущем разделе мы использовали `Func<Stream>` в качестве детали реализации. В некоторых случаях `Func<T>` может использоваться в качестве полноценной фабрики и передаваться классу извне его клиентами.

Данный вариант фабрики является допустимым во внутреннем (internal) коде и только для функций с небольшим количеством аргументов. Понять, что делает `Func<Stream>`, довольно просто, но разобраться в назначении `Func<int, string, int, ValidationResult> factory` без контекста будет практически невозможно. В случае повторно используемого кода понятность кода очень важна, поэтому именованная фабрика является предпочтительным вариантом.

Конструктор vs. фабричный метод

В объектно-ориентированных языках программирования конструктор отвечает за корректную инициализацию создаваемого объекта. В большинстве случаев они прекрасно справляются со своей задачей, но иногда лучше воспользоваться статическим фабричным методом.

Именованные конструкторы. В языке C# имя конструктора совпадает с именем класса, что делает невозможным использование двух конструкторов с одним набором и типом параметров. Хорошим примером такого ограничения является структура `Timespan`, которая представляет собой интервал времени. Очень удобно создавать интервал времени по количеству секунд, минут, часов и дней, но сделать несколько конструкторов, каждый из которых принимает один параметр типа `double`, невозможно. Для этого структура `Timespan` содержит набор фабричных методов (листинг 10.6).

Листинг 10.6. Пример использования фабричных методов в качестве именованных конструкторов

```
public struct Timespan
{
    public Timespan(double ticks) { ... }
    public static Timespan FromMilliseoncds(double value) {...}
    public static Timespan FromSeconds(double value) {...}
    public static Timespan FromMinutes(double value) {...}
    // Остальные фабричные методы
}
```

Тяжеловесный процесс создания. Конструктор отвечает за корректную инициализацию объекта, после которой объект должен быть готов для использования своими клиентами. Обычно логика инициализации относительно простая и должна выполняться конструктором, но слишком тяжеловесную логику лучше вынести из конструктора в статический фабричный метод[1].

Я использую статические фабричные методы, когда сложность или время исполнения конструктора переходит определенную черту. Если для конструирования объекта требуется обращение к внешним ресурсам, то я предпочитаю сразу же выделять эту логику в фабричный метод. Это позволяет сделать фабричный метод асинхронным, а также упростить эволюцию решения и разбиение данного класса на более мелкие составляющие в случае необходимости.

Применимость паттерна «Фабричный метод»

Разные виды фабричных методов применяются для решения разных задач.

Применимость классического фабричного метода

Классический фабричный метод очень редко появляется в результате тщательного проектирования. Будучи частным случаем шаблонного метода, он естественным образом возникает в иерархии наследования, когда базовый класс определяет некоторый алгоритм, одним из этапов которого является конструирование объекта. При этом решение о типе объекта не может быть принято на его уровне и переносится на уровень наследников.

Применимость полиморфного фабричного метода

- **Стратегия создания объектов.** Полиморфный фабричный метод применяется для выделения стратегии создания объектов, чтобы отвязать клиентов от создаваемых ими зависимостей.
- **Конфигурирование процесса создания.** В некоторых случаях экземплярная фабрика может быть использована для создания объектов с нужными характеристиками. Например, класс `TaskFactory` из Task Parallel Library используется

[1] Некоторые специалисты считают, что наличие в конструкторе инструкции if является признаком плохого кода. Достаточно поискать в вашем любимом поисковом сервисе фразу bloated constructor, чтобы понять, что я не шучу!

для создания задач с заданным планировщиком, маркером отмены и т. д. При этом гибкость обеспечивается не за счет наследования, а за счет конфигурирования объекта фабрики.

Применимость статического фабричного метода

- **Сокрытие иерархии наследования.** Набор статических фабричных методов может скрывать от своих клиентов глубину и даже наличие иерархии наследования, как это делают `WebRequest.Create(string)` и класс `LogReader`, рассмотренный ранее.
- **Именованный конструктор.** Фабрика используется для устранения неоднозначности, когда объект может быть создан по аргументам одного типа, но с разным значением. Например, `Timespan.FromSeconds`, `Timespan.FromMilliseconds` и т. п.
- **Вызов виртуального метода после создания объектов.** Как мы видели ранее, статический фабричный метод может быть использован для эмуляции вызова виртуального метода в конструкторе базового класса, когда требуется выполнить обязательные действия после создания всех объектов определенной иерархии типов.
- **Фасад для сложного процесса создания.** В некоторых случаях процесс создания может быть достаточно сложным для того, чтобы изолировать его в отдельном статическом методе. Например, фабричный метод может использовать паттерн «Строитель», создавать объект довольно сложным образом или обращаться к внешним ресурсам.
- **Асинхронный конструктор.** Если длительность конструирования относительно велика, то статический фабричный метод можно сделать асинхронным, что невозможно в случае использования конструктора.
- **Кэширование.** Статический фабричный метод позволит возвращать экземпляры из кэша, а не создавать их каждый раз заново.

Примеры в .NET Framework

В .NET Framework применяется огромное количество фабрик.

- Классический фабричный метод: `Stream.CreateWaitHandle`, `SecurityAttribute.CreatePermission`, `ChannelFactory.CreateChannel`, `XmlNode.CreateNavigator`.

- Полиморфная фабрика: `IControllerFactory` в ASP.NET MVC, `IHttpHandlerFactory` в ASP.NET, `ServiceHostFactory` и `IChannelFactory<T>` в WCF, `IQueryProvider` в LINQ.
- Неполиморфная фабрика: `TaskFactory` в TPL.
- Обобщенная статическая фабрика: `Activator.CreateInstance`, `Array.CreateInstance`, `StringComparer.Create`.
- Сокрытие наследников: `RandomNumberGenerator.Create`, `WebRequest.Create`, `BufferManager.CreateBufferManager`, `Message.CreateMessage`, `MessageFault.CreateFault`.
- Фасадные фабричные методы: `File.Create`, `File.CreateText`.
- Именованные конструкторы: `Timespan.FromSecond`, `Timespan.FromMilliseconds`, `GCHandle.FromIntPtr`, `Color.FromArgb`.

Глава 11

Паттерн «Строитель» (Builder)

Назначение: строитель отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Мотивация

Фабричный метод скрывает от своих клиентов процесс создания и конкретный тип возвращаемого объекта. Фабрика позволяет развивать иерархию типов, не затрагивая существующих клиентов, но в некоторых случаях требуется гибкость иного рода.

Иногда процесс создания является довольно сложным, состоит из нескольких этапов. Создаваемому объекту нужно передать множества аргументов, часть из которых нужны одним клиентам, но не нужны другим. В этом случае фабричный метод с десятью аргументами, девять из которых будут регулярно повторяться, нельзя назвать удачным решением. В этом случае поможет другой порождающий паттерн — «Строитель».

Давайте рассмотрим задачу создания объектов `Email` для отправки сообщений электронной почты. Часть свойств объекта `Email` являются обязательными,

а часть — нет, количество приложений (attachment) и получателей может быть произвольным и т. п. Электронные сообщения можно создавать руками, задавая нужные свойства, как это делается при работе с классом `MailMessage` из .NET Framework, а можно выделить процесс создания в отдельный класс — `MailMessageBuilder` (листинг 11.1).

Листинг 11.1. Строитель сообщений электронной почты

```
public sealed class MailMessageBuilder
{
    private readonly MailMessage _mailMessage = new MailMessage();

    public MailMessageBuilder From(string address)
    {
        _mailMessage.From = new MailAddress(address);
        return this;
    }

    public MailMessageBuilder To(string address)
    {
        _mailMessage.To.Add(address);
        return this;
    }

    public MailMessageBuilder Cc(string address)
    {
        _mailMessage.CC.Add(address);
        return this;
    }

    public MailMessageBuilder Subject(string subject)
    {
        _mailMessage.Subject = subject;
        return this;
    }
```

```
    public MailMessageBuilder Body(string body, Encoding enconding)
    {
        _mailMessage.Body = body;
        _mailMessage.BodyEncoding = enconding;
        return this;
    }

    public MailMessage Build()
    {
        return _mailMessage;
    }
}
```

Теперь этот класс можно использовать так (листинг 11.2).

Листинг 11.2. Пример использования класса MailMessageBuilder

```
var mail = new MailMessageBuilder()
    .From("st@unknown.com")
    .To("support@microsof.com")
    .Cc("my_boss@unknown.com")
    .Subject("Msdn is down!")
    .Body("Please fix!", Encoding.UTF8)
    .Build();

new SmtpClient().Send(mail);
```

Использование паттерна «Строитель» позволяет более четко разграничить ответственность между создателем и потребителем объектов, а также делает процесс создания более удобным.

Паттерн «Строитель» довольно часто применяется в современных приложениях, но не в том виде, в котором он был описан «бандой четырех». На практике строитель все так же отвечает за создание объектов, но гораздо реже обладает всеми изначальными свойствами. Классическая диаграмма классов паттерна «Строитель» приведена на рис. 11.1.

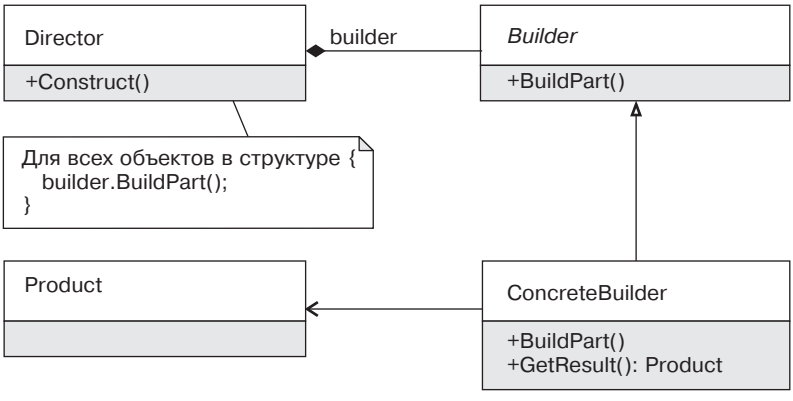


Рис. 11.1. Классическая диаграмма паттерна «Строитель»

Участники:

- `Builder` определяет интерфейс конструирования продукта по частям;
- `Director` управляет процессом создания, не зная, какой продукт будет создан в результате;
- `ConcreteBuilder` — конкретный строитель, который создает только известный ему объект класса `Product`.

Обратите внимание на два момента: наличие наследования и то, что о классе `Product` знает только конкретный строитель (`ConcreteBuilder`). Ни базовый класс строителя, ни его клиент (класс `Director`) не знают о типе создаваемого продукта. Это развязывает руки конкретным строителям, которые могут формировать совершенно разнородные объекты, от сообщения электронной почты до строки в формате Json. Но это оставляет открытыми многие вопросы. Кто потребляет созданный продукт? Как конкретный строитель узнает об этом потребителе? Всегда ли процесс формирования и потребления продуктов должен быть так удален друг от друга, что продукт конструируется классом `Director`, а потребляется непонятно кем?

На практике обычно используется более простая разновидность паттерна «Строитель», без наследования и с более явной моделью взаимодействия между участниками (рис. 11.2).

Участники в этом случае остались теми же самыми, но ответственность немного изменяется. `Director` (клиент класса `MailMessageBuilder`) управляет созданием сложного объекта и получает созданный объект путем вызова метода `Build`. `Builder` (`MailMessageBuilder`) отвечает за создание конкретного продукта (`MailMessage`).

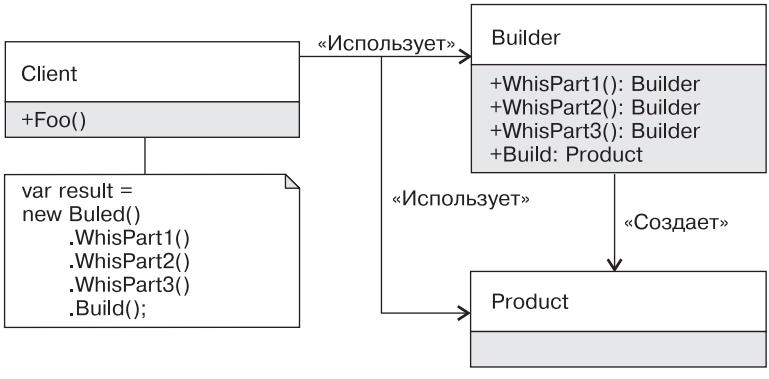


Рис. 11.2. Упрощенная диаграмма паттерна «Строитель»

Особенности реализации в .NET

При работе со строителями в .NET есть два специфических момента: использование текучего интерфейса (fluent interface) и возможность использования методов расширения.

Использование текучего интерфейса

Текучий интерфейс предназначен для повышения читабельности кода (листинг 11.3). Каждый метод возвращает объект, над которым выполняется текущая операция, что позволяет связывать методы в цепочку.

Листинг 11.3. Пример использования текучего интерфейса

```
var result = Enumerable.Range(1, 10).Select(n => n + 1).Count();
```

Текучий интерфейс изначально появился в мире Java и активно используется для создания мини-DSL (Domain-Specific Language). При реализации строителей данная идиома используется постоянно, поскольку позволяет создавать требуемый объект одним оператором (листинг 11.4).

Листинг 11.4. Пример использования класса StringBuilder

```
public override string ToString()
{
    return new StringBuilder()
        .AppendFormat("Id: {0}", Id).AppendLine()
```

```
        .AppendFormat("Name: {0}", Name).AppendLine()
        .ToString();
}
```

Методы расширения

Методы расширения позволяют создать видимость того, что мы добавляем операции в существующие типы. В некоторых случаях можно создать набор методов расширения, которые будут играть роль строителя для существующих типов.

Вместо использования объекта `MailMessageBuilder` можно обойтись методами расширения над классом `MailMessage` (листинг 11.5).

Листинг 11.5. Использование методов расширения для реализации строителя

```
public static class MailMessageBuilderEx
{
    public static MailMessage From(this MailMessage mailMessage,
        string address)
    {
        mailMessage.From = new MailAddress(address);
        return mailMessage;
    }

    public static MailMessage To(this MailMessage mailMessage,
        string address)
    {
        mailMessage.To.Add(address);
        return mailMessage;
    }

    public static MailMessage Cc(this MailMessage mailMessage,
        string address)
    {
        mailMessage.CC.Add(address);
        return mailMessage;
    }

    public static MailMessage Subject(
```

```
        this MailMessage mailMessage, string subject)
    {
        mailMessage.Subject = subject;
        return mailMessage;
    }

    public static MailMessage Body(this MailMessage mailMessage,
        string body, Encoding enconding)
    {
        mailMessage.Body = body;
        mailMessage.BodyEncoding = enconding;
        return mailMessage;
    }
}
```

Пример использования будет практически аналогичен примеру с классом `MailMessageBuilder` (листинг 11.6).

Листинг 11.6. Пример использования методов расширения

```
var mail = new MailMessage()
    .From("st@unknown.com")
    .To("support@microsof.com")
    .Cc("my_boss@unknown.com")
    .Subject("Msdn is down!")
    .Body("Please fix!", Encoding.UTF8);

new SmtpClient().Send(mail);
```

Разница лишь в том, что методы расширения манипулируют уже созданным продуктом, просто делают это более декларативным образом.

ПРИМЕЧАНИЕ

Подобные методы расширения не всегда могут заменить полноценный строитель. Строитель обладает значительной гибкостью: он может накапливать состояние в собственных полях, что даст больше свободы при валидации состояния создаваемого объекта. Строитель может проверять накопленное состояние и генерировать исключение в методе Build, если клиент еще не вызвал один из обязательных методов.

Обсуждение паттерна «Строитель»

При описании диаграммы классов паттерна «Строитель» уже было сказано, что в классическом виде этот паттерн применяется довольно редко. Возможность создания разнообразных продуктов нужна относительно редко, и возрастающая при этом сложность не всегда окупается.

Если посмотреть на примеры строителей в .NET Framework, то все они аналогичны рассмотренному ранее классу `MailMessageBuilder` — это специализированные классы для создания конкретных продуктов — строк, коллекций, контейнеров и т. д.

В случае применения паттерна «Строитель» разумно следовать той же логике, что и при выделении стратегий. Начните с конкретного класса `Builder`, предназначенного для создания конкретных продуктов. Выделяйте стратегию конструирования, такую как `IBuilder`, лишь тогда, когда в этом будет необходимость.

Строго типизированный строитель

В некоторых случаях процесс создания может содержать обязательные и необязательные этапы. Так, в случае класса `MailMessageBuilder` можно запретить конструирование объекта `MailMessage` с пустым списком получателей. Для этого метод `Build` может проверять состояние конструированного объекта и генерировать исключение (например, `InvalidOperationException`), если поле `To` пусто (листинг 11.7).

Листинг 11.7. Генерация исключения в методе Build

```
public MailMessage Build()
{
    if (_mailMessage.To.Count == 0)
    {
        throw new InvalidOperationException(
            "Can't create a mail message with empty To. Please call
               'To' method first");
    }

    return _mailMessage;
}
```

Вместо генерации исключений во время исполнения можно изменить дизайн таким образом, чтобы эта ошибка была недопустимой во время компиляции.

Использование конструктора. Самый простой способ добиться этого — передать обязательные аргументы в конструкторе строителя (листинг 11.8).

Листинг 11.8. Добавление конструктора в класс MailMessageBuilder

```
public class MailMessageBuilder
{
    public MailMessageBuilder(string address)
    {
        To(address);
    }

    // Остальные методы остались без изменения
```

Проблема этого подхода в том, что он плохо расширяем, если количество обязательных этапов станет больше 3–4. Если обязательная часть более сложная, то можно воспользоваться вторым подходом.

Использование паттерна «Состояние». Более сложный в реализации подход основан на использовании варианта паттерна «Состояние». Можно создать набор строителей, каждый из которых будет содержать лишь операцию для перевода объекта на следующий этап. При этом метод `Build` будет доступен лишь на последнем этапе, когда все нужные этапы были проинициализированы.

В простом виде данный подход будет выглядеть так (листинг 11.9).

Листинг 11.9. Пример строго типизированного строителя

```
public class MailMessageBuilder
{
    private readonly MailMessage _mailMessage = new MailMessage();

    internal MailMessageBuilder(MailMessage mailMessage)
    {
        _mailMessage = mailMessage;
    }

    public FinalMailMessageBuilder To(string address)
    {
```

```
        _mailMessage.To(address);
        // Для большей эффективности может быть добавлено кэширование
        return new FinalMailMessageBuilder(_mailMessage);
    }

    // Остальные методы остались без изменения
}

public class FinalMailMessageBuilder
{
    private readonly MailMessage _mailMessage;

    internal FinalMailMessageBuilder(MailMessage mailMessage)
    {
        _mailMessage = mailMessage;
    }

    public MailMessage Build()
    {
        return _mailMessage;
    }
}
```

Для обязательного этапа выделен специализированный тип строителя — `FinalMailMessageBuilder`, который можно получить вызовом метода `To` основного строителя. Этот подход гарантирует, что при вызове метода `Build` получатель был установлен и создаваемый объект будет в корректном состоянии.

ПРИМЕЧАНИЕ

Данный подход устраняет ошибки времени исполнения, но за это приходится платить дополнительными усилиями во время разработки. В простых случаях в методе Build достаточно генерировать исключение InvalidOperationException с понятным сообщением об ошибке, а строго типизированный вариант оставить на тот случай, когда важность корректного использования очень высока. Еще одна проблема этого подхода заключается в выделении нового экземпляра строителя для каждого нового этапа. Избавиться от нее можно заменой классов на структуры.

Создание неизменяемых объектов

Паттерн «Строитель» четко отделяет процесс создания продукта от его потребления. Это особенно полезно, если продукт физически или логически является неизменяемым (immutable).

Конструирование неизменяемых объектов является дорогостоящим, поскольку каждая команда (операция, изменяющая состояние) создает новый экземпляр. В .NET строки являются неизменяемыми, поэтому в состав .NET Framework был добавлен класс `StringBuilder`, предназначенный для более эффективного создания строк.

Паттерн «Строитель» часто используется в комплекте с другими неизменяемыми структурами данных, например неизменяемыми коллекциями[1]. Добавление нового элемента в неизменяемую коллекцию позволяет гарантированно выделять память и создавать новую коллекцию. В примере в листинге 11.10 будут созданы десять экземпляров коллекции `ImmutableList<int>`.

Листинг 11.10. Пример работы с неизменяемой коллекцией

```
var list = ImmutableList<int>.Empty;
foreach (var element in Enumerable.Range(1, 10))
{
    // Объект list пересоздается 10 раз!
    list = list.Add(element);
}
```

Избежать дополнительных аллокаций памяти помогает то, что каждая неизменяемая коллекция содержит внутренний строитель, с помощью которого создание коллекции будет более эффективным (листинг 11.11).

Листинг 11.11. Использование строителя для создания неизменяемой коллекции

```
ImmutableList<int>.Builder builder =
    ImmutableList<int>.Empty.ToBuilder();

foreach (var element in Enumerable.Range(1, 10))
{
```

[1] Неизменяемые коллекции входят в состав .NET Framework, но распространяются отдельным пакетом. Для их использования с помощью пакетного менеджера nuget нужно добавить пакет System.Collections.Immutable for .NET 4.0.

```
    builder.Add(element);
}

ImmutableList<int> list = builder.ToImmutable();
```

Частичная изменяемость

Неизменяемые коллекции показывают хороший пример частичной неизменяемости. С точки зрения внешних клиентов объекты класса `ImmutableList<T>` являются неизменяемыми. Но есть одно исключение — вложенный класс `ImmutableList<T>.Builder`, который может получить непосредственный доступ к закрытым полям класса `ImmutableList<T>`.

В некоторых случаях можно пойти еще дальше и убрать из продукта все команды, оставив лишь набор свойств только для чтения. И предоставить единственный интерфейс создания объектов через строитель. Если бы я создавал класс `MailMessage` самостоятельно, то мог бы прийти к такому решению (листинг 11.12).

Листинг 11.12. Неизменяемый класс MailMessage

```
public class MailMessage
{
    private string _to;
    private string _from;
    private string _subject;
    private string _body;

    private MailMessage()
    {}

    public static MailMessageBuilder With()
    {
        return new MailMessageBuilder(new MailMessage());
    }

    public string To { get { return _to; } }
    public string From { get { return _from; } }
```

```
    public string Subject { get { return _subject; } }
    public string Body { get { return _body; } }

    public class MailMessageBuilder {...}
}
```

Класс `MailMessage` является полностью неизменяемым с единственным фабричным методом `With`, который возвращает строитель. `MailMessageBuilder` является вложенным классом в `MailMessage`, а значит, имеет доступ к его закрытым членам (листинг 11.13).

Листинг 11.13. Реализация вложенного строителя

```
// Объявлен внутри класса MailMessage
public class MailMessageBuilder
{
    private readonly MailMessage _mailMessage;

    internal MailMessageBuilder(MailMessage mailMessage)
    {
        _mailMessage = mailMessage;
    }

    public MailMessageBuilder To(string to)
    {
        _mailMessage._to = to;
        return this;
    }

    public MailMessageBuilder From(string from)
    {
        _mailMessage._from = from;
        return this;
    }

    public MailMessageBuilder Subject(string subject)
    {
```

```
        _mailMessage._subject = subject;
        return this;
    }

    public MailMessageBuilder Body(string body)
    {
        _mailMessage._body = body;
        return this;
    }

    public MailMessage Build()
    {
        return _mailMessage;
    }
}
```

Клиенты класса `MailMessage` вынуждены использовать строитель для задания объекта `MailMessage`, который после создания является неизменяемым (листинг 11.14).

Листинг 11.14. Пример использования неизменяемого типа со строителем

```
var mailMessage =
    MailMessage.With()
    .From("st@unknown.com")
    .To("support@microsof.com")
    .Subject("Msdn is down!")
    .Body("Please fix!")
    .Build();
Console.WriteLine(mailMessage.To);
```

Применимость

Паттерн «Строитель» идеально подходит для ситуаций, когда процесс создания является сложным и состоит из нескольких этапов, при этом одним клиентам нужно устанавливать одни параметры создаваемого объекта, а другим — другие. Строитель

может устанавливать разумные значения по умолчанию, позволяя клиентам сосредоточиться лишь на важных для них параметрах.

Строитель обычно обладает текучим интерфейсом, что делает его использование более читабельным и декларативным.

Я обратил внимание на то, что в последнее время довольно часто пользуюсь этим паттерном при создании тестовых данных, создании и конфигурировании классов экспорта данных (источник, тип получателя, то, какие данные нужно экспортировать), формировании слабо типизированных объектов, таких как JSON или Xml, с помощью строго типизированного Fluent API и т. п.

Строитель идеально сочетается с неизменяемыми классами. Неизменяемость упрощает понимание кода и прекрасно подходит для использования в многопоточной среде. Наличие строителей позволяет обойти ограничения неизменяемости и решить проблемы эффективности, которые обязательно возникнут при работе с такими типами.

Примеры в .NET Framework

Паттерн «Строитель» довольно часто используется в .NET Framework.

- `StringBuilder`, `UriBuilder`, `DbCommandBuilder` и `DbConnectionStringBuilder` из ADO.NET.
- Строители неизменяемых коллекций `ImmutableList<T>.Builder`, `ImmutableDictionary<TKey, TValue>.Builder` и т. д.
- Типы из Reflection.Emit: `ModuleBuilder`, `TypeBuilder`, `EnumBuilder`, `MethodBuilder` и т. д.
- В WCF используется довольно много внутренних строителей: `ChannelBuilder`, `DispatcherBuilder`, `EndpointAddressBuilder` и т. д.
- В autofac, довольно популярном IoC-контейнере, паттерн «Строитель» применяется для разделения этапов конфигурирования и использования контейнеров.

Использование порождающих паттернов в юнит-тестах

Данная врезка подразумевает, что у читателя есть представление о том, что такое юнит-тесты и для чего они нужны. Если вы не вполне уверены в своих знаниях, то стоит обратиться к другим источникам. Хорошей стартовой точкой может быть статья «Об автоматизированном тестировании» (bit.ly/OnAutomatedTesting).

Смысл юнит-тестов заключается в проверке кода на соответствие некоторым требованиям. Другими словами, тесты показывают, работает ли приложение так, как задумано. Кроме того, тесты играют важную коммуникативную роль: они выступают в роли спецификации и описывают ожидаемое поведение системы. Но для того, чтобы тесты справлялись с этой функцией, их читабельность должна быть достаточно высокой.

К сожалению, на практике тесты далеко не всегда справляются с этой задачей. Для того чтобы тесты читались как книга, они должны быть декларативными, с минимальным количеством лишних деталей. Любой тест состоит из трех основных этапов: создания контекста исполнения, выполнения операции и проверки результата (паттерн AAA — Arrange, Act, Assert). Каждый из этих этапов может быть достаточно сложным, а значит, наивный подход к написанию тестов может привести к обилию тестового кода, который сложно читать, понимать и исправлять в случае поломок.

Рассмотрим несколько простых тестов, необходимых для проверки функциональности класса LogEntryReader, который отвечает за чтение и разбор записей лог-файлов.

Наивный подход к написанию юнит-тестов

Класс `LogEntryReader` принимает в аргументах конструктора поток ввода/вывода, а также стратегию разбора записей лог-файла. Затем он создает объект `StreamReader` и делегирует основную работу объекту `ILogEntryParser` (листинг 11.5).

Листинг 11.5. Класс LogEntryReader

```
public interface ILogEntryParser
{
    bool TryParse(string s, out LogEntry logEntry);
}

public class SimpleLogEntryParser : ILogEntryParser
{
    // Простая реализация разбора записей лог-файла
    public bool TryParse(string s, out LogEntry logEntry)
    {
        // ...
    }
}

public class LogEntryReader
{
```

```
    private readonly Stream _stream;
    private readonly ILogEntryParser _logEntryParser;

    public LogEntryReader(Stream stream, ILogEntryParser logEntryParser)
    {
        _stream = stream;
        _logEntryParser = logEntryParser;
    }

    public IEnumerable<LogEntry> Read()
    {
        using (var sr = new StreamReader(_stream))
        {
            string line;
            while ((line = sr.ReadLine()) != null)
            {
                LogEntry logEntry;
                if (_logEntryParser.TryParse(line, out logEntry))
                {
                    yield return logEntry;
                }
            }
        }
    }
}
```

Класс `LogEntryReader` довольно легко покрыть тестами, поскольку его ответственность четко выражена, а зависимости достаточно простые. В тесте достаточно создать объект `MemoryStream`, заполнить его нужными данными, создать объект тестируемого класса, а затем проверить результат. Решение «в лоб» будет выглядеть так, как показано в листинге 11.6.

Листинг 11.6. Проверка чтения одной записи

```
[Test]
public void Test_Stream_With_One_Entry()
{
```

```
    // Arrange
    var memoryStream = new MemoryStream();
    StreamWriter sw = new StreamWriter(memoryStream);
    sw.WriteLine("[2014/11/01][Info] This is message!");
    sw.Flush();
    memoryStream.Position = 0;

    ILogEntryParser logEntryParser = new SimpleLogEntryParser();
    var reader = new LogEntryReader(memoryStream, logEntryParser);

    // Act
    var entries = reader.Read().ToList();

    // Assert
    Assert.That(entries.Count, Is.EqualTo(1));
    LogEntry entry = entries.Single();
    Assert.That(entry.EntryDateTime,
                Is.EqualTo(new DateTime(2014, 11, 01)));
    Assert.That(entry.Severity, Is.EqualTo(Severity.Info));
    Assert.That(entry.Message, Is.EqualTo("This is message!"));
}
```

Для простого класса с двумя зависимостями и простым поведением количество вспомогательного кода кажется весьма значительным. В реальном мире код будет еще более сложным, а значит, и читабельность тестов будет очень низкой. Умножьте ненужное число строк на количество тестов, и вы придете к неутешительным выводам. Теперь несложно догадаться, почему многие разработчики так не любят юнит-тесты: подобный код едва ли решает больше проблем, чем привносит новых.

Большинство тестовых фреймворков предоставляют возможность выполнить код инициализации перед вызовом каждого тестового метода. Для Xunit эту роль выполняет конструктор класса, а для NUnit — метод, помеченный атрибутом `SetUp` (листинг 11.7).

Листинг 11.7. Пример использования атрибута SetUp

```
private LogEntryReader _classUnderTest;

[SetUp]
public void SetUp()
{
```

```
    var memoryStream = new MemoryStream();
    StreamWriter sw = new StreamWriter(memoryStream);
    sw.WriteLine("[2014/11/01][Info] This is message!");
    sw.Flush();
    memoryStream.Position = 0;

    ILogEntryParser logEntryParser = new SimpleLogEntryParser();
    _classUnderTest = new LogEntryReader(memoryStream, logEntryParser);
}

[Test]
public void Test_Stream_With_One_Entry()
{
    // Act
    var entries = _classUnderTest.Read().ToList();

    // Assert
    Assert.That(entries.Count, Is.EqualTo(1));
    LogEntry entry = entries.Single();
    Assert.That(entry.EntryDateTime,
                Is.EqualTo(new DateTime(2014, 11, 01)));
    Assert.That(entry.Severity, Is.EqualTo(Severity.Info));
    Assert.That(entry.Message, Is.EqualTo("This is message!"));
}
```

Этот подход довольно активно применяется на практике, но у него есть ряд существенных недостатков. Обычно этапы инициализации (Arrange), исполнения (Act) и проверки (Assert) связаны между собой. На первом этапе тестируемый класс переводится в определенное состояние, на втором вызывается некоторый метод, а на третьем проверяются результаты, зависящие от начального состояния объекта.

Использование фабричного метода

Инициализация тестируемого класса в методе `SetUp` часто дает лишнюю связанность (coupling) между методом инициализации и самими тестами, что приводит к постоянному «переключению контекста» при чтении кода. К тому же мы не можем контролировать из теста процесс создания тестируемого объекта и не можем задать нужные аргументы конструктора.

Когда создание тестируемого класса может отличаться от теста к тесту, следует выделить логику создания тестируемого класса в отдельный фабричный метод (листинг 11.8).

Листинг 11.8. Использование фабричного метода в тестах

```
public static LogEntryReader CreateLogEntryReader(string content)
{
    var memoryStream = new MemoryStream();
    StreamWriter sw = new StreamWriter(memoryStream);
    sw.WriteLine(content);
    sw.Flush();
    memoryStream.Position = 0;

    ILogEntryParser logEntryParser = new SimpleLogEntryParser();
    return new LogEntryReader(memoryStream, logEntryParser);
}

[Test]
public void Test_Stream_With_One_Entry()
{
    // Arrange
    var classUnderTest = CreateLogEntryReader(
        "[2014/11/01][Info] This is message!");
    // Act
    var entries = classUnderTest.Read().ToList();

    // Assert
    Assert.That(entries.Count, Is.EqualTo(1));
    LogEntry entry = entries.Single();
    Assert.That(entry.EntryDateTime,
                Is.EqualTo(new DateTime(2014, 11, 01)));
    Assert.That(entry.Severity, Is.EqualTo(Severity.Info));
    Assert.That(entry.Message, Is.EqualTo("This is message!"));
}
```

Разница кажется небольшой, но теперь вся нужная информация для анализа логики теста находится в одном месте. Теперь метод `CreateLogEntryReader` может использоваться в разных тестах и даже разных классах, а его применение делает тест более читабельным. Данный фабричный метод также выступает в роли фасада, что упрощает адаптацию изменений тестируемого кода.

Методы `SetUp` все еще остаются очень полезным инструментом борьбы со сложностью, но они должны использоваться для инициализации контекста, одинакового для всех тестов. В общем случае сложные методы инициализации тестов говорят о проблемах дизайна и существенно усложняют сопровождаемость тестового кода.

ПРИМЕЧАНИЕ

Одним из главных недостатков тестов является их хрупкость. Изменения в поведении или интерфейсе тестируемого класса должны приводить к поломкам и изменениям тестов. Это нормально и ожидаемо. Но главное, чтобы изменения касались лишь тестов, проверяющих измененную функциональность. Если конструктор класса используется в 300 тестах напрямую, то добавление одного аргумента потребует изменения 300 строк кода. Это не слишком сложное изменение, но это лишние действия, которые отвлекают от решения основной задачи — внесения изменений. Выделение фасадных фабричных методов повышает читабельность и упрощает рефакторинг, поскольку изменять придется лишь пару методов, а не пару сотен тестов.

Паттерн Test Fixture

Фабричные методы позволяют спрятать детали создания тестируемых классов, но в некоторых случаях тесты должны сильнее контролировать этот процесс. Даже в нашем случае для отдельных тестов может потребоваться другая реализация `ILogEntryParser` или же может возникнуть необходимость передать определенные аргументы конструктора тестируемого класса, важные лишь для некоторых тестов. Тогда вместо фабричного метода следует использовать паттерн «Строитель».

Чтобы сделать тесты максимально простыми, логично отделить все аспекты по созданию тестируемого класса от самих тестов. Обычно такой класс называется `Fixture` и для рассматриваемого случая он будет выглядеть так, как показано в листинге 11.9.

Листинг 11.9. Пример класса LogEntryReaderFixture

```
// Прячет все вопросы инициализации класса LogEntryReader
internal class LogEntryReaderFixture
{
    private ILogEntryParser _parser = new SimpleLogEntryParser();
```

```
    private readonly Lazy<LogEntryReader> _lazyCut;
    private string _streamContent;

    public LogEntryReaderFixture()
    {
        _lazyCut = new Lazy<LogEntryReader>(
            () => LogEntryReaderTestFactory.Create
                                        (_streamContent, _parser));
    }

    public LogEntryReaderFixture WithParser(ILogEntryParser parser)
    {
        _parser = parser;
        return this;
    }

    public LogEntryReaderFixture WithStreamContent(string streamContent)
    {
        _streamContent = streamContent;
        return this;
    }

    public LogEntryReader LogEntryReader
    {
        get { return _lazyCut.Value; }
    }
}
```

Теперь, когда вся подготовительная работа завершена, тесты могут заниматься исключительно своей работой: проверять, что функциональность тестируемого кода соответствует ожиданиям (листинг 11.10).

Листинг 11.10. Пример использования LogEntryReaderFixture

```
[Test]
public void Test_Stream_With_One_Entry()
{
```

```
    // Arrange
    var classUnderTest = new LogEntryReaderFixture()
        .WithParser(new SimpleLogEntryParser())
        .WithStreamContent("[2014/11/01][Info] This is message!")
        .LogEntryReader;

    // Act
    var entries = classUnderTest.Read().ToList();

    // Assert
    Assert.That(entries.Count, Is.EqualTo(1));
}
```

Выделение `LogEntryReaderFixture` позволяет следовать принципу единственной обязанности. Такой подход оправдан, когда процесс инициализации сложен и фабричные методы уже не обеспечивают нужной гибкости.

ПРИМЕЧАНИЕ

Если вы устали от создания классов Fixture вручную, возможно, пришло время посмотреть на библиотеку AutoFixture, которая позволяет автоматизировать этот процесс. AutoFixture генерирует самостоятельно аргументы методов, а также позволяет создавать объекты тестируемых классов более элегантным образом. Существует интеграция AutoFixture с большинством современных IoC-фреймворков, таких как Unity, StructureMap и др., что позволяет ей находить нужные зависимости создаваемых объектов самостоятельно. Об этой библиотеке можно почитать на ее официальной странице на github: github.com/AutoFixture.

Параметризованные юнит-тесты

Во многих случаях тестируемый класс обладает достаточно простым поведением, которое выражается в получении результата в зависимости от аргументов. Например, разбор отдельной строки лог-файла — это простая операция, которая заключается в анализе содержимого строки и создании объекта `LogEntry`. Покрытие всех граничных условий этого процесса с помощью обычных тестов будет неудобно, поскольку приведет к обилию дублирующегося кода.

Параметризованные юнит-тесты помогают свести дублируемый код к минимуму и позволяют добавлять новый тестовый сценарий путем добавления одной-двух строк кода.

Разные тестовые фреймворки поддерживают разные виды параметризованных тестов, но я бы хотел здесь остановиться на примере, который использует паттерн

«Строитель». NUnit содержит атрибут `TestCaseSource`, в котором указывается фабричный метод, возвращающий описание теста: набор входных данных и ожидаемый результат (листинг 11.11).

Листинг 11.11. Пример параметризованного теста

```
[TestCaseSource("SimpleLogEntrySource")]
public LogEntry Test_SimpleLogEntry_Parse(string logEntry)
{
    // Arrange
    var parser = new SimpleLogEntryParser();

    // Act
    LogEntry result = parser.Parse(logEntry);

    // Assert
    return result;
}
```

Сам тест очень простой, и вся основная работа содержится в методе `SimpleLogEntrySource` (листинг 11.12).

Листинг 11.12. Использование строителя для описания тестовых сценариев

```
// Вспомогательный фабричный метод
private static TestCaseData Entry(string entry)
{
    return new TestCaseData(entry);
}

private static IEnumerable<TestCaseData> SimpleLogEntrySource()
{
    yield return
        Entry("[2014/01/12] [DEBUG] message")
        .Returns(new SimpleLogEntry(DateTime.Parse("2014-01-12"),
                                    Severity.Debug, "message"));

    yield return
        Entry("[2015/01/12] [Info] another message")
```

```
        .Returns(new SimpleLogEntry(DateTime.Parse("2015/01/12"),
            Severity.Info, "another message"));

    yield return
        Entry("corrupted message")
            .Throws(typeof(InvalidOperationException));
}
```

NUnit предоставляет класс TestCaseData, который является строителем, и позволяет описать входные данные метода (передаются через конструктор объекта TestCaseData), ожидаемый результат (с помощью метода Returns) или генерируемое исключение (с помощью метода Throws). Обратите внимание на использование простого фабричного метода Entry, который делает решение более строго типизированным и более читабельным.

Параметризованные тесты позволяют добавлять новый сценарий тестирования путем добавления нескольких строк кода, а также приводят к улучшению дизайна тестируемого кода. Возможность использования параметризованных юнит-тестов говорит о четких обязанностях класса и об отсутствии у тестируемых операций сложных побочных эффектов.

ПРИМЕЧАНИЕ

Более подробно о параметризованных юнит-тестах можно прочитать в моей статье «Параметризованные юнит-тесты» по ссылке http://bit.ly/Parametrized-UnitTests.

Часть III

Структурные паттерны

Любая грамотно спроектированная программная система является иерархической. В корне приложения, например в методе `Main`, создается набор высокоуровневых компонентов, каждый из которых опирается на компоненты более низкого уровня. Эти компоненты, в свою очередь, также могут быть разбиты на более простые составляющие.

Существует набор типовых решений, которые помогают бороться со сложностью, создавать и развивать современные системы. Так, например, довольно часто возникает потребность в полиморфном использовании классов, которые выполняют схожую задачу, но не обладают единым интерфейсом. Такая ситуация возможна, когда классы разрабатывались в разное время, разными людьми, а иногда и разными организациями. Связать такие классы вместе позволит паттерн **«Адаптер»**. Он дает возможность подстроить разные реализации к одному интерфейсу и использовать их полиморфным образом даже тогда, когда часть существующего кода находится вне вашего контроля.

Еще одним распространенным структурным паттерном является **«Фасад»**. Он представляет высокоуровневый интерфейс к сторонней библиотеке или модулю, что упрощает код клиентов и делает их менее зависимыми от стороннего кода. Это позволяет существенно упростить логику приложения и делает ее менее зависимой от ошибок и изменений сторонних библиотек. Фасад упрощает миграцию на новую версию библиотеки, поскольку придется не проверять весь код приложения, а лишь запустить тесты фасада и убедиться в том, что поведение осталось неизменным.

Паттерн **«Декоратор»** позволяет нанизывать дополнительные аспекты поведения один на другой без создания чрезмерного числа производных классов. Декоратор прекрасно справляется с задачами кэширования, логирования или с ограничением числа вызовов. При этом за каждый аспект поведения будет отвечать выделенный класс, что упростит понимание и развитие системы.

Многим приложениям приходится работать с иерархическими данными. При этом часто возникает потребность скрыть иерархическую природу или упростить работу с ней таким образом, чтобы приложение работало с составными и одиночными объектами унифицированным образом. Паттерн **«Компоновщик»** предназначен для решения именно этой задачи: он позволяет объединить простые элементы

в составные объекты и дает возможность клиентам работать с такими структурами единообразно.

Современные системы невозможно представить без удаленного взаимодействия. И в этой области паттерн **«Прокси»** давно стал классическим. Никто уже не может представить себе, что для взаимодействия с удаленным сервером код приложения будет создавать TCP соединение, сериализовать аргументы и анализировать результаты[1]. Все современные библиотеки, такие как WCF или .NETRemoting, используют прокси-классы, которые сглаживают грань между внутрипроцессным и межпроцессным взаимодействием.

[1] Это не значит, что приложение не может использовать самописные протоколы сетевого взаимодействия. Но даже если вам не подошел WCF и вы решили работать с TCP-соединениями напрямую, то все равно логично спрятать низкоуровневые подробности в самодельном прокси-классе, а не «размазывать» эту логику по коду приложения.

Глава 12

Паттерн «Адаптер» (Adapter)

Назначение: преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер делает возможной совместную работу классов с несовместимыми интерфейсами.

Мотивация

Далеко не все системы обладают прекрасным дизайном. Даже если модуль или приложение были хорошо продуманы изначально, внесение изменений разными разработчиками в течение длительного времени может привести к неприятным последствиям. Одно из таких последствий — рассогласованность реализации однотипных задач.

Вернемся к приложению для импорта лог-файлов. На ранних этапах разработки может быть неочевидным, появится ли необходимость в обобщенном решении, базовых классах и гибкости. Если изначально было принято решение хранить данные в SqlServer, то вместо использования иерархии наследования разработчики создали простой класс `SqlServerLogSaver`. Со временем требования или понимание задачи могли измениться, в результате чего команда решила добавить еще одно хранилище данных, например Elasticsearch. Новый класс, `ElasticsearchLogSaver`, проектировал другой разработчик, в результате чего он стал обладать интерфейсом, отличным от интерфейса своего предшественника (рис. 12.1).

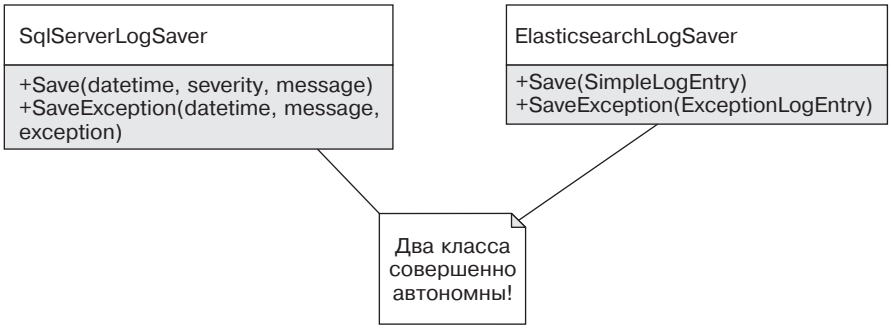


Рис. 12.1. Диаграмма классов SqlServerLogSaver и ElasticsearchLogSaver

Классы сохранения данных являются независимыми (что хорошо), но это не позволяет использовать их полиморфным образом (что плохо). Решение заключается в создании еще одного абстрактного слоя путем выделения интерфейса `ILogSaver`. Но вместо того, чтобы изменять существующие классы и реализовывать в них новый интерфейс, можно создать промежуточный слой адаптеров, которые будут связывать старый и новый миры воедино (рис. 12.2).

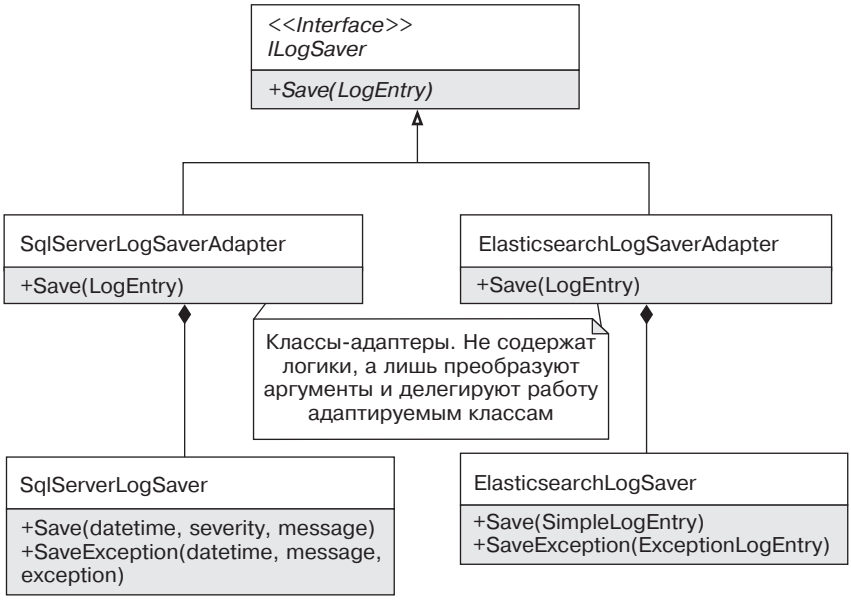


Рис. 12.2. Использование адаптеров для сохранения лог-файлов

Реализация каждого из адаптеров будет довольно тривиальной. Нужно преобразовать входные параметры, вызвать метод адаптируемого класса и, возможно, преобразовать полученный результат (листинг 12.1).

Листинг 12.1. Реализация SqlServerLogSaverAdapter

```
public class SqlServerLogSaverAdapter : ILogSaver
{
    private readonlySqlServerLogSaver _sqlServerLogSaver =
                                      new SqlServerLogSaver();

    public void Save(LogEntrylogEntry)
    {
        var simpleEntry = logEntry as SimpleLogEntry;
        if (simpleEntry != null)
        {
            _sqlServerLogSaver.Save(simpleEntry.EntryDateTime,
                simpleEntry.Severity.ToString(),
                simpleEntry.Message);
            return;
        }

        varexceptionEntry = (ExceptionLogEntry) logEntry;
        _sqlServerLogSaver.SaveException(
            exceptionEntry.EntryDateTime, exceptionEntry.Message,
            exceptionEntry.Exception);
    }
}
```

Адаптер — это клей, который связывает воедино два мира путем подгонки текущих классов к требуемому интерфейсу.

Классическая диаграмма классов паттерна «Адаптер» приведена на рис. 12.3.

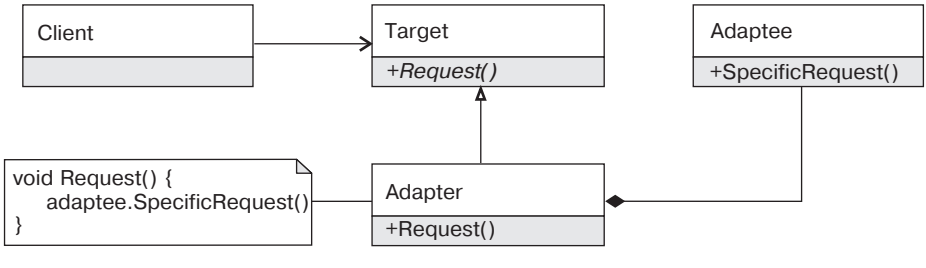


Рис. 12.3. Классическая диаграмма классов паттерна «Адаптер»

Участники:

- `Target` (`ILogSaver`) — целевой интерфейс, к которому нужно преобразовать интерфейс существующих классов;
- `Adaptee` (`SqlServerLogSaver`) — существующий класс, чей интерфейс нужно преобразовать;
- `Adapter` (`SqlServerLogSaverAdapter`) — класс-адаптер, который преобразует интерфейс адаптируемого класса к целевому;
- `Client` (клиенты `ILogSaver`) — клиенты нового интерфейса, которые работают с адаптированными классами полиморфным образом.

Обсуждение паттерна «Адаптер»

Адаптер является одним из тех паттернов проектирования, которые мы используем, почти не задумываясь. Диаграмма классов этого паттерна настолько общая, что практически любую композицию объектов можно считать примером использования адаптеров.

Адаптер классов и объектов

«Бандой четырех» были описаны два вида адаптеров — адаптеры классов и адаптеры объектов. Ранее был рассмотрен пример адаптера объектов. В этом случае создается новый класс, который реализует требуемый интерфейс и делегирует всю работу адаптируемому объекту, хранящемуся в виде закрытого поля.

Классический адаптер классов использует множественное наследование. Адаптер реализует новый интерфейс, но также использует адаптируемый класс в качестве базового класса. Обычно в этом случае используется закрытое наследование, что предотвращает возможность конвертации адаптера к адаптируемому объекту (рис. 12.4).

Наследование от адаптируемого объекта позволяет получить доступ к защищенному представлению, а также реализовать лишь несколько методов, если новый интерфейс не слишком отличается от интерфейса адаптируемого класса.

В языке C# отсутствует множественное наследование реализации, что делает адаптеры классов неприменимыми. В некоторых случаях можно воспользоваться упрощенной версией адаптеров классов — адаптерами интерфейсов (рис. 12.5).

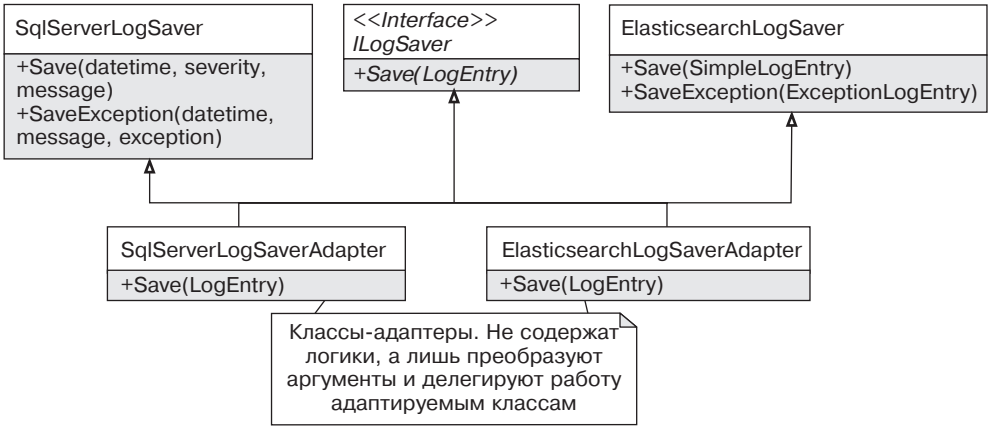


Рис. 12.4. Пример адаптера классов

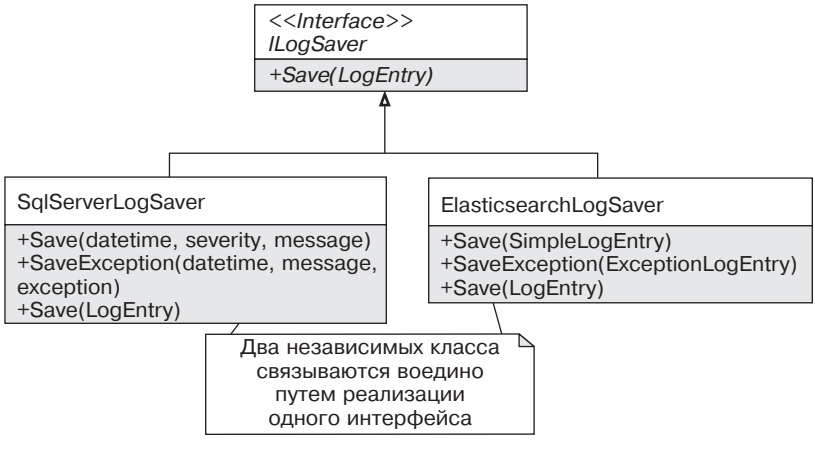


Рис. 12.5. Пример адаптера интерфейсов

Адаптер интерфейсов не всегда можно применить, поскольку это требует изменения адаптируемого класса. Но, если такое изменение возможно, это может быть самой простой реализацией паттерна «Адаптер».

Адаптивный рефакторинг

Зачастую адаптеры применяются для адаптации классов к существующим интерфейсам. Но в некоторых случаях адаптеры могут упростить эволюцию дизайна и рефакторинг.

Давайте снова вернемся к задаче импорта логов, но подойдем к проблеме сохранения логов с другой стороны. Предположим, что у нас есть иерархия классов `LogSaver`, но мы хотим модифицировать ее таким образом, чтобы операция сохранения стала асинхронной.

Можно, конечно, переписать все классы этой иерархии, а можно пойти более итеративным путем.

1. Проектируем новый интерфейс `IAsyncLogSaver` с набором нужных асинхронных методов сохранения.
2. Создаем первую реализацию `AsyncLogSaverAdapter`, которая реализует `IAsyncLogSaver`, но делегирует все операции старой реализации.
3. Переводим клиентов иерархии `LogSaver` на асинхронные рельсы и проверяем, что клиенты работают нормально.
4. Создаем полноценную асинхронную реализацию сохранения логов.
5. Удаляем `AsyncLogSaverAdapter` за ненадобностью.

Диаграмма классов этого решения будет выглядеть так (рис. 12.6).

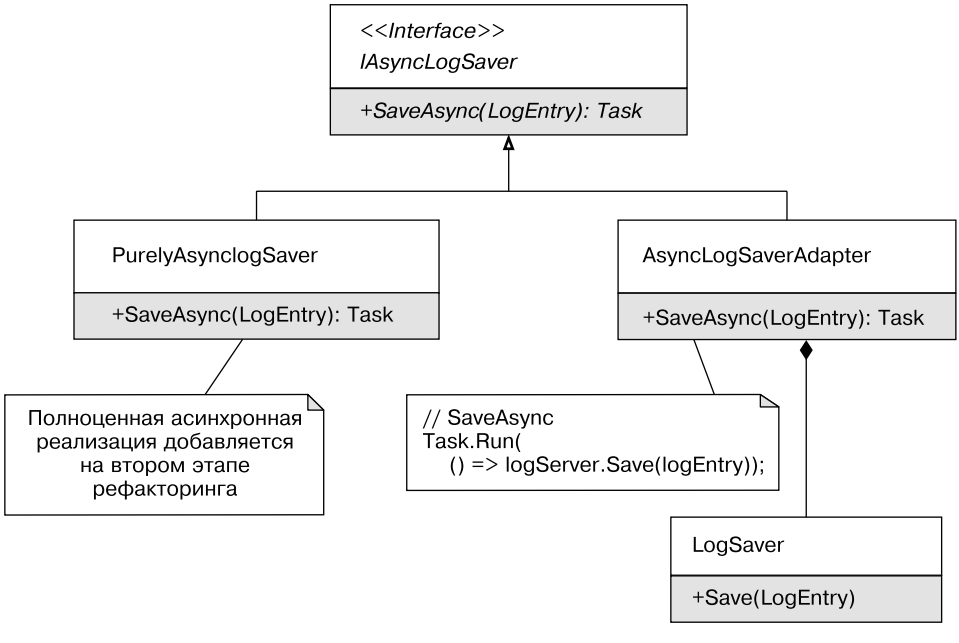


Рис. 12.6. Диаграмма классов адаптивного рефакторинга

Преимущество такого подхода в том, что изменения происходят постепенно, а не одним большим скачком. Реализация `AsyncLogSaverAdapter` будет примитивной, но это позволит уменьшить количество ошибок в единицу времени (листинг 12.2).

Листинг 12.2. Реализация класса AsyncLogSaverAdapter

```
public interface IAsyncLogSaver
{
    Task SaveAsync(LogEntrylogEntry);
}

public class AsyncLogSaverAdapter : IAsyncLogSaver
{
    private readonlyLogSaver _logSaver;

    public AsyncLogSaverAdapter(LogSaverlogSaver)
    {
        _logSaver = logSaver;
    }

    public Task SaveAsync(LogEntrylogEntry)
    {
        return Task.Run(() => _logSaver.Save(logEntry));
    }
}
```

Языковые адаптеры

Компилятор языка C# налагает определенные ограничения на применение пользовательских типов в некоторых языковых конструкциях. Например, для использования типа в LINQ-выражениях вида `fromsinobj` требуются экземплярные методы `Select`, `Where`, `OrderBy` и др., но подойдут и методы расширения. Для использования в цикле `foreach` типу необходимо предоставить метод `GetEnumerator`, который возвращает тип с методом `boolMoveNext` и свойством `Current`. Для применения `async/await` требуется метод `GetAwaiter`

и `GetResult`[1]. Для использования инициализатора коллекций требуется реализация интерфейса `IEnumerable<T>` (или `IEnumerable`) и метода `Add`.

Вместо того чтобы модифицировать код приложения так, чтобы он удовлетворял этим требованиям, можно создать набор классов-адаптеров или методов расширения. Начиная с 6-й версии языка C#, для использования инициализатора коллекций достаточно иметь метод расширения `Add` (листинг 12.3).

Листинг 12.3. Класс CustomCollection и его методы расширения

```
public class CustomCollection : IEnumerable<int>
{
    private readonly List<int> _list = new List<int>();

    public void Insert(int value)
    {
        _list.Add(value);
    }

    public IEnumerator<int>GetEnumerator()
    {
        return _list.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public static class CustomCollectionExtensions
{
    public static void Add(this CustomCollectioncustomCollection, int value)
    {
```

[1] Все это примеры «утиной» типизации (ducktyping) на уровне языка программирования.

```
            customCollection.Insert(value);
        }
    }
```

В этом случае класс `CustomCollectionExtensions` выступает своеобразным адаптером, который позволяет использовать класс `CustomCollection` с инициализатором коллекций (листинг 12.4).

Листинг 12.4. Пример использования инициализатора коллекций с CustomCollection

```
// При наличии импорта нужного пространства имен
// код успешно компилируется и запускается!
varcc = newCustomCollection {42};
```

Применимость

Адаптер позволяет использовать существующие типы в новом контексте.

- **Повторное использование чужого кода.** В некоторых случаях у нас уже есть код, который решает нужную задачу, но его интерфейс не подходит для текущего приложения. Вместо изменения кода библиотеки можно создать слой адаптеров.
- **Адаптивный рефакторинг.** Адаптеры позволяют плавно изменять существующую функциональность путем выделения нового «правильного» интерфейса, но с использованием старой проверенной функциональности.

Примеры в .NET Framework

- `TextReader/TextWriter` — адаптеры над классами `Stream` для чтения/записи текстовых данных в потоки ввода/вывода.
- `BinaryReader/BinaryWriter` — аналогичные адаптеры для работы с бинарными данными потоков ввода/вывода.
- `ReadonlyCollection<T>` — адаптирует произвольный список (`IList<T>`) к коллекции только для чтения (`IReadonlyCollection<T>`).
- Любая реализация LINQ-провайдера (интерфейса `IQueryProvider`) служит своеобразным адаптером, поскольку подстраивает классы для работы с внешним источником данных к интерфейсу `IQueryProvider`.

Глава 13

Паттерн «Фасад» (Facade)

Назначение: предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Мотивация

Библиотеки классов обычно предназначены для решения целого спектра задач. С помощью ADO.NET можно выполнять простые команды на стороне базы данных, а можно использовать транзакции и выходные (output) параметры хранимых процедур. Наличие сложных сценариев делает библиотеку полезной, но это же может усложнить решение с ее помощью простых задач. Это приводит к появлению всевозможных оболочек даже над стандартными библиотеками, которые упрощают решение простых задач, или сценариев, специфичных для конкретного приложения.

Такие оболочки являются фасадами, которые скрывают исходную сложность библиотеки или модуля за более простым и, возможно, специфичным для приложения интерфейсом.

Давайте рассмотрим конкретный пример небольшой оболочки, которая упрощает работу с SQLServer (рис. 13.1).

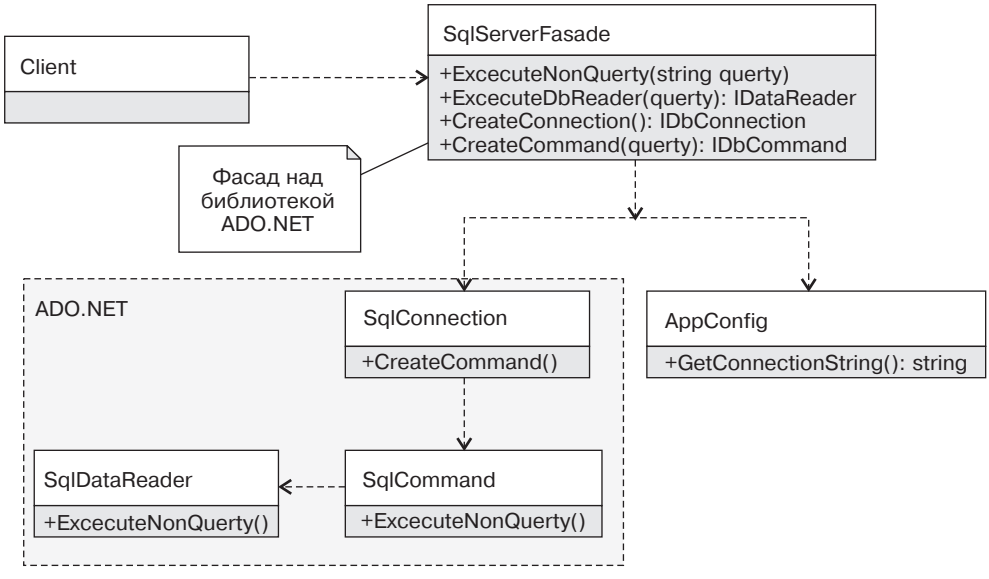


Рис. 13.1. Фасад для работы с SQLServer

В большинстве приложений используется лишь часть функциональности сложной библиотеки, и фасад позволяет сделать более простой интерфейс, максимально подходящий для специфических сценариев. Класс `SqlServerFacade` выступает в роли фасада, который упрощает решение приложением типовых задач взаимодействия с базой данных: прячет логику получения строки подключения, а также ряд второстепенных деталей библиотеки ADO.NET.

Классическая диаграмма классов паттерна фасад приведена на рис. 13.2.

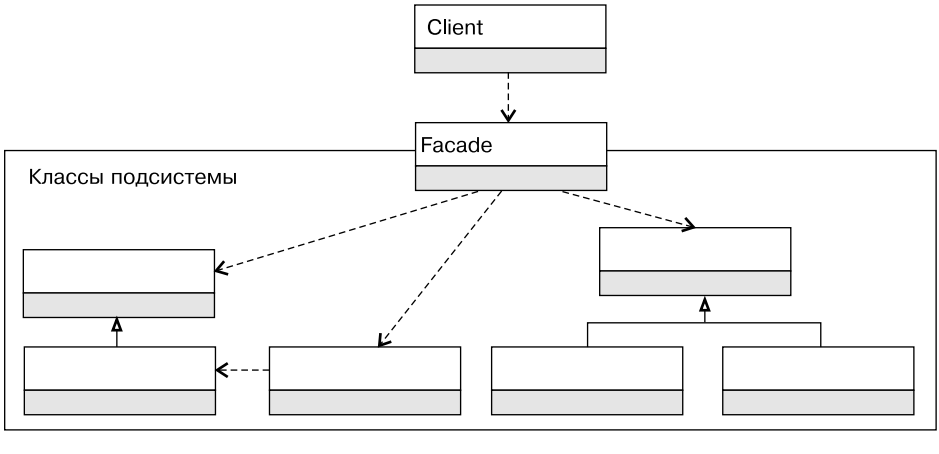


Рис. 13.2. Классическая диаграмма классов паттерна Фасад

Участники:

- `Facade` (`SqlServerFacade`) — фасадный класс, который прячет детали реализации подсистемы от клиентов;
- `Client` — клиент фасада, который работает с фасадом, а не с классами подсистемы.

Обсуждение паттерна «Фасад»

Очень многие библиотеки или подсистемы приложений содержат встроенные фасады, которые являются высокоуровневыми классами, предназначенными для решения типовых операций. Фасады делают базовые сценарии простыми, а сложные сценарии — возможными. Если клиенту нужна лишь базовая функциональность, достаточно воспользоваться фасадом; если же его функциональности недостаточно, можно использовать более низкоуровневые классы модуля или библиотеки напрямую.

Инкапсуляция стороннего кода

Использование фасадов не только упрощает использование библиотек или сторонних компонентов, но и решает ряд насущных проблем.

- Повторное использование кода и лучших практик. Многие библиотеки довольно сложные, поэтому их корректное использование требует определенных навыков. Инкапсуляция работы с ними в одном месте позволяет корректно использовать их всеми разработчиками независимо от их опыта.
- Переход на новую версию библиотеки. При выходе новой версии библиотеки достаточно будет протестировать лишь фасад, чтобы принять решение, стоит на нее переходить или нет.
- Переход с одной библиотеки на другую. Благодаря фасаду приложение не так сильно завязано на библиотеку, так что переход на другую библиотеку потребует лишь создание еще одного фасада. А использование адаптера[1] сделает этот переход менее болезненным.

Повышение уровня абстракции

Фасад повышает уровень абстракции и упрощает решение задач, специфичных для текущего приложения. Фасад скрывает низкоуровневые детали, но может

[1] Паттерн «Адаптер» был рассмотрен в главе 12.

предоставлять интерфейс, специфичный для конкретного приложения, за счет использования в качестве параметров доменных объектов.

Например, при использовании фасада для работы с `SqlServer` класс `SqlServerProvider` может принимать класс `Configuration`, из которого он будет получать строку подключения и другие параметры (листинг 13.1).

Листинг 13.1. Пример использования фасадом доменного типа

```
public static IDbCommandCreateCommand(Configuration config) {...}
```

Применимость

Я предпочитаю использовать фасады для работы с большинством сторонних библиотек или подсистем. Это уменьшает связанность (coupling) системы с внешними зависимостями, позволяет лучше понять сторонний код, контролировать качество новых версий, а также избавляет код приложения от излишних низкоуровневых деталей.

Фасад **не нужно применять**, когда в повышении уровня абстракции нет никакого смысла. В большинстве случаев нет смысла в фасаде для библиотеки логирования, достаточно выбрать одно из решений (log4net, NLog, .NETTraces) и использовать его в коде приложения. Фасады бесполезны, когда имеют громоздкий интерфейс, и пользоваться ими сложнее, чем исходной библиотекой.

Примеры в .NET Framework

- `XmlSerializer` прячет сложность сериализации, генерацию временной сборки и многие другие низкоуровневые подробности.
- Класс `ThreadPool.QueueUserWorkItem` является фасадом для работы с пулом потоков.
- Класс `Parallel` из TPL является фасадом для упрощения параллельного программирования. Он скрывает низкоуровневые детали работы планировщиков, стратегии разбивки данных (partitioning) и другие подробности создания объектов `Task<T>` и управления ими.
- Класс `System.Runtime.CompilerServices.RuntimeHelpers` является фасадом с низкоуровневыми служебными операциями, такими как вызов статического конструктора типа и т. п.
- Класс `System.Console` является фасадом для работы с консолью ввода-вывода.

Глава 14

Паттерн «Декоратор» (Decorator)

Назначение: динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Мотивация

Хорошо спроектированный класс отвечает за определенную функциональность, не распыляясь на решение второстепенных задач. Но что делать, если второстепенные задачи, такие как логирование[1], кэширование, замеры времени исполнения, проникают в код класса и непомерно увеличивают сложность реализации? Можно выделить эти аспекты поведения во вспомогательные классы, но все равно останется проблема их координации. Паттерн «Декоратор» элегантно решает задачу нанизывания одних обязанностей на другие.

---

[1] Использование декораторов для обычного логирования в подавляющем большинстве случаев будет неоправданно. В некоторых случаях мне приходилось выделять из сложного класса часть, ответственную за логирование. Но поскольку трассировочные сообщения переплетаются с вызовами сторонних методов, реализовать это с помощью декораторов довольно сложно. Однако некоторые аспекты поведения очень хорошо логировать с помощью декораторов. Например, очень удобно замерять время исполнения методов.

Давайте рассмотрим задачу импорта логов несколько с иной стороны. Помимо консольной утилиты или локального Windows-сервиса, мы можем разработать облачное приложение, которое будет принимать трассировочные сообщения от приложений пользователя и предоставлять интерфейс для последующего поиска[1]. Большинство облачных приложений не позволяют конкретному пользователю передавать произвольное количество данных. При достижении определенного лимита, например десять сообщений в секунду от конкретного пользователя, включается режим ограничений входящих запросов (throttling).

В простом случае логику троттлинга можно смешать с логикой сохранения, но такое решение сложно поддерживать в длительной перспективе. Вместо этого можно воспользоваться декоратором (листинг 14.1).

Листинг 14.1. Исходный класс декоратора сохранения данных

```
public interface ILogSaver
{
    Task SaveLogEntry(string applicationId, LogEntrylogEntry);
}

public sealed class ElasticsearchLogSaver : ILogSaver
{
    public Task SaveLogEntry(string applicationId, LogEntrylogEntry)
    {
        // Сохраняем переданную запись в Elasticsearch
        return Task.FromResult<object>(null);
    }
}

public abstract class LogSaverDecorator : ILogSaver
{
    protected readonly ILogSaver _decoratee;
```

[1] Именно этим занимается ApplicationInsights — набор облачных сервисов от компании Microsoft, которые принимают, хранят и анализируют телеметрические данные приложений. ApplicationInsights предоставляет API для сохранения телеметрических данных и веб-интерфейс для анализа сохраненных данных, включая полнотекстовый поиск в логах, исключениях и других данных.

```
    protected LogSaverDecorator(ILogSaver decoratee)
    {
        _decoratee = decoratee;
    }

    public abstract Task SaveLogEntry(
        string applicationId, LogEntry logEntry);
}
```

Идея паттерна «Декоратор» в том, что у интерфейса (`ILogSaver`) появляется два вида реализаций: основная реализация бизнес-функциональности (`ElasticsearchLogSaver`) и набор классов-декораторов, которые реализуют тот же интерфейс, но довольно специфическим образом. Декоратор принимает в конструкторе тот же самый интерфейс, а в реализации делегирует работу декорируемому объекту с присоединением некоторого поведения до или после вызова метода (листинг 14.2).

Листинг 14.2. Реализация декоратора ThrottlingLogSaverDecorator

```
public class ThrottlingLogSaverDecorator : LogSaverDecorator
{
    public ThrottlingLogSaverDecorator(ILogSaverdecoratee)
        : base(decoratee)
    {}

    public override async Task SaveLogEntry(
        string applicationId, LogEntrylogEntry)
    {
        if (!QuotaReached(applicationId))
        {
            IncrementUsedQuota();

            // Сохраняем записи. Обращаемся к декорируемому объекту!
            await _decoratee.SaveLogEntry(applicationId, logEntry);
            return;
        }
```

```
            // Сохранение невозможно! Лимит приложения исчерпан!
            throw new QuotaReachedException();
        }

        private boolQuotaReached(string applicationId)
        {
            // Проверяем, израсходована ли квота приложения
        }

        private void IncrementUsedQuota()
        {
            //...
        }
}
```

Декоратор с ограничением числа вызовов (`ThrottlingLogSaverDecorator`) вначале проверяет, не исчерпано ли число запросов со стороны текущего приложения, и лишь затем делегирует работу основному объекту (полю `decoratee`). Если квота не достигла лимита, то вызывается основной метод и данные успешно сохраняются. В противном случае генерируется исключение `QuotaReached-Exception`, которое прикажет клиенту попробовать выполнить тот же самый запрос через некоторое время[1].

Теперь останется правильно сконструировать экземпляр `ILogSaver` и «навесить» на него нужный набор декораторов. При этом клиенты интерфейса будут работать с ним, как и раньше, не замечая наличия дополнительного поведения (листинг 14.3).

Листинг 14.3. Пример инициализации декоратора

```
ILogSaverlogSaver = new ThrottlingLogSaverDecorator(
    new ElasticsearchLogSaver());

var controller = new LogSaverController(logSaver);
```

[1] Если функциональность сохранения записей лог-файлов находится в сервисе, доступном по протоколу HTTP, то можно даже воспользоваться специальным возвращаемым статусом 429 — TooManyRequests, чтобы сообщить клиенту о чрезмерной нагрузке с его стороны.

Классическая диаграмма классов паттерна «Декоратор» приведена на рис. 14.1.

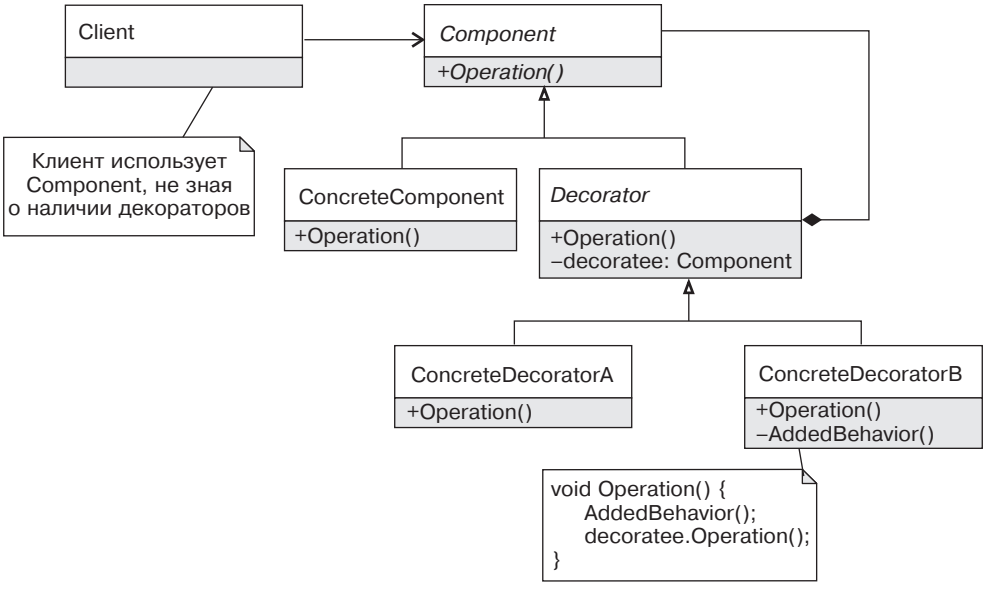


Рис. 14.1. Классическая диаграмма классов паттерна «Декоратор»

Участники:

- `Component` (`ILogSaver`) — базовый класс компонента, чье поведение будет расширяться декораторами;
- `Client` (`LogSaverController`) — работает с компонентом, не зная о существовании декораторов;
- `ConcreteComponent` (`ElasticsearchLogSaver`) — конкретная реализация компонента;
- `Decorator` (`LogSaverDecorator`) — базовый класс декоратора, предназначенный для расширения поведения компонента;
- `ConcreteDecoratorA` (`ThrottlingLogSaverDecorator`) — конкретный декоратор, который добавляет декорируемому объекту специфическое поведение.

Обсуждение паттерна «Декоратор»

Композиция vs. наследование

Задачу контроля количества сохраненных лог-файлов можно было бы решить и с помощью наследования. Для этого достаточно было создать наследник

ElasticsearchLogSaver и добавить в него логику ограничения числа запросов (рис. 14.2).

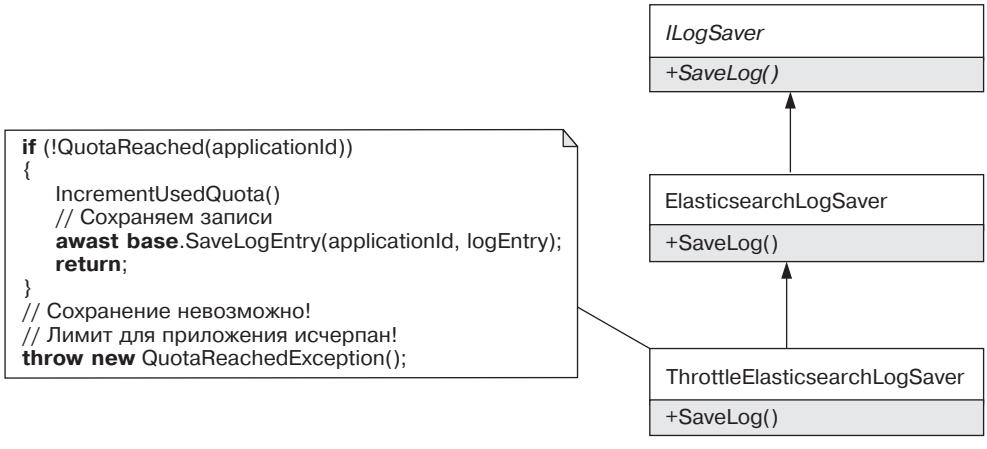


Рис. 14.2. Троттлинг на основе наследования

Разница между этими подходами в том, что наследование обеспечивает более жесткую связь по сравнению с агрегацией. Класс ThrottleElasticsearchLogSaver жестко привязан к своему базовому классу и не может быть использован с другой реализацией интерфейса ILogSaver.

Декораторы же могут использоваться с любой реализацией интерфейса ILogSaver, включая другие декораторы. Агрегация позволяет добавлять поведение объектам во время исполнения, а наследование расширяет поведение лишь во время компиляции. Гибкость агрегации позволяет нанизывать несколько декораторов друг на друга, создавая довольно сложное поведение.

Помимо троттлинга, легко добавить еще несколько реализаций декораторов, например для кэширования или замера времени исполнения (листинг 14.4).

Листинг 14.4. Использование нескольких декораторов

```
// Декоратор для замера времени исполнения метода
public class TraceLogSaverDecorator : LogSaverDecorator
{
    publicTraceLogSaverDecorator(ILogSaver decoratee) : base(decoratee)
    {}
```

```
    public override async Task SaveLogEntry(string applicationId,
        LogEntry logEntry)
    {
        var sw = Stopwatch.StartNew();
        try
        {
            await _decoratee.SaveLogEntry(applicationId, logEntry);
        }
        finally
        {
            Trace.TraceInformation("Операция сохранения завершена за {0}мс",
                sw.ElapsedMilliseconds);
        }
    }
}

ILogSaver logSaver = new ThrottlingLogSaverDecorator(
        new TraceLogSaverDecorator(new ElasticsearchLogSaver()));

// Используем logSaver с двумя декораторами
```

Инициализация декораторов

Наличие декораторов делает процесс инициализации компонентов более сложным. В простых случаях код инициализации может находиться в корне приложения, например в методе `Main`. В более сложных случаях декоратор может инициализироваться в IoC-контейнере или же создаваться фабрикой.

Если есть ряд предустановленных конфигураций, то достаточно выделить фабричный метод, отвечающий за создание объекта (листинг 14.5).

Листинг 14.5. Фабричный метод создания объекта ILogSaver

```
public static class LogSaverFactory
{
    public static ILogSaverCreateLogSaver()
    {
```

```
        return
            new ThrottlingLogSaverDecorator(
                new TraceLogSaverDecorator(
                    new ElasticsearchLogSaver()));
    }
}
```

Статический фабричный метод не позволит изменить конфигурацию декоратора на лету, но обычно такой гибкости вполне достаточно. При изменении требований достаточно будет заменить реализацию фабрики и развернуть приложение заново.

Недостатки декораторов

Декоратор, как и большинство других паттернов, имеет некоторые недостатки.

- **Чувствительность к порядку.** Код инициализации декораторов очень важен, поскольку именно в процессе создания определяются вложенность и порядок исполнения разных декораторов.
- **Сложность отладки.** Разработчику, незнакомому с этим паттерном, замер времени исполнения или кэширование результатов декораторами может показаться черной магией. Отлаживать проблемы, которые возникли внутри декоратора, может быть довольно сложно.
- **Увеличение сложности.** Декоратор является довольно тяжеловесным паттерном, к которому стоит прибегать тогда, когда выделяемый аспект поведения достаточно сложен. Если нужно кэшировать результаты в одном из десяти методов, то сложность, привнесенная декоратором, будет неоправданна.

Генерация декораторов

Многие IoC-контейнеры, такие как Unity или StructureMap, поддерживают генерацию декораторов на лету с помощью перехватчиков (Interceptors). Идея заключается в генерации IL-кода (IntermediateLanguage) во время исполнения, который будет выполнять пользовательский код до или после вызова декорируемого метода. Я предпочитаю использовать более простые решения и переходить к средствам вроде генерации кода лишь в случае необходимости. Если вас интересует эта тема, достаточно запросить в поисковом сервисе (кто сказал «Гугл»?) материалы по теме Unityinterceptors и найти массу примеров.

Применимость

Декоратор позволяет динамически расширять поведение объектов. Он идеально подходит для расширения поведения всех методов интерфейса, которое не является частью основной функциональности. Если кэшировать нужно лишь результаты одного метода класса, то использование декоратора будет слишком тяжеловесным.

Декораторы применяются для добавления всем методам интерфейса некоторого поведения, которое не является частью основной функциональности. Декораторы отлично подходят для решения следующих задач:

- кэширования результатов работы;
- замера времени исполнения методов;
- логирования аргументов;
- управления доступом пользователей;
- модификации аргументов или результата работы методов упаковки/распаковки, шифрования и т. п.

Динамическая природа позволяет нанизывать аспекты один на другой, обходя ограничения наследования, использование которого привело бы к комбинаторному взрыву числа наследников.

ПРИМЕЧАНИЕ

За последние несколько лет я неоднократно применял декораторы на практике. Пример, рассмотренный в разделе «Мотивация», основан на практическом опыте использования декоратора в одном из сервисов ApplicationInsights и решает задачу исключения чрезмерного числа запросов со стороны одного пользователя.

До этого я неоднократно использовал декораторы для замера длительности вызова методов, а также кэширования. Пример использования декораторов для кэширования объектов Task я описывал в статье «Кэширующий декоратор на деревьях выражений» (http://bit.ly/CachedDecorator).

Примеры в .NET Framework

В .NET Framework существует довольно большое количество декораторов. Большая их часть предназначена для работы с потоками ввода/вывода, но есть и исключения.

- `System.IO.BufferedStream` — добавляет буферизацию потоку ввода/вывода.
- `System.IO.Compression.GZipStream`, `System.IO.Compression.DeflateStream` — добавляют возможности сжатия потоку ввода/вывода.
- `System.CodeDom.Compiler.IndentedTextWriter` — управляет форматированием объекта `System.IO.TextWriter`.
- `System.Reflection.TypeDelegator` — декоратор для добавления дополнительных аспектов поведения объекту `System.Type`.
- `System.Collections.SortedList.SyncSortedList` — декоратор, который является вложенным классом `SortedList` и вызывает все методы декорируемого списка внутри конструкции `lock`.

Потоки ввода/вывода в .NET

Потоки ввода/вывода — это одна из областей, в которой отчетливо видно влияние паттернов проектирования. Я довольно длительное время путался в обязанностях между классами `TextReader`, `StreamReader`, `StringReader` и др. Если у вас была схожая проблема, то этот раздел позволит расставить все по своим местам.

Чтобы понять обязанности классов пространства имен `System.IO`, нужно нарисовать диаграмму с ключевыми классами и их отношениями (рис. 14.3).

Потоки ввода/вывода используют следующие ключевые паттерны.

- Абстракция потоков ввода/вывода: классы `System.IO.Stream` и наследники — `FileStream`, `MemoryStream`, `NetworkStream` и др. Стратегия потоков позволяет абстрагироваться от конкретной реализации и работать с разными потоками ввода/вывода единообразно.
- Декораторы (`BufferedStream`, `GZipStream` и др.). Добавляют определенное поведение потокам ввода/вывода, типа буферизации, сжатия и т. п. Многие другие библиотеки, такие как WCF, добавляют свои декораторы для собственных нужд.
- Адаптеры (текстовые `TextReader`/`TextWriter` и бинарные `BinaryReader`/`BinaryWriter`). Упрощают чтение специфических данных из любых потоков ввода/вывода, например чтение текстовых данных или примитивных данных из двоичного потока.
- Фасады и фабрики. Класс `File` является фасадом, который упрощает работу с файлами. Он же содержит ряд фабричных методов для открытия файлов в текстовом/двоичном режимах, добавления данных или создания нового файла (`File.OpenRead`, `File.OpenWrite`, `File.Create`, `File.AppendText`, `File.OpenText` и др.).

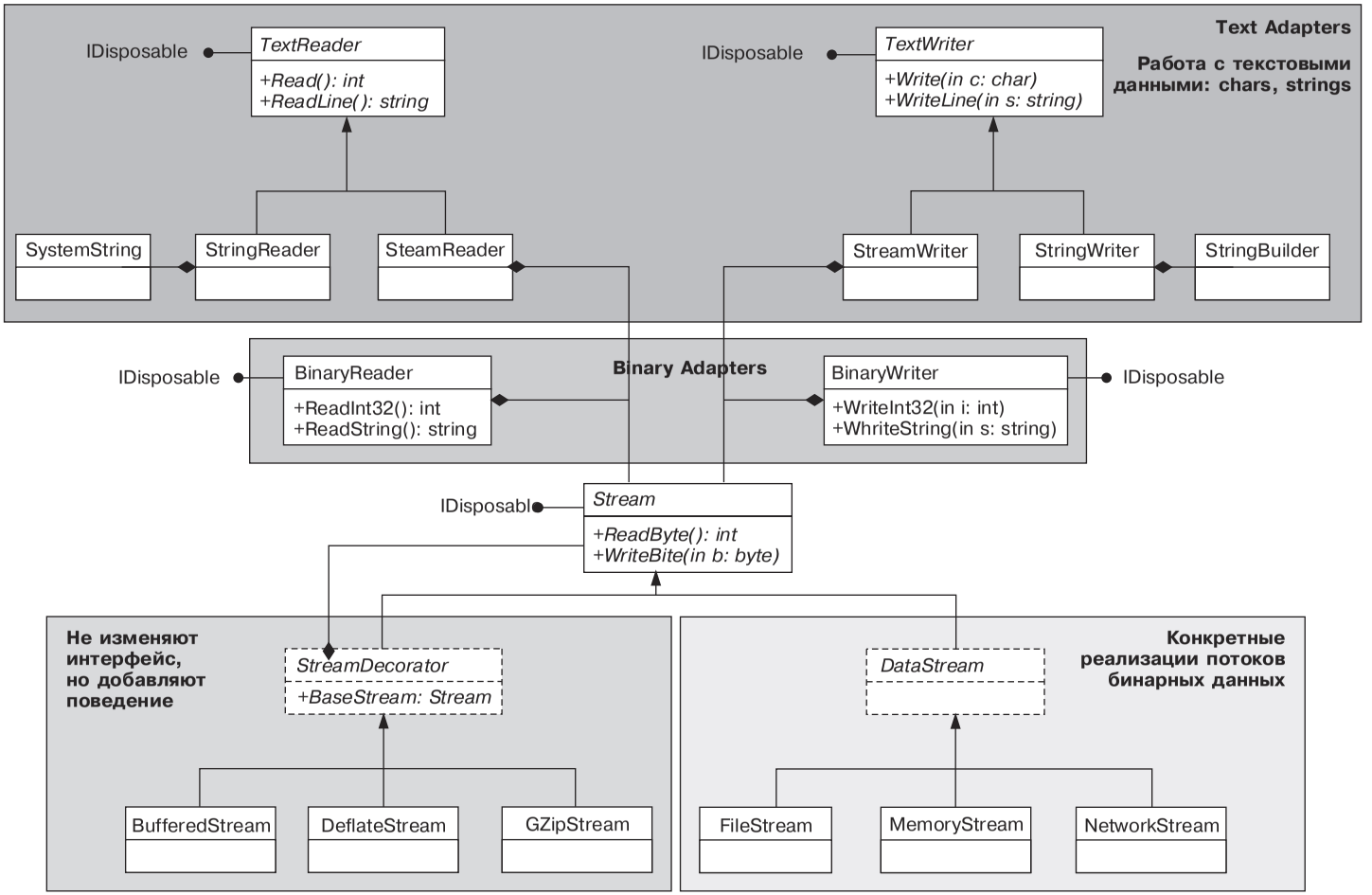


Рис. 14.3. Классы пространства имен System.IO

Встроенная поддержка тестируемости

Библиотека потоков ввода/вывода содержит множество абстракций, которые позволяют сделать решение тестируемым. Классы `TextReader`/`TextWriter` отлично подходят для этих целей, поскольку, помимо классов `StreamReader`/`StreamWriter`, которые оперируют потоками ввода/вывода, существуют классы `StringReader`/`StringWriter`, которые оперируют строками.

Это значит, что для обеспечения тестируемости классам достаточно работать не с потоками ввода/вывода, а с классами `TextReader`/`TextWriter` и не изобретать собственные абстракции[1] (листинг 14.6).

Листинг 14.6. Класс LogEntryParser

```
public class LogEntryParser
{
    private readonly TextReader _reader;

    public LogEntryParser(TextReader reader)
    {
        _reader = reader;
    }

    public IEnumerable<LogEntry> Parse()
    {
        string line;
        while ((line = _reader.ReadLine()) != null)
        {
            yield return ParseLine(line);
        }
    }

    private LogEntry ParseLine(string line)
    {
```

[1] Пример тестируемости классов на основе StreamReader уже был рассмотрен в главе о паттерне «Фабричный метод». Здесь он дублируется в несколько измененном виде для наглядности.

```
        return new SimpleLogEntry();
    }
}
```

В тесте достаточно создать строку с нужным содержимым и затем передать ее конструктору `StringReader` (листинг 14.7).

Листинг 14.7. Тест класса LogEntryParser

```
[TestFixture]
class LogEntryParserTests
{
    [Test]
    public void OneLineSequenceProducesOneItem()
    {
        var sr = new StringReader("2015-01-19 [INFO] Message");
        var cut = new LogEntryParser(sr);

        Assert.That(cut.Parse().Count(), Is.EqualTo(1));
    }
}
```

Этому подходу следуют многие классы .NET Framework, например `XmlReader`/`XmlWriter`, `XElement`/`XDocument` и др. `TextReader`/`TextWriter` представляют собой абстракции для работы с текстовыми данными, которые устраняют необходимость в «самописных» интерфейсах, таких как `IReader`/`IWriter`.

Глава 15

Паттерн «Компоновщик» (Composite)

Назначение: компонует объекты в древовидные структуры для представления иерархий «часть — целое». Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Мотивация

Во многих приложениях часто возникает потребность в древовидных структурах данных, одни узлы которых являются «листьями», а другие содержат потомков — дочерние узлы. При этом часто возникает потребность в унифицированном протоколе работы с такими структурами данных, когда интерфейс объектов является одинаковым для одиночных и составных узлов. Композитные структуры активно применяются для создания элементов управления пользовательского интерфейса, деревьев выражений и решения ряда других задач.

При импорте лог-файлов может возникнуть необходимость задавать бизнес-правила, которые будут определять, нужно сохранять запись или нет. Данные правила могут быть простыми: проверять важность записи по свойству `Severity` или отвергать очень старые записи, свойство `DateTime` которых меньше определенной величины. Но поскольку такие правила должны накладываться одно на другое, то рано или поздно возникнет потребность в композиции этих правил.

На помощь могут прийти DDD[1] и паттерн «Спецификация», который для данной задачи будет выглядеть так (рис. 15.1).

Спецификация позволяет гибко комбинировать бизнес-правила, создавая более сложные правила из набора простых условий. Правила моделируются классом `LogImportRule` и его наследниками. Простые правила (`SingleLogImportRule`) определяются на основе предиката `Func<LogEntry, bool>`, переданного пользователем. Составные правила моделируются классом `CompositeLogImportRule` и его наследниками. Самыми простыми составными правилами являются `AndCompositeLogImportRule` и `OrCompositeLogImportRule`, которые объединяют переданный набор правил по «И» и «ИЛИ» соответственно.

Для удобства работы можно создать класс `ImportRuleFactory` с набором фабричных методов для создания и объединения правил (листинг 15.1).

Листинг 15.1. Фабричный метод для создания правил сохранения записей

```
public static class LogRuleFactory
{
    public static LogImportRule Import(Func<LogEntry, bool> predicate)
    {
        return new SingleLogImportRule(predicate);
    }

    public static LogImportRule Or(this LogImportRule left,
        Func<LogEntry, bool> predicate)
    {
        return new OrLogImportRule(left, Import(predicate));
    }

    public static LogImportRule And(this LogImportRule left,
        Func<LogEntry, bool> predicate)
    {
        return new AndLogImportRule(left, Import(predicate));
    }
```

[1] DDD (Domain-Driven Design) — проблемно-ориентированное проектирование, с которым можно познакомиться в замечательной книге Эрика Эванса Domain-Driven Design: Tackling Complexity in the Heart of Software, Prentice Hall, 2003: http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215.

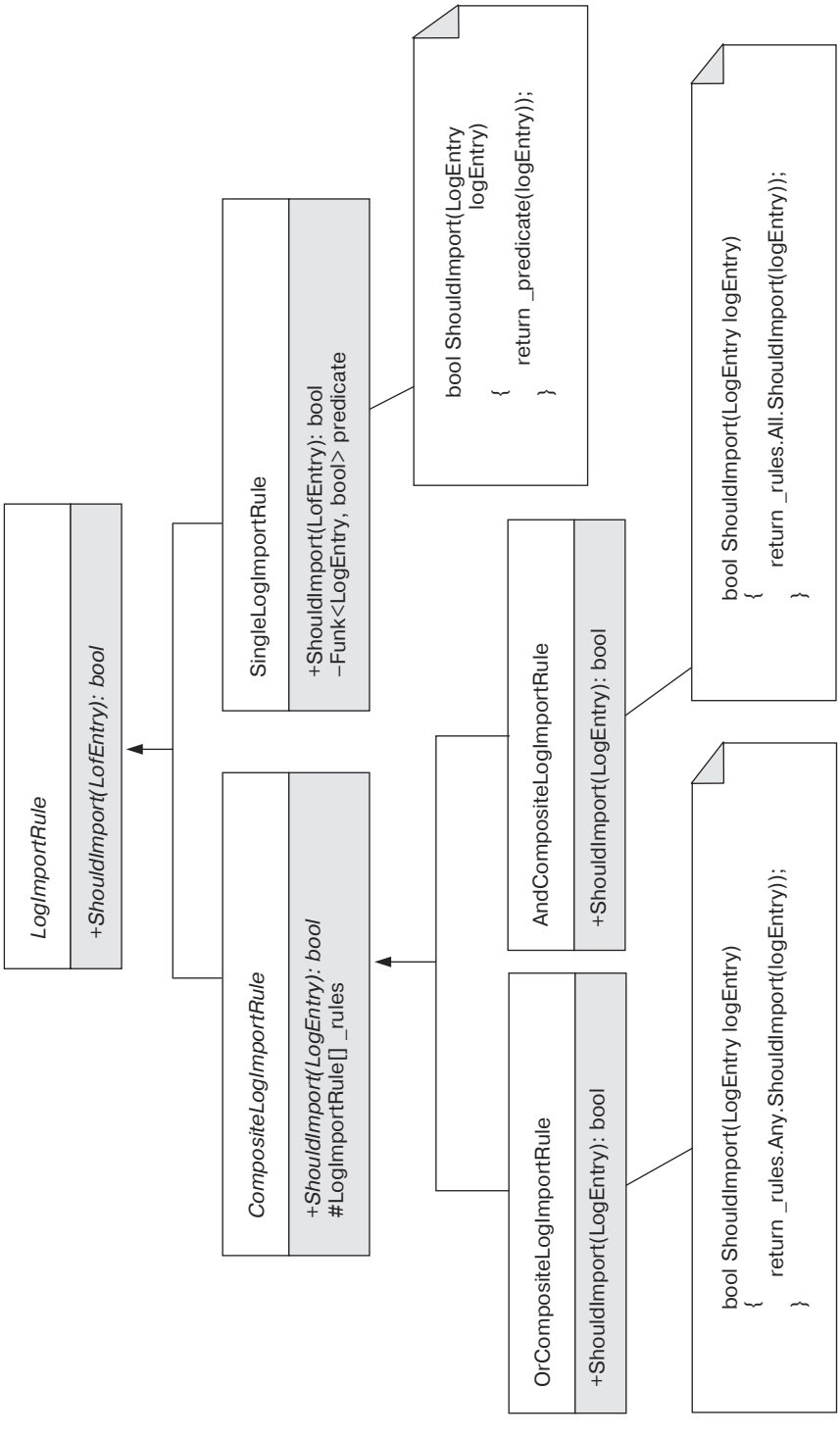


Рис. 15.1. Использование паттерна «Спецификация» для импорта лог-файлов

```
    public static LogImportRuleRejectOldEntriesWithLowSeverity(
        TimeSpan period)
    {
        return
            // Импортируем исключения
            Import(le => le is ExceptionLogEntry)
            // или старые сообщения с высокой важностью
            .Or(le => (DateTime.Now – le.EntryDateTime) > period)
                        .And(le =>le.Severity>= Severity.Warning)
            // или новые сообщения с любой важностью
            .Or(le => (DateTime.Now – le.EntryDateTime) <= period);
    }
}
```

Фабричные методы, такие как `Use`, `Or` и `And`, предназначены для создания встроенного языка, который позволяет получить более читабельные правила. Основной фабричный метод `RejectOldEntriesWithLowSeverity` создает составное правило, которое позволит экспортировать записи с исключениями, но отвергать старые записи с низким приоритетом.

Теперь осталось написать несколько тестов, которые покажут, что данная реализация работает (листинг 15.2).

Листинг 15.2. Тесты правила сохранения лог-файлов

```
var rule = ImportRuleFactory.RejectOldEntriesWithLowSeverity(
    TimeSpan.FromDays(7));

LogEntry logEntry = new ExceptionLogEntry();
Assert.IsTrue(rule.ShouldImport(logEntry));

logEntry = new SimpleLogEntry(){ EntryDateTime = DateTime.Now.AddDays(-10) };
Assert.IsFalse(rule.ShouldImport(logEntry));

logEntry.Severity = Severity.Critical;
Assert.IsTrue(rule.ShouldImport(logEntry));

logEntry = new SimpleLogEntry()
{
```

```
    EntryDateTime = DateTime.Now.AddDays(-5),
    Severity = Severity.Debug
};
Assert.IsTrue(rule.ShouldImport(logEntry));
```

Паттерн «Компоновщик» позволяет использовать составные объекты так же, как и одиночные объекты, что делает код использования более простым и понятным.

Классическая диаграмма классов паттерна «Компоновщик» приведена на рис. 15.2.

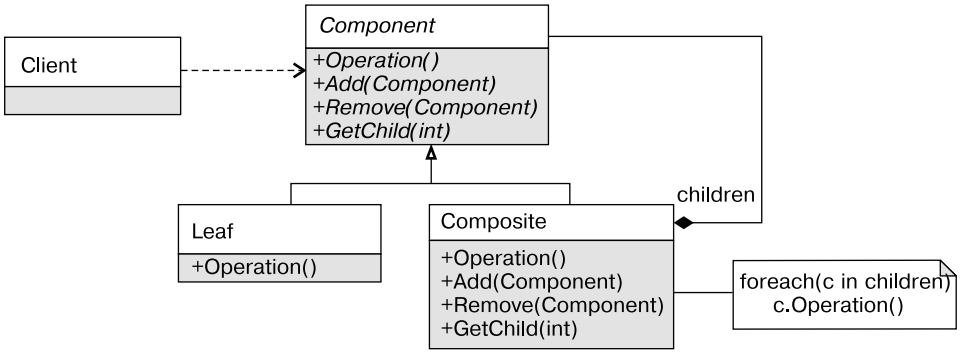


Рис. 15.2. Классическая диаграмма классов паттерна «Компоновщик»

Участники:

- `Component` (`LogImportRule`) — базовый класс компонента. Содержит операцию (`ShouldImport`), также может содержать операции по добавлению/удалению компонентов;
- `Composite` (`CompositeLogImportRule`) — составной компонент, который делегирует выполнение основной операции всем дочерним компонентам;
- `Leaf` (`SingleLogImportRule`) — одиночный компонент, который не может содержать дочерних элементов;
- `Client` — потребитель компонента, который единообразно работает с одиночными и составными объектами.

Обсуждение паттерна «Компоновщик»

Интерфейс составного объекта. При работе с паттерном «Компоновщик» возникает вопрос о том, как формировать составные объекты и каким должен быть интер-

фейс базового класса `Component`: должен он содержать операции по добавлению/удалению компонентов или нет.

Здесь мы сталкиваемся с компромиссом в вопросе о том, чем жертвовать — согласованностью интерфейса компонентов или безопасностью. С одной стороны, клиенты должны работать с простыми и составными объектами единообразно, что делает разумным добавление операций `Add/Remove` в базовый класс `Component`. С другой стороны, эти методы не могут быть нормально реализованы в классе `Leaf`, а значит, такой дизайн будет нарушать принцип подстановки Лисков[1].

Есть три варианта решения этой проблемы, каждый со своими достоинствами и недостатками.

- Использование фабричных методов. Самый простой вариант решить проблему согласованности — сделать классы компонентов неизменяемыми. В этом случае составной объект будет формироваться фабричным методом или конструктором, а необходимость в методах `Add/Remove` полностью пропадет.
- Методы `Add/Remove` находятся в составном компоненте. Если формировать составные объекты с помощью конструкторов неудобно, но процессы формирования и использования компонентов четко разделены, то разумно поместить операции по добавлению/удалению компонентов в класс `CompositeComponent`. В этом случае часть клиентов будут знать о классе `CompositeComponent`, но остальные клиенты станут использовать классы `Component` с более простым интерфейсом.
- Методы `Add/Remove` находятся в базовом компоненте. Если же составной объект является базовым сценарием, а одиночный компонент — частным случаем, то гораздо проще добавить операции `Add/Remove` в базовый класс. В этом случае операции класса `Leaf` могут генерировать исключение `InvalidOperationException` или просто ничего не делать в зависимости от того, является вызов операции `Add` на одиночном компоненте ошибкой или нет.

Применимость

Компоновщик — это относительно низкоуровневый паттерн проектирования, который лежит в основе других паттернов. Команды объединяются в составные команды, декоратор является составным объектом с одним дочерним элементом, посетитель очень часто обходит составные объекты иерархической формы.

---

[1] О принципе подстановки Лисков речь пойдет в части IV.

Паттерн «Компоновщик» применяется для моделирования иерархических структур данных, простые элементы которых объединяются в более сложные компоненты. Паттерн «Компоновщик» позволяет работать с такими объектами единообразно, скрывая от клиента разницу между одиночным и составным объектами.

Примеры в .NET Framework

Существует ряд предметных областей, в которых паттерн «Компоновщик» используется практически постоянно. Каждый из представленных далее классов является примером использования паттерна «Компоновщик».

- Построение компонентов пользовательского интерфейса — `System.Windows.Forms.Control` в WindowsForms, `FrameworkElement` в WPF, `CompositeControl` в ASP.NET.
- Работа с Xml — классы `XmlNode` и `XElement`.
- Деревья выражений — класс `Expression`.

Глава 16

Паттерн «Заместитель» (Proxy)

Назначение: является суррогатом другого объекта и контролирует доступ к нему.

Мотивация

Изменение требований и эволюция системы могут вызвать необходимость внесения серьезных архитектурных изменений. Если на ранних этапах некая операция выполнялась на стороне клиента или же приложение состояло из одного процесса, то со временем исполнение операции может быть перенесено на сервер, а приложение разбито на несколько процессов. В результате возникает задача взаимодействия с удаленным процессом, реализация которой должна быть максимально похожей на локальное взаимодействие. Именно для таких целей предназначен паттерн «Заместитель».

В случае с приложением для импорта лог-файлов его разработчики могут прийти к заключению, что прямая работа с удаленным хранилищем является неудачным решением и необходимо выделить отдельный сервис для сохранения данных. При использовании современных библиотек, таких как WCF, ASP.NET WebAPI или .NET Remoting, выполнить такой переход будет несложно.

При использовании WCF старый интерфейс `ILogSaver` и все объекты, используемые в качестве аргументов и возвращаемых значений, нужно декорировать специальными атрибутами (`ServiceContract` + `OperationContract` и `DataContract` + `DataMember`). Затем сконфигурировать приложение, добавив в его конфигурационный файл информацию о привязках (binding), контрактах (contracts) и точках доступа (endpoint). После чего останется создать простой класс-заместитель (или прокси-класс) и пользоваться им так, как будто реализация `ILogSaver` находится в этом же процессе[1] (листинг 16.1).

Листинг 16.1. Пример использования класса-заместителя для работы с WCF

```
class LogSaverClient : ClientBase<ILogSaver>, ILogSaver
{
    public void SaveLogEntry(LogEntrylogEntry)
    {
        // Делегируем всю работу внутреннему свойству Channel
        // с типом ILogSaver
        Channel.SaveLogEntry(logEntry);
    }
}

//...
publicvoidSaveLogEntryAtTheBackend(LogEntrylogEntry)
{
    var proxy = newLogSaverClient();
    proxy.SaveLogEntry(logEntry);
    proxy.Close();
}
```

Основная «магия» заключена в классе `ClientBase<T>` и нижележащей инфраструктуре WCF, которая отвечает за сериализацию аргументов, передачу их «по проводам» и десериализацию ответа. Класс `LogSaverClient` является классом-заместителем, который позволяет работать с удаленным WCF-сервисом так, как будто экземпляр `ILogSaver` находится в текущем процессе.

---

[1] Как будет показано далее, полностью игнорировать распределенную природу не следует, поскольку это может привести к негативным последствиям. Распределенное взаимодействие налагает слишком большое число дополнительных ограничений, игнорировать которые опасно.

Классическая диаграмма классов паттерна Заместитель представлена на рис. 16.1.

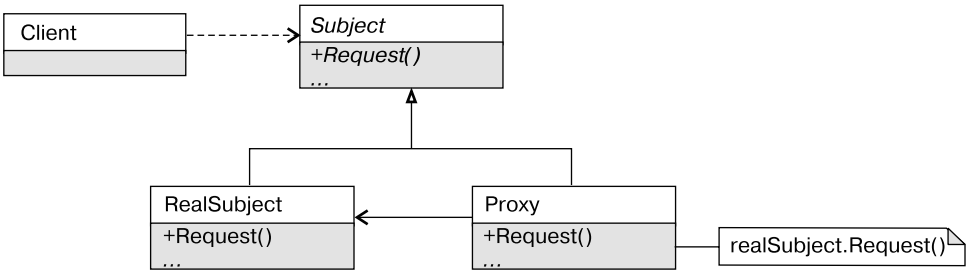


Рис. 16.1. Классическая диаграмма классов паттерна «Заместитель»

Участники:

- `Client` — работает с абстрактным компонентом, не зная, является он настоящим или нет;
- `Subject` (`ILogSaver`) — определяет интерфейс компонента;
- `Proxy` (`LogSaverClient`) — объект-заместитель, который реализует интерфейс компонента, но делегирует всю работу настоящему объекту;
- `RealSubject` — реальный компонент, доступ к которому осуществляется через заместитель.

Обсуждение паттерна «Заместитель»

«Заместитель» является одним из немногих паттернов проектирования, который с течением времени претерпел довольно серьезные изменения. В классическом труде «банды четырех» описаны три основных сценария использования паттерна «Заместитель».

- Удаленный заместитель отвечает за кодирование запроса и его аргументов для работы с компонентом в другом адресном пространстве.
- Виртуальный заместитель может кэшировать дополнительную информацию о реальном компоненте, чтобы отложить его создание.
- Защищающий заместитель проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права.

Второй и третий варианты паттерна «Заместитель» применяются на практике, но своей известностью этот паттерн обязан первому варианту. Классы-заместители

являются стандартным паттерном в подавляющем большинстве современных технологий построения распределенных приложений. Однако внешний вид классов-заместителей и подходы к их реализации за последние 15 лет претерпели очень серьезные изменения.

Прозрачный удаленный заместитель

Основная идея удаленного заместителя заключается в том, что клиент может работать с классом-заместителем так, как будто он работает с объектом в собственном адресном пространстве. Этот подход настолько понравился разработчикам распределенных технологий конца 1990-х — начала 2000-х, что многие технологии того времени постарались стереть грань между внутрипроцессным и межпроцессным взаимодействием. Такой подход получил название *прозрачность местоположения* и лег в основу таких технологий, как DCOM и .NETRemoting.

Однако со временем разработчики начали осознавать, что постоянно сталкиваются с проблемами «дырявых абстракций» и не могут игнорировать особенности сетевого взаимодействия. Особенности обработки ошибок, необходимость того, чтобы все параметры методов были сериализуемыми, задержки и длительное время исполнения методов — все это требует другого подхода с точки зрения разработки[1]. Поэтому в последних технологиях, таких как WCF и ASP.NETWebAPI, классы-заместители перестали быть прозрачными. Разработчик может работать с интерфейсом `ISomeInterface`, каждый вызов которого исполняется на удаленной стороне, но программисту самому придется стереть грань между распределенным и локальным мирами.

Такие классы, как `ChannelBase<T>` в WCF, сильно упрощают взаимодействие с удаленным процессом, но они выполняют роль скорее фасадов, чем классов-заместителей!

Заместитель vs. декоратор

Структуры паттернов «Заместитель» и «Декоратор» очень похожи. Каждый из них содержит ссылку на базовый компонент и делегирует ему выполнение всей работы. Но у этих паттернов разное назначение.

---

[1] Можно смело говорить, что удаленный заместитель нарушает принцип замещения Лисков. При замене локальной реализации ILogSaver удаленным заместителем поведение приложения обязательно изменится. Изменения коснутся времени исполнения и обработки ошибок. Зачастую может потребоваться полное перепроектирование клиента ILogSaver с учетом его новой удаленной природы.

Декоратор добавляет поведение всем методам интерфейса, позволяя нанизывать расширения одно на другое. Класс-заместитель может выполнять определенные действия, например создавать настоящий компонент по мере необходимости, но он не должен ничего подмешивать в результаты исполнения операции.

Виртуальный заместитель и Lazy<T>

Класс `Lazy<T>` можно считать универсальным строительным блоком, с помощью которого легко создавать виртуальные классы-заместители (листинг 16.2).

Листинг 16.2. Реализация виртуального заместителя с помощью Lazy<T>

```
public interface IHeavyweight
{
    void Foo();
}

// Стоимость создания класса очень высока
public class Heavyweight : IHeavyweight
{
    public void Foo()
    {}
}

// Виртуальный заместитель, который будет создавать
// тяжеловесный объект лишь при необходимости
public class HeavyweightProxy : IHeavyweight
{
    private readonly Lazy<Heavyweight> _lazy = new Lazy<Heavyweight>();

    public void Foo()
    {
        _lazy.Value.Foo();
    }
}
```

Применимость

Классы-заместители активно применяются там, где нужно спрятать исходный объект и добавить к его методам некоторое поведение: позволить отложить создание дорогостоящего объекта, контролировать количество вызовов метода или спрятать удаленную природу объекта.

Примеры в .NET Framework

- Удаленные классы-заместители. В коммуникационных технологиях в .NET Framework применяются удаленные прокси-классы: `ChannelBase<T>` в WCF, `RealProxy` в .NETRemoting.
- Виртуальные классы-заместители в ORM-фреймворках. В современных ORM (Object-RelationalMapping — объектно-реляционное отображение), таких как NHibernate или EntityFramework, применяются специализированные виртуальные классы-заместители. ORM генерирует оболочку над сущностями (dataentities), чтобы отслеживать изменения и генерировать корректный SQL-код для обновления записей в базе данных.

Часть IV

Принципы проектирования

Любая индустрия по мере взросления старается делиться своим опытом с подрастающим поколением. Паттерны проектирования, рассмотренные ранее, являются отличным примером повторного использования знаний и опыта более опытных проектировщиков. Паттерны весьма полезны (иначе зачем бы я решил написать еще одну книгу на эту тему?), но они показывают типовые решения типовых задач. Некоторые паттерны весьма общие, другие специфичны для конкретной предметной области, но многим программистам все равно будет не хватать более фундаментальных путеводных нитей, следуя которым станет развиваться дизайн.

В этой части мы рассмотрим популярные принципы проектирования, обозначенные аббревиатурой SOLID. В середине 1990-х годов Роберт Мартин начал публиковать в журнале C++ Report статьи на тему проектирования. В качестве основы было взято несколько известных ранее принципов проектирования, добавлены собственные мысли, и на свет появились фундаментальные принципы объектно-ориентированного проектирования.

Изначально эти принципы были описаны в несколько ином порядке и в архиве «дядюшки Боба» значатся под аббревиатурой SOLID. Через несколько лет они перекочевали в книгу Agile Software Development, Principles, Patterns, and Practices, а со временем — в аналогичную книгу с примерами на языке C# «Принципы, практики и методики гибкой разработки на языке C#». Здесь порядок уже был иным и появились «цельные» (SOLID) принципы, звучность названия которых в немалой степени обеспечила им успех.

Оригинальные статьи были опубликованы почти 20 лет назад, когда объектно-ориентированное мышление только становилось мейнстримом, в качестве примеров использовался язык C++. Столь почтенный возраст дает о себе знать, и оригинальные статьи Роберта Мартина содержат ряд советов, полезных лишь для языка C++ двадцатилетней давности. Основной упор делается на критике структурного программирования и восхвалении объектно-ориентированного подхода. Любопытно, что многие «современные» описания SOLID-принципов все еще показывают пользу полиморфизма на примере иерархии фигур и говорят о проблемах транзитивных зависимостей, которые далеко не столь актуальны в C# или Java.

Я же хочу рассмотреть SOLID-принципы с абстрактной точки зрения, подумать о том, какую проблему они призваны решать (и решают ли) и как мы должны смотреть на них сегодня, когда ООП уже давно стало широко распространенной парадигмой программирования.

Принцип единственной обязанности (SRP — The **S**ingle **R**esponsibility **P**rinciple): *у класса/модуля должна быть лишь одна причина для изменения*. Данный принцип говорит о борьбе с изменениями, но на самом деле суть его сводится к борьбе со сложностью (tackling the complexity). Любой сложный класс должен быть разбит на несколько простых составляющих, отвечающих за определенный аспект поведения. Это упрощает как понимание, так и развитие класса в будущем. Простой класс с небольшим числом зависимостей легко изменить независимо от того, сколько причин для изменения существует. Разработчик очень редко знает, как требования изменятся в будущем, что делает простое решение лучшим способом обеспечения гибкости.

Принцип «открыт/закрыт» (OCP — The **O**pen-**C**losed **P**rinciple): *программные сущности (классы, модули, функции и т. п.) должны быть открытыми для расширения, но закрытыми для модификации*. Любой готовый модуль должен быть стабильным (закрытым) с точки зрения своего интерфейса, но открытым с точки зрения реализации. Закрытость модулей означает стабильность интерфейса и возможность использования модулей его клиентами. Открытость модулей означает возможность его изменения путем изменения реализации или же путем переопределения поведения за счет создания наследников. Сокрытие информации и полиморфизм позволяют ограничить количество изменений, которые понадобится внести в систему при очередном изменении требований, что сделает этот процесс более простым и управляемым.

Принцип замещения Лисков (LSP — The **L**iskov **S**ubstitution **P**rinciple): *должна существовать возможность вместо базового типа подставить любой его подтип*. Поскольку наследование является одним из ключевых механизмов объектно-ориентированного проектирования, очень важно использовать его корректным образом. Данный принцип дает четкие рекомендации о том, в каких пределах может изменяться поведение методов, переопределенных в производных классах, чтобы между классами сохранялось отношение «ЯВЛЯЕТСЯ».

Принцип разделения интерфейсов (ISP — The **I**nterface **S**egregation **P**rinciple): *клиенты не должны вынужденно зависеть от методов, которыми не пользуются*. Интерфейс класса определяется некоторым контрактом, которому он должен следовать ради своих клиентов. Иногда возникают ситуации, когда у разных клиентов появляются различные сценарии использования класса, в результате чего его интерфейс становится несогласованным и неудобным в использовании. Данный принцип говорит о том, что клиенты хотят иметь цельный и согласованный интерфейс сервисов независимо от того, пользуется ли этими сервисами еще кто-то, кроме них, или нет.

Принцип инверсии зависимостей (DIP — The **D**ependency **I**nversion **P**rinciple): *модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.* Слишком большое число зависимостей класса говорит о проблемах в его дизайне. Возможно, класс делает слишком многое или же он неудачно спроектирован, что приводит к необходимости вызова по одному методу у большого числа зависимостей. Любая объектная система представляет собой граф взаимодействующих объектов. При этом некоторые зависимости являются деталями реализации и контролируются классами самостоятельно, а некоторые должны передаваться извне при конструировании объектов. Данный принцип говорит о необходимости выделения и передачи ключевых зависимостей через аргументы конструктора, что позволяет перенести проблемы создания и выбора конкретных зависимостей на вызывающий код.

Глава 17

Принцип единственной обязанности

Нельзя объять необъятное.

Козьма Прутков

Принцип единственной обязанности (Single-Responsibility Principle, SRP): *«У класса должна быть только одна причина для изменения»* (*Мартин Р.* Принципы, паттерны и практики гибкой разработки. — 2006).

В разработке ПО есть одна неизменная составляющая — неизбежность изменений. Как бы мы ни старались, как бы ни пытались продумать все до мелочей, рано или поздно требования изменятся. Требования меняются из-за изначального недопонимания задачи, изменений во внешнем мире, более точного осознания собственных нужд заказчиком или десятков других причин. На самом деле важны не столько причины изменений, сколько наши возможности по адаптации системы к новым требованиям и легкость внесения изменений.

Многие разработчики считают, что обобщенные решения являются лучшим способом борьбы с изменениями требований. Полагается, что если предусмотреть все возможные сценарии и предоставить различные точки расширения, то система сможет решать любые задачи пользователя. Но у этого подхода есть несколько серьезных недостатков. Мы (разработчики) очень плохо разбираемся в вопросах предсказания будущего, не знаем, как изменятся требования в будущем и в какой

плоскости система должна легко расширяться. А поскольку гибкость всегда приводит к увеличению сложности, то полученное решение не всегда справляется с исходной задачей и плохо поддается модификации.

Другой подход к обеспечению расширяемости заключается в использовании наиболее простых решений: программная сущность (класс, модуль, метод) должна по возможности решать лишь одну задачу, но делать это хорошо. Как это предотвращает проблему изменений? Очень просто. Чем меньше у метода, класса или модуля вспомогательных задач, тем ниже вероятность случайных изменений.

Фредерик Брукс в своей книге «Мифический человеко-месяц» вводит понятия естественной сложности (essential complexity) и привнесенной или случайной сложности (accidental complexity). Естественная сложность исходит из предметной области и является неотъемлемой частью любого решения. Привнесенная сложность внесена нами в процессе реализации за счет плохого дизайна, неудачно выбранных библиотек или неподходящего языка программирования.

Изменения, вносимые в систему, также можно разделить на естественные, которые возникают из-за изменения бизнес-логики или требуемого поведения, и случайные, которые мы вынуждены вносить во второстепенные модули из-за неудачного дизайна. Изменения в законодательстве приведут к изменениям в классе вычисления заработной платы и являются естественными. Но мы не хотим его менять при изменении пользовательского интерфейса, смене базы данных или из-за изменений формата одного из отчетов.

Существует ряд патологических случаев нарушения принципа единственной обязанности: классы пользовательского интерфейса, которые знают о бизнес-правилах или работают напрямую с базой данных, или классы Windows-сервисов с обилием бизнес-логики. Есть примеры нарушения SRP на уровне приложений: Windows Forms-приложение, в котором располагается WCF-сервис, Windows-сервис, взаимодействующий с пользователем с помощью диалоговых окон. Эти примеры показывают, что нарушения SRP бывают как на микроуровне — на уровне классов или методов, так и на макроуровне — на уровне модулей, подсистем и целых приложений.

Приведенные ранее примеры не столько нарушают SRP, сколько противоречат здравому смыслу. В реальном мире проблемы бывают более тонкими, когда один разработчик говорит, что дизайн хорош, а для другого он серьезно «попахивает» и вам хочется найти весомые аргументы против текущего решения. Чтобы понять, нарушает ли код принцип единственной обязанности, важно понимать, какую проблему он должен решать.

Для чего нужен SRP

Принцип единственной обязанности предназначен для борьбы со сложностью. Когда в приложении всего 200 строк, то дизайн как таковой вообще не нужен. Достаточно аккуратно написать 5–7 методов и решить задачу любым доступным способом. Проблемы возникают, когда система растет и увеличивается в размере.

Зависимость между числом строк кода и сложностью решения является нелинейной. Добавление каждой новой функции в систему требует все больше и больше усилий. Когда речь касается десятков и сотен тысяч строк кода, приходится вспоминать такие «страшные» понятия, как «абстракция» и «сокрытие информации», и лучше продумывать обязанности каждого класса. При разработке крупной системы очень важно иметь возможность сосредоточиться на главной задаче метода, класса или модуля и выбросить из рассмотрения все второстепенные детали.

Основным строительным блоком объектно-ориентированного приложения является класс, поэтому обычно принцип единственной обязанности рассматривается в контексте класса. Но поскольку основную работу выполняют методы, то очень важно, чтобы они также были нацелены на решение одной задачи.

Основная сложность принципа SRP в том, что понятие «обязанности» является относительным. Если мы говорим, что у класса или метода должна быть лишь одна обязанность, то может ли метод валидировать свои аргументы? Или логировать определенные этапы своей работы? А может ли класс читать и сохранять данные?

Наличие или отсутствие нарушения SRP очень зависит от того, насколько сложным является каждый из описанных ранее шагов. Метод будет нарушать SRP, если валидация аргументов занимает 40 строк кода и находится в разных его частях. Метод также будет нарушать SRP, если за обилием трассировочных сообщений не видно его основной логики. Но класс может и не нарушать SRP, если он читает и сохраняет данные, но на каждую операцию требуется две строки кода.

Принцип единственной обязанности на практике

Развитие программного проекта приводит к увеличению хаоса. Спешка, непродуманные решения, неполное понимание назначения оригинального решения — все это приводит к постепенному ухудшению качества дизайна. При внесении изменений очень важно оценивать качество решения на предмет его чистоты, соответствия

принципам проектирования и здравому смыслу. То, что на первых этапах должно было находиться в одном классе, необходимо переместить в другое место, а логика, которая помещалась в одном методе, может перерасти в целую иерархию классов.

Приведенная в предыдущих главах задача импорта лог-файлов для полнотекстового поиска является вполне реальной. Наша команда разработала утилиту для импорта лог-файлов в Elasticsearch с целью изучения узких мест производительности и быстрого поиска определенных сообщений. Первая версия этой утилиты предназначалась для экспорта логов определенного приложения и была реализована за несколько часов. Это было простое консольное приложение в пару сотен строк, и главный его класс выглядел примерно так (листинг 17.1).

Листинг 17.1. Первая реализация утилиты импорта лог-файлов

```
public class LogImporter
{
    public void ImportLogs()
    {
        string[] logFileNames = GetListOfFilesFromAppConfig();
        foreach (var file in logFileNames)
        {
            var logEntries = ReadLogEntries(file);
            SaveLogEntries(logEntries);
        }
    }

    private string[] GetListOfFilesFromAppConfig()
    {
        // Читаем список файлов из конфигурационного файла приложения
    }

    private IEnumerable<string> ReadLogEntries(string fileName)
    {
        // Читаем файл построчно
    }

    private void SaveLogEntries(IEnumerable<string> entries)
    {
```

```
        foreach (var entry in entries)
        {
            DateTime dateTime = ParseEntryTime(entry);
            // Сохраняем для полнотестового поиска
            LogSaver.SaveEntry(dateTime, entry);
        }
    }

    private DateTime ParseEntryTime(string entry)
    {
        // Получаем время из лог-файла, зная его формат
    }
}
```

Класс `LogImporter` полностью отвечает за процесс импорта лог-файлов. Он читал конфигурационный файл приложения для получения списка импортируемых файлов, анализировал прочитанные строки и сохранял записи в очень простом формате с помощью фасадного класса `LogSaver`.

Насколько это решение удачное? Все зависит от поставленных целей. Если требуется одноразовая утилита для анализа логов «продакшн»-сервера в воскресенье вечером, то это решение вполне оправданно. Но оно не подходит, если поставлена цель разработать полноценную утилиту, способную экспортировать логи разного формата или хотя бы учитывать важность (severity) записей и обрабатывать сохраненные исключения особым образом.

Посмотрим, как следование принципу единственной обязанности может улучшить дизайн этого кода. Вот как я подхожу к внесению изменений.

1. **Убираем ненужные обязанности.** Первое, что нужно сделать, — это найти обязанности, которые можно переложить на плечи вызывающего кода. В данном случае это чтение конфигурации. Работа с конфигурационным файлом приложения делает этот код менее автономным, затрудняет юнит-тестирование и повторное использование (листинг 17.2).

   Листинг 17.2. Улучшенная версия класса LogImporter

   ```
   public class LogImporter
   {
       private readonly ICollection<string> _logFileNames;
   ```

```
    public LogImporter(ICollection<string> logFileNames)
    {
        _logFileNames = logFileNames;
    }

    public void ImportLogs()
    {
        foreach (var file in _logFileNames)
        {
            var logEntries = ReadLogEntries(file);
            SaveLogEntries(logEntries);
        }
    }

    // Остальные методы остались без изменения
}
```

Этот шаг кажется незначительным, но он очень важен. Формирование списка анализируемых файлов является обязанностью вызывающего кода и не должно контролироваться классом `LogImporter`. Новое решение является более простым и гибким одновременно. Так, например, теперь нам не придется изменять код этого класса, если список анализируемых файлов будет передаваться из командной строки, а не читаться из конфигурации.

ПРИМЕЧАНИЕ

Некоторые читатели могут сказать, что выделение стратегии конфигурирования тоже решило бы поставленную задачу. Выделение интерфейса ILogFileListProvider позволит иметь несколько разных реализаций и обеспечит требуемую гибкость. Но такой подход гораздо сложнее, обладает дополнительным уровнем косвенности и требует от клиентов класса LogImporter значительно больших усилий. Если классу для работы требуется коллекция строк, отразите это требование в своем решении наиболее простым и ясным способом!

2. **Выделяем класс разбора записей лог-файла.** У класса может быть несколько зон ответственности, пока все они являются относительно простыми. Какой аспект класса `LogImporter` является самым сложным? Разбор прочитанных записей! У этого аспекта множество граничных условий: разные форматы даты/

времени, разные уровни важности сообщений, многострочные записи с исключениями и т. п. Даже для анализа лог-файлов одного приложения сложность этого класса будет достаточно высокой. Все это говорит о необходимости выделения класса `LogEntryParser` (листинг 17.3).

Листинг 17.3. Выделенный класс LogEntryParser

```
Class LogEntryParser
{
    public bool TryParse(string line, out LogEntry logEntry)
    {
        // Используем регулярное выражение для анализа содержимого строки.
        // Метод возвращает true, если запись полностью прочитана.
        // Возвращает false, если мы столкнулись с многострочной записью
        // и для получения записи нужно проанализировать еще одну строку
        // или более
    }
}

public class LogImporter
{
    private readonly LogEntryParser _parser = new LogEntryParser();

    public void ImportLogs()
    {
        foreach (varfilein _logFileNames)
        {
            IEnumerable<LogEntry>logEntries = ReadLogEntries(file);
            SaveLogEntries(logEntries);
        }
    }

    private IEnumerable<LogEntry> ReadLogEntries(string fileName)
    {
        // Читаем файл построчно
        using (var file = File.OpenText(fileName))
        {
```

```
            string line = null;
            while ((line = file.ReadLine()) != null)
            {
                LogEntrylogEntry;
                if (_parser.TryParse(line, outlogEntry))
                {
                    yield return logEntry;
                }
            }
        }
    }
    // Остальные методы остались без изменения
}
```

Наличие класса `LogEntryParser` имеет ряд важных преимуществ.

- Сложная логика анализа записей изолирована и может развиваться независимо.
- Логику анализа записей легко тестировать в изоляции путем передачи строк нужного формата.
- Класс `LogImporter` превратился в довольно простого посредника, который отвечает за пересылку данных из одного источника в другой.

3. **Тестируем код.** Многие разработчики считают, что каждый аспект реализации должен быть проверен юнит-тестами. Такое стремление не во всех случаях оправдывает затраченные усилия, а иногда вызывает ложное чувство безопасности. Более разумным является смешанный подход к тестированию, в котором наиболее сложные аспекты проверяют юнит-тестами, а вспомогательную логику — интеграционными тестами.

   В процессе работы над этой задачей я бы начал с тестов класса `LogEntryParser` (листинг 17.4).

Листинг 17.4. Юнит-тесты класса LogEntryParser

```
[TestFixture]
public class LogEntryParserTests
{
    [TestCase("2014-01-12 [DEBUG] message", Result = Severity.Debug)]
```

```
    [TestCase("[Info] Message", Result = Severity.Info)]
    public Severity ParseSeverity(string line)
    {
        // Arrange
        var parser = new LogEntryParser();

        // Act & Assert
        return parser.Parse(line).Severity;
    }
}

static class LogEntryParserEx
{
     public static LogEntry Parse(this LogEntryParser parser, string
line)
    {
        LogEntry logEntry;
        parser.TryParse(line, out logEntry);
        return logEntry;
    }
}
```

Наличие у класса `LogEntryParser` четкого интерфейса с минимальным числом зависимостей делает его тестирование очень простым. Использование параметризованных тестов NUnit[1] обеспечивают высокое покрытие кода тестами с минимальным числом усилий, а добавление нового тест-кейса требует добавления лишь одной строки кода.

ПРИМЕЧАНИЕ

Обратите внимание на метод расширения Parse класса LogEntryParser. Качество тестов не менее важно, чем качество основного кода. Интерфейс класса LogEntryParser специализирован для решения своих основных задач, но он не так удобен для других сценариев. Использование методов расширения для тестов показывает применение еще одного принципа проектирования — принципа разделения интерфейсов.

[1] Подробнее о пользе параметризованных тестов можно почитать в моей статье «Параметризованные юнит-тесты»: http://bit.ly/ParametrizedUnitTests.

Интеграционные тесты показывают корректность реализации в реальном окружении. В данном случае класс `LogImporter` работает с двумя внешними источниками: файлами и хранилищем лог-файлов. Поскольку его логика весьма проста, то вместо выделения дополнительных уровней косвенности для юнит-тестирования будет достаточно нескольких интеграционных тестов (листинг 17.5).

Листинг 17.5. Интеграционные тесты класса LogImporter

```
[TestFixture]
public class LogImporterIntegrationTests
{
    [Test]
    public void ImportFileWithTenEntries()
    {
        string fileName = "10Entry.log";

        // Cleanup
        TryRemoveExistingEntries(fileName);

        // Arrange
        var importer = new LogImporter(new[] {fileName});

        // Act
        importer.ImportLogs();

        // Assert
        Assert.That(GetEntriesCountForLogFile(fileName), Is.EqualTo(10));
    }
```

Интеграционные тесты более хрупки по своей природе и требуют особого подхода при реализации. К трем основным этапам: инициализации, действию, утверждению (Arrange, Act, Assert, AAA) — может быть добавлен еще один — очистка состояния. Это предотвратит прохождение теста из-за наличия в хранилище старых данных, хотя и потребует дополнительного времени при каждом запуске.

Подводя итоги, скажем следующее: использование принципа единственной обязанности привело нас к следующему дизайну класса `LogImporter` (рис. 17.1).

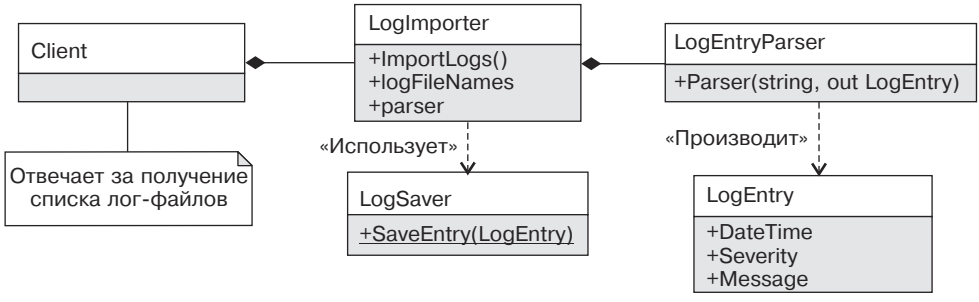


Рис. 17.1. Переработанный дизайн класса LogImporter

Нельзя сказать, что каждый класс этого решения имеет лишь одну обязанность, но ведь следование принципам проектирования не должно быть самоцелью. Сложность дизайна определяется сложностью решаемой задачи. На данный момент класс `LogImporter` наиболее простым образом решает поставленную перед ним задачу: импортирует лог-файлы определенного приложения.

Полученный дизайн является довольно простым, без ненужных уровней абстракции или интерфейсов. В последующих главах мы рассмотрим использование других принципов проектирования, которые помогут справиться с последующими изменениями требований.

Типичные примеры нарушения SRP

Типичными примерами нарушения SRP являются:

- **смешивание логики с инфраструктурой.** Бизнес-логика смешана с представлением, слоем персистентности, находится внутри WCF или Windows-сервисов. Должна быть возможность сосредоточиться на бизнес-правилах, не обращая внимания на второстепенные инфраструктурные детали;
- **слабая связность (lowcohesion).** Класс/модуль/метод не является цельным и решает несколько несвязанных задач. Проявляются несколько групп методов, каждая из которых обращается к подмножеству полей, не используемых другими методами;
- **выполнение нескольких несвязанных задач.** Класс/модуль может быть цельным, но решать несколько несвязанных задач (вычисление заработной платы и построение отчета). Класс/модуль/метод должен быть сфокусированным на решении минимального числа задач;

- **решение задач разных уровней абстракции.** Класс/метод не должен отвечать за задачи разного уровня. Например, класс удаленного заместителя не должен самостоятельно проверять аргументы, заниматься сериализацией и шифрованием. Каждый из этих аспектов должен решаться отдельным классом.

Выводы

Следование принципам проектирования является не статической, а динамической характеристикой дизайна. Наибольшая опасность заключается в «загнивании» дизайна, когда внесение нескольких изменений приводит к разрастанию обязанностей и увеличению сложности. То, что вчера казалось лишь одной обязанностью, сегодня может потребовать целой иерархии классов. Каждый раз при изменении логики нужно анализировать дизайн на соответствие здравому смыслу и принципам проектирования.

Важность принципа единственной обязанности резко возрастает при увеличении сложности. Если решение перестает помещаться в голове, то пришло время разбить его на более простые составляющие, каждая из которых будет решать лишь одну задачу.

Глава 18

Принцип «открыт/закрыт»

Эффективные проекты контролируют изменения;
неэффективные проекты находятся
под контролем изменений.

Стив Макконнелл. Остаться в живых[1]

Принцип «открыт/закрыт» (Open-Closed Principle, OCP): *«Программные сущности (классы, модули, функции и т. п.) должны быть открытыми для расширения, но закрытыми для модификации»* (*Мартин Р.* Принципы, паттерны и практики гибкой разработки. — 2006).

Одна из причин «загнивания» дизайна кроется в страхе внесения изменений. Разработчики и менеджеры должны быть уверены в том, что изменения являются корректными и не приведут к появлению ошибок в других частях системы. Простые классы и модули, которые соответствуют принципу единственной обязанности, являются хорошей стартовой точкой для получения адаптивного дизайна, но этого не всегда достаточно.

По мере развития в системе появляются семейства типов с общим поведением и схожими интерфейсами. Возникают иерархии наследования, в базовых классах

---

[1] *Макконнелл С.* Остаться в живых! Руководство для менеджера программных проектов. — СПб.: Питер, 2006.

которых помещается общее поведение, которое наследники изменяют при необходимости. Это позволяет повторно использовать значительную часть логики базовых классов, а также упрощает добавление типов с новым поведением.

Полученные иерархии типов одновременно являются открытыми и закрытыми. Открытость говорит о простоте добавления новых типов, а закрытость — о стабильности интерфейсов базовых классов иерархии.

Путаница с определениями

Принцип «открыт/закрыт» является самым неоднозначным из всех SOLID-принципов. Его неоднозначность кроется в противоречивости его определения, а подкрепляется разнообразными описаниями этого принципа в разных источниках. Неудивительно, что даже такие яркие представители нашей отрасли, как Эрик Липперт и Джон Скит[1], относятся к этому принципу неоднозначно и признаются в его непонимании.

Принцип «открыт/закрыт» был изначально сформулирован Бертраном Мейером в первом издании его книги «Объектно-ориентированное конструирование программных систем» еще в 1988 году[2], но популярность этот принцип завоевал благодаря трудам Роберта Мартина.

Определение от Роберта Мартина: программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации. Таким образом, у модулей есть две основные характеристики.

- Они открыты для расширения. Это означает, что поведение модуля можно расширить. Когда требования к приложению изменяются, мы добавляем в модуль новое поведение, отвечающее изменившимся требованиям. Иными словами, мы можем изменить состав функций модуля.
- Они закрыты для модификации. Расширение поведения модуля не сопряжено с изменениями в исходном или двоичном коде модуля. Двоичное исполняемое представление модуля, будь то компонуемая библиотека, DLL или EXE-файл, остается неизменным.

---

1 Эрик Липперт является автором известного среди .NET-разработчиков блога Fabulousa dventuresincoding, а Джон «Чак Норрис» Скит занимает первую строчку в рейтинге сайта stackoverflow.com — самого популярного сайта вопросов и ответов по разработке ПО.

2 Бертран Мейер является признанным гуру в мире объектно-ориентированного программирования и автором одной из наиболее значимых книг по разработке ПО — «Объектно-ориентированное конструирование программных систем» (Интернет-университет, 2005).

На основе этого определения может сложиться впечатление, что следование принципу «открыт/закрыт» подразумевает использование расширяемых решений на основе подключаемых модулей (pluggable architecture) и обилие наследования. На самом деле это не так, и даже сам автор определения со временем несколько изменил свое отношение к этому принципу. В своей статье An Open and Closed Case[1] Роберт Мартин написал, что с годами он стал более мудрым и менее категоричным в своих высказываниях (хотя на момент описания принципа «открыт/закрыт» в статье 1996 года ему было всего 43).

Теперь давайте перейдем к определению, которое дал Бертран Мейер, и попробуем понять, что же имел в виду действительный автор этого принципа.

Определение от Бертрана Мейера: модули должны иметь возможность быть как открытыми, так и закрытыми. При этом понятия открытости и закрытости определяются так.

- Модуль называют открытым, если он еще доступен для расширения. Например, имеется возможность расширить множество операций в нем или добавить поля к его структурам данных.
- Модуль называют закрытым, если он доступен для использования другими модулями. Это означает, что модуль (его интерфейс — с точки зрения сокрытия информации) уже имеет строго определенное окончательное описание. На уровне реализации закрытое состояние модуля означает, что можно компилировать модуль, сохранять в библиотеке и делать его доступным для использования другими модулями (его клиентами).

Так что же означает принцип «открыт/закрыт»?

Исходный посыл Мейера заключается не столько в расширяемости системы, сколько в управляемости процесса разработки: «Необходимо, особенно с точки зрения руководителя проекта, закрывать модули. В системе, состоящей из многих модулей, большинство модулей зависимы. Например, модуль интерфейса пользователя может зависеть от модуля синтаксического разбора — синтаксического анализатора и модуля графики. Синтаксический анализатор может зависеть от модуля лексического анализа и т. д. Если не закрывать модуль до тех пор, пока не будет уверенности, что он уже содержит все необходимые компоненты, то невозможно будет завершить разработку многомодульной программы: каждый из разработчиков будет вынужден ожидать, когда же завершат свою работу все остальные».

---

[1] An Open and Closed Case by Uncle Bob (http://blog.8thlight.com/uncle-bob/2013/03/08/AnOpenAndClosedCase.html).

Другими словами, Мейер говорит о том, что интерфейс модуля должен быть закрытым, а реализация и точное поведение могут варьироваться и оставаться открытыми для изменений.

Когда нам может понадобиться изменять поведение без изменения интерфейса? Например, когда у существующего класса появляется вторая группа клиентов, которой требуется аналогичное поведение, но с небольшими изменениями. В объектно-ориентированном мире это означает создание наследника, который использует повторно весь код базового класса и переопределяет ряд методов для обеспечения нового поведения.

В результате добавления наследника интерфейс и поведение нашего исходного класса (класс `A`) остается неизменным, что гарантирует правильное функционирование его старых клиентов, а новые клиенты начинают использовать модифицированную версию класса (класс `A'`).

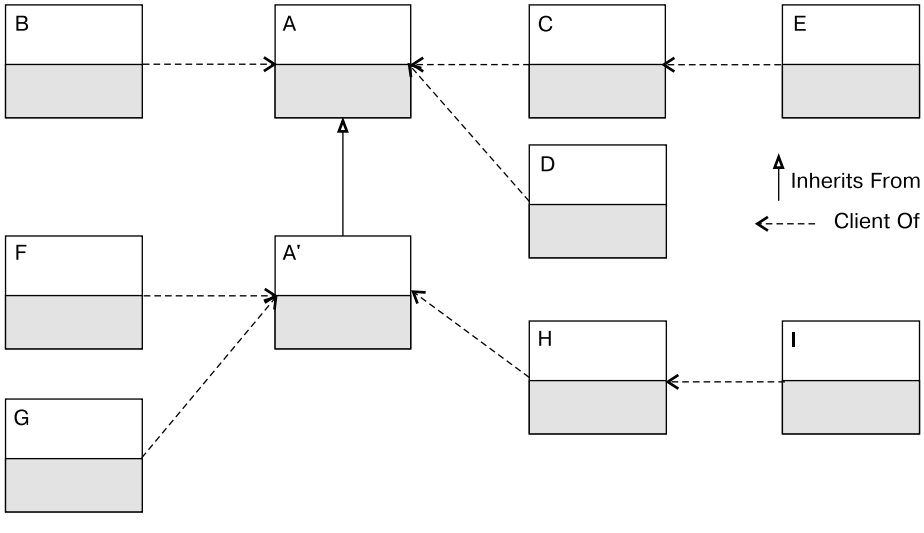


Рис. 18.1. Открытость и закрытость классов

Данный принцип не говорит, что создание наследника — это единственный возможный или же необходимый способ адаптации существующего кода к новым требованиям. Бертран Мейер является признанным гуру в области объектно-ориентированного программирования и описывает в своей книге 12 видов наследования, но даже он относится к расширяемости модулей путем создания наследников с разумным прагматизмом: «Если имеется возможность переписать исходную программу так, чтобы она, без излишнего усложнения, смогла удовлетворять потребности нескольких разновидностей клиентов, то следует это сделать».

Естественно, модуль должен модифицироваться при наличии в нем ошибок: «Как принцип «открыт/закрыт», так и переопределение в механизме наследования не позволяет справиться с дефектами разработки, не говоря уже об ошибках в программе. Если в модуле что-то не в порядке, то следует это сразу исправить в исходной программе, не пытаясь разбираться с возникающей проблемой в производном модуле[1]».

Какую проблему призван решить принцип «открыт/закрыт»

Смысл принципа OCP: дизайн системы должен быть простым и устойчивым к изменениям.

Это значит, что, когда требования изменятся (не «если», а именно «когда»), вы должны быть к этому готовы. Это не означает, что нужно создавать дополнительные уровни абстракции в приложении без необходимости. Мы просто должны ограничить каскад изменений и свести их количество к минимуму.

Как этого добиться?

Во-первых, за счет абстракции и инкапсуляции. Выделение существенных (важных) частей системы в открытой части класса позволяет сосредоточиться на важных аспектах поведения, не задумываясь о реализации, скрытой от клиентов в закрытой его части. Важно понимать, что абстракция не требует наличия интерфейсов или абстрактных классов. Класс `String` абстрагирует нас от конкретного представления строки и многих других подробностей, хотя и не реализует интерфейс `IString`.

Сокрытие информации (information hiding) заключается не только в наличии закрытых полей, недоступных клиентам класса/модуля. Сокрытие информации (или даже сокрытие реализации — implementation hiding) позволяет думать о классе как о черном ящике, который предоставляет определенные услуги лишь ему известным способом (Буч Г. — 2004).

Лисков прямо утверждает, что «абстракция будет работать только вместе с инкапсуляцией». На практике это означает наличие в классе двух частей: интерфейса и реализации. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения.

---

[1] Для справки: Мейер трактует понятия «модуль» и «класс» одинаково!

Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Во-вторых, за счет наследования. Выделение интерфейсов и абстрактных классов позволяет думать о задаче еще более абстрактно. Полиморфное поведение позволяет заменить один вариант реализации на другой во время исполнения, а также позволит использовать повторно значительные фрагменты кода.

Принцип «открыт/закрыт» на практике

Закрытость интерфейсов

Давайте рассмотрим принцип «открыт/закрыт» на примере развития утилиты для импорта лог-файлов.

Эта утилита может работать в одном из двух режимов:

- режим командной строки для одноразового импорта лог-файлов;
- режим сервиса, который будет отслеживать и импортировать новые записи лог-файлов по мере поступления.

Первый режим работы является относительно простым, но второй требует внесения серьезных изменений в текущий дизайн.

В данный момент чтение записей осуществляется с помощью pull-модели[1] взаимодействия и находится непосредственно в классе `LogImporter`. Нам же требуется push-модель взаимодействия, когда некоторый класс будет отслеживать состояние файла и уведомлять о наличии новых записей.

Принцип единственной обязанности говорит нам, что класс `LogImporter` уже не может отвечать за такой сложный аспект поведения. Нужно выделить отдельный класс — `LogFileReader`, который будет построчно читать лог-файл и уведомлять об этом наблюдателей. Он также может отслеживать изменения лог-файла и отправлять своим подписчикам новые фрагменты.

---

[1] Более подробно pull- и push-модели взаимодействия были рассмотрены в главе 5, посвященной паттерну «Наблюдатель».

Поскольку реализация данного поведения может потребовать существенных усилий, нам нужно зафиксировать интерфейс нового типа и дать возможность его клиентам реализовать свою часть обязанностей. В языке C# существует множество вариантов реализации паттерна «Наблюдатель». В данном случае идеально подходит использование интерфейса `IObservable<string>`[1], с помощью которого мы будем моделировать push-последовательность прочитанных записей (листинг 18.1).

Листинг 18.1. Реализация класса LogFileReader

```
public class LogFileReader : IObservable<string>
{
    private readonly string _fileName;

    public LogFileReader(string fileName)
    {
        _fileName = fileName;
    }

    public virtual IDisposable Subscribe(IObserver<string> observer)
    {
        using (var file = File.OpenText(_fileName))
        {
            string line = null;
            while ((line = file.ReadLine()) != null)
            {
                observer.OnNext(line);
            }
        }

        observer.OnCompleted(); return Disposable.Empty;
    }
}
```

[1] Подробнее о паттерне «Наблюдатель» и об использовании интерфейса IObservable<T> для его реализации вы можете прочитать в главе 5.

Текущая реализация класса `LogFileReader` весьма примитивна, но ее наличие позволит разработчику класса `LogImporter` сосредоточиться на своей части функциональности (листинг 18.2).

Листинг 18.2. Реализация класса LogImporter[1] на основе IObservable<string>

```
public class LogImporter
{
    private readonly IObservable<string> _logFileReader;
    private readonly LogEntryParser _parser = new LogEntryParser();

    public LogImporter(string logFile)
    {
        _logFileReader = new LogFileReader(logFile);
    }

    public void ImportLogs()
    {
        _logFileReader.Subscribe(ProcessString);
    }

    private void ProcessString(string line)
    {
        LogEntry logEntry;
        if (_parser.TryParse(line, out logEntry))
        {
            LogSaver.SaveEntry(logEntry);
        }
    }
}
```

В классе `LogImporter` происходит «подписка» на события чтения данных объектом `_logFileReader`. Библиотека реактивных расширений (Reactive Extensions, Rx) предоставляет набор методов расширения для интерфейса `IObserver<T>`, которые

[1] В главе 17 класс LogImporter принимал список лог-файлов в аргументах конструктора. В целях повышения читаемости кода в данной главе все версии класса LogImporter будут принимать имя лишь одного файла.

позволяют обрабатывать события более декларативным способом — используя LINQ-синтаксис. Эта же возможность позволяет обрабатывать push-последовательности с помощью анонимных или именованных методов, таких как `ProcessString`, а не путем создания классов, реализующих интерфейс `IObservable<T>`[1].

Разработчик класса `LogImporter` может абстрагироваться от процесса чтения еще сильнее и потребовать интерфейс `IObservable<string>` через аргументы конструктора. В любом случае теперь будет довольно просто развивать модули чтения и импорта независимо. Уже сейчас можно написать интеграционные тесты, которые будут проверять функциональность от начала до конца. Параллельно этому автор класса `LogFileReader` может заняться разработкой более «умной» версии этого класса, которая будет отслеживать текущую позицию файла и уведомлять подписчиков при появлении новых записей (рис. 18.2). Это может быть сделано путем опроса файла вручную через определенный период времени или же с помощью типа `FileSystemWatcher`[2].

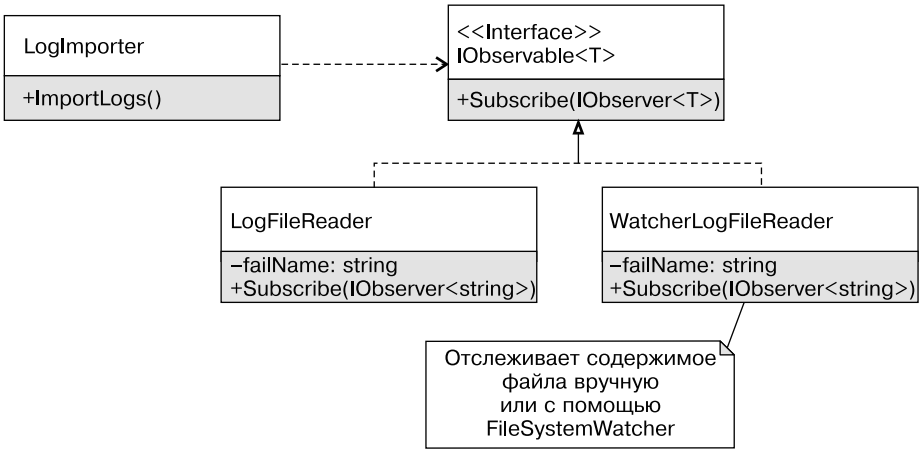


Рис. 18.2. «Открытость» реализации анализаторов лог-файлов

Открытость поведения

Помимо фиксации интерфейсов, принцип «открыт/закрыт» позволяет расширять реализацию модулей, без существенных изменений остальных частей системы.

---

[1] Подробнее о реактивном программировании и библиотеке Reactive Extensions можно прочесть в замечательной онлайн-книге Ли Кэмпбелла Introduction to Rx на сайте introrx.com.

[2] Класс FilSystemWatcher позволяет отслеживать состояние файла или папки и получать уведомления в случае изменения их содержимого.

Исходная версия импорта лог-файлов предназначалась для импорта логов лишь одного приложения. Для этого был выделен конкретный класс `LogEntryParser`, экземпляр которого создавался напрямую в классе `LogImporter`. Такой подход был совершенно оправдан, поскольку сложность парсинга была спрятана в отдельном классе и тщательно проверена, а также выполнена базовая подготовка к будущим изменениям. Дальнейшие изменения дизайна зависят от того, как именно будут меняться требования.

Вполне возможно, новым требованием будет поддержка импорта логов разного формата. Например, помимо логов серверной части, мы захотим отслеживать еще и логи нашей базы данных. В этом случае потребуется стратегия разбора лог-файлов, создание экземпляра которой может быть спрятано за фабричным методом. Для этого класс `LogEntryParser` делается абстрактным, текущая реализация перемещается в наследника и создается столько дополнительных производных классов, сколько форматов мы хотим поддерживать. При этом в класс `LogImporter` вносится лишь одно небольшое изменение (листинг 18.3).

Листинг 18.3. Использование абстрактного класса LogEntryParser

```
public class LogImporter
{
    // Старая версия
    // private readonly LogEntryParser _parser
    //    = new LogEntryParser();
    private readonly LogEntryParser _parser;

    public LogImporter(string logFile)
    {
        // Создаем нужный парсер в зависимости от имени файла
        // или его содержимого
        _parser = LogEntryParser.Create(logFile);
    }
```

Наличие стратегии анализа логов позволяет использовать один и тот же класс импорта данных для лог-файлов разного формата. Добавление нового вида лог-файлов требует изменения класса `LogEntryParser`, но не требует изменения его клиентов.

Иерархия классов для поддержки импорта логов сервера (`BackendLogEntryParser`) и логов базы данных (`PostgreLogEntryParser`) будет выглядеть следующим образом (рис. 18.3).

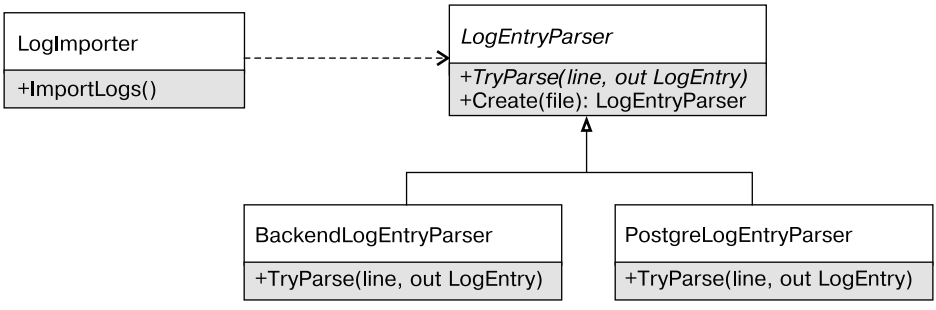


Рис. 18.3. Иерархия классов LogEntryParser

Открытость дизайна не означает расширения функциональности совсем без внесения изменений. Гибкий (supple) дизайн позволяет изменять поведение путем внесения изменений в модули, ответственные за данное поведение. При этом изменения второстепенных модулей отсутствуют либо их число сведено к минимуму.

Принцип единственного выбора

Разные интерпретации принципа «открыт/закрыт» по-разному трактуют понятие открытости. Давайте рассмотрим реализацию фабричного метода `LogEntryParser.Create` и подумаем, отвечает ли она принципу «открыт/закрыт». Данный метод может анализировать имя файла (возможно, и его содержимое) и создавать один из экземпляров — `BackendLogEntryParser` или `PostgreLogEntryParser` (листинг 18.4).

Листинг 18.4. Реализация фабричного метода LogEntryParser.Create

```
public abstract class LogEntryParser
{
    public static LogEntryParser Create(string fileName)
    {
        if (fileName.StartsWith("Server"))
        {
            return new BackendLogEntryParser(fileName);
        }

        if (fileName.StartsWith("Postgre"))
        {
```

```
            return new PostgreLogEntryParser(fileName);
        }

        throw new InvalidOperationException("Неизвестный формат файла");
    }

    // ...
}
```

Отвечает ли реализация такой фабрики принципу «открыт/закрыт»?

Вот что пишет Бертран Мейер по этому поводу: «Необходимо допускать возможность того, что список вариантов, заданных и известных на некотором этапе разработки программы, может в последующем быть изменен путем добавления или удаления вариантов. Чтобы обеспечить реализацию такого подхода к процессу разработки программного обеспечения, нужно найти способ защитить структуру программы от воздействия подобных изменений. Отсюда следует принцип Единственного Выбора».

Принцип единственного выбора: всякий раз, когда система программного обеспечения должна поддерживать множество альтернатив, их полный список должен быть известен только одному модулю системы.

Это означает, что фабрика отвечает принципу «открыт/закрыт», если список вариантов является ее деталью реализации. Если же информация о конкретных типах иерархии начинает распространяться по коду приложения и в нем появляются проверки типов (`as` или `is`), то это решение уже перестанет следовать принципу «открыт/закрыт». В этом случае добавление нового типа обязательно потребует каскадных изменений в других модулях, что негативно отразится на стоимости изменения.

Расширяемость: объектно-ориентированный и функциональный подходы

При рассмотрении принципа «открыт/закрыт» часто сравнивают подходы структурного и объектно-ориентированного программирования на примере рисования фигур (листинг 18.5).

Листинг 18.5. Примитивная реализация метода DrawShape

```
public static void DrawShape(Shape shape)
{
  switch(shape.ShapeType)
  {
    case ShapeType.Circle:
      DrawCircle((Circle)shape);
      break;
    case ShapeType.Square:
      DrawSquare((Square)shape);
      break;
    case ShapeType.Rectangle:
      DrawRectangle((Rectangle)shape);
      break;
    default:
      throw new InvalidOperationException("Неизвестный тип фигуры");
  }
}
```

Данный код является нерасширяемым, поскольку его придется обновлять каждый раз при добавлении новой фигуры. После чего приводится объектно-ориентированное решение, расширяемость которого реализуется за счет использования наследования и полиморфного метода `Draw` (листинг 18.5).

Листинг 18.6. Объектно-ориентированная версия метода DrawShape

```
public static void DrawShape(Shape shape)
{
  shape.Draw();
}
```

Теперь мы можем сказать, что избавились от конструкции `switch` в функции `Draw` и перенесли всю логику в класс `Shape` и его наследники. Решение легко расширяется и соответствует принципу «открыт/закрыт», поскольку к квадрату и треугольнику мы можем добавить еще и ромб с кругом!

Действительно, добавить новый класс в иерархию фигур довольно легко, но что, если мы хотим добавить новую операцию, например метод `GetArea`? Добавление

нового абстрактного метода в класс `Shape` является «ломающим» изменением (breaking change) и потребует изменения всех классов наследников. Когда всю иерархию контролирует один человек, внести такие изменения легко, но в случае библиотеки или широко используемых классов стоимость изменений будет очень высокой.

Объектно-ориентированное решение на основе полиморфизма позволяет легко расширять функциональность лишь в определенном направлении, но не является открытым к любым изменениям. Задача добавления новой операции в существующую иерархию типов решается с помощью паттерна «Посетитель». Для этого в базовый класс `Shape` добавляется абстрактный метод `Accept`, который принимает `IShapeVisitor`, а каждый конкретный класс иерархии просто вызывает метод `Visit` (листинг 18.7).

Листинг 18.7. Паттерн «Посетитель» и иерархия фигур

```
public interface IShapeVisitor
{
    void Visit(Circle circle);
    void Visit(Square square);
    void Visit(Rectangle rectangle);
}
public abstract class Shape
{
    public abstract void Accept(IShapeVisitor visitor);
}

public class Circle : Shape
{
    public override void Accept(IShapeVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

Теперь для добавления операции `GetArea` достаточно будет реализовать интерфейс `IShapeVisitor` и добавить метод расширения в класс `ShapeEx` (листинг 18.8).

Листинг 18.8. Добавление новой операции в существующую иерархию

```
public class ComputeAreaVisitor : IShapeVisitor
{
    public double Area { get; private set; }
    public void Visit(Circle circle)
    {
        Area = Math.PI * Math.Sqrt(circle.Radius);
    }

    public void Visit(Square square)
    {
        Area = Math.Sqrt(square.X);
    }

    public void Visit(Rectangle rectangle)
    {
        Area = rectangle.X * rectangle.Y;
    }
}

public static class ShapeEx
{
    public static double GetArea(this Shape shape)
    {
        var visitor = new ComputeAreaVisitor();
        shape.Accept(visitor);
        return visitor.Area;
    }
}

// Где-то в коде приложения
Shape shape = GetShape();
var area = shape.GetArea();
```

Открытость иерархий типов относительна. Если вы ожидаете, что более вероятным является добавление нового типа, то следует использовать классическую иерархию наследования. Если же иерархия типов стабильна, а все операции определяются клиентами, то более подходящим будет подход на основе паттерна «Посетитель»[1].

Паттерн «Посетитель» показывает функциональный подход к расширяемости семейства типов. В функциональном программировании операции четко отделены от данных. Свободные функции принимают на входе экземпляр неизменяемого типа данных и вычисляют результат в зависимости от типа. При этом добавить новую функцию очень просто, но добавление нового варианта в семейство типов может потребовать множества изменений.

Типичные примеры нарушения принципа «открыт/закрыт»

Типичными примерами нарушения принципа «открыт/закрыт» являются следующие.

- **Интерфейс класса является нестабильным.** Постоянные изменения интерфейса класса, используемого во множестве мест, приводят к постоянным изменениям во многих частях системы.
- **«Размазывание» информации об иерархии типов.** В коде постоянно используются понижающие приведения типов (downcasting), что «размазывает» информацию об иерархии типов по коду приложения. Это затрудняет добавление новых типов и усложняет понимание текущего решения.

Выводы

Что такое OCP? Фиксация интерфейса класса/модуля и возможность изменения реализации/поведения.

Цели OCP: борьба со сложностью и ограничение изменений минимальным числом модулей.

Как мы реализуем OCP? С помощью инкапсуляции, которая дает возможность изменять реализацию без изменения интерфейса, и посредством наследования, что позволяет заменить реализацию, не затрагивая существующих клиентов базового класса.

---

[1] Особенности паттерна «Посетитель», включая функциональную реализацию на основе делегатов, рассмотрены в главе 6.

Приведенные ранее примеры показывают «ортогональность» объектного и функционального подходов. Классический объектный подход позволяет легко добавлять новые типы в существующую иерархию типов, а функциональный подход позволяет легко добавлять новые операции. Проблема однобокости каждого из решений является одной из классических проблем программирования и носит название Expression Problem[1].

Во время дизайна модуля нужно подумать о том, в каком направлении упрощать расширяемость. Если наиболее вероятным является добавление новых типов, то более подходящим будет классический объектный подход на основе наследования. Если более вероятным является добавление новых операций в существующую иерархию типов, то лучше подойдет функциональный подход на основе размеченных объединений (discriminate dunions) или на основе паттерна «Посетитель» в объектном мире.

[1] Одно из лучших описаний данной проблемы можно найти здесь: http://c2.com/cgi/wiki?ExpressionProblem.

Глава 19

Принцип подстановки Лисков

Отыщи всему начало, и ты многое поймешь.

Где начало того конца, которым оканчивается начало?

Козьма Прутков

Принцип подстановки Лисков (Liskov Substitution Principle, LSP): «*Должна быть возможность вместо базового типа подставить любой его подтип*» (*Мартин Р.* Принципы, паттерны и практики гибкой разработки. — 2006).

«*...Если для каждого объекта o1 типа S существует объект o2 типа T такой, что для всех программ P, определенных в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом (subtype) для T*» (*Лисков Б.* Абстракция данных и иерархия. — 1988).

Наследование и полиморфизм являются ключевыми инструментами «объектно-ориентированного» разработчика для борьбы со сложностью и получения простого и расширяемого решения. Наследование используется в большинстве паттернов проектирования и лежит в основе принципа «открыт/закрыт» и принципа инверсии зависимостей.

Большинство опытных разработчиков знают, что с наследованием не все так просто. Наследование — это одна из самых сильных связей в объектно-ориентированном

мире, которая крепко привязывает наследников к базовому классу (сильнее только отношение дружбы[1]). Кроме того, не всегда легко ответить на два простых вопроса.

1. Когда наследование уместно?
2. Как его реализовать корректно?

Наследование обычно моделирует отношение «ЯВЛЯЕТСЯ» (IS-A Relationship) между классами. Говорят, что экземпляр наследника также ЯВЛЯЕТСЯ экземпляром базового класса, что выражается в возможности использования экземпляров наследника везде, где ожидается использование базового класса. Данный вид наследования называется также наследованием подтипов (Subtype Inheritance), но он не является единственно возможным. Бертран Мейер в своей книге «Объектно-ориентированное конструирование программных систем» приводит 12 (!) различных видов наследования, включая наследование реализации (закрытое наследование), IS-A, Can-Do (реализация интерфейсов) и т. п.

Большинство современных объектно-ориентированных языков программирования не поддерживают множественного или закрытого наследования, поэтому наибольший интерес вызывает именно наследование подтипов. Принцип подстановки Лисков призван помочь в корректной реализации этого вида наследования, что также должно помочь отказаться от наследования, если его корректная реализация невозможна.

Чтобы принцип подстановки был полезнее, нужно подобрать для него более звучное определение. Проблема с приведенными ранее определениями в том, что определение Роберта Мартина повторяет определение отношения «ЯВЛЯЕТСЯ», а исходное определение от Барбары Лисков выглядит слишком академичным и также слабо применимым на практике. Одно из лучших определений этого принципа находится на сайте Уорда Каннингема `c2.com`[2]: *«Должна существовать возможность использовать объекты производного класса вместо объектов базового класса. Это значит, что объекты производного класса должны вести себя согласованно, согласно контракту базового класса».*

---

[1] Дружеские отношения (friendship relationship) между классами полноценно поддерживаются в языке C++ и дают другу полный доступ к внутренней реализации класса. В языке C# отсутствует отношение дружбы между типами. Схожего поведения можно добиться путем использования вложенных классов. Вложенный класс определяется внутри другого класса и получает полный доступ к закрытым полям внешнего класса.

[2] Уорд Каннингем является создателем языка разметки Wiki, а также одним из соавторов экстремального программирования. Данное определение взято из его заметки Liskov Substitution Principle (http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple).

Для чего нужен принцип подстановки Лисков

Основной смысл любой иерархии наследования в том, что она позволяет использовать базовые классы полиморфным образом, не задумываясь о том, экземпляр какого конкретного класса был передан.

В своем приложении мы можем абстрагироваться от конкретной реализации потоков ввода-вывода (Streams), коллекций и последовательностей (Enumerables), провайдеров, репозиториев и т. п. Но чтобы поведение приложения оставалось корректным, оно должно отталкиваться от некоторых допущений, справедливых для всех реализаций абстракции: возвращаемое значение никогда не равно `null`, будут генерироваться исключения лишь определенного типа, любая реализация должна сохранить данные в базу данных (неважно какую), или метод добавления элемента обязательно его добавит и размер коллекции увеличится на единицу.

Здесь мы сталкиваемся с такой особенностью: с одной стороны, любая реализация должна следовать некоторому абстрактному протоколу или контракту, а с другой — она должна иметь возможность выбрать конкретный способ реализации этого протокола. Именно контракт, неважно, формальный или нет, описывает ожидаемое видимое поведение абстракции, оставляя реализации решать, каким образом это поведение будет реализовано.

Если же реализация (то есть наследники) не будет знать об этом протоколе или не будет ему следовать, то в приложении мы будем вынуждены обрабатывать конкретную реализацию специальным образом, что сводит на нет идею использования наследования и полиморфизма.

Почему важно следовать принципу подстановки Лисков? Потому что в противном случае:

- иерархии наследования приведут к неразберихе. Она будет заключаться в том, что передача в метод экземпляра класса-наследника вызовет странное поведение существующего кода;
- юнит-тесты базового класса никогда не будут проходить для наследников[1].

[1] Взято из все той же заметки Уорда Каннингема (http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple).

Классический пример нарушения: квадраты и прямоугольники

Наследование моделирует отношение «ЯВЛЯЕТСЯ». Но поскольку это лишь слово, мы не можем считать возможность его использования безоговорочным доказательством возможности применения наследования. Можем ли мы сказать, что цветная фигура является фигурой, контрактник является сотрудником, который, в свою очередь, является человеком, квадрат является прямоугольником, а круг — овалом?

Приведенные утверждения могут быть корректными или некорректными в зависимости от того, какое поведение мы будем приписывать фигурам, сотрудникам или квадратам. Именно в зависимости от спецификации или контракта этих сущностей (то есть ожидаемого поведения) мы можем говорить о возможности использования наследования.

Давайте более подробно рассмотрим пример с квадратами и прямоугольниками. С точки зрения математики квадрат является прямоугольником, но актуально ли это отношение для классов `Rectangle` и `Square` (рис. 19.1)?

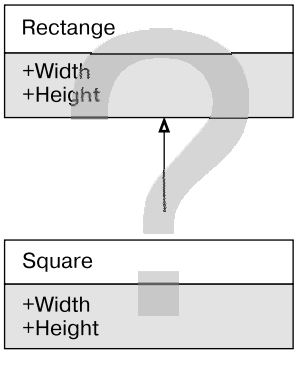


Рис. 19.1. Является ли квадрат прямоугольником?

Чтобы понять, будет ли нарушать данная иерархия классов принцип подстановки, нужно постараться сформулировать контракты этих классов.

- **Контракт прямоугольника** (инвариант): ширина и высота положительны.
- **Контракт квадрата** (инвариант): ширина и высота положительны, ширина и высота равны.

Пока нельзя сказать, являются ли контракты согласованными. Поскольку у квадрата все стороны равны, то изменение его ширины должно приводить к изменению высоты и наоборот. Это значит, контракт свойств `Width` и `Height` квадрата становится несогласованным с контрактом этих свойств прямоугольника! (С точки зрения клиента прямоугольника свойства `Width` и `Height` полностью независимы, а значит, замена прямоугольника квадратом во время исполнения нарушит это предположение клиента.)

Но это не значит, что данная иерархия наследования является невозможной. Квадрат перестает быть нормальным прямоугольником, только если квадрат и прямоугольник являются изменяемыми! Так, если мы сделаем их неизменяемыми (immutable), то проблема с контрактами, принципом подстановки и нарушением поведения клиентского кода при замене прямоугольников квадратами пропадет. Если клиент не может изменить ширину и высоту, то его поведение будет неизменным при замене прямоугольников квадратами!

ПРИМЕЧАНИЕ

Именно этой же логике следуют правила ковариантности и контравариантности обобщений в языке C#. Так, именно отсутствие изменяемости позволяет трактовать объект IEnumerable‹string› как IEnumerable‹object›. Поскольку мы можем лишь извлечь элемент из последовательности и не можем поместить его обратно, то такое преобразование является типобезопасным.

Данный пример показывает несколько важных моментов. Во-первых, именно наличие контракта позволяет четко понять, нарушает производный класс принцип подстановки Лисков или нет. А во-вторых, пример с неизменяемостью показывает пользу неизменяемости в объектно-ориентированном мире: контракт неизменяемых типов проще, поскольку контролируется лишь конструктором и инвариантом класса.

В-третьих, этот пример показывает, почему некоторые специалисты рекомендуют, чтобы классы были либо абстрактными, либо запечатанными (sealed)[1] и не было возможности создавать экземпляры классов из середины иерархии наследования. Так, классическим решением проблемы квадратов/прямоугольников является выделение промежуточного абстрактного класса «четырехугольник», от которого уже наследуются квадрат и прямоугольник. И хотя в некоторых случаях такой совет оправдан, слишком строго следовать ему не стоит. Вместо этого лучше точнее описывать контракт базовых классов с помощью контрактов.

[1] Запечатанные классы не могут иметь наследников.

Принцип подстановки Лисков и контракты

Не будет преувеличением сказать, что лишь с помощью принципов проектирования по контракту вы можете точно понять, что представляет собой наследование (*Мейер Б.* Объектно-ориентированное конструирование программных систем, раздел 16.1).

Существует несколько способов описать ожидаемое поведение типа, то есть его спецификацию. Для этого мы можем использовать комментарии (их никто не читает!), юнит-тесты (это лучше, но кто нам даст тесты для классов из BCL?) или контракты (заставить бы всех ими пользоваться!).

Пример с квадратами и прямоугольниками показал, что мы не можем доказать, нарушает конкретный класс принцип подстановки Лисков или нет, пока не определимся с тем, чего ожидают клиенты от поведения базового класса.

Если посмотреть на исходное описание принципа подстановки в трудах Барбары Лисков, то с удивлением можно обнаружить, что оно полностью основано на таких понятиях, как предусловия, постусловия и инварианты. Другими словами, описание этого принципа полностью основано на принципах проектирования по контракту.

1. Производные классы не должны усиливать предусловия (не должны требовать большего от своих клиентов).
2. Производные классы не должны ослаблять постусловия (должны гарантировать как минимум то же, что и базовый класс).
3. Производные классы не должны нарушать инварианты базового класса (инварианты базового класса и наследников суммируются).
4. Производные классы не должны генерировать исключения, не описанные базовым классом.

Попытка формализации обязанностей класса в терминах предусловий, постусловий и инвариантов позволит более четко сказать, является поведение наследника согласованным или нет.

О сложностях наследования в реальном мире

Отношение «ЯВЛЯЕТСЯ» подразумевает, что поведение клиентского кода должно быть предсказуемым при использовании любого конкретного класса иерархии типов. Существует мнение, что генерация методами наследника исключений

`InvalidOperationException` или `NotSupportedException` будет нарушать работу клиентского кода, а значит, такие наследники будут нарушать принцип подстановки Лисков. В некоторых случаях такое мнение справедливо, но в общем случае так судить не следует.

Давайте рассмотрим, является ли нарушением LSP то, что `ImmutableList<T>` и `ReadOnlyCollection<T>` реализуют `IList<T>`, ведь попытка добавления элемента в такую коллекцию приводит к генерации `NotSupportedException`. Наличие или отсутствие нарушения принципа подстановки определяется не наличием исключений, а контрактом реализуемого интерфейса. В «контракте» интерфейса `IList<T>` (точнее, интерфейса `ICollection<T>`) четко сказано, что метод `Add` может добавить элемент только в случае выполнения «предусловия»: коллекция должна быть изменяемой — свойство `IsReadOnly` возвращает `False`! Аналогично дела обстоят с потоками ввода/вывода, методы `Read`/`Write` которых могут генерировать исключения, если свойства `CanRead`/`CanWrite` возвращают `False`.

ПРИМЕЧАНИЕ

«Контракт» и «предусловие» специально взяты в кавычки. На самом деле это логический контракт и логическое предусловие, описанные в документации интерфейса ICollection‹T›, а не в классе его контракта — классе ICollectionContract‹T›.

Контракты списков и коллекций

Между постусловиями метода `Add` интерфейсов `IList<T>` и `ICollection<T>` существует небольшое, но очень важное различие. В случае выполнения предусловия метода `Add` реализация списка обязана добавить элемент в коллекцию. Постусловие интерфейса `ICollection<T>` слабее, и такой гарантии нет. Существуют некоторые виды коллекций, вызов метода `Add` которых не увеличивает количество элементов на 1. Любые наборы (sets) подразумевают наличие в коллекции уникальных элементов, а значит, вызов метода `Add` на экземпляре `HashSet<int>` не обязательно приведет к увеличению числа элементов.

Если поведение метода зависит от состояния объекта, то мы можем говорить о неудачном дизайне, но не можем утверждать, что реализация нарушает принцип подстановки Лисков! Даже если метод наследника всегда генерирует `InvalidOperationException`, это говорит лишь о том, что не все методы интерфейса или базового класса применимы к наследнику, но это не означает и некорректного использования наследования.

Принцип подстановки Лисков говорит о том, что если при замене объектов класса `T` объектами класса `S` поведение клиентского кода не меняется, то `S` является подтипом `T` (`T` является базовым классом для `S`). А что, если поведение при такой замене все-таки изменится? Означает ли это, что нарушается некоторый принцип проектирования, или это значит, что `S` не является подтипом `T`?

Согласно принципу наименьшего удивления мы привыкли считать, что должна существовать возможность использовать экземпляр наследника вместо экземпляра базового класса. Но ведь это не значит, что мы никогда не используем классы-наследники напрямую! Мы добавляем свойства и методы в производные классы, хотя и понимаем, что они не будут доступны через призму базового класса. Нас при этом совсем не волнует, что клиент должен знать о классе-наследнике, чтобы ими воспользоваться.

Наследование подтипов (subtype inheritance) является одним из многих видов наследования, но далеко не единственным. Наследник имеет право добавлять новые методы (расширяющее наследование), «удалять» или не реализовывать некоторые методы базового класса (сужающее наследование), переименовывать методы базового класса для более точной передачи семантики производного класса. Наследование вообще может быть закрытым (private inheritance), и тогда преобразование экземпляров класса-наследника к базовому классу будет запрещено компилятором!

Данные техники наследования в полном виде поддерживаются лишь несколькими языками программирования. Язык Eiffel, разработанный Бертраном Мейером, поддерживает большинство таких техник, но и в практике современного .NET-разработчика они также встречаются.

Явная реализация интерфейсов в C# (explicit interface implementation) является примером сужающего наследования: класс реализует интерфейс, но при этом методы интерфейса видны лишь в случае явного приведения экземпляра класса к интерфейсу (листинг 19.1).

Листинг 19.1. Работа с коллекциями только для чтения

```
var list = new List<int> { 42 }.AsReadOnly();


//list.Add(42); // ошибка компиляции!
((IList<int>)list).Add(42); // ошибка времени исполнения
IList<int>list2 = list; // к сожалению, здесь явное приведение не требуется!
list2.Add(42); // ошибка времени исполнения
```

К сожалению, в языке C# остается возможность неявного приведения экземпляра класса к явно реализованному интерфейсу. В противном случае сходство с сужающим наследованием было бы максимальным: нам обязательно пришлось бы приводить экземпляр класса к нужному интерфейсу, чтобы воспользоваться его членами.

Явная реализация интерфейса используется также для переименования членов интерфейса — возможности, активно используемой в языке Eiffel при решении конфликтов множественного наследования, а также для использования в классах-наследниках более подходящих имен. Классическим примером использования

этой возможности является класс `Socket`, который реализует интерфейс `IDisposable` явно и при этом вводит «синоним» спрятанного метода `Dispose` с именем `Close` (листинг 19.2).

Листинг 19.2. Пример работы с сокетами

```
Socket s = CreateSocket();
s.Dispose(); // Не компилируется до .NET 4.0
((IDisposable)s).Dispose(); // Всегда OK
s.Close(); // Всегда OK
```

Об этой технике явно говорится в книге Framework Design Guidelines[1] в разделе Dispose Pattern Guidelines: автор класса имеет полное право при реализации интерфейса воспользоваться более подходящим именем и «спрятать» от клиентов имя из интерфейса.

ПРИМЕЧАНИЕ

В плане наследования C++ является существенно более функциональным языком, нежели C#. Помимо множественного наследования, C++ поддерживает большую часть техник, описанных Б. Мейером. Так, в C++ есть наследование реализации (закрытое наследование), при этом автоматическая конвертация экземпляров наследников в экземпляры базовых классов производиться не будет. C++ поддерживает изменение видимости функции при переопределении: видимость можно как расширить (от защищенной в базовом классе к открытой в наследнике), так и сузить (от открытой в базовом классе к закрытой/защищенной в наследнике). В C++ легче моделировать «примеси» и др.

Когда наследования бывает слишком мало

Обычно проблемы с наследованием возникают из-за слишком «широких» или «глубоких» иерархий наследования. Но иногда проблемы в дизайне возникают из-за того, что класс унаследован не от того базового класса или не реализует нужный интерфейс.

Классическим нарушением принципа «открыт/закрыт» является перебор типов: если переданный объект — это круг, рисуем круг, если квадрат — рисуем квадрат, если слон — рисуем слона, при этом такой перебор «размазан» по коду приложения. Обычно это характерно для структурного подхода, но часто применяется и в объектно-ориентированных решениях.

[1] *Abrams B., Cwalina K.* Framework Design Guidelines, 2nd Ed. — Addison-Wesley Professional, 2008.

В идеальном объектно-ориентированном мире все, что делает иерархия классов, находится прямо в ней, однако в реальном мире часть логики периодически оказывается за ее пределами. Хорошим примером является LINQ (Language Integrated Query) — интегрированный язык запросов. LINQ содержит унифицированную логику работы с любыми последовательностями — сущностями, которые реализуют интерфейс `IEnumerable<T>`. Это не обязательно должны быть коллекции в памяти, это могут быть бесконечные генераторы, реализованные с помощью блока итераторов и ключевого слова `yieldreturn`, или XML-файлы, или записи из базы данных.

Но даже когда дело касается коллекций, класс `Enumerable` содержит специализированные реализации некоторых операций, таких как `Count()`, `ElementAt()`, `Last()` и др., для конкретных типов коллекций, например `Collection<T>`, `List<T>` и т. п.

Примерный вариант реализации метода `Enumerable.Count()` представлен в листинге 19.3.

Листинг 19.3. Реализация метода расширения Count

```
public static int Count<TSource>(this IEnumerable<TSource> source)
{
    var collectionoft = source as ICollection<TSource>;
    if (collectionoft != null) return collectionoft.Count;

    var collection = source as ICollection;
    if (collection != null) return collection.Count;

    int count = 0;
    using (IEnumerator<TSource> e = source.GetEnumerator())
    {
        while (e.MoveNext()) count++;
    }

    return count;
}
```

Данная специализация позволяет получить значительный прирост производительности при вызове метода `Count()` на коллекциях (списках, массивах и любых

других классах, прямо или косвенно реализующих интерфейс `ICollection`). Однако, поскольку проверяется лишь интерфейс `ICollection`, эта реализация будет неэффективной для других коллекций, которые знают о своем размере, но не реализуют интерфейс `ICollection`.

В .NET Framework существует класс «коллекции» со свойством `Count`, который не реализует `ICollection`, — класс `Lookup<T>` (листинг 19.4).

Листинг 19.4. Использование метода Count для типа Lookup

```
var lookup = Enumerable.Range(1, 10000).ToLookup(x => x);
Console.WriteLine(lookup.Count); // Сложность O(1)[1]
Console.WriteLine(lookup.Count()); // Сложность O(n)
```

В данном случае использование `lookup.Count` является значительно более эффективным, поскольку не зависит от числа элементов. Вычислительная сложность `lookup.Count()` является линейной, а значит, время вызова будет пропорционально числу элементов данной коллекции.

Этот пример показывает, что проблемы с наследованием бывают разных типов: когда поведение наследника противоречит поведению базового класса или когда класс не реализует нужный интерфейс или базовый класс.

Принцип подстановки Лисков на практике

В предыдущей главе мы добавили иерархию парсеров лог-файлов (листинг 19.5).

Листинг 19.5. Интерфейс класса LogEntryParser

```
public abstract class LogEntryParser
{
    public abstract bool TryParse(string line, out LogEntry logEntry);
}
```

---

[1] O-нотация предназначена для обозначения асимптотической сложности алгоритмов. Другими словами, она говорит о зависимости времени исполнения или количества используемой алгоритмом памяти от количества входных элементов. O(1) означает, что время алгоритма не зависит от количества входных элементов. O(n) говорит о линейной зависимости: при увеличении числа элементов в два раза время алгоритма также изменится вдвое. Подробнее об асимптотической сложности алгоритмов можно прочитать на «Вики» (http://ru.wikipedia.org/wiki/Вычислительная_сложность) или в вашей любимой книге по алгоритмам и структурам данных.

Глядя на объявление данного класса, многие опытные разработчики поймут его неформальный контракт. Метод должен возвращать `true` в случае успешного разбора строки и `false` — в противном случае. При успешном разборе выходной аргумент `logEntry` должен содержать рассмотренное значение.

Но данный контракт не является формальным. Он не описывает возможные типы генерируемых исключений, а также поведение при передаче `null` в качестве параметра `line`. Неформальность контрактов часто приводит к тому, что замена одной реализации на другую ломает поведение существующего кода.

Лучший способ избежать неоднозначности — постараться максимально формализовать контракт абстрактного метода. Сделать это можно с помощью предусловий, или постусловий, или по крайней мере комментариев (листинг 19.6).

Листинг 19.6. Контракт интерфейса LogEntryParser

```
public class LogEntryParserException : Exception
{}

[ContractClass(typeof (LogEntryParserContract))]
public abstract class LogEntryParser
{
    /// <summary>
    /// Анализирует переданную строку и возвращает результат
    /// через <paramref name="logEntry"/>
    /// </summary>
    /// <exception cref="LogEntryParserException">
    /// Генерируется, если переданная строка не соответствует
    /// ожидаемому формату.
    /// </exception>
    public abstract bool TryParse(string line, out LogEntry logEntry);
}

[ContractClassFor(typeof (LogEntryParser))]
abstract class LogEntryParserContract : LogEntryParser
{
    public override bool TryParse(string line, out LogEntry logEntry)
    {
```

```
        Contract.Requires(line != null);
        Contract.Ensures(!Contract.Result<bool>() ||
                         Contract.ValueAtReturn(out logEntry) != null,
                         "Если результат True, то logEntry не может
                         быть null");
        throw new NotImplementedException();
    }
}
```

При наличии контракта наследники будут знать, чего ожидают клиенты базового класса, что им можно делать при переопределении методов, а что — нет. Следующий перечень действий наследников класса `LogEntryParser` нарушает принцип подстановки Лисков.

- Генерация исключений, отличных от `LogEntryParserException` или его наследников. Другие исключения возможны, но они будут означать ошибки (баги) в реализации классов-наследников (листинг 19.7).

  Листинг 19.7. Генерация некорректного типа исключения методом TryParse

  ```
  public class IncorrectLogEntryParser : LogEntryParser
  {
      public override bool TryParse(string line, out LogEntry logEntry)
      {
          if (LineIsIncorrect(line))
          {
              // Неизвестный тип исключения! Нарушение LSP!
              throw new FormatException("Can't parse the line!");
          }
          // ...
      }
  }
  ```

- Метод `TryParse` возвращает `false` при передаче `null` в качестве значения аргумента `line`. В этом случае должно генерироваться исключение `ArgumentNullException` или `ContractException` (листинг 19.8).

  Листинг 19.8. Некорректная проверка аргументов методом TryParse

  ```
  public override bool TryParse(string line, out LogEntry logEntry)
  {
  ```

```
    logEntry = null;
    // Отсутствие генерации исключения, когда line == null!
    // Нарушение LSP!
    if (string.IsNullOrEmpty(line))
        return false;
    // ...
}
```

- Метод `TryParse` возвращает `true`, но не возвращает значение через возвращаемый аргумент `logEntry` (листинг 19.9).

Листинг 19.9. Некорректное возвращаемое значение методом TryParse

```
public override bool TryParse(string line, out LogEntry logEntry)
{
    logEntry = null;
    // logEntry равен null! Нарушение LSP!
    return true;
    // ...
}
```

Далеко не всегда нужно описывать контракт базовых классов столь формально. Но чем лучше вы и другие разработчики будете его понимать, тем меньше шансов, что поведение текущего кода будет нарушено при замене одной реализации на другую.

Типичные примеры нарушения LSP

Типичными примерами нарушения LSP являются следующие.

- Производные классы используются полиморфным образом, но их поведение не согласуется с поведением базового класса: генерируются исключения, не описанные контрактом базового класса, или не выполняются действия, предполагаемые контрактом базового класса.
- Контракт базового класса настолько нечеткий, что реализовать согласованное поведение наследником просто невозможно.

Выводы

Принцип подстановки Лисков не является панацеей в вопросах наследования, он лишь помогает формализовать, в каких пределах может варьироваться поведение наследника

с точки зрения контракта базового класса. В своих трудах Барбара Лисков строила анализ на основе контрактов класса: предусловий, постусловий и инвариантов. Именно с помощью контрактов мы можем хотя бы с некоторой долей уверенности утверждать, что поведение наследника и базового класса являются согласованными.

Когда речь касается наследования, разработчик должен четко понимать, для чего он его использует и каким образом клиенты будут пользоваться наследниками: лишь через призму базового класса, напрямую или же и так и этак. Когда тип предназначен лишь для полиморфного использования, то такое наследование является наследованием подтипов и должно соответствовать принципу подстановки Лисков. Если же создание наследника нужно для повторного использования кода базового класса или производный класс будет всегда использоваться напрямую, то вполне возможно, что его интерфейс и контракт будут изменены: добавлены новые методы и/или не реализованы некоторые методы базового класса.

Нельзя говорить, что второй пример использования наследования вреден или не должен использоваться на практике — как минимум он должен быть обдуман и четко описан в документации класса-наследника, чтобы пользователь вашего класса знал о таком решении. Создание иерархий наследования, удобных для развития и использования, — это сложный итеративный процесс. Четкое следование принципам проектирования вообще и принципу замещения в частности не гарантирует хорошего дизайна или удобной в использовании иерархии наследования.

Дополнительные ссылки

- *Barbara Liskov and Jannette Wing.* A Behavioural Notion of Subtyping (http://web.cse.ohio-state.edu/~neelam/courses/788/lwb.pdf). Это та самая статья, в которой Барбара Лисков описывает свой принцип, опираясь на такие понятия, как предусловия, постусловия и инварианты.
- Обсуждение принципа подстановки Лисков на c2.com — http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple. На этой странице принцип подстановки обсуждают легенды современной разработки ПО Майкл Физерс, Алистер Кокберн, Роберт Мартин и др.
- Programming Stuff. Цикл статей о проектировании по контракту (первая статья серии — http://bit.ly/DbCBasics). Контрактное программирование является очень полезной методикой проектирования. В контексте принципа подстановки просто необходимо понимать, хотя бы на базовом уровне, что такое контракт класса, независимо от того, доступны ли инструменты контрактного программирования для вашего языка или нет.

Глава 20

Принцип разделения интерфейсов

Многие вещи нам непонятны не потому,
что наши понятия слабы; но потому,
что сии вещи не входят в круг наших понятий.

Козьма Прутков

Принцип разделения интерфейсов (Interface Segregation Principle, ISP): *«Клиенты не должны вынужденно зависеть от методов, которыми не пользуются»* (*Мартин Р.* Принципы, паттерны и практики гибкой разработки. — 2006).

Необходимым условием повторного применения кода является простота его использования. Разработчик никогда не будет использовать чужой код, если ему проще реализовать эту возможность самостоятельно, чем искать чье-то решение и разбираться в нем. С одной стороны, программисты ленивы и не склонны делать больше, чем того требует задача, но с другой — они достаточно горды и упрямы для того, чтобы считать любое чужое решение по определению менее удачным, чем собственное.

Синдром «я бы сделал это не так» является главным камнем преткновения во время рецензирования кода. Этот же синдром останавливает многих разработчиков от использования чужого кода. На своем горьком опыте разработчики уяснили, что разработка повторно используемого кода — дело сложное. Поэтому они нечасто

применяют чужой код, но активно продвигают повторное использование собственного.

Одной из причин скептического отношения к повторно используемому коду является сложность проектирования библиотек, простых и удобных в применении. С одной стороны, библиотека должна решать базовые вещи простым и интуитивно понятным способом, с другой — она должна делать возможными и более сложные сценарии. Добиться простоты использования кода можно за счет проектирования чистых интерфейсов — интерфейсов, предоставляющие цельный набор операций, которого будет необходимо и достаточно большинству клиентов.

Для чего нужен принцип разделения интерфейса

Принцип разделения интерфейса предназначен для получения простого и слабосвязного кода. Он гласит, что клиенты должны зависеть лишь от тех методов, которые используют, и не должны знать о существовании не интересующих их частей в интерфейсе применяемых ими сервисов. Как мы увидим позднее, разработчик сервиса не всегда знает о том, кто и как его будет использовать. Поэтому может потребоваться несколько итераций для перегруппировки методов таким образом, чтобы их использование было удобным максимальному числу клиентов.

Данный принцип является частным случаем принципа наименьшего знания, который много лет используется в нашей индустрии. Для получения простого в сопровождении кода каждый класс должен знать минимум информации об окружающем коде, необходимой для решения своей задачи. На практике это проявляется в минимизации числа зависимостей и стремлении к использованию наиболее простых типов зависимостей.

Сопровождаемость класса определяется его внутренней сложностью и связанностью с другими классами. Чем больше у класса зависимостей, тем сложнее понять его роль, сложнее тестировать и использовать повторно. К тому же большое число связей увеличивает вероятность поломки класса при изменении зависимостей. Стабильность зависимостей играет важную роль. Чем ниже вероятность изменения интерфейса зависимости или его поведения, тем меньше вероятность поломки вашего кода. Далее представлены виды зависимостей, стабильность которых уменьшается от очень стабильной до нестабильной:

- примитивные типы;
- объекты-значения (неизменяемые пользовательские типы);
- объекты со стабильным интерфейсом и поведением (пользовательские типы, интерфейс которых стабилен, а поведение не зависит от внешнего окружения);
- объекты с изменчивыми интерфейсом и поведением (типы расположены на стыке модулей, которые постоянно подвергаются изменениям, или типы, которые работают с внешним окружением: файлами, базами данных, сокетами и т. п.).

ПРИМЕЧАНИЕ

Мы затронули очень сложную тему проектирования, связанную с управлением зависимостями. Чтобы раскрыть ее, требуется несколько глав или даже целая книга. Дополнительными источниками по этой теме могут служить книга Марка Симана Dependency Injectionin .NET и цикл моих статей по управлению зависимостями (http://bit.ly/DependencyManagement).

Принцип разделения интерфейсов также соответствует одному из принципов контрактного программирования: **требуй меньше, но гарантируй больше**. В терминах контрактов это проявляется в виде использования более слабого предусловия и более строгого постусловия. Чем проще аргументы метода, тем проще удовлетворять предусловиям метода. На практике это правило выражается в использовании наиболее простых типов (согласно приведенной ранее шкале), а также базовых типов в качестве входных аргументов (листинг 20.1).

Листинг 20.1. Использование базовых и производных типов в качестве аргументов метода

```
public Result Process1(List<RequestArgs> input)
{
  foreach(var ra in input)
  {
    // ...
  }
}

public Result Process2(IEnumerable<RequestArgs> input)
{
  foreach(var ra in input)
  {
    // ...
  }
}
```

Если метод использует лишь члены интерфейса `IEnumerable<T>`, то нет смысла заявлять, что он требует `List<T>`. Если метод может работать с любым потоком ввода-вывода, то лучше ему принимать `Stream`, а не `MemoryStream`. Если классу требуется конфигурация, то лучше передавать в аргументах конструктора экземпляр класса `Configuration` (объект-значение), а не провайдер `IConfigurationProvider`, который будет читать конфигурацию в методе `ReadConfiguration`.

ПРИМЕЧАНИЕ

В случае с коллекциями и последовательностями нужно очень обдуманно подходить к выбору типа аргументов. Часто бывает, что в качестве аргумента используют IEnumerable‹T›, но при этом рассчитывают на то, что будет передаваться коллекция в памяти. Например, метод может несколько раз вызывать метод расширения Count() в расчете на то, что он будет выполняться быстро, поскольку в метод передается лишь список или массив. Это является примером неявной связи (implicit coupling) и может привести к серьезным проблемам при сопровождении, когда в качестве аргумента метода начнет передаваться последовательность, генерируемая на лету с помощью блока итераторов.

SRP vs. ISP

Принцип разделения интерфейсов является довольно простым и очень полезным принципом проектирования. Но его иногда путают с принципом единственной обязанности. Причина такого недопонимания лежит в классическом описании ISP (Мартин Р. — 2006).

Этот принцип относится к недостаткам «жирных» интерфейсов. Говорят, что класс имеет «жирный» интерфейс, если функции этого интерфейса недостаточно хорошо сцеплены (not cohesive), иными словами, если интерфейс класса можно разбить на группу методов. Каждая группа предназначена для обслуживания разных клиентов. Одним клиентам нужна одна группа методов, другим — другая.

Проблема этого определения в том, что в нем акцент делается на «жирности». «Жирный» интерфейс, который содержит несколько групп методов, недостаточно сцепленных между собой, нарушает принцип единственной обязанности. Такие методы используются разными клиентами и будут развиваться независимо. Однако все не так просто.

Давайте вспомним, как мы определяем, что класс или модуль нарушает принцип единственной обязанности. Мы открываем исходный код этого класса или модуля и смотрим, не делает ли он слишком многого. Если класс или модуль отвечает за

выполнение разнородных задач, значит, он нарушает принцип единственной обязанности. Но можем ли мы, глядя на класс или его интерфейс, сказать, нарушает он принцип разделения интерфейсов или нет?

Например, у нас есть класс репозитория, который содержит CRUD-операции[1]. Нарушает ли он ISP? Мы не знаем! Нарушение этого принципа зависит не столько от самого класса, сколько от сценариев его использования. Если в нашей бизнес-модели четко разделяются операции чтения и обновления данных, то наличие одного класса со всеми операциями работы с данными однозначно делает интерфейс слишком «жирным». В то же время если приложение содержит множество простых форм пользовательского интерфейса, которые соотносятся как 1 × 1 с нашими поставщиками данных, то принцип ISP не нарушается.

Из предыдущего обсуждения можно вывести важное различие принципов SRP и ISP. Следование принципу единственной обязанности приводит к связным (cohesive) классам, что позволяет с меньшими усилиями их понимать и развивать. Следование принципу разделения интерфейсов уменьшает связанность (coupling) между классами и их клиентами, ведь теперь клиенты используют более простые зависимости, чем раньше.

Принцип разделения интерфейсов на практике

На практике периодически возникает необходимость выделения у класса вспомогательных интерфейсов. Класс может наследоваться от базового класса для повторного использования кода, но реализовывать дополнительный интерфейс, который нужен лишь части его клиентов.

Одна из реализаций паттерна «Адаптер»[2] заключается в реализации адаптируемым классом дополнительного интерфейса. Итак, у нас есть два класса сохранения данных с несовместимыми интерфейсами — `SqlServerLogSaver` и `ElasticsearchLogSaver`. При наличии доступа к их исходному коду мы можем не создавать новые классы-адаптеры, а просто заставить текущие классы реализовать новый интерфейс — `ILogSaver` (рис. 20.1).

---

1 CRUD (Create, Read, Update, Delete) — набор ключевых операций для работы с постоянными хранилищами данных, такими как база данных.

2 Подробнее паттерн «Адаптер» рассмотрен в главе 12.

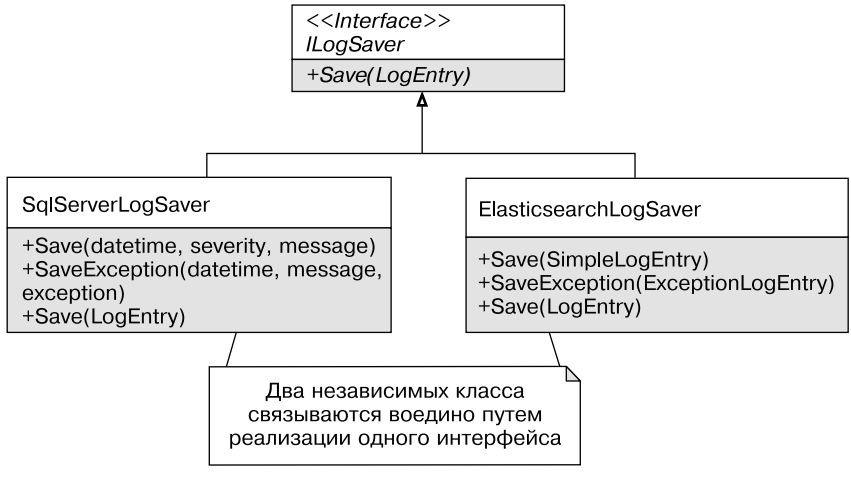


Рис. 20.1. Использование паттерна «Адаптер» для разделения интерфейсов

Теперь у классов `SqlServerLogSaver` и `ElasticsearchLogSaver` появляются два вида клиентов: старые и новые. При этом реализовать интерфейс следует явным образом (explicit interface implementation), что позволит изолировать данных клиентов друг от друга (листинг 20.2).

Листинг 20.2. Явная реализация интерфейса ILogSaver

```
public class SqlServerLogSaver : ILogSaver
{
    public void Save(DateTime dt, string severity, string message) {...}
    public void SaveException(DateTime dt, Exception e) {...}

    // Явная реализация интерфейса!
    ILogSaver.Save(LogEntry logEntry)
    {
        var exception = logEntry as ExceptionLogEntry;
        if (exceptiony != null)
        {
            SaveException(exception.EntryDateTime, exception.Exception);
        }
        else
        {
```

```
            Save(logEntry.EntryDateTime, logEntry.Severity.ToString(),
                logEntry.Message);
        }
    }
}
```

Явная реализация интерфейсов позволяет вызывать методы интерфейса лишь при явном приведении объекта к типу интерфейса. Это значит, что обычные клиенты класса `SqlServerLogSaver` не будут «видеть» метод `Save(LogEntry)`, а смогут им пользоваться лишь в случае явного приведения типов (листинг 20.3).

Листинг 20.3. Явная реализация интерфейсов для четкого разделения интерфейсов

```
var saver = new SqlServerLogSaver();
LogEntry logEntry = GetLogEntry();
saver.Save(logEntry); // Ошибка компиляции

ILogSaver logSaver = saver;
logSaver.Save(logEntry); // OK
```

Еще один пример следования принципу разделения интерфейсов заключается в использовании методов расширения. Вместо создания «жирного» интерфейса, которым будет удобно пользоваться всем клиентам, можно выделить базовый интерфейс, а вспомогательные методы реализовать в виде методов расширения. В этом случае базовый интерфейс будет максимально простым и разные клиенты самостоятельно решат, какие методы расширения использовать.

Подобный пример был рассмотрен в главе 17, посвященной принципу единственной обязанности. Интерфейс класса `LogEntryParser` содержит универсальный метод `TryParse`, пользоваться которым не всегда удобно. Поэтому для тестов был добавлен метод расширения `Parse`, который всегда возвращает результат и не предназначен для разбора многострочных записей (листинг 20.4).

Листинг 20.4. Использование принципа разделения интерфейсов с методами расширения

```
// В основном коде приложения
var parser = new LogEntryParser();
LogEntry logEntry;
if (parser.TryParse(stringEntry, outlogEntry))
{
```

```
    SaveEntry(logEntry);
}

// В тестах
// Импортируем пространство имен с методом расширения
varparser = newLogEntryParser();
varlogEntry = parser.Parse(sampleStringEntry);
Assert.That(logEntry.Severity, Is.EqualTo(Severity.Debug));
```

Явная реализация интерфейсов и методы расширения позволяют четко следовать принципу разделения интерфейсов. Каждый клиент знает лишь о подмножестве операций и не видит лишних деталей, которые ему неинтересны.

Типичные примеры нарушения ISP

Типичными примерами нарушения ISP являются следующие.

- Метод принимает аргументы производного класса, хотя достаточно использовать базовый класс.
- У класса два или более ярко выраженных вида клиентов.
- Класс зависит от более сложной зависимости, чем нужно: принимает интерфейс провайдера вместо результатов его работы и т. п.
- Класс зависит от сложного интерфейса, что делает его зависимым от всех типов, используемых в этом интерфейсе.

Выводы

Принцип разделения интерфейсов является частным случаем управления зависимостями и борьбы со сложностью. Чем проще зависимости класса, тем легче понять, что класс делает, поскольку в голове приходится держать лишь минимальное число ненужных деталей. Чем стабильнее зависимости класса, тем меньше вероятность того, что его поведение будет нарушено при внесении изменений в другие части системы.

Принцип разделения интерфейсов лежит на стыке классов или модулей. Глядя на исходный код некоторого класса, мы можем сказать, соответствует ли реализация

принципу единственной обязанности, принципу замещения Лисков или принципу «открыт/закрыт». Но лишь по исходному коду класса или его интерфейса мы не можем судить о том, нарушает он принцип разделения интерфейсов или нет. Для этого нужно посмотреть контекст его использования: есть ли разные группы клиентов, которые используют его по-разному, или нет.

Если класс используется разными клиентами, это может говорить о слишком большом числе обязанностей, поэтому его нужно упростить. В некоторых случаях у класса может быть одна обязанность, которая рассматривается клиентами с разных точек зрения. Тогда этот факт нужно сделать явным путем реализации двух или более интерфейсов.

Глава 21

Принцип инверсии зависимостей

Когда душа уходит в пятки,
встань вверх ногами и встряхнись!

Козьма Прутков

Принцип инверсии зависимостей (Dependency Inversion Principle, DIP): «*Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций*» (*Мартин Р.* Принципы, паттерны и практики гибкой разработки. — 2006).

Принцип инверсии зависимостей — один из самых известных сегодня принципов проектирования, который лежит в основе популярных техник внедрения зависимостей (Dependency Injection). Однако, если посмотреть лишь на его название и описание, будет довольно сложно понять, что же он означает. А если спросить простых обывателей о том, как они понимают этот принцип, они начнут что-то говорить о пользе интерфейсов и абстракций и вообще будут путаться в показаниях.

Интерфейсы

В основе принципа инверсии зависимостей лежит идея использования интерфейсов[1]. Одна группа классов реализует некоторый набор интерфейсов, а другая — принимает эти интерфейсы в качестве аргументов конструктора (листинг 21.1).

Листинг 21.1. Пример наивного использования DIP

```
interface IFileReader
{
  string ReadLine();
}

class LogEntryParser
{
    public LogEntryParser(IFileReader fileReader)
    {}

    public IEnumerable<LogEntry> ParseLogEntries()
    {}
}

class FileReader : IFileReader {...}
```

Использование интерфейсов приводит к слабосвязанному (loosely couplied) дизайну, поскольку класс `LogEntryParser` знает лишь об интерфейсе `IFileReader` и не знает о конкретной реализации этого интерфейса. А следование принципу замещения Лисков позволит заменить одну реализацию другой и получить гибкое решение, соответствующее принципу «открыт/закрыт».

И хотя данные рассуждения вполне логичны, не стоит забывать, что у такого решения есть и обратная сторона. Наличие интерфейсов образует дополнительный уровень косвенности (или дополнительный уровень абстракции), что затрудняет понимание системы. Полиморфизм — это GOTO объектного мира, который делает

[1] Под интерфейсами в данном случае понимаются программные конструкции, объявленные с помощью ключевого слова interface. В большинстве случаев вместо интерфейсов можно использовать абстрактные классы, и все рассуждения данной главы еще будут актуальными.

решение более гибким ценой увеличения сложности. Понять логику исполнения лишь путем чтения кода становится довольно сложно, поскольку конкретный тип наследника определяется во время исполнения.

В нашем случае клиентам класса `LogEntryParser` приходится решать проблему с поиском подходящей зависимости, даже если им это неинтересно. Только представьте себе, что у класса `List<T>` появилась бы зависимость вида `IList-GrowingPolicy`, которая отвечала бы за способ роста списка, а у класса `String` появилась бы зависимость `IInterningPolicy`, которая отвечала бы за политику интернирования! Подобные решения обеспечивают гибкость не там, где нужно, и могут подрывать инкапсуляцию.

Не для всех классов нужно выделять интерфейсы, и не все зависимости следует требовать извне в виде интерфейсов. Принцип инверсии зависимостей довольно четко дает понять, когда нужно зависимость инвертировать, а когда можно создавать зависимости непосредственно в месте их использования.

Слои

У лука есть слои, у торта есть слои,
и у Людоеда есть слои.

Шрек

Любое современное приложение разбито на слои, каждый из которых отвечает за определенный аспект поведения. На нижних уровнях находятся повторно используемые компоненты и инфраструктурный код, а слои более высокого уровня отвечают за логику приложения и пользовательский интерфейс (Ларман К. — 2006) (рис. 21.1).

Каждый слой отвечает за определенную область и использует сервисы нижележащих уровней для решения своих задач. Принцип единственной обязанности говорит, что каждый класс, модуль или слой должен решать лишь одну задачу. Это значит, что инфраструктурный код доступа к данным не должен содержать бизнес-логики, а бизнес-логика не должна знать о пользовательском интерфейсе.

Слои нижнего уровня не знают и не должны знать о слоях верхнего уровня. Это позволяет использовать низкоуровневые слои повторно, упрощает понимание и развитие каждого из них, а также ограничивает распространение изменений.

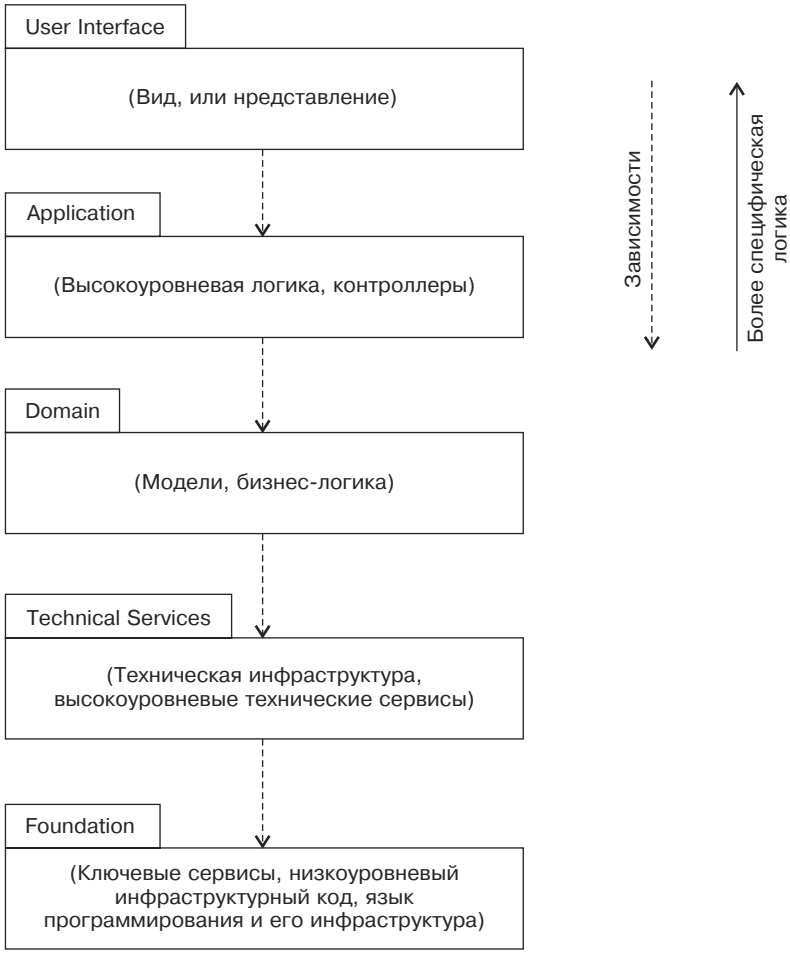


Рис. 21.1. Многоуровневая архитектура современного приложения

Чем выше слой, тем специфичнее он для каждого конкретного приложения. На нижних уровнях находятся инфраструктурный код, сервисы и ключевая бизнес-логика (core business logic). На верхних уровнях находятся высокоуровневая бизнес-логика, которая отличает одно приложение от другого, а также пользовательский интерфейс.

Иерархичность и «слоеность» присуща как программной системе в целом, так и отдельным крупным модулям. По мере роста сложности предметной области или за счет появления дополнительных деталей компоненты начинают дробиться на более мелкие составляющие, в основе которых будут лежать уже примитивные типы платформы и языка программирования.

Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые, в свою очередь, также могут быть разделены на подсистемы, вплоть до самого низкого уровня (*Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений. — 2010).

Классы более высокого уровня обычно контролируют процесс создания и передачу управления классам нижних уровней. В предыдущей главе мы остановились на следующем дизайне класса `LogImporter` (рис. 21.2). Класс `LogImporter` создает экземпляр одного из наследников класса `LogEntryParser` и просит его проанализировать записи, прочитанные из файла. В некоторых случаях класс верхнего уровня не знает, какой точно класс нижнего уровня использовать. В этом случае выделяется фабричный метод, который прячет процесс создания экземпляра нужной стратегии[1].

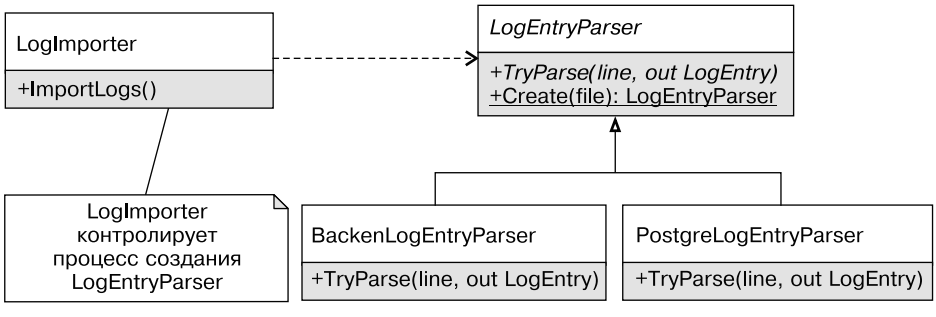


Рис. 21.2. Высокоуровневый дизайн класса LogImporter

Однако иногда возникает необходимость обеспечить передачу управления не только сверху вниз по слоям приложения, но и снизу вверх. Классы бизнес-логики могут потребовать что-то от слоя пользовательского интерфейса. Инфраструктурному коду может потребоваться информация, недоступная на его уровне абстракции. Иногда классы вынуждены зависеть от чего-то, что находится вне поля их зрения.

Наблюдатели

Классическим решением задачи связи нижних слоев с верхними является паттерн «Наблюдатель»[2]. Самый простой способ отвязать класс от внешних зависимо-

[1] Паттерн «Стратегия» был рассмотрен в главе 1, а паттерн «Фабричный метод» — в главе 10.

[2] Более подробно паттерн «Наблюдатель» был рассмотрен в главе 5.

стей — добавить в него событие, с помощью которого он будет уведомлять всех заинтересованных подписчиков об изменении состояния. При этом логически управление будет передано с нижнего уровня на верхний, однако знать о том, что будет происходить при вызове этого делегата, текущий класс не будет.

Хорошим примером использования наблюдателей для передачи управления снизу вверх по слоям приложения является паттерн MVC (Model — View — Controller, «Модель — представление — контроллер») (рис. 21.3). Каждый класс в этом паттерне отвечает за отдельный аспект: модель определяет бизнес-правила, представление отвечает за пользовательский интерфейс, а контроллер обрабатывает входные события от пользователя и отвечает за логику приложения.

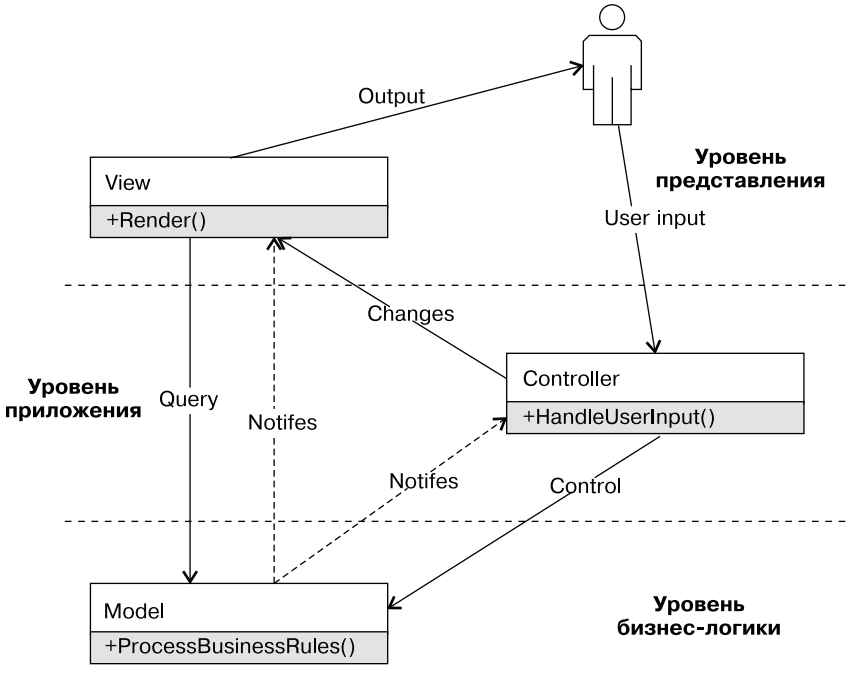


Рис. 21.3. Паттерн «Модель — представление — контроллер»[1]

[1] Распределение модели, контроллера и представления по слоям приложения может быть неоднозначным. Что считать верхним слоем, а что — нижним? Согласно общепринятой классификации, описанной Крэгом Ларманом в его книге «Применение UML и шаблонов проектирования», верхним слоем приложения является пользовательский интерфейс. Следом за ним идет логика приложения, которая связывает слой пользовательского интерфейса с доменными объектами. Именно этим объясняется порядок слоев на рисунке.

Классы моделей могут требовать чего-то от вышестоящих слоев. Но поскольку слои нижнего уровня не должны знать напрямую о классах верхнего уровня, то данное «общение» осуществляется с помощью наблюдателей. Так, модель не знает ничего о контроллере и представлении и общается с ними опосредованно, с помощью событий. Паттерн MVC несколько нарушает данное правило, и контроллер обычно знает о пользовательском интерфейсе больше, чем нам бы хотелось. Контроллер также обрабатывает входные данные от пользовательского интерфейса, что делает этот паттерн широко используемым в веб-приложениях, но практически неприменимым в других видах приложений, например WPF или Windows Forms.

Справиться с этими особенностями помогает другой паттерн, под названием MVP (Model — View — Presenter, «Модель — представление — презентер») (рис. 21.4).

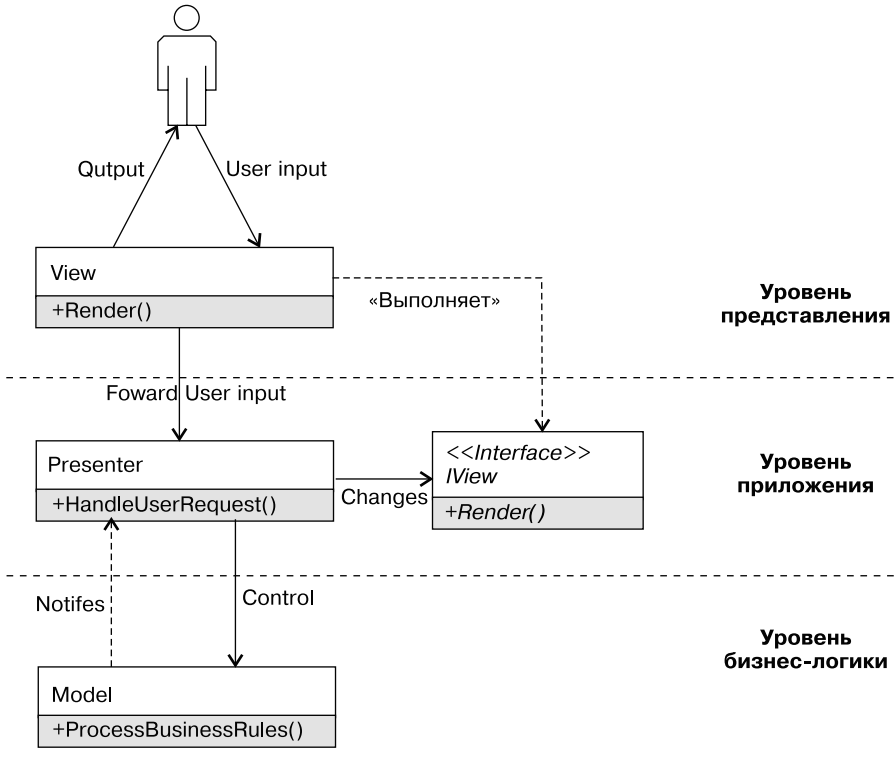


Рис. 21.4. Паттерн «Модель — представление — презентер»

Главное различие между паттернами MVP и MVC в том, что презентер не обрабатывает входные данные пользователя, а также в том, как он взаимодействует с представлением. Презентер знает лишь об абстрактном представлении (`IView`),

интерфейс которого объявляется на его уровне (например, в той же сборке), а реализуется уровнем представления.

Помимо двух перечисленных паттернов, существует еще один паттерн из семейства MVx — MVVM (Model — View — View Model, «Модель — представление — модель представления»). MVVM похож на MVP, однако вместо интерфейсов `IView` модель представления общается с верхним уровнем с помощью событий, спрятанных в инфраструктурном коде за технологией привязки данных (Data Binding).

Для чего нужен принцип инверсии зависимостей

Принцип инверсии зависимостей предназначен для устранения прямых связей между классами или модулями с зависимостями более высокого уровня (рис. 21.5).

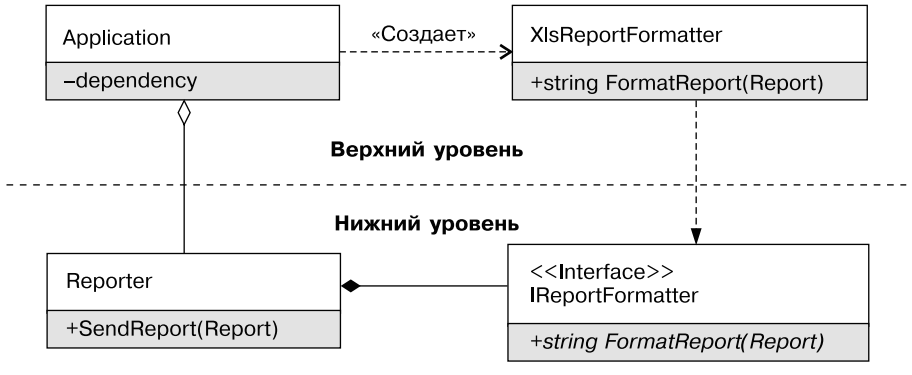


Рис. 21.5. Принцип инверсии зависимостей

Название принципа отражает нетипичность направления зависимостей: классы нижнего уровня определяют некоторый контракт, которому должны следовать классы верхнего уровня. Классы верхнего уровня вынуждены выступать в роли адаптеров и подстраиваться под протокол, определенный на уровне ниже. При этом вместо слаботипизированных наблюдателей текущий класс выделяет именованную зависимость в виде интерфейса и требует ее в своем конструкторе от вышестоящего кода (листинг 21.2).

Листинг 21.2. Использование DIP для организации взаимодействия разных уровней приложения

```
// Reporter.dll — Нижний уровень
public interface IReportFormatter
{
```

```
  string FormatReport(Report report);
}

public class Reporter
{
    private readonly IReportFormatter _formatter;

    public Reporter(IReportFormatter formatter)
    {
      _formatter = formatter;
    }

  public void SendReport(Report report)
  {
    var formattedReport = _formatter.FormatReport(report);
    SendFormattedReport(formattedReport);
  }

  private void SendFormattedReport(string formattedReport)
  {}
}

// Application.dll — Верхний уровень
class XlsFormatter : IReportFormatter
{
  public string FormatReport(Report report)
  {
    // Форматируем отчет для отображения его в Excel
  }
}

class Application
{
  public void Run()
  {
```

```
    var reporter = new Reporter(new XlsFormatter());
    reporter.SendReport(GenerateReport());
  }
}
```

Класс `Reporter` определяет свои зависимости в виде интерфейса `IReport-Formatter`, а класс `Application` адаптируется под его требования. Поэтому интерфейс `IReportFormatter` определяется на том же уровне, что и `Reporter`, а реализация этого интерфейса находится на более высоком уровне — уровне приложения. В результате прямые зависимости направлены сверху вниз — от приложения к модулю отчетов, а косвенные зависимости направлены в обратную сторону, что позволяет классу `Reporter` взаимодействовать во время исполнения с `XlsFormatter` полиморфным образом.

Остерегайтесь неправильного понимания DIP

Чрезмерное или необдуманное использование любого принципа проектирования может привести к переусложнению дизайна, и принцип инверсии зависимостей здесь не исключение. Распространение библиотек управления зависимостями — IoC-контейнеров[1] — может привести к выделению чрезмерного количества интерфейсов, что делает решение настолько слабосвязным, что разобраться в нем становится практически невозможно.

Причина неправильного использования принципа инверсии зависимостей кроется в недопонимании его целей, а также в описании этого принципа его автором — Робертом Мартином. Вот что он пишет в своей книге: *«DIP выражается простым эвристическим правилом: "Зависеть надо от абстракций". Оно гласит, что не должно быть зависимостей от конкретных классов; все связи в программе должны вести на абстрактный класс или интерфейс*.

---

[1] IoC-контейнеры (Inversion of Control Containers) представляют собой библиотеки для автоматического управления зависимостями. Их главным действующим лицом является контейнер, который может создавать экземпляры любого типа в приложении. Для этого контейнер конфигурируется в точке входа (Entry Point) приложения путем задания ассоциации между интерфейсами и их реализацией. Теперь если пользователь попросит у контейнера экземпляр некоторого типа, то контейнер самостоятельно найдет все его зависимости и передаст их требуемому классу через аргументы конструктора или установит соответствующие свойства. Подробнее о них говорится в книге Марка Сиимана Dependency Injectionin .NET.

- *Не должно быть переменных, в которых хранятся ссылки на конкретные классы.*
- *Не должно быть классов, производных от конкретных классов.*
- *Не должно быть методов, переопределяющих метод, реализованный в одном из базовых классов».*

И далее:

«Конечно, эта эвристика хотя бы раз, да нарушается в любой программе... В большинстве систем класс, описывающий строку, конкретный. Такой класс изменяется редко, поэтому в прямой зависимости от него нет никакой беды. Однако конкретные классы, являющиеся частью прикладной программы, которые пишем мы сами, в большинстве случаев изменчивы. Именно от таких конкретных классов мы и не хотим зависеть напрямую. Их изменчивость можно изолировать, скрыв их за абстрактным интерфейсом».

Буквальное следование принципу DIP по такому описанию чревато серьезными последствиями для дизайна. В последнее время легко столкнуться с проблемой чрезмерно абстрактных решений, когда интерфейсов слишком много и решение настолько гибкое и слабосвязное, что просто невозможно понять, кто за что отвечает и откуда начать изучение системы.

Тестируемость решения vs. подрыв инкапсуляции

Сегодня выделение интерфейсов часто объясняют необходимостью юнит-тестирования. Дескать, если класс создает свои зависимости самостоятельно, то как же мы сможем протестировать его в изоляции? Если же класс принимает все свои зависимости извне, да еще и в виде интерфейсов, то мы сможем «замокать»[1] все что угодно и добиться 100%-ного покрытия тестами!

---

[1] Существует несколько видов объектов-подделок, используемых в юнит-тестах, которые объединяются общим названием test doubles. Моки (mocks) и стабы (stubs) являются двумя наиболее распространенными видами подделок, которые предназначены для проверки поведения и эмуляции состояния. Далеко не все изоляционные фреймворки разделяют эти два понятия, поэтому многие из них (такие как Moq) любые подделки называют моками (mocks), что сделало этот термин применимым для описания любых подделок. Подробнее о разнице между стабами и моками можно узнать из моей статьи «Programming Stuff: Стабы и моки» (bit.ly/StubsVsMocks).

Проблема такого подхода в том, что он легко может подорвать инкапсуляцию текущего класса! Если зависимость находится на более низком уровне абстракции, то вызывающему коду она будет неинтересна. Пользовательскому интерфейсу неважно, как именно реализована модель и какая именно инфраструктура используется двумя слоями ниже. А слою бизнес-логики вряд ли интересны подробности реализации инфраструктурного слоя для удаленного взаимодействия с клиентами. С колокольни высокого уровня бывает просто невозможно решить, что же нужно передать в качестве `IPartitioningStrategy` нашей модели и почему вообще нас это должно интересовать.

Исходный смысл принципа инверсии зависимостей в том, чтобы классы нижнего уровня взаимодействовали с верхним уровнем косвенно, ничего не зная о нем. Но это совершенно не означает, что теперь классы верхнего уровня должны знать обо всех внутренних проблемах нижележащих слоев.

Существует несколько причин того, почему класс может требовать зависимости в виде аргумента конструктора или метода.

- Реализация зависимости находится на более высоком уровне (следование принципу DIP).
- Существует множество реализаций зависимости, и класс на этом уровне не может решать, какой из них выбрать (использование паттерна «Стратегия» и следование принципу «открыт/закрыт»).
- Поведение зависимости может быть завязано на внешнее окружение — файлы, базы данных, сокеты.

В первых двух случаях класс самостоятельно не может решить, какой конкретный тип зависимости использовать, поэтому просит верхний уровень помочь ему в этом. Последний случай более специфичен. Иногда мы можем осознанно пожертвовать инкапсуляцией в угоду тестируемости, особенно при наличии сложной логики в текущем классе и невозможности разделить его на более простые составляющие для тестирования их в изоляции.

Принцип инверсии зависимостей на практике

Вернемся к примеру с классом `LogEntryParser` и интерфейсом `IFileReader` (листинг 21.3).

Листинг 21.3. Наивная реализация класса LogEntryParser

```
class LogEntryParser
{
```

```
    private readonly IFileReader _fileReader;

    public LogEntryParser(IFileReader fileReader)
    {
        _fileReader = fileReader;
    }

    public IEnumerable<LogEntry> ParseLogEntries()
    {
        // Читает файл с помощью _fileReader
        // и разбирает прочитанные строки
        yield break;
    }
}
```

Код выглядит весьма неплохо, но отвечает ли он принципам проектирования?

- Класс нарушает SRP. Данный парсер умеет разбирать множество строк, что является дополнительной ответственностью. Поскольку одной записи `LogEntry` могут соответствовать несколько строк лог-файла (например, для исключений), такой подход может показаться более разумным. Я же предпочитаю, чтобы интерфейс класса был максимально простым.
- Класс нарушает ISP[1]. Самой простой зависимостью класса `LogEntryParser` является строка (`System.String`), и именно ее должен принимать метод `Parse` в качестве аргумента метода.
- Класс нарушает DIP. Класс `LogEntryParser` использует `IFileReader`, интерфейс, который нельзя назвать высокоуровневой зависимостью. Первое, что должно приходить на ум при попытке абстрагироваться от ввода/вывода, — это тип `System.Stream`. Если он по какой-то причине не подходит, то вместо интерфейса `IFileReader` нужно выделять интерфейс `IRawLogEntryStream`. `IRawLogEntryStream` позволит работать с любым источником сырых записей, будь это файл, сетевой поток или что-то еще.

---

[1] Здесь происходит очень тонкое нарушение принципа ISP. С точки зрения этого принципа класс должен обладать самыми простыми из возможных зависимостей. Самой простой и необходимой зависимостью класса LogEntryParser является строка, которую необходимо проанализировать, а не зависимости, такие как IFileReader.

Давайте посмотрим, как я изменил бы этот дизайн.

Поведение класса `LogEntryParser` неточно отражало его название. Если нам нужен класс или интерфейс, который отвечает за чтение и разбор записей, то ему нужно подходящее имя, например `LogEntryReader` (листинг 21.4).

Листинг 21.4. Реализация класса LogEntryReader

```
class LogEntryReader : IDisposable
{
    private readonly Stream _stream;
    private readonly LogEntryParser _parser;

    public LogEntryReader(Stream stream, LogEntryParser parser)
    {
        _stream = stream;
        _parser = parser;
    }

    public void Dispose()
    {
        _stream.Close();
    }

    public IEnumerable<LogEntry> Read()
    {
        using (var sr = new StreamReader(_stream))
        {
            string line = null;
            while ((line = sr.ReadLine()) != null)
            {
                LogEntry logEntry;
                if (_parser.TryParse(line, out logEntry))
                {
                    yield return logEntry;
                }
            }
        }
    }
```

Теперь ответственности классов разделены более четко. Класс `LogEntryReader` отвечает за чтение прочитанных записей, при этом за разбор записей отвечают класс `LogEntryParser` и его наследники.

Мы можем пойти еще дальше и создать фасадный фабричный метод, который будет создавать экземпляр `LogEntryReader` для чтения конкретного файла (листинг 21.5).

Листинг 21.5. Фабричный метод LogEntryReader.FromFile

```
public static LogEntryReader FromFile(string fileName)
{
    var fs = new FileStream(fileName, FileMode.Open);
    var parser = LogEntryParser.Create(fileName);

    return new LogEntryReader(fs, parser);
}
```

Данный фабричный метод упрощает импорт конкретных лог-файлов, поскольку прячет процесс открытия файла и создание конкретного парсера[1]. Наличие конструктора делает решение гибким и тестируемым, а фабричный метод упрощает использование класса большинством клиентов.

Нужно ли выделять интерфейс `ILogEntryReader`? Ответить на этот вопрос сложно. Сам класс `LogEntryReader` не является автономным и будет использоваться для реализации класса `LogImporter` (листинг 21.6).

Листинг 21.6. Реализация класса LogImporter

```
public class LogImporter : IDisposable
{
    private readonly LogEntryReader _logEntryReader;

    public LogImporter(string logFile)
    {
        _logEntryReader = LogEntryReader.FromFile(logFile);
    }
```

[1] Фабричный метод LogEntryParser.Create приведен в главе 18. Парсеру нужно имя файла, чтобы определить, какой конкретно экземпляр парсера нужно создавать. В зависимости от имени файла будет использован парсер логов разных приложений.

```
    public void Dispose()
    {
        _logEntryReader.Dispose();
    }

    public void ImportLogs()
    {
        foreach (var logEntry in _logEntryReader.Read())
        {
            LogSaver.SaveEntry(logEntry);
        }
    }
}
```

Класс `LogImporter` является довольно простым посредником, который использует классы `LogEntryReader` и `LogSaver` и практически не содержит бизнес-логики. На данном этапе может быть достаточно добавить несколько интеграционных тестов, которые проверят корректность поведения класса в реальном окружении. Если же логика класса `LogImporter` начнет усложняться, то выделение интерфейсов `ILogEntryReader` и `ILogEntrySaver` станет оправданным (листинг 21.7).

Листинг 21.7. Выделение интерфейсов ILogEntryReader и ILogEntrySaver

```
public class LogImporter
{
    private readonly ILogEntryReader _reader;
    private readonly ILogEntrySaver _saver;

    public LogImporter(ILogEntryReader reader, ILogEntrySaver saver)
    {
        _reader = reader;
        _saver = saver;
    }

    public void ImportLogs()
    {
```

```
        foreach (var logEntry in _reader.Read())
        {
            _saver.SaveEntry(logEntry);
        }
    }
    // Много другой логики!
}
```

ПРИМЕЧАНИЕ

Обратите внимание на то, что я сделал акцент на выделении интерфейсов ILogEntryReader и ILogEntrySaver лишь в случае необходимости. Если класс LogImporter будет содержать лишь код, приведенный в листинге 21.7, то никакой необходимости в интерфейсах нет! Разделения обязанностей между классами LogImporter, LogSaver и LogEntryReader уже достаточно для успешной эволюции приложения.

Примеры нарушения принципа инверсии зависимостей

Нарушение принципа инверсии зависимостей происходит в следующих случаях.

- Низкоуровневые классы напрямую общаются с высокоуровневыми классами — модели знают о пользовательском интерфейсе или инфраструктурный код знает о бизнес-логике.
- Классы принимают слишком низкоуровневые интерфейсы, такие как `IFileStream`, что может привести к подрыву инкапсуляции и излишнему увеличению сложности.

Выводы

Простое эвристическое правило — зависеть нужно от абстракций — на деле оказалось более сложным, чем могло показаться изначально.

Принцип инверсии зависимостей не сводится лишь к выделению интерфейсов и передаче их через конструктор. DIP объясняет, для чего нужно это делать. Классы имеют право контролировать свои детали реализации, но некоторые аспекты находятся за пределами их компетенции. Чтобы не завязываться на классы верхнего уровня, класс может объявить некоторый интерфейс и потребовать его экзем-

пляр через аргументы конструктора. Таким образом мы можем инвертировать зависимости и позволить классам нижних уровней взаимодействовать с другими частями системы, ничего конкретного о них не зная.

DI vs. DIP vs. IoC

Существуют три схожих понятия, связанных с передачей зависимостей, в каждом из которых есть слово «инверсия» (inversion) или «зависимость» (dependency):

- IoC — Inversion of Control (инверсия управления);
- DI — Dependency Injection (внедрение зависимостей);
- DIP — Dependency Inversion Principle (принцип инверсии зависимостей).

Подливает масло в огонь рассогласованность использования этих терминов. Так, например, контейнеры иногда называют DI-контейнерами, а иногда IoC-контейнерами. Большинство разработчиков не различают DI и DIP, хотя за каждой из этих аббревиатур скрываются разные понятия.

Inversion of Control

Инверсия управления (Inversion of Control, IoC) — это довольно общее понятие, которое отличает библиотеку от фреймворка. Классическая модель подразумевает, что вызывающий код контролирует внешнее окружение, время и порядок вызова библиотечных методов. Однако в случае фреймворка обязанности меняются местами: фреймворк предоставляет некоторые точки расширения, через которые он вызывает определенные методы пользовательского кода.

Простой метод обратного вызова или любая другая форма паттерна «Наблюдатель» является примером инверсии управления. Зная значение понятия IoC, мы понимаем, что такое понятие, как IoC-контейнер, лишено смысла, если только данный контейнер не предназначен для упрощения создания фрейморков (рис. 21.6).

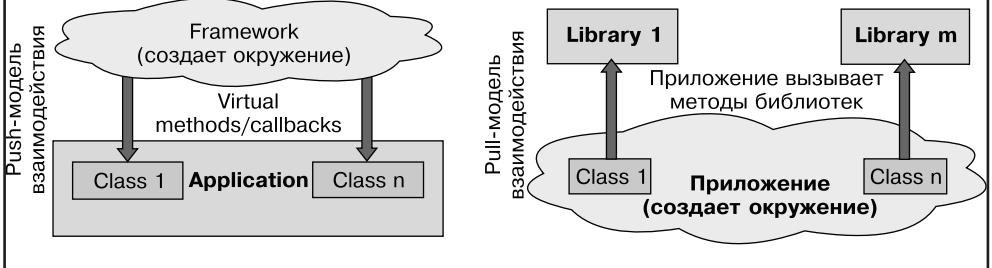


Рис. 21.6. Разница между фреймворком и библиотекой

Dependency Injection

Внедрение зависимостей (Dependency Injection, DI) — это механизм передачи классу его зависимостей. Существует несколько конкретных видов или паттернов

внедрения зависимостей: внедрение зависимости через конструктор (Constructor Injection), через метод (Method Injection) и через свойство (Property Injection).

Далее приведены примеры использования каждого вида внедрения зависимостей (листинг 21.8).

Листинг 21.8. Виды внедрения зависимостей

```
class ReportProcessor
{
  private readonly IReportSender _reportSender;

  // Constuctor Injection: передача обязательной зависимости
  public ReportProcessor(IReportSender reportSender)
  {
    _reportSender = reportSender;
    Logger = LogManager.DefaultLogger;
  }

  // Method Injection: передача обязательных зависимостей метода
  public void SendReport(Report report, IReportFormatter formatter)
  {
     Logger.Info("Sending report...");
     var formattedReport = formatter.Format(report);
     _reportSender.SendReport(formattedReport);
     Logger.Info("Report has been sent");
  }

  // Property Injection: установка необязательных
  // инфраструктурных зависимостей
  public ILogger Logger {get; set;}
}
```

Разные виды внедрения зависимостей предназначены для решения разных задач. Через конструктор передаются обязательные зависимости класса, без которых работа класса невозможна (в нашем примере `IReportSender` — обязательная зависимость класса `ReportProcessor`). Через метод передаются зависимости, которые нужны лишь одному методу, а не всем методам класса (`IReportFormatter`

необходим только методу отправки отчета, а не классу `ReportProcessor` целиком). Через свойства должны устанавливаться лишь необязательные зависимости (обычно инфраструктурные), для которых существует значение по умолчанию (свойство `Logger` содержит разумное значение по умолчанию, но может быть заменено позднее).

Очень важно понимать, что DI-паттерны не говорят о том, как должна выглядеть зависимость, к какому уровню она относится, сколько их должно быть и т. п. Это лишь инструмент передачи зависимостей от одного класса другому. Ни больше ни меньше!

Dependency Inversion Principle

Принцип инверсии зависимости (Dependency Inversion Principle. DIP) говорит о том, какого вида зависимости необходимо передавать классу извне, а какие зависимости класс должен создавать самостоятельно. Важно, чтобы зависимости класса были понятны и важны вызывающему коду. Зависимости класса должны располагаться на текущем или более высоком уровне абстракции. Другими словами, не любой класс, которого требует интерфейс в конструкторе, следует принципу инверсии зависимостей (листинг 21.9).

Листинг 21.9. Пример нарушения DIP при наличии DI

```
class ReportProcessor
{
    private readonly ISocket _socket;
    public ReportProcessor(ISocket socket)
    {
        _socket = socket;
    }

    public void SendReport(Report report, IStringBuilder stringBuilder)
    {
        stringBuilder.AppendFormat(CreateHeader(report));
        stringBuilder.AppendFormat(CreateBody(report));
        stringBuilder.AppendFormat(CreateFooter(report));
        _socket.Connect();
        _socket.Send(ConvertToByteArray(stringBuilder));
    }
}
```

Класс `ReportProcessor` все еще принимает абстракцию в аргументах конструктора — `ISocket`, но эта абстракция находится на несколько уровней ниже уровня формирования и отправки отчетов. Аналогично дела обстоят и с аргументом метода `SendReport`: абстракция `IStringBuilder` не соответствует принципу инверсии зависимостей, поскольку оперирует более низкоуровневыми понятиями, чем требуется. На этом уровне нужно оперировать не строками, а отчетами. В результате в данном примере используется внедрение зависимостей (DI), но данный код не следует принципу инверсии зависимостей (DIP).

Подведем итоги. Инверсия управления (IoC) говорит об изменении потока исполнения, присуща фреймворкам и функциям обратного вызова и не имеет никакого отношения к управлению зависимостями. Передача зависимостей (DI) — это инструмент передачи классу его зависимости через конструктор, метод или свойство. Принцип инверсии зависимостей (DIP) — это принцип проектирования, который говорит, что классы должны зависеть от высокоуровневых абстракций.

Дополнительные ссылки

- DI-паттерны: Constructor Injection (http://bit.ly/CustructorInjection).
- DI-паттерны: Method Injection (bit.ly/MethodInjection).
- DI-паттерны: Property Injection (bit.ly/PropertyInjection).
- DIP in the Wild (http://martinfowler.com/articles/dipInTheWild.html).
- Inversion of Control Containers and the Dependency Injection pattern (http://martinfowler.com/articles/injection.html).
- Фреймворки, библиотеки и зависимости (http://bit.ly/FrameworkDependencies).

Глава 22

Размышления о принципах проектирования

Век живи — век учись! И ты наконец достигнешь того,
что, подобно мудрецу, будешь иметь право сказать,
что ничего не знаешь.

Козьма Прутков

Для чего понадобилось кому-то выдумывать разные паттерны проектирования, принципы и методики разработки? Разве не было бы проще научить разработчиков хорошему проектированию? Или почему тогда не формализовать процесс разработки и не ввести четкие количественные метрики дизайна[1], которые бы говорили о качестве решения?

---

[1] Сложности перевода. В русскоязычной литературе нет однозначного перевода термина software design, который бы точно передавал смысл оригинала. Когда понятие design описывает процесс, то его принято переводить как «проектирование». Но все становится сложнее, когда понятие design описывает результаты этого процесса. Что мы получаем в результате процесса проектирования? Проект? Дизайн? Первый вариант не подходит, поскольку термин «проект» в области программирования означает проектную деятельность. Второй вариант тоже неидеален, поскольку его легко спутать с дизайном пользовательских интерфейсов. Меньшим злом, с моей точки зрения, является использование разных вариантов перевода для описания разных значений этого термина. Поэтому в данной книге будет использоваться термин «проектирование» для описания процесса проектирования и термин «дизайн» для описания артефактов процесса проектирования.

«Правильный дизайн» — это святой Грааль молодых разработчиков и молодых менеджеров. И те и другие мечтают найти ответ на главный вопрос разработки ПО: как добиться качественного дизайна в сжатые сроки и приложив минимум усилий? Со временем и молодой разработчик, и молодой менеджер приходят к пониманию того, что это невозможно. Невозможно найти идеальный абстрактный дизайн, поскольку слова «идеальный» и «абстрактный» противоречат друг другу. Проектирование — это постоянный поиск компромисса между противоречивыми требованиями: производительностью и читабельностью, простотой и расширяемостью, тестируемостью и цельностью решения. Даже если учитывать, что разработчик всегда решает правильную задачу, а не борется с ветряными мельницами, нельзя «абстрактно» сказать, какие характеристики дизайна являются ключевыми здесь и сейчас.

Существуют формальные критерии, которые описывают качество кода или дизайна: цикломатическая сложность методов, глубина иерархии наследования, число входящих и исходящих связей класса или модуля, число строк метода, в конце концов. Эти количественные показатели полезны, однако попадание их в заданные границы является необходимым, но недостаточным условием хорошего дизайна. Если классы разбиты неумело, а важные абстракции предметной области не выявлены, то какими бы количественными характеристиками ни обладал дизайн, он никогда не будет хорошим.

Помимо формальных критериев, есть универсальные понятия дизайна — связанность (coupling) и связность (cohesion)[1]. Качественный дизайн обладает слабой связанностью (low coupling) и сильной связностью (high cohesion). Это значит, что программный компонент имеет небольшое число внешних связей и отвечает за решение близких по смыслу задач. Эти свойства полезны, но слишком неформальны.

Между формальными и неформальными критериями находятся принципы проектирования — набор правил, на которые опираются опытные проектировщики. Цель принципов — простыми словами описать, «что такое хорошо, а что такое плохо» в вопросах проектирования. Идеальный класс должен иметь лишь одну причину для изменения, обладать минимальным интерфейсом, правильно реализовывать наследование и предотвращать каскадные изменения в коде при изменении требований.

---

[1] Это еще одна пара терминов, перевод которых на русский язык вызывает сложности. Существует несколько популярных вариантов перевода этих понятий, но ни один из них не вызывает четких ассоциаций в голове у читателя. Чтобы устранить эту неоднозначность, при использовании этих понятий в тексте книги я всегда буду приводить оригинальное значение.

Бертран Мейер в своей книге Agile!: The Good, The Hype, and The Ugly[1] дает довольное четкое определение того, что такое принцип проектирования: «Принцип — это методологическое правило, которое выражает общий взгляд на разработку ПО. Хороший принцип является одновременно *абстрактным* и *опровергаемым* (falsifiable). Абстрактность отличает принцип от практик, а опровергаемость отличает принцип от банальности (platitude). Абстрактность означает, что принцип должен описывать универсальное правило, а не конкретную практику. Опровергаемость означает, что у разумного человека должна быть возможность не согласиться с принципом. Если никто в здравом уме не будет оспаривать предложенный принцип, то это правило будет полезным, но неинтересным. Чтобы правило являлось принципом — независимо от вашего мнения, — вы должны предполагать наличие людей, придерживающихся противоположной точки зрения».

Использование принципов проектирования

Разработчик за свою профессиональную карьеру проходит несколько стадий владения таким инструментом, как принципы проектирования.

- **На первой стадии** молодой разработчик еще не дорос до абстрактных принципов, вместо этого он ищет набор конкретных практик, которые позволят ему получить качественное решение: «Я хочу узнать набор шагов, четкое следование которым приведет меня к нужному результату!» На этом этапе разработчик хорошо копирует чужое решение и сталкивается с серьезными трудностями, если описание слишком абстрактное или не абсолютно подходит к его случаю.
- **На второй стадии** разработчик начинает понимать, что лежит в основе конкретной практики и для чего нужны принципы проектирования: «Ага, этот класс нарушает принцип единой обязанности, поскольку он "ходит" в базу и содержит бизнес-логику! Получается, что у него есть две четкие причины для изменения!»

  Несмотря на возросший уровень мастерства, именно на этом этапе разработчик наиболее часто использует принципы или паттерны не по назначению. Легко доказать, что конкретный класс нарушает данный принцип, но на этом этапе развития не всегда очевидно, когда это нарушение оправданно, а когда — нет.
- **На третьей стадии** у разработчика (или уже скорее архитектора) развивается чутье и появляется довольно четкое понимание того, какую проблему призван решить конкретный принцип проектирования. Поскольку по своему определению

---

[1] *Meyer B.*, Agile!: The Good, The Hype, and The Ugly. — Springer, 2014.

принцип не является однозначным, опытный разработчик начинает понимать, когда нарушение оправданно, когда с ним можно жить, а когда пришло время браться за исправление дизайна.

При достаточном опыте разработчик начинает использовать принципы на подсознательном уровне. Разделение ответственностей классов и методов происходит на этапе их разработки, а не во время рецензирования кода. На этом этапе принципы не управляют дизайном приложения, а начинают играть скорее коммуникативную роль для общения с младшими коллегами: «Смотри, у тебя класс делает слишком многое, значит, он нарушает принцип единой обязанности! А у этого класса есть два вида клиентов, каждый из которых использует этот класс по-своему, значит, он нарушает принцип разделения интерфейсов!»

ПРИМЕЧАНИЕ

В боевых искусствах принято выделять три стадии мастерства: сю, ха и ри (Shu, Ha, Ri — https://en.wikipedia.org/wiki/Shuhari). На первой ступени находится ученик, который лишь повторяет движения за мастером. На второй ступени ученик начинает освобождаться от правил и сам принимает решение, когда им следовать, а когда —нет. На третьей стадии правила пропадают, ученик становится мастером и может сам эти правила создавать. Эта же модель, но под несколько иным соусом, появилась и в американской культуре под названием «модель Дрейфуса» — http://en.wikipedia.org/wiki/Dreyfus_model_of_skill_acquisition.

Правильное использование принципов проектирования

Применение любого принципа проектирования имеет свою цену. Дробление класса на более мелкие составляющие, чтобы он отвечал принципу единственной обязанности, может привести к «размазыванию» логики по нескольким классам. А это может привести к уменьшению связности (lowcohesion) и читабельности, а иногда и к падению производительности.

Нарушение принципа «открыт/закрыт» может быть оправдано вопросами обратной совместимости. Мы можем игнорировать принцип замещения Лисков, поскольку наследование не всегда определяет отношение «ЯВЛЯЕТСЯ». Интерфейс может быть «жирным» из-за обратной совместимости или удобства использования. А инверсия зависимостей легко может подорвать инкапсуляцию и привести к чрезмерному числу уровней косвенности.

Чтобы эффективно использовать принципы, нужно знать, какую проблему они на самом деле решают и является ли она для вас актуальной. Обычно важность прин-

ципов возрастает с увеличением сложности или при увеличении стоимости внесения изменений. Ключевая бизнес-логика приложения является одним из примеров того, что должно быть изолировано от остального мира. Любые публично доступные классы должны иметь минимум дополнительных связей с внешним миром и легко позволять делать простые вещи. Классы, которые находятся на стыке модулей, должны быть продуманы лучше остальных.

Ключом для получения хорошего дизайна и эффективного использования принципов проектирования является итеративный подход к проектированию. На ранних этапах мы мало что понимаем в предметной области, поэтому все, что мы можем сделать, — это разбить модули по ролям: инфраструктуру отдельно, логику отдельно. При этом автономность классов и минимизация побочных эффектов делает решение проще. Когда класс обладает минимальным числом зависимостей, его влияние на систему и влияние системы на него строго ограничены.

По мере развития приложения улучшается понимание предметной области. Одновременно с этим приложение обрастает заплатками и неуклюжими решениями. Накапливается технический долг, который усложняет добавление новых возможностей, а исправление одной ошибки приводит к появлению двух других. Чтобы не доводить систему до такого состояния, лучше постоянно думать о дизайне и улучшать его по ходу развития. Перед реализацией новой возможности или исправлением ошибки я стараюсь ответить на такой вопрос: отвечает ли текущий дизайн моему новому пониманию задачи, или требуется его переработка?

Главной движущей силой изменений в конечном счете является возрастающая сложность. Как только я перестаю удерживать в голове текущее решение или простое изменение требований приводит к изменению целой группы классов, значит, пришло время подумать над изменением дизайна: «Класс стал слишком сложным, он нарушает принцип единой обязанности, пришло время разбить его на два. Информация об иерархии наследования распространилась по всему модулю. Наверное, стоит перенести часть логики в базовый класс, а саму иерархию спрятать за фабрикой. Число зависимостей класса стало слишком большим, значит, нужно пересмотреть обязанности класса или объединить низкоуровневые зависимости».

При этом я всегда проверяю, не приведут ли меня принципы и паттерны в мир ненужной сложности (overengineering): стал ли мой дизайн после внесения изменений проще? Не решаю ли я проблему, которой на самом деле не существует? Не попал ли я в сети преждевременного обобщения (premature generalization)? Иногда приходится выполнить несколько итераций, прежде чем удается найти разумное решение задачи.

Антипринципы проектирования

Популярность принципов проектирования легко может сыграть с вами и вашей командой злую шутку. Чрезмерная любовь к принципам и паттернам может проявиться в виде оверинжиниринга и чрезмерной сложности. Но, зная об этом, мы можем подготовиться и предугадать, как именно будет проявляться чрезмерная любовь к принципам в коде приложения.

- **Anti-SRP** — принцип размытой обязанности. Классы разбиты на множество мелких классов, в результате чего логика «размазывается» по нескольким классам/модулям.
- **Anti-OCP** — принцип фабрики фабрик. Дизайн является слишком обобщенным, его можно расширять почти неограниченно, выделяется слишком большое число уровней абстракции.
- **Anti-LCP** — принцип непонятного наследования. Принцип проявляется либо в чрезмерном количестве наследования, либо в его полном отсутствии в зависимости от опыта и взглядов местного главного архитектора.
- **Anti-ISP** — принцип тысячи интерфейсов. Интерфейсы классов разбиваются на слишком большое число составляющих, что делает их неудобными для использования всеми клиентами.
- **Anti-DIP** — принцип инверсии сознания, или DI головного мозга. Интерфейсы выделяются для каждого класса и пачками передаются через конструкторы. Понять, где находится логика, становится практически невозможно.

Заключение

Если оглянуться на пройденный нами путь, то будет довольно легко найти связь между принципами и паттернами проектирования. Принципы проектирования более фундаментальны и лежат в основе практически любого решения, которое принимается во время проектирования. Паттерны проектирования описывают каркас решения определенных проблем и могут меняться от одной предметной области к другой. Эти инструменты находятся на разных уровнях абстракции, но они преследуют единую цель: получить дизайн, который будет легко понимать, развивать и тестировать.

Те или иные принципы проектирования лежат в основе любого паттерна. Паттерны проектирования предназначены для борьбы со сложностью (SRP), ограничения распространения изменений и получения расширяемого решения (OCP). В основе многих из них лежит наследование, корректное использование которого невозможно без LSP. Применение паттернов приводит к слабосвязанному дизайну, в основе которого лежит принцип наименьшего знания (ISP), а четкое разграничение ответственности приводит к минимизации связей между слоями приложения (DIP).

Принцип единственной обязанности лежит в основе многих паттернов проектирования. Суть большинства паттернов заключается в изоляции сложного аспекта поведения в отдельном классе или группе классов. **«Фабрика»** позволяет изолировать процесс создания объекта, когда смешивание этой обязанности с логикой самого класса становится неприемлемым. Главный смысл использования **«Декоратора»** заключается в выделении отдельного аспекта поведения в отдельный класс, что нужно лишь тогда, когда сложность этого аспекта является очень высокой. Паттерн **«Итератор»** позволяет вынести логику обхода коллекции в отдельный класс, а основная роль **«Адаптера»** заключается лишь в адаптации интерфейса класса без изменения поведения.

Принцип «открыт/закрыт» отражает одну из самых важных задач паттернов — изоляцию изменений. Многие паттерны проектирования представляют собой своеобразные шлюзы, которые затормозят нахлынувшую волну изменений. **«Фасад»** позволяет отделить клиентов от внутреннего устройства внешних библиотек, а **«Адаптер»** и **«Посредник»** позволяют ограничить изменения лишь несколькими классами. Очень многие паттерны проектирования отражают второе свойство принципа «открыт/закрыт» — расширяемость решения. **«Стратегия»** позволяет переходить от одной реализации алгоритма к другой без изменения клиентов. **«Фабрики»** прячут процесс создания объектов или целые иерархии наследования, позволяя гибко расширять поведение системы. **«Декоратор»** дает возможность нанизывать поведение объектов, а **«Посетитель»** позволяет добавлять операции в закрытые иерархии типов.

Принцип подстановки Лисков лежит в основе корректной реализации иерархий наследования и вместе с **Шаблонным методом** является инструментом получения расширяемого и сопровождаемого решения.

Принципы разделения интерфейсов и **инверсии зависимостей** не лежат в основе паттернов проектирования, тут роль скорее обратная: применение паттернов проектирования позволяет следовать этим принципам. Паттерны **«Посредник»** и **«Фасад»** избавляют классы от лишней информации друг о друге, а использование **«Стратегии»** и **«Наблюдателя»** позволяет следовать принципу инверсии зависимостей.

Разумное следование принципам проектирования и умелое применение паттернов проектирования не являются гарантией успеха проекта. Но они хотя бы подскажут направление, куда нужно двигаться, чтобы получить хороший дизайн.

Источники информации

Книги о дизайне и ООП

1. *Гамма Э. и др.* Приемы объектно-ориентированного проектирования. — СПб.: Питер, 1994. Это та самая знаменитая книга «банды четырех», после выхода которой началось стремительное развитие идеи шаблонов проектирования в мире разработки ПО. После ее выхода идея шаблонов начала распространяться и развиваться, и сегодня идея шаблонов применяется не только в контексте проектирования, но и практически в каждой области разработки программного обеспечения.
2. *Фриман Э.* Паттерны проектирования. — СПб.: Питер, 2003. Лучший учебник по ООП, паттернам и принципам проектирования. Несмотря на «желтоватое» название (Head-First Design Patterns), книга действительно очень хороша: темы рассмотрены в игровой форме, и при этом акцент делается не на конкретных паттернах, а на принципах проектирования и их связи с паттернами.
3. *Мейер Б.* Объектно-ориентированное конструирование программных систем. — СПб.: Интернет-университет, 2005. Книга, которую многие из-за ее фундаментальности в области объектно-ориентированного программирования сравнивают с творением Дональда Кнута (причем совершенно без преувеличения) в области алгоритмов и структур данных. Эта книга является наиболее фундаментальным трудом по объектной парадигме, когда-либо выходившим на русском или английском языках. Книга охватывает широкий круг вопросов, начиная от вопросов наследования, инкапсуляции, модульности, повторного использования и заканчивая автоматическим управлением

памятью, шаблонами проектирования и проектированием по контракту (которое только спустя два десятилетия начинает набирать обороты в мейнстрим-языках и технологиях).

4. *Ларман К.* Применение UML 2.0 и шаблонов проектирования. 3-е изд. — М.: Вильямс, 2004. Еще один учебник по разработке ПО с уклоном в объектно-ориентированные методы: анализ и проектирование.

5. *Буч Г.* Объектно-ориентированный анализ и проектирование. — М.: Вильямс, 2003. Еще одна классическая книга по объектно-ориентированному программированию, но, в отличие от книги Бертрана Мейера, имеет менее формальный и более описательный характер. В книге потрясающе описаны проблема сложности ПО, роли абстракции и иерархии, наиболее популярные на сегодняшний день методологии разработки и многое другое.

6. *Эванс Э.* Предметно-ориентированное проектирование. — М.: Вильямс, 2003. Многие ассоциируют DDD (Domain-Driven Design) со всякими новомодными штучками наподобие Event Sourcing, CQRS или на худой конец с рядом enterprise-паттернов и обязательным выделением сборок Domain и DataAccess. На самом деле основная цель любого вида проектирования заключается в моделировании предметной области, и любая книга по ООП или ФП говорит о том, как создавать модели, наиболее близкие к реальному миру. Классическая книга Эрика Эванса дала толчок развитию этой новой области проектирования. Прочесть ее обязательно.

7. *Мартин Р.* Принципы, паттерны и методики гибкой разработки на языке C#. — СПб.: Символ-Плюс, 2011. Многие разработчики считают книгу Роберта Мартина чуть ли не библией разработки, но я с этим категорически не согласен. В отличие от других книг из этого списка, книга Роберта довольно поверхностна, эмоциональна и содержит большое количество спорных моментов. Я не советовал бы читать эту книгу неопытным разработчикам, поскольку она может существенно исказить понимание вопросов проектирования. Более подробный критический анализ книги Мартина можно найти в моей статье «Критика книги Боба Мартина “Принципы, паттерны и методики гибкой разработки на языке C#”» в блоге по адресу http://sergeyteplyakov.blogspot.ru/2013/12/about-agile-principles-patterns-and.html.

8. *Beck K.* Test-Driven Development: By Example. — 2002. Возрастная книга, но если вы хотите понять, что же такое TDD с точки зрения его автора, то лучшего источника не найти. Книга легко читается, в ней довольно много практических примеров, и она не слишком устарела, несмотря на свой возраст.

9. *Freeman S., Pryce N.* Growing Object-Oriented Software Guided by Tests. — 2009. Одна из лучших книг для изучения TDD вообще и тестирования в частности.

Отлично описаны процесс «выращивания» системы, ее эволюция и адаптация под новые требования. Хорошо изложена связь тестов и дизайна.

10. *Фаулер М.* Рефакторинг. Улучшение существующего кода. — СПб.: Символ-Плюс, 1999. Классическая книга Мартина Фаулера, с которой началось распространение практики рефакторинга в массах. Книга состоит из двух частей: потрясающей вводной части, в которой рассматриваются вопросы качества кода и дизайна, и части с перечнем рефакторингов. И хотя вторая часть уже немного неактуальна, первая половина книги очень полезна.

11. *Osherove R.* The Art of Unit Testing: with examples in C#. — 2014. Паттерны не ограничиваются классическими, описанными в книге «банды четырех». Паттерны повсюду: есть архитектурные паттерны, паттерны проектирования, DI-паттерны, DDD-паттерны, паттерны рефакторинга, даже паттерны поведения. Точно так же существуют паттерны разработки юнит-тестов. Есть типовые подходы к организации тестового кода для решения тех или иных задач. Лучшим источником по этой теме является фундаментальный труд xUnit Test Patterns: Refactoring Test Code, а книга Роя является отличным изданием по этой же тематике в контексте платформы .NET.

12. *Физерс М.* Эффективная работа с унаследованным кодом. — М.: Вильямс, 2004. Лучшая книга на рынке с перечнем подходов к работе с унаследованным кодом. Здесь описаны вопросы качества кода, подходы к юнит-тестированию и проблемы модификации кода без тестов.

13. *Cwalina K., Abrams B.* Framework Design Guidelines. — 2009. Разработка качественных систем является весьма сложной задачей, а разработка качественных библиотек (особенно фреймворков) — поистине вершина мастерства архитекторов и разработчиков. Сложность здесь кроется в специфике принимаемых решений, ведь акцент серьезно смещается в сторону простоты и удобства использования, расширяемости и надежности. И хотя именно тема разработки библиотек является центральной, книга будет также невероятно полезна и простым разработчикам, ведь знание ключевых идиом языка является совершенно необходимым, когда команда смотрит хотя бы немного дальше своего носа и заботится не только о написании кода, но и о его последующем сопровождении. Кроме того, книга часто выступает арбитром во многих спорах, касающихся идиом именования, обработки исключений, проектирования собственных классов или использования других идиом языка C#. А поскольку такие дискуссии происходят с завидным постоянством, подобный козырь лишним точно не будет.

14. *Seeman M.* Dependency Injection in .NET, 2011. Лучшая книга об управлении зависимостями и о DI-паттернах. Книга очень полезна для корректного применения

популярных ныне принципов инверсии управления в своих приложениях. Рассмотрены ключевые паттерны, такие как Constructor Injection, Method Injection и др., а также негативные стороны использования контейнеров в коде приложения. Однозначно Must Read для каждого .NET-программиста!

15. *Petrisek T.* Real-World Functional Programming: With Examples in F# and C#. — 2010. Очень полезная книга, если вы хотите на примерах рассмотреть разницу функционального и объектно-ориентированного подходов. Книга Томаса Петрисека очень практична, а в примерах используются как язык C#, так и функционально-объектный язык F#.
16. *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. — СПб.: Символ-Плюс, 1995. Данная книга известна прежде всего как один из лучших трудов в области управления проектами, но из нее можно почерпнуть много полезной информации и в вопросах проектирования.
17. *Brooks F.* The Design of Design: Essays from a Computer Scientist. — 2010. Одна из лучших книг о проектировании как общей дисциплине. Книга заставляет задуматься о философских вещах вроде борьбы со сложностью и забыть на время об особенностях языка программирования или паттернах проектирования.

Более подробное описание книг о проектировании можно найти в моем блоге в статье «Книги по дизайну и ООП» (http://bit.ly/BestOODBooks). Описание лучших книг о языке C# и платформе .NET можно найти в другой моей статье — «Книги для изучения C#/.NET» (http://bit.ly/BestDotNetBooks).

Статьи

Более чем за пять лет ведения блога я опубликовал в нем более 200 заметок, многие из которых являются полноценными статьями. Часть из них были опубликованы в других источниках, например в журнале RSDN Magazine, на портале SoftwarePeople.Ru и на Habrahabr.ru.

Чтобы упростить поиск и навигацию по блогу, я создал отдельную страницу, куда добавляю наиболее значимые публикации. Легко перейти на нее со стартовой страницы моего блога или найти ее по адресу http://sergeyteplyakov.blogspot.com/2013/10/articles.html.

С. Тепляков

Паттерны проектирования на платформе .NET

Заведующий редакцией	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художник	*С. Заматевская*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*А. Барцевич*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 25.03.15. Формат 70×100/16. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru факс: 8(496) 726-54-10, телефон: (495) 988-63-87

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

  Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

---

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
