# Принцип открыт/закрыт

**ЦИТАТА:** *Эффективные проекты контролируют изменения; неэффективные проекты находятся под контролем изменений.* Стив Макконнелл "Остаться в живых" (\*)

Сноска: Стив Макконнелл "Остаться в живых! Руководство для менеджера программных проектов", Питер, 2006

**TODO: была мысль использовать в этой части только лишь цитаты Козьмы Пруткова, а в другой части использовать другие цитаты. Можно отойти от этой практики.**

**Принцип открыт/закрыт (Open-Closed Principle, OCP)**: *Программные сущности (классы, модули, функции и т.п.) должны быть открытыми для расширения, но закрытыми для модификации.*  
Роберт Мартин. "Принципы, паттерны и практики гибкой разработки" [Martin2006]

Одна из причин "загнивания" дизайна кроется в страхе внесения изменений. Разработчики и менеджеры должны быть уверенными в том, что изменения являются корректными и они не приведут к появлению ошибок в других частях системы. Простые классы и модули, которые следуют 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 которых помещается общее поведение, а наследники его 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 говорит о стабильности интерфейсов базовых классов иерархии.

## Путаница с определениями

Из всех SOLID-принципов, принцип Открыт/Закрыт является самым неоднозначным. Его неоднозначность кроется в противоречивости его определения, а подкрепляется разнообразными описаниями этого принципа в разных источниках. Не удивительно, что даже такие яркие представители нашей отрасли, как Эрик Липперт и Джон Скит (\*) , относятся к этому принципу неоднозначно и признаются в его непонимании.

Сноска (\*): Эрик Липперт является автором известного среди .NET разработчиков блога "Fabulous adventures in coding", а Джон "Чак Норрис" Скит занимает первую строчку в рейтинге сайта stackoverflow.com - самого популярного сайта вопросов и ответов по разработке ПО.

Принцип Открыт-Закрыт был изначально сформулирован Бертраном Мейером в первом издании его книги "Объектно-ориентированное конструирование программных систем" еще в 1988 году (\*), но популярность этот принцип завоевал благодаря трудам Роберта Мартина.

Сноска (\*): Бертран Мейер является признанным гуру в мире объектно-ориентированного программирования и автором одной из наиболее влиятельных книг по разработке ПО - "Объектно-ориентированное конструирование программных систем", Интернет-Университет, 2005.

**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.

Из этого определения может сложится впечатление, что следствие 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 этих слов (Роберт Мартин) со временем несколько изменил свое отношение к этому принципу. В своей статье "An Open and Closed Case" (\*) Роберт написал, что с годами он стал более мудрым и менее категоричным в своих высказываниях (хотя, на момент описания принципа Открыт-Закрыт в статье 1996-го года ему было "всего" 43).

(\*) Сноска: An Open and Closed Case by Uncle Bob - http://blog.8thlight.com/uncle-bob/2013/03/08/AnOpenAndClosedCase.html.

Теперь давайте перейдем к определению от Бертрана Мейера и попробуем понять, что же имел ввиду исходный 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 скрытия информации) уже имеет строго определенное окончательное описание. На уровне реализации закрытое состояние модуля означает, что модуль можно компилировать, 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 так далее. 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".

Другими словами, Мейер говорит о том, что интерфейс модуля должен быть закрытым, а реализация и точное поведение могут варьироваться и оставаться открытыми к изменениям.

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 (класса А) остается неизменным, что гарантирует правильное функционирование его старых клиентов, а новые клиенты начинают использовать модифицированную версию класса (класс А’).

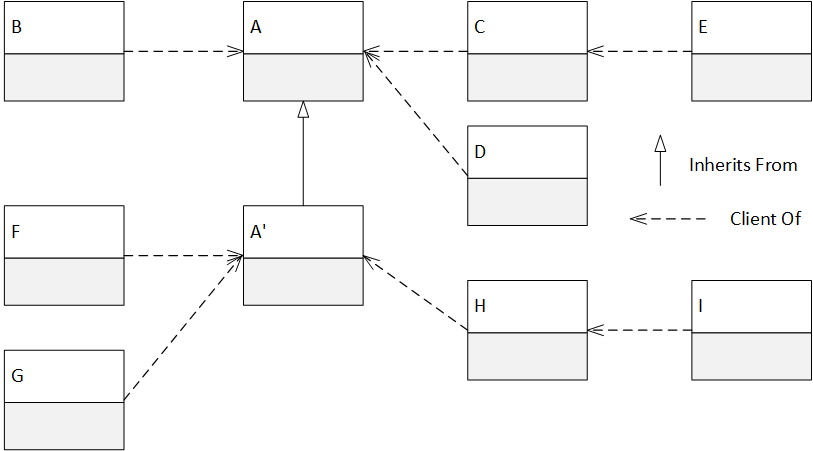


Рисунок 2.1 - Открытость и закрытость классов

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. (\*)"

(\*) Сноска: Ради справки, Мейер трактует понятия «модуля» и «класса» одинаковым образом!

## 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 представляет определенные услуги лишь ему известным способом [Booch2004]:

Лисков прямо утверждает, что "абстракция будет работать только вместе с инкапсуляцией". Практически это означает наличие двух частей в классе: 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. 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

TODO: название пока не нравится! Вообще, не уверен, что этот раздел нужен!! Возможно, следующего раздела будет достаточно!! FEEDBACK REQUIRED!

Давайте рассмотрим Принцип открыт/закрыт на примере развития утилиты для импорта лог-файлов.

Эта утилита может работать в одном из двух режимов: \* в режиме командной строки для одноразового импорта лог-файлов, и \* в режиме сервиса, который будет отслеживать и импортировать новые записи лог-файлов по мере поступления.

Первый режим работы является относительно простым, но второй режим работы требует внесения серьезных изменений в текущий дизайн.

В данный момент, чтение записей осуществляется с помощью pull-модели (\*) 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.

(\*) Сноска: более подробно pull и push-модели взаимодействия были рассмотрены в главе, посвященной паттерну Наблюдатель.

Принцип единственной обязанности говорит нам, что класс LogImporter уже не может отвечать за такой сложный аспект поведения. Нам 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.

Поскольку реализация данного поведения может потребовать существенных усилий, то 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>(\*), с помощью которого мы будем моделировать push-последовательность прочитанных записей.

(\*) Подробнее о паттерне Наблюдатель и об использовании интерфейса IObservable<T> для его реализации, вы можете прочитать в главе 5 первой части книги.

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;  
 }  
}

Листинг 2.1 - Реализация класса LogFileReader

Текущая реализация класса LogFileReader весьма примитивна, но ее наличие позволит разработчику класса LogImporter сосредоточиться на своей части функциональности.

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);  
 }  
 }  
}

Листинг 2.2 - Реализация класса LogImporter (\*) на основе IObservable<string>

(\*) Сноска: в предыдущей главе, 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.

В классе LogImporter происходит "подписка" на события чтения данных объектом \_logFileReader. Библиотека реаткинвых расширений (Reactive Extensions, Rx) предоставляет набор методов расширения для интерфейса IObserver<T>, которые 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>(\*).

(\*) Сноска: подробнее о реактивном программировании и библиотеке Reactive Extensions можно в замечательной он-лайн книге Ли Кэмпбелла Introduction to Rx на сайте introrx.com.

Разработчик класса LogImporter может абстрагироваться от процесса чтения еще сильнее и потребовать интерфейс IObservable<string> через аргументы конструктора. В любом случае, теперь будет достаточно просто развивать модули чтения и импорта независимо. Уже сейчас можно написать интеграционные тесты, которые будут проверять функциональность от начала и до конца. Параллельно этому, автор класса LogFileReader может заняться разработкой более "умной" версии класса LogFileReader, которая будет отслеживать текущую позицию файла, и уведомлять подписчиков при появлении новых записей (рис. 2.2). Это может быть сделано путем опроса файла вручную через определенный период времени, или же с помощью типа FileSystemWatcher (\*).

(\*) Сноска: Класс FilSystemWatcher позволяет отслеживать состояние файла или папки и получать уведомления в случае изменения их содержимого.

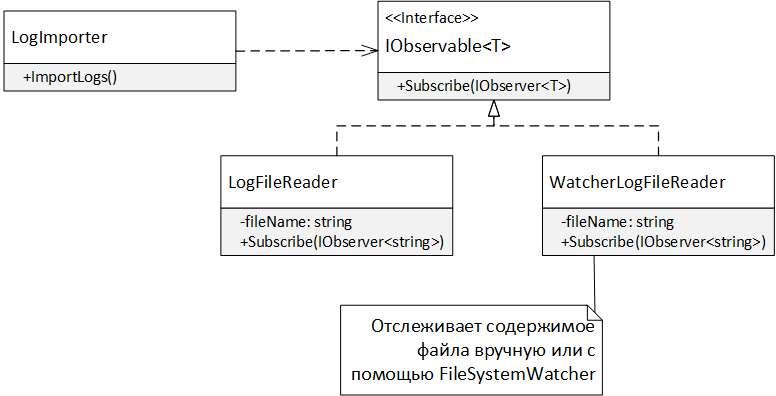


Рисунок 2.2 - "Открытость" реализации анализаторов лог-файлов

### Открытость поведения

Помимо фиксации интерфейсов, Принцип открыт-закрыт позволяет расширять реализацию модулей, без существенных изменений остальных частей системы.

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, тщательно проверена, и сделана 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, нам 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:

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);  
 }

Листинг 2.3 - Использование абстрактного класса LogEntryParser

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. 2.3).

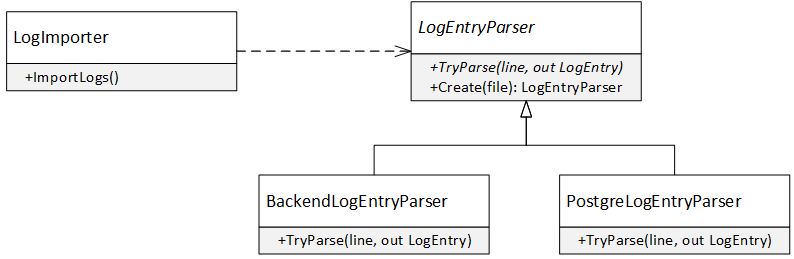


Рисунок 2.3 - Иерархия классов LogEntryParser

Открытость дизайна не означает расширение 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:

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("Unknown file format");  
 }  
  
 // ...  
}

Листинг 2.4 - Реализация фабричного метода LogEntryParser.Create

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 негативным образом отразится на стоимости изменения.

## Расширяемость: объектно-ориентированный и функциональный подходы

При рассмотрении Принципа открыт/закрыт очень часто сравнивают подходы структурного и объектно-ориентированного программирования на примере рисования фигур:

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");  
 }  
}

Листинг 2.5 - Примитивная реализация метода DrawShape

Данный код является нерасширяемым, поскольку его придется обновлять каждый раз при добавлении новой фигуры. После чего приводится объектно-ориентированное решение, расширяемость которого осуществляется за счет использования наследования и полиморфного метода Draw:

public static void DrawShape(Shape shape)  
{  
 shape.Draw();  
}

Листинг 2.6 - Объектно-ориентированная версия метода DrawShape

Теперь мы можем сказать, что избавились от конструкции switch в функции Draw и перенесли всю логику в класс Shape и его наследники. И теперь 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 вся иерархия контролируется одним человеком, то 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:

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);  
 }  
}

Листинг 2.7 - Паттерн Посетитель и иерархия фигур

Теперь, для добавления операции GetArea будет достаточно реализовать интерфейс IShapeVisitor и добавить метод расширения в класс ShapeEx:

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();

Листинг 2.8 - Добавление новой операции в существующую иерархию

"Открытость" иерархий типов является относительной. 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 (\*).

(\*) Сноска: особенности паттерна Посетитель, включая функциональную реализацию на основе делегатов, были рассмотрены в Главе 6 первой части книги.

Паттерн Посетитель показывает "функциональный" подход к расширяемости семейства типов. В функциональном программировании операции четко отделены от данных. Свободные функции принимают на вход 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 множество изменений.

## Типичные примеры нарушения принципа Открыт-Закрыт

* **Интерфейс класса является нестабильным**. Постоянные изменения интерфейса класса, используемого во множестве мест приводит 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 позволяет изменять реализацию без изменения интерфейса и с помощью наследования, что позволяет заменить реализацию, не затрагивая существующих клиентов базового класса.

Приведенные выше 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 нашего мира и носит название "Expression Problem" (\*).

(\*)СНОСКА: одно из лучших описаний данной проблемы можно найти по следующей ссылке: http://c2.com/cgi/wiki?ExpressionProblem

Во время дизайна модуля нужно подумать о том, в каком направлении упрощать расширяемость. Если более 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 более подходящим будет функциональный подход на основе размеченных объединений (discriminated unions) или на основе паттерна "Посетитель" в объектном мире.