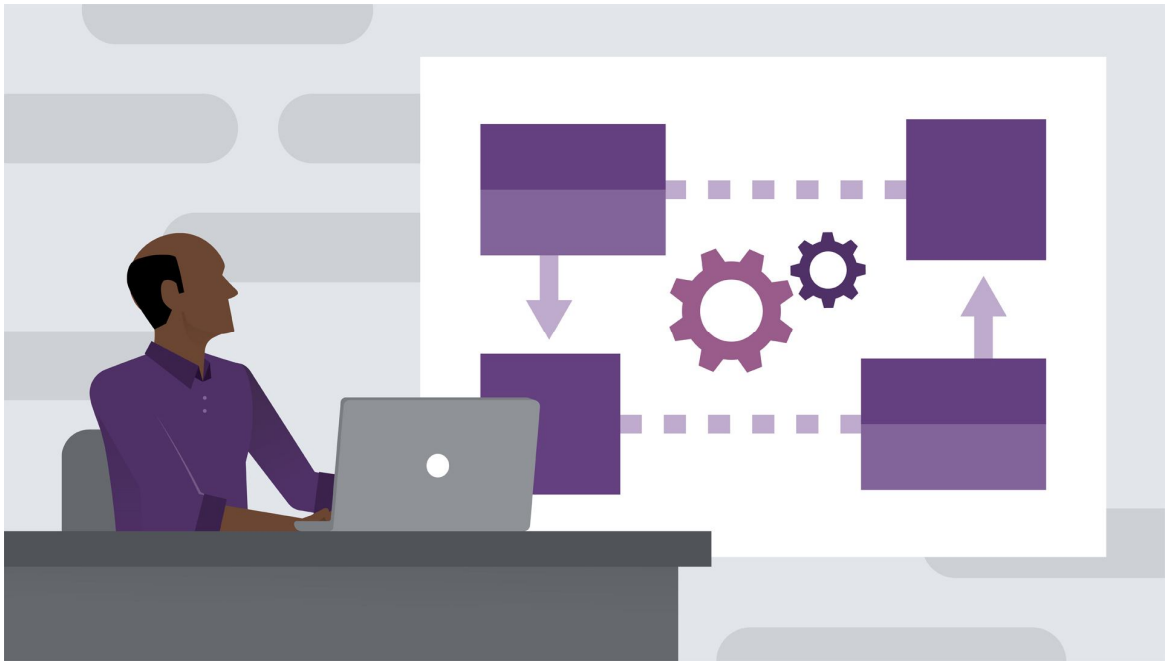
Рыбанов А.А.

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ НА C#



Волжский
2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛЖСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ (ФИЛИАЛ)
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Рыбанов А.А.

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ НА C#

Электронное учебное пособие

Волжский

2023

УДК 004.45(07)

ББК 32.973я73
Р 931

Рецензенты:
заведующий кафедрой методики преподавания математики и физики, ИКТ, профессор, д.п.н. ВГСПУ
Смыковская Т.А.
директор, ООО Научно-производственный центр «АИР»
Шуревский А.Н.

Издается по решению редакционно-издательского совета
Волгоградского технического университета

Рыбанов, А.А.
Паттерны проектирования на C# [Электронный ресурс] : учебное пособие / А.А.. Рыбанов ; Министерство науки и высшего образования Российской Федерации, ВПИ (филиал) ФГБОУ ВО ВолгГТУ. – Электрон. текстовые дан. (1 файл: 776 КБ). – Волжский, 2023. – Режим доступа: http://lib.volpi.ru. – Загл. с титул. экрана.
ISBN 978-5-9948-4548-6

В учебном пособии рассматриваются существующие принципы разработки программных продуктов, такие как SOLID, а также порождающие, структурные и поведенческие паттерны проектирования GoF. Приводятся сильные и слабые стороны существующих методологий разработки программного обеспечения. Учебное пособие соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования. Учебное пособие адресовано студентам высших учебных заведений, обучающимся по направлениям 09.03.01 «Информатика и вычислительная техника» и 09.03.04 «Программная инженерия».

Илл. 16, библиограф.: 6 назв.

ISBN 978-5-9948-4548-6

ВВЕДЕНИЕ

Популярность объектного подхода обусловлена объективными факторами усложнения программных систем и неуклонным повышением требований к интеллектуальности, производительности, эргономичности, доступности и адаптивности программного обеспечения и средств разработки. Особая роль объектно-ориентированных технологий приводит к необходимости детального изучения принципов построения программных компонент информационных систем на базе объектных технологий. При этом объектно-ориентированное программирование (ООП) – это только одно из нескольких самостоятельных направлений изучения и использования теории, в основе которой лежат термины «объект» и «класс». Современные технологии объектно-ориентированного программирования интенсивно развиваются – на данный момент программисту недостаточно понимать простейшие принципы ООП (инкапсуляция, полиморфизм, наследование). ООП, как технология, должна реагировать на появление новых требований современного высокотехнологичного мира: параллельный характер процессов в информационных системах; распределенный характер информационных систем; повышение требований к защищенности программного обеспечения: слияние различных технологий разработки приложений и востребованность унифицированного подхода к проектированию и разработке веб-приложений, сервисов, интерфейсов. Высококвалифицированный программист должен понимать основные ошибки, которые приводят к созданию неэффективных приложений. Программирование приложений должно быть критическим: необходимо использовать различные системы принципов (например, S.O.L.I.D.) и активно применять проверенные паттерны проектирования. Данное пособие ориентировано на

студентов, освоивших синтаксические правила и базовые технологии языка программирования высоко уровня C#.

1. ОСНОВЫ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

1.1. Введение в паттерны проектирования

Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

Хотя идея паттернов как способ описания решения распространенных проблем в области проектирования появилась довольно давно, но их популярность стала расти во многом благодаря известной работе четырех авторов Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона, Джона Влиссидеса, которая называлась "Design Patterns: Elements of Reusable Object-Oriented Software" (на русском языке известна как "Приемы объектно-ориентированного проектирования. Паттерны проектирования") и которая вышла в свет в 1994 году. А сам коллектив авторов нередко называют "Банда четырёх" или Gang of Four или сокращенно GoF. Данная книга по сути являлась первой масштабной попыткой описать распространенные способы проектирования программ. И со временем применение паттернов стало считаться хорошей практикой программирования.

При написании программ мы можем формализовать проблему в виде классов и объектов и связей между ними. И применить один из существующих паттернов для ее решения. В итоге нам не надо ничего придумывать. У нас уже есть готовый шаблон, и нам только надо его применить в конкретной программе.

Причем паттерны, как правило, не зависят от языка программирования. Их принципы применения будут аналогичны и в C#, и в Jave, и в других языках. Хотя в рамках данного руководства мы будем говорить о паттернах в контексте языка C#.

Также мышление паттернами упрощает групповую разработку программ. Зная применяемый паттерн проектирования и его основные принципы, другому программисту будет проще понять его реализацию и использовать ее.

В то же время не стоит применять паттерны ради самих паттернов. Хорошая программа предполагает использование паттернов. Однако не всегда паттерны упрощают и улучшают программу. Неоправданное их использование может привести к усложнению программного кода, уменьшению его качества. Паттерн должен быть оправданным и эффективным способом решения проблемы.

Существует множество различных паттернов, которые решают разные проблемы и выполняют различные задачи. Но по своему действию их можно объединить в ряд групп. Рассмотрим некоторые группы паттернов. В основу классификации основных паттернов положена цель или задачи, которые определенный паттерн выполняет.

Порождающие паттерны – это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов. Среди них выделяются следующие:

- Абстрактная фабрика (Abstract Factory)
- Строитель (Builder)
- Фабричный метод (Factory Method)
- Прототип (Prototype)
- Одиночка (Singleton)

Другая группа паттернов – структурные паттерны – рассматривает, как классы и объекты образуют более крупные структуры – более сложные по характеру классы и объекты. К таким шаблонам относятся:

- Адаптер (Adapter)
- Мост (Bridge)
- Компоновщик (Composite)
- Декоратор (Decorator)
- Фасад (Facade)
- Приспособленец (Flyweight)
- Заместитель (Proxy)

Третья группа паттернов называются поведенческими – они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие:

- Цепочка обязанностей (Chain of responsibility)
- Команда (Command)
- Интерпретатор (Interpreter)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)
- Наблюдатель (Observer)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный метод (Template method)
- Посетитель (Visitor)

Существуют и другие классификации паттернов в зависимости от того, относится паттерн к классам или объектам.

Паттерны классов описывают отношения между классами посредством наследования. Отношения между классами определяются на стадии компиляции. К таким паттернам относятся:

- Фабричный метод (Factory Method)
- Интерпретатор (Interpreter)
- Шаблонный метод (Template Method)
- Адаптер (Adapter)

Другая часть паттернов – паттерны объектов описывают отношения между объектами. Эти отношения возникают на этапе выполнения, поэтому обладают большей гибкостью. К паттернам объектов относят следующие:

- Абстрактная фабрика (Abstract Factory)
- Строитель (Builder)
- Прототип (Prototype)
- Одиночка (Singleton)
- Мост (Bridge)
- Компоновщик (Composite)
- Декоратор (Decorator)
- Фасад (Facade)
- Приспособленец (Flyweight)
- Заместитель (Proxy)
- Цепочка обязанностей (Chain of responsibility)
- Команда (Command)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)
- Наблюдатель (Observer)
- Состояние (State)

- Стратегия (Strategy)
- Посетитель (Visitor)

И это только некоторые основные паттерны. А вообще различных шаблонов проектирования гораздо больше. Одни из них только начинают применяться, другие являются популярными на текущий момент, а некоторые уже менее распространены, чем раньше.

В данном учебном пособии рассмотрим основные и наиболее распространенные паттерны и принципы их использования применительно к языку C#.

Для выбора паттерна при решении какой-нибудь проблемы надо выделить все используемые сущности и связи между ними и абстрагировать их от конкретной ситуации. Затем надо посмотреть, вписывается ли абстрактная форма решения задачи в определенный паттерн. Например, суть решаемой задачи может состоять в создании новых объектов. В этом случае, возможно, стоит посмотреть на порождающие паттерны. Причем лучше не сразу взять какой-то определенный паттерн – первый, который показался нужным, а посмотреть на несколько родственных паттернов из одной группы, которые решают одну и ту же задачу.

При этом важно понимать смысл и назначение паттерна, явно представлять его абстрактную организацию и его возможные конкретные реализации. Один паттерн может иметь различные реализации, и чем чаще вы будете сталкиваться с этими реализациями, тем лучше вы будете понимать смысл паттерна. Но не стоит использовать паттерн, если вы его не понимаете, даже если он на первый взгляд поможет вам в решении задачи. И, в конечном счете, надо придерживаться принципа KISS (Keep It Simple, Stupid) – сохранять код программы по возможности простым и ясным. Ведь смысл паттернов не в усложнении кода программы, а наоборот в его упрощении.

1.2. Отношения между классами и объектами

Прежде чем приступить к изучению основных паттернов, также рассмотрим основные отношения между объектами, которые помогут нам понять связи между сущностями при их использовании в паттернах. Мы можем выделить несколько основных отношений: наследование, реализация, ассоциация, композиция и агрегация.

Наследование. Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение IS A, то есть "является". Например:

```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Manager : User
{
    public string Company { get; set; }
}
```

В данном случае используется наследование, а объекты класса Manager также являются и объектами класса User.

С помощью диаграмм UML отношение между классами выражается в незакрашенной стрелочке от класса-наследника к классу-родителю (рис. 1).

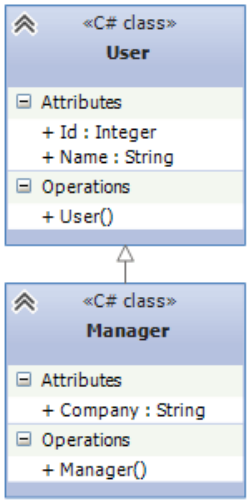


Рис. 1. Отношение наследования

Реализация. Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

```
public interface IMovable
{
    void Move();
}
public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

С помощью диаграмм UML отношение реализации также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная (рис. 2).

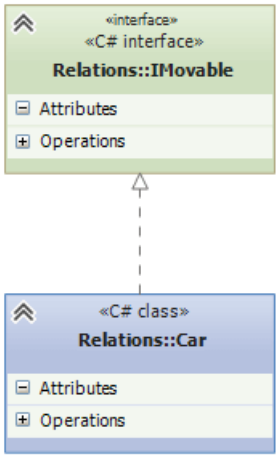


Рис. 2. Отношение реализации

Ассоциация. Ассоциация – это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

```
class Team
{

}
class Player
{
    public Team Team { get; set; }
}
```

Класс Player связан отношением ассоциации с классом Team. На схемах UML ассоциация обозначается в виде обычно стрелки (рис. 3).

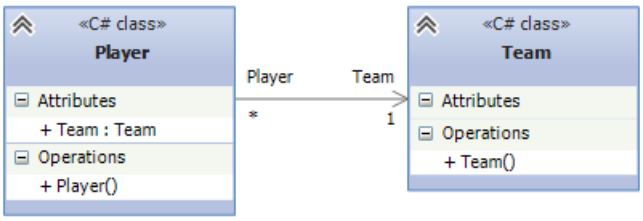


Рис. 3. Отношение ассоциации

Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

Агрегация и композиция являются частными случаями ассоциации.

Композиция. Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя:

```
public class ElectricEngine
{ }

public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя – зависимой.

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит объект второй сущности, располагается закрашенный ромбик (рис. 4).

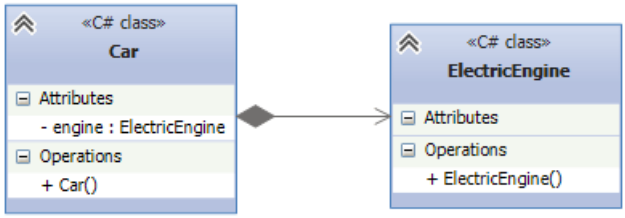


Рис. 4. Отношение композиции

Агрегация. От композиции следует отличать агрегацию. Она также предполагает отношение HAS A, но реализуется она иначе:

```
public abstract class Engine
{ }

public class Car
{
    Engine engine;
    public Car(Engine eng)
    {
        engine = eng;
    }
}
```

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Отношение агрегации на диаграммах UML отображается так же, как и отношение композиции, только теперь ромбик будет незакрашенным (рис. 5).

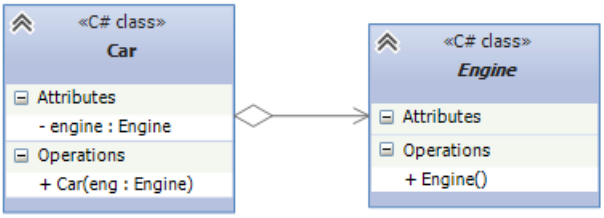


Рис. 5. Отношение агрегации

При проектировании отношений между классами надо учитывать некоторые общие рекомендации. В частности, вместо наследования следует предпочитать композицию. При наследовании весь функционал класса-наследника жестко определен на этапе компиляции. И во время выполнения программы мы не можем его динамически переопределить. А класс-наследник не всегда может переопределить код, который определен в родительском классе. Композиция же позволяет динамически определять поведение объекта во время выполнения и поэтому является более гибкой.

Вместо композиции следует предпочитать агрегацию, как более гибкий способ связи компонентов. В то же время не всегда агрегация уместна. Например, у нас есть класс человека, который содержит объект нервной системы. Понятно, что в реальности, по крайней мере на текущий момент, невозможно вовне определить нервную систему и внедрить ее в человека. То есть в данном случае человек будет главным компонентом, а нервная система – зависимым, подчиненным, и их создание и жизненный цикл будет происходить совместно, поэтому здесь лучше выбрать композицию.

1.3. Интерфейсы или абстрактные классы

Один из принципов проектирования гласит, что при создании системы классов надо программировать на уровне интерфейсов, а не их конкретных реализаций. Под интерфейсами в данном случае понимаются не только типы C#, определенные с помощью ключевого слова interface, а определение функционала без его конкретной реализации. То есть под

данное определение попадают как собственно интерфейсы, так и абстрактные классы, которые могут иметь абстрактные методы без конкретной реализации.

В этом плане у абстрактных классов и интерфейсов много общего. Нередко при проектировании программ в паттернах мы можем заменять абстрактные классы на интерфейсы и наоборот. Однако все же они имеют некоторые отличия.

Когда следует использовать абстрактные классы:

- если надо определить общий функционал для родственных объектов;
- если мы проектируем довольно большую функциональную единицу, которая содержит много базового функционала;
- если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию. При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе;
- если же нам вдруг надо будет поменять название или параметры метода интерфейса, то придется вносить изменения и также во всех классы, которые данный интерфейс реализуют.

Когда следует использовать интерфейсы:

- если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой;
- если мы проектируем небольшой функциональный тип.

Ключевыми здесь являются первые пункты, которые можно свести к следующему принципу: если классы относятся к единой системе классификации, то выбирается абстрактный класс. Иначе выбирается интерфейс. Посмотрим на примере. Допустим, у нас есть система транспортных средств: легковой автомобиль, автобус, трамвай, поезд и т.д.

Поскольку данные объекты являются родственными, мы можем выделить у них общие признаки, то в данном случае можно использовать абстрактные классы:

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Машина едет");
    }
}

public class Bus : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Автобус едет");
    }
}

public class Tram : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Трамвай едет");
    }
}
```

Абстрактный класс Vehicle определяет абстрактный метод перемещения Move(), а классы-наследники его реализуют.

Но предположим, что наша система транспорта не ограничивается вышеперечисленными транспортными средствами. Например, мы можем добавить самолеты, лодки. Возможно, также мы добавим лошадь – животное, которое может также выполнять роль транспортного средства. Также можно добавить дирижабль. В общем, получается довольно широкий круг объектов, которые связаны только тем, что являются транспортным средством и должны реализовать некоторый метод Move(), выполняющий перемещение.

Так как объекты малосвязанные между собой, то для определения общего для всех них функционала лучше определить интерфейс. Тем более некоторые из этих объектов могут существовать в рамках параллельных

систем классификаций. Например, лошадь может быть классом в структуре системы классов животного мира.

Возможная реализация интерфейса могла бы выглядеть следующим образом:

```
public interface IMovable
{
    void Move();
}

public abstract class Vehicle : IMovable
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move() => Console.WriteLine("Машина едет");
}

public class Bus : Vehicle
{
    public override void Move() => Console.WriteLine("Автобус едет");
}

public class Hourse : IMovable
{
    public void Move() => Console.WriteLine("Лошадь скачет");
}

public class Aircraft : IMovable
{
    public void Move() => Console.WriteLine("Самолет летит");
}
```

Теперь метод Move() определяется в интерфейсе IMovable, а конкретные классы его реализуют.

Говоря об использовании абстрактных классов и интерфейсов, можно привести еще такую аналогию, как состояние и действие. Как правило, абстрактные классы фокусируются на общем состоянии классов-наследников. В то время как интерфейсы строятся вокруг какого-либо общего действия.

Например, солнце, костер, батарея отопления и электрический нагреватель выполняют функцию нагревания или излучения тепла. По большому счету выделение тепла – это единственный общий между ними признак. Можно ли для них создать общий абстрактный класс? Можно, но

это не будет оптимальным решением, тем более у нас могут быть какие-то родственные сущности, которые мы, возможно, тоже захотим использовать. Поэтому для каждой вышеперечисленной сущности мы можем определить свою систему классификации. Например, в одной системе классов, которые наследуются от общего абстрактного класса, были бы звезды, в том числе и солнце, планеты, астероиды и так далее – то есть все те объекты, которые могут иметь какое-то общее с солнцем состояние. В рамках другой системы классов мы могли бы определить электрические приборы, в том числе электронагреватель. И так для каждой разноплановой сущности можно было бы составить свою систему классов, исходящую от определенного абстрактного класса. А для общего действия определить интерфейс, например, IHeatable, в котором бы был метод Heat, и этот интерфейс реализовать во всех необходимых классах.

Таким образом, если разноплановые классы обладают каким-то общим действием, то это действие лучше выносить в интерфейс. А для одноплановых классов, которые имеют общее состояние, лучше определять абстрактный класс.

2. ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

2.1. Фабричный метод (Factory Method)

Фабричный метод (Factory Method) – это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Когда надо применять паттерн:

- когда заранее неизвестно, объекты каких типов необходимо создавать;

- когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;
- когда создание новых объектов необходимо делегировать из базового класса классам наследникам.

Описание паттерна на языке UML приведено на рисунке 6.

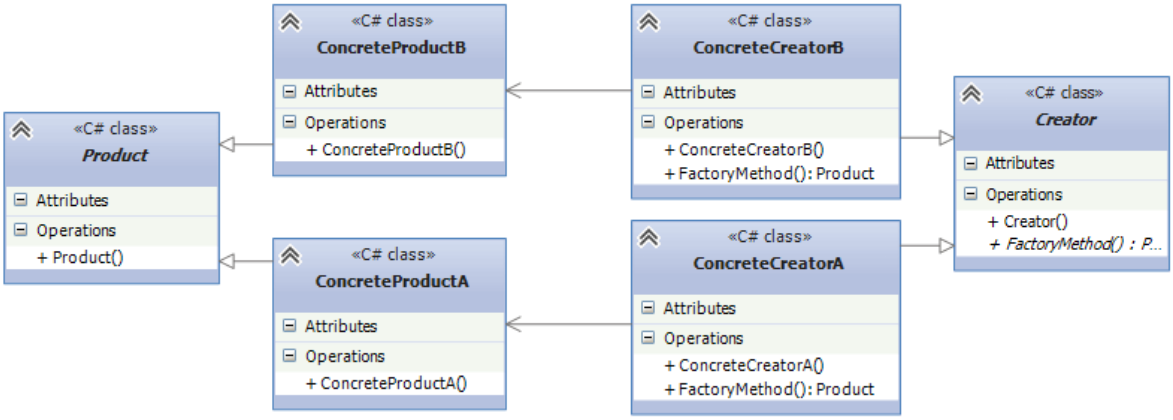


Рис. 6. Фабричный метод (Factory Method)

Формальное определение паттерна на языке C# может выглядеть следующим образом:

```
abstract class Product
{ }

class ConcreteProductA : Product
{ }

class ConcreteProductB : Product
{ }

abstract class Creator
{
    public abstract Product FactoryMethod();
}

class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductA(); }
}

class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductB(); }
}
```

Участники:

- Абстрактный класс Product определяет интерфейс класса, объекты которого надо создавать.
- Конкретные классы ConcreteProductA и ConcreteProductB представляют реализацию класса Product. Таких классов может быть множество
- Абстрактный класс Creator определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
- Конкретные классы ConcreteCreatorA и ConcreteCreatorB – наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.

Теперь рассмотрим на реальном примере. Допустим, мы создаем программу для сферы строительства. Возможно, вначале мы захотим построить многоэтажный панельный дом. И для этого выбирается соответствующий подрядчик, который возводит каменные дома. Затем нам захочется построить деревянный дом и для этого также надо будет выбрать нужного подрядчика:

```
class Program
{
    static void Main(string[] args)
    {
        Developer dev = new PanelDeveloper("ООО КирпичСтрой");
        House house2 = dev.Create();

        dev = new WoodDeveloper("Частный застройщик");
        House house = dev.Create();

        Console.ReadLine();
    }
}
// абстрактный класс строительной компании
abstract class Developer
```

```
{
    public string Name { get; set; }

    public Developer(string n)
    {
        Name = n;
    }
    // фабричный метод
    abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}
// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}

abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}
class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

В качестве абстрактного класса Product здесь выступает класс House. Его две конкретные реализации – PanelHouse и WoodHouse представляют типы домов, которые будут строить подрядчики. В качестве абстрактного класса создателя выступает Developer, определяющий абстрактный метод Create(). Этот метод реализуется в классах-наследниках WoodDeveloper и PanelDeveloper. И если в будущем нам потребуется построить дома какого-

то другого типа, например, кирпичные, то мы можем с легкостью создать новый класс кирпичных домов, унаследованный от House, и определить класс соответствующего подрядчика. Таким образом, система получится легко расширяемой. Правда, недостатки паттерна тоже очевидны – для каждого нового продукта необходимо создавать свой класс создателя.

2.2. Абстрактная фабрика (Abstract Factory)

Паттерн "Абстрактная фабрика" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

Когда использовать абстрактную фабрику?

- Когда система не должна зависеть от способа создания и компоновки новых объектов.
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными.

Описание паттерна на языке UML приведено на рисунке 7.

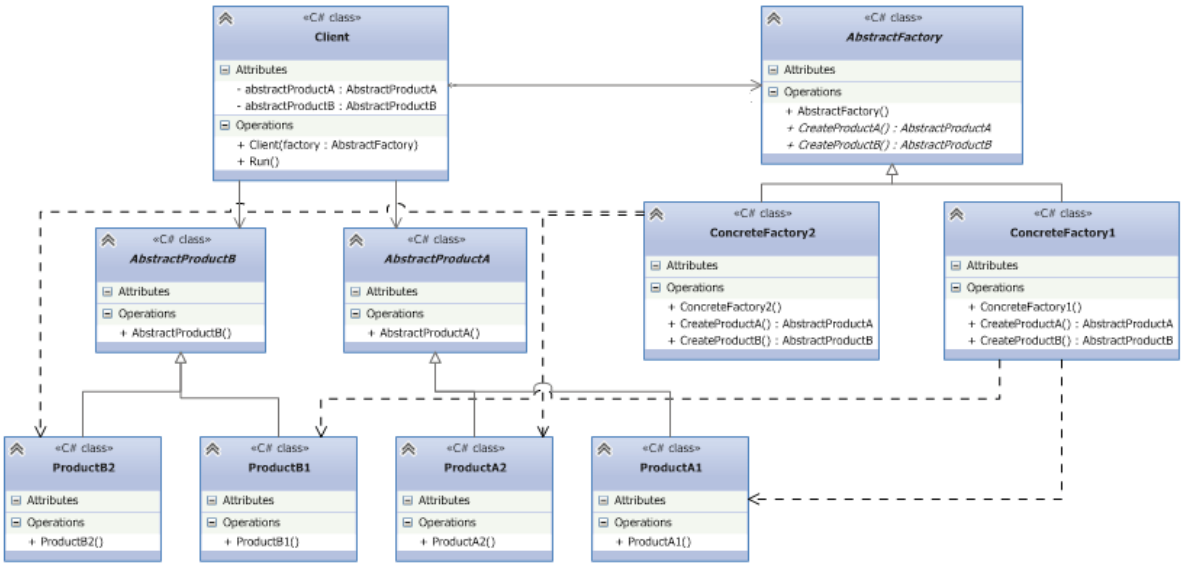


Рис. 7. Абстрактная фабрика (Abstract Factory)

Формальное определение паттерна на языке C# может выглядеть следующим образом:

```
abstract class AbstractFactory
{
```

```
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
class ConcreteFactory1 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}
class ConcreteFactory2 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}

abstract class AbstractProductA
{ }

abstract class AbstractProductB
{ }

class ProductA1 : AbstractProductA
{ }

class ProductB1 : AbstractProductB
{ }

class ProductA2 : AbstractProductA
{ }

class ProductB2 : AbstractProductB
{ }

class Client
{
    private AbstractProductA abstractProductA;
    private AbstractProductB abstractProductB;

    public Client(AbstractFactory factory)
    {
        abstractProductB = factory.CreateProductB();
        abstractProductA = factory.CreateProductA();
    }
    public void Run()
    { }
}
```

Паттерн определяет следующих участников:

- Абстрактные классы AbstractProductA и AbstractProductB определяют интерфейс для классов, объекты которых будут создаваться в программе.
- Конкретные классы ProductA1 / ProductA2 и ProductB1 / ProductB2 представляют конкретную реализацию абстрактных классов.
- Абстрактный класс фабрики AbstractFactory определяет методы для создания объектов. Причем методы возвращают абстрактные продукты, а не их конкретные реализации.
- Конкретные классы фабрик ConcreteFactory1 и ConcreteFactory2 реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать.
- Класс клиента Client использует класс фабрики для создания объектов. При этом он использует исключительно абстрактный класс фабрики AbstractFactory и абстрактные классы продуктов AbstractProductA и AbstractProductB и никак не зависит от их конкретных реализаций.

Посмотрим, как мы можем применить паттерн. Например, мы делаем игру, где пользователь должен управлять некими супергероями, при этом каждый супергерой имеет определенное оружие и определенную модель передвижения. Различные супергерои могут определяться комплексом признаков. Например, эльф может летать и должен стрелять из арбалета, другой супергерой должен бегать и управлять мечом. Таким образом, получается, что сущность оружия и модель передвижения являются взаимосвязанными и используются в комплексе. То есть имеется один из доводов в пользу использования абстрактной фабрики.

И кроме того, наша задача при проектировании игры абстрагировать создание супергероев от самого класса супергероя, чтобы создать более гибкую архитектуру. И для этого применим абстрактную фабрику:

```
class Program
{
```

```
    static void Main(string[] args)
    {
        Hero elf = new Hero(new ElfFactory());
        elf.Hit();
        elf.Run();

        Hero voin = new Hero(new VoinFactory());
        voin.Hit();
        voin.Run();

        Console.ReadLine();
    }
}
//абстрактный класс - оружие
abstract class Weapon
{
    public abstract void Hit();
}
// абстрактный класс движение
abstract class Movement
{
    public abstract void Move();
}

// класс арбалет
class Arbalet : Weapon
{
    public override void Hit()
    {
        Console.WriteLine("Стреляем из арбалета");
    }
}
// класс меч
class Sword : Weapon
{
    public override void Hit()
    {
        Console.WriteLine("Бьем мечом");
    }
}
// движение полета
class FlyMovement : Movement
{
    public override void Move()
    {
        Console.WriteLine("Летим");
    }
}
// движение - бег
class RunMovement : Movement
{
    public override void Move()
    {
        Console.WriteLine("Бежим");
    }
}
// класс абстрактной фабрики
abstract class HeroFactory
{
    public abstract Movement CreateMovement();
    public abstract Weapon CreateWeapon();
}
```

```
// Фабрика создания летящего героя с арбалетом
class ElfFactory : HeroFactory
{
    public override Movement CreateMovement()
    {
        return new FlyMovement();
    }

    public override Weapon CreateWeapon()
    {
        return new Arbalet();
    }
}
// Фабрика создания бегущего героя с мечом
class VoinFactory : HeroFactory
{
    public override Movement CreateMovement()
    {
        return new RunMovement();
    }

    public override Weapon CreateWeapon()
    {
        return new Sword();
    }
}
// клиент - сам супергерой
class Hero
{
    private Weapon weapon;
    private Movement movement;
    public Hero(HeroFactory factory)
    {
        weapon = factory.CreateWeapon();
        movement = factory.CreateMovement();
    }
    public void Run()
    {
        movement.Move();
    }
    public void Hit()
    {
        weapon.Hit();
    }
}
```

Таким образом, создание супергероя абстрагируется от самого класса супергероя. В то же время нельзя не отметить и недостатки шаблона. В частности, если нам захочется добавить в конфигурацию супергероя новый объект, например, тип одежды, то придется переделывать классы фабрик и класс супергероя. Поэтому возможности по расширению в данном паттерне имеют некоторые ограничения.

2.3. Одиночка (Singleton)

Одиночка (Singleton, Синглтон) – порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон?

- Когда необходимо, чтобы для класса существовал только один экземпляр.
- Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

```
class Singleton
{
    private static Singleton instance;

    private Singleton()
    { }

    public static Singleton getInstance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

В классе определяется статическая переменная – ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

Для применения паттерна Одиночка создадим небольшую программу. Например, на каждом компьютере можно одномоментно запустить только одну операционную систему. В этом плане операционная система будет реализоваться через паттерн синглтон:

```
class Program
{
    static void Main(string[] args)
    {
        Computer comp = new Computer();
        comp.Launch("Windows 8.1");
        Console.WriteLine(comp.OS.Name);
```

```
        // у нас не получится изменить ОС, так как объект уже создан
        comp.OS = OS.getInstance("Windows 10");
        Console.WriteLine(comp.OS.Name);

        Console.ReadLine();
    }
}
class Computer
{
    public OS OS { get; set; }
    public void Launch(string osName)
    {
        OS = OS.getInstance(osName);
    }
}
class OS
{
    private static OS instance;

    public string Name { get; private set; }

    protected OS(string name)
    {
        this.Name = name;
    }

    public static OS getInstance(string name)
    {
        if (instance == null)
            instance = new OS(name);
        return instance;
    }
}
```

3. ПАТТЕРНЫ ПОВЕДЕНИЯ

3.1. Стратегия (Strategy)

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

Когда использовать стратегию?

- Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные вариан-

ты поведения вынести в отдельные классы и при необходимости их применять.

- Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий.
- Когда необходимо менять поведение объектов на стадии выполнения программы.
- Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Описание паттерна на языке UML приведено на рисунке 8.

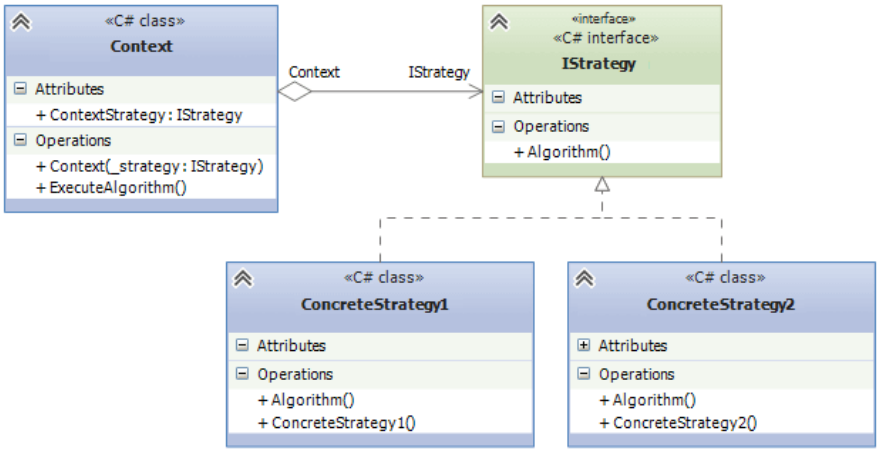


Рис. 8. Паттерн Стратегия (Strategy)

Формальное определение паттерна на языке C# может выглядеть следующим образом:

```
public interface IStrategy
{
    void Algorithm();
}

public class ConcreteStrategy1 : IStrategy
{
    public void Algorithm()
    { }
}

public class ConcreteStrategy2 : IStrategy
{
    public void Algorithm()
    { }
}

public class Context
{
    public IStrategy ContextStrategy { get; set; }
```

```
    public Context(IStrategy _strategy)
    {
        ContextStrategy = _strategy;
    }

    public void ExecuteAlgorithm()
    {
        ContextStrategy.Algorithm();
    }
}
```

Участники:

- Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
- Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
- Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

В данном случае объект IStrategy заключена в свойстве ContextStrategy, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

Теперь рассмотрим конкретный пример. Существуют различные легковые машины, которые используют разные источники энергии: электричество, бензин, газ и так далее. Есть гибридные автомобили. В целом они похожи и отличаются преимущественно видом источника энергии. Не говоря уже о том, что мы можем изменить применяемый источник энергии, модифицировав автомобиль. И в данном случае вполне можно применить паттерн стратегию:

```
class Program
{
    static void Main(string[] args)
    {
        Car auto = new Car(4, "Volvo", new PetrolMove());
        auto.Move();
        auto.Movable = new ElectricMove();
```

```
        auto.Move();

        Console.ReadLine();
    }
}
interface IMovable
{
    void Move();
}

class PetrolMove : IMovable
{
    public void Move()
    {
        Console.WriteLine("Перемещение на бензине");
    }
}

class ElectricMove : IMovable
{
    public void Move()
    {
        Console.WriteLine("Перемещение на электричестве");
    }
}
class Car
{
    protected int passengers; // кол-во пассажиров
    protected string model; // модель автомобиля

    public Car(int num, string model, IMovable mov)
    {
        this.passengers = num;
        this.model = model;
        Movable = mov;
    }
    public IMovable Movable { private get; set; }
    public void Move()
    {
        Movable.Move();
    }
}
```

В данном случае в качестве IStrategy выступает интерфейс IMovable, определяющий метод Move(). А реализующее этот интерфейс семейство алгоритмов представлено классами ElectricMove и PetroleMove. И данные алгоритмы использует класс Car.

3.2. Наблюдатель (Observer)

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдате-

лей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

Когда использовать паттерн Наблюдатель?

- Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях.
- Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
- Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

Описание паттерна на языке UML приведено на рисунке 9.

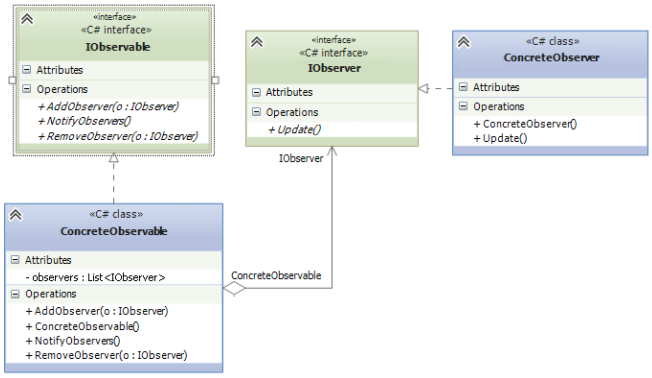


Рис. 9. Наблюдатель (Observer)

Формальное определение паттерна на языке C# может выглядеть следующим образом:

```
interface IObservable
{
    void AddObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
}
class ConcreteObservable : IObservable
{
    private List<IObserver> observers;
    public ConcreteObservable()
    {
        observers = new List<IObserver>();
    }
    public void AddObserver(IObserver o)
    {
        observers.Add(o);
    }

    public void RemoveObserver(IObserver o)
    {
        observers.Remove(o);
    }

    public void NotifyObservers()
    {
        foreach (IObserver observer in observers)
            observer.Update();
    }
}

interface IObserver
{
    void Update();
}
class ConcreteObserver : IObserver
{
    public void Update()
    {
    }
}
```

Участники:

- IObservable: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление наблюдателя) и NotifyObservers() (уведомление наблюдателей).
- ConcreteObservable: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.

- IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
- ConcreteObserver: конкретная реализация интерфейса IObserver.

При этом наблюдаемому объекту не надо ничего знать о наблюдателе кроме того, что тот реализует метод Update(). С помощью отношения агрегации реализуется слабосвязанность обоих компонентов. Изменения в наблюдаемом объекте не виляют на наблюдателя и наоборот.

В определенный момент наблюдатель может прекратить наблюдение. И после этого оба объекта – наблюдатель и наблюдаемый могут продолжать существовать в системе независимо друг от друга.

Рассмотрим реальный пример применения шаблона. Допустим, у нас есть биржа, где проходят торги, и есть брокеры и банки, которые следят за поступающей информацией и в зависимости от поступившей информации производят определенные действия:

```
class Program
{
    static void Main(string[] args)
    {
        Stock stock = new Stock();
        Bank bank = new Bank("ЮнитБанк", stock);
        Broker broker = new Broker("Иван Иваныч", stock);
        // имитация торгов
        stock.Market();
        // брокер прекращает наблюдать за торгами
        broker.StopTrade();
        // имитация торгов
        stock.Market();

        Console.Read();
    }
}

interface IObserver
{
    void Update(Object ob);
}

interface IObservable
{
    void RegisterObserver(IObserver o);
    void RemoveObserver(IObserver o);
    void NotifyObservers();
```

```
}

class Stock : IObservable
{
    StockInfo sInfo; // информация о торгах

    List<IObserver> observers;
    public Stock()
    {
        observers = new List<IObserver>();
        sInfo = new StockInfo();
    }
    public void RegisterObserver(IObserver o)
    {
        observers.Add(o);
    }

    public void RemoveObserver(IObserver o)
    {
        observers.Remove(o);
    }

    public void NotifyObservers()
    {
        foreach (IObserver o in observers)
        {
            o.Update(sInfo);
        }
    }

    public void Market()
    {
        Random rnd = new Random();
        sInfo.USD = rnd.Next(20, 40);
        sInfo.Euro = rnd.Next(30, 50);
        NotifyObservers();
    }
}

class StockInfo
{
    public int USD { get; set; }
    public int Euro { get; set; }
}

class Broker : IObserver
{
    public string Name { get; set; }
    IObservable stock;
    public Broker(string name, IObservable obs)
    {
        this.Name = name;
        stock = obs;
        stock.RegisterObserver(this);
    }
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.USD > 30)
            Console.WriteLine("Брокер {0} продает доллары;  Курс доллара: {1}",
this.Name, sInfo.USD);
```

```
        else
            Console.WriteLine("Брокер {0} покупает доллары;  Курс доллара: {1}",
this.Name, sInfo.USD);
    }
    public void StopTrade()
    {
        stock.RemoveObserver(this);
        stock = null;
    }
}

class Bank : IObserver
{
    public string Name { get; set; }
    IObservable stock;
    public Bank(string name, IObservable obs)
    {
        this.Name = name;
        stock = obs;
        stock.RegisterObserver(this);
    }
    public void Update(object ob)
    {
        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.Euro > 40)
            Console.WriteLine("Банк {0} продает евро;  Курс евро: {1}", this.Name,
sInfo.Euro);
        else
            Console.WriteLine("Банк {0} покупает евро;  Курс евро: {1}", this.Name,
sInfo.Euro);
    }
}
```

Итак, здесь наблюдаемый объект представлен интерфейсом IObservable, а наблюдатель – интерфейсом IObserver. Реализацией интерфейса IObservable является класс Stock, который символизирует валютную биржу. В этом классе определен метод Market(), который имитирует торги и инкапсулирует всю информацию о валютных курсах в объекте StockInfo. После проведения торгов производится уведомление всех наблюдателей.

Реализациями интерфейса IObserver являются классы Broker, представляющий брокера, и Bank, представляющий банк. При этом метод Update() интерфейса IObserver принимает в качестве параметра некоторый объект. Реализация этого метода подразумевает получение через данный параметр объекта StockInfo с текущей информацией о торгах и произведение некоторых действий: покупка или продажа долларов и евро. Дело в том, что часто необходимо информировать наблюдателя об изменении со-

стояния наблюдаемого объекта. В данном случае состояние заключено в объекте StockInfo. И одним из вариантом информирования наблюдателя о состоянии является push-модель, при которой наблюдаемый объект передает (иначе говоря толкает – push) данные о своем состоянии, то есть передаем в виде параметра метода Update().

Альтернативой push-модели является pull-модель, когда наблюдатель вытягивает (pull) из наблюдаемого объекта данные о состоянии с помощью дополнительных методов.

Также в классе брокера определен дополнительный метод StopTrade(), с помощью которого брокер может отписаться от уведомлений биржи и перестать быть наблюдателем.

3.3. Команда (Command)

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

Когда использовать команды?

- Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.
- Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.

- Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы – для этого необходимо будет использовать последовательность запротоколированных команд.

Описание паттерна на языке UML приведено на рисунке 10.

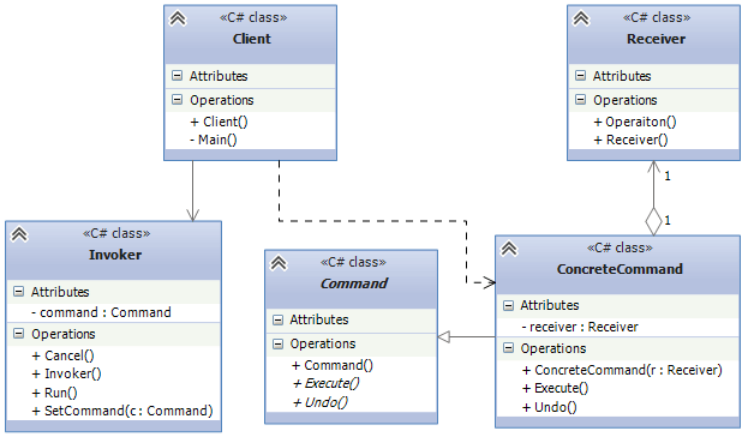


Рис. 10. Команда (Command)

Формальное определение на языке C# может выглядеть следующим образом:

```
abstract class Command
{
    public abstract void Execute();
    public abstract void Undo();
}
// конкретная команда
class ConcreteCommand : Command
{
    Receiver receiver;
    public ConcreteCommand(Receiver r)
    {
        receiver = r;
    }
    public override void Execute()
    {
        receiver.Operation();
    }

    public override void Undo()
    { }
}

// получатель команды
class Receiver
{
    public void Operation()
    { }
}
// инициатор команды
class Invoker
```

```
{
    Command command;
    public void SetCommand(Command c)
    {
        command = c;
    }
    public void Run()
    {
        command.Execute();
    }
    public void Cancel()
    {
        command.Undo();
    }
}
class Client
{
    void Main()
    {
        Invoker invoker = new Invoker();
        Receiver receiver = new Receiver();
        ConcreteCommand command = new ConcreteCommand(receiver);
        invoker.SetCommand(command);
        invoker.Run();
    }
}
```

Участники:

- Command: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды.
- ConcreteCommand: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver.
- Receiver: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
- Invoker: инициатор команды – вызывает команду для выполнения определенного запроса.
- Client: клиент – создает команду и устанавливает ее получателя с помощью метода SetCommand().

Таким образом, инициатор, отправляющий запрос, ничего не знает о получателе, который и будет выполнять команду. Кроме того, если нам потребуется применить какие-то новые команды, мы можем просто унасле-

довать классы от абстрактного класса Command и реализовать его методы Execute и Undo.

В программах на C# команды находят довольно широкое применение. Так, в технологии WPF и других технологиях, которые используют XAML и подход MVVM, на командах во многом базируется взаимодействие с пользователем. В некоторых архитектурах, например, в архитектуре CQRS, команды являются одним из ключевых компонентов.

Нередко в роли инициатора команд выступают панели управления или кнопки интерфейса. Самая простая ситуация – надо программно организовать включение и выключение прибора, например, телевизора. Решение данной задачи с помощью команд могло бы выглядеть так:

```
class Program
{
    static void Main(string[] args)
    {
        Pult pult = new Pult();
        TV tv = new TV();
        pult.SetCommand(new TVOnCommand(tv));
        pult.PressButton();
        pult.PressUndo();

        Console.Read();
    }
}

interface ICommand
{
    void Execute();
    void Undo();
}

// Receiver - Получатель
class TV
{
    public void On()
    {
        Console.WriteLine("Телевизор включен!");
    }

    public void Off()
    {
        Console.WriteLine("Телевизор выключен...");
    }
}

class TVOnCommand : ICommand
{
    TV tv;
    public TVOnCommand(TV tvSet)
```

```
    {
        tv = tvSet;
    }
    public void Execute()
    {
        tv.On();
    }
    public void Undo()
    {
        tv.Off();
    }
}

// Invoker - инициатор
class Pult
{
    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)
    {
        command = com;
    }

    public void PressButton()
    {
        command.Execute();
    }
    public void PressUndo()
    {
        command.Undo();
    }
}
```

Итак, в этой программе есть интерфейс команды – ICommand, есть ее реализация в виде класса TVOnCommand, есть инициатор команды – класс Pult, некий прибор – пульт, управляющий телевизором. И есть получатель команды – класс TV, представляющий телевизор. В качестве клиента используется класс Program.

При этом пульт ничего не знает об объекте TV. Он только знает, как отправить команду. В итоге мы получаем гибкую систему, в которой мы легко можем заменять одни команды на другие, создавать последовательности команд. Например, в нашей программе кроме телевизора появилась микроволновка, которой тоже неплохо было бы управлять с помощью одного интерфейса. Для этого достаточно добавить соответствующие классы и установить команду:

```
class Program
{
```

```
    static void Main(string[] args)
    {
        Pult pult = new Pult();
        TV tv = new TV();
        pult.SetCommand(new TVOnCommand(tv));
        pult.PressButton();
        pult.PressUndo();

        Microwave microwave = new Microwave
        // 5000 - время нагрева пищи
        pult.SetCommand(new MicrowaveCommand(microwave, 5000));
        pult.PressButton();

        Console.Read();
    }
}
//.....ранее описанные классы

class Microwave
{
    public void StartCooking(int time)
    {
        Console.WriteLine("Подогреваем еду");
        // имитация работы с помощью асинхронного метода Task.Delay
        Task.Delay(time).GetAwaiter().GetResult();
    }

    public void StopCooking()
    {
        Console.WriteLine("Еда подогрета!");
    }
}
class MicrowaveCommand : ICommand
{
    Microwave microwave;
    int time;
    public MicrowaveCommand(Microwave m, int t)
    {
        microwave = m;
        time = t;
    }
    public void Execute()
    {
        microwave.StartCooking(time);
        microwave.StopCooking();
    }

    public void Undo()
    {
        microwave.StopCooking();
    }
}
```

Теперь еще одним получателем запроса является класс Microwave, функциональностью которого можно управлять через команды MicrowaveCommand.

Правда, в вышеописанной системе есть один изъян: если мы попытаемся выполнить команду до ее назначения, то программа выдаст исключе-

ние, так как команда будет не установлена. Эту проблему мы могли бы решить, проверяя команду на значение null в классе инициатора:

```
class Pult
{
    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)
    {
        command = com;
    }

    public void PressButton()
    {
        if (command != null)
            command.Execute();
    }
    public void PressUndo()
    {
        if (command != null)
            command.Undo();
    }
}
```

Либо можно определить класс пустой команды, которая будет устанавливаться по умолчанию:

```
class NoCommand : ICommand
{
    public void Execute()
    {
    }
    public void Undo()
    {
    }
}
class Pult
{
    ICommand command;

    public Pult()
    {
        command = new NoCommand();
    }

    public void SetCommand(ICommand com)
    {
        command = com;
    }

    public void PressButton()
    {
        command.Execute();
    }
    public void PressUndo()
    {
        command.Undo();
    }
```

```
}
```

При этом инициатор необязательно указывает на одну команду. Он может управлять множеством команд. Например, на пульте от телевизора есть как кнопка для включения, так и кнопки для регулировки звука:

```
class Program
{
    static void Main(string[] args)
    {
        TV tv = new TV();
        Volume volume = new Volume();
        MultiPult mPult = new MultiPult();
        mPult.SetCommand(0, new TVOnCommand(tv));
        mPult.SetCommand(1, new VolumeCommand(volume));
        // включаем телевизор
        mPult.PressButton(0);
        // увеличиваем громкость
        mPult.PressButton(1);
        mPult.PressButton(1);
        mPult.PressButton(1);
        // действия отмены
        mPult.PressUndoButton();
        mPult.PressUndoButton();
        mPult.PressUndoButton();
        mPult.PressUndoButton();

        Console.Read();
    }
}
interface Command
{
    void Execute();
    void Undo();
}

class TV
{
    public void On()
    {
        Console.WriteLine("Телевизор включен!");
    }

    public void Off()
    {
        Console.WriteLine("Телевизор выключен...");
    }
}

class TVOnCommand : ICommand
{
    TV tv;
    public TVOnCommand(TV tvSet)
    {
        tv = tvSet;
    }
    public void Execute()
    {
        tv.On();
    }
```

```
    public void Undo()
    {
        tv.Off();
    }
}
class Volume
{
    public const int OFF = 0;
    public const int HIGH = 20;
    private int level;

    public Volume()
    {
        level = OFF;
    }

    public void RaiseLevel()
    {
        if (level < HIGH)
            level++;
        Console.WriteLine("Уровень звука {0}", level);
    }
    public void DropLevel()
    {
        if (level > OFF)
            level--;
        Console.WriteLine("Уровень звука {0}", level);
    }
}

class VolumeCommand : ICommand
{
    Volume volume;
    public VolumeCommand(Volume v)
    {
        volume = v;
    }
    public void Execute()
    {
        volume.RaiseLevel();
    }

    public void Undo()
    {
        volume.DropLevel();
    }
}

class NoCommand : ICommand
{
    public void Execute()
    {
    }
    public void Undo()
    {
    }
}

class MultiPult
{
    ICommand[] buttons;
    Stack<ICommand> commandsHistory;
```

```
    public MultiPult()
    {
        buttons = new ICommand[2];
        for (int i = 0; i < buttons.Length; i++)
        {
            buttons[i] = new NoCommand();
        }
        commandsHistory = new Stack<ICommand>();
    }

    public void SetCommand(int number, ICommand com)
    {
        buttons[number] = com;
    }

    public void PressButton(int number)
    {
        buttons[number].Execute();
        // добавляем выполненную команду в историю команд
        commandsHistory.Push(buttons[number]);
    }
    public void PressUndoButton()
    {
        if (commandsHistory.Count > 0)
        {
            ICommand undoCommand = commandsHistory.Pop();
            undoCommand.Undo();
        }
    }
}
```

Здесь два получателя команд – классы TV и Volume. Volume управляет уровнем звука и сохраняет текущий уровень в переменной level. Также есть две команды TVOnCommand и VolumeCommand.

Инициатор – MultiPult имеет две кнопки в виде массива buttons: первая предназначена для TV, а вторая – для увеличения уровня звука. Чтобы сохранить историю команд используется стек. При отправке команды в стек добавляется новый элемент, а при ее отмене, наоборот, происходит удаление из стека. В данном случае стек выполняет роль примитивного лога команд.

3.4. Шаблонный метод (Template Method)

Шаблонный метод (Template Method) определяет общий алгоритм поведения подклассов, позволяя им переопределить отдельные шаги этого алгоритма без изменения его структуры.

Когда использовать шаблонный метод?

- Когда планируется, что в будущем подклассы должны будут переопределять различные этапы алгоритма без изменения его структуры.
- Когда в классах, реализующих схожий алгоритм, происходит дублирование кода. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.

Описание паттерна на языке UML приведено на рисунке 11.

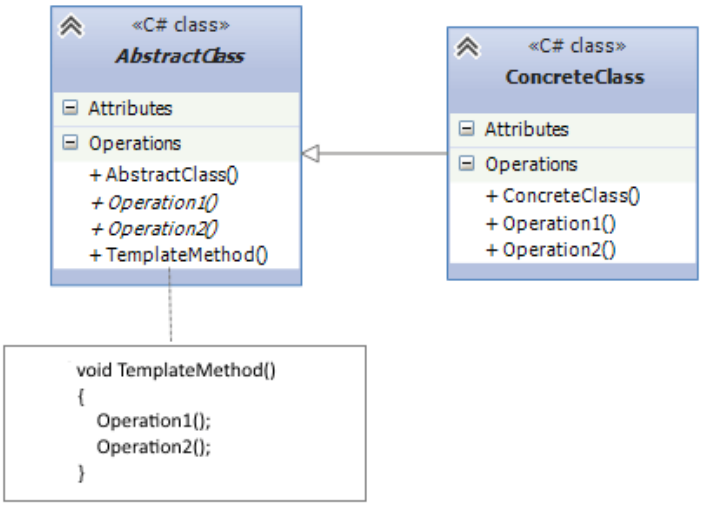


Рис. 11. Шаблонный метод (Template Method)

Формальная реализация паттерна на C#:

```
abstract class AbstractClass
{
    public void TemplateMethod()
    {
        Operation1();
        Operation2();
    }
    public abstract void Operation1();
    public abstract void Operation2();
}

class ConcreteClass : AbstractClass
{
    public override void Operation1()
    {
    }

    public override void Operation2()
    {
    }
}
```

Участники:

- AbstractClass: определяет шаблонный метод TemplateMethod(), который реализует алгоритм. Алгоритм может состоять из последовательности вызовов других методов, часть из которых может быть абстрактными и должны быть переопределены в классах-наследниках. При этом сам метод TemplateMethod(), представляющий структуру алгоритма, переопределяться не должен.
- ConcreteClass: подкласс, который может переопределять различные методы родительского класса.

Рассмотрим применение на конкретном примере. Допустим, в нашей программе используются объекты, представляющие учебу в школе и в вузе:

```
class School
{
    // идем в первый класс
    public void Enter() { }
    // обучение
    public void Study() { }
    // сдаем выпускные экзамены
    public void PassExams() { }
    // получение аттестата об окончании
    public void GetAttestat() { }
}

class University
{
    // поступление в университет
    public void Enter() { }
    // обучение
    public void Study() { }
    // проходим практику
    public void Practice() { }
    // сдаем выпускные экзамены
    public void PassExams() { }
    // получение диплома
    public void GetDiploma() { }
}
```

Как видно, эти классы очень похожи и, самое главное, реализуют примерно общий алгоритм. Да, где-то будет отличаться реализация методов, где-то чуть больше методов, но в целом мы имеем общий алгоритм, а функциональность обоих классов по большому счету дублируется. Поэто-

му для улучшения структуры классов мы могли бы применить шаблонный метод:

```
class Program
{
    static void Main(string[] args)
    {
        School school = new School();
        University university = new University();

        school.Learn();
        university.Learn();

        Console.Read();
    }
}
abstract class Education
{
    public void Learn()
    {
        Enter();
        Study();
        PassExams();
        GetDocument();
    }
    public abstract void Enter();
    public abstract void Study();
    public virtual void PassExams()
    {
        Console.WriteLine("Сдаем выпускные экзамены");
    }
    public abstract void GetDocument();
}

class School : Education
{
    public override void Enter()
    {
        Console.WriteLine("Идем в первый класс");
    }

    public override void Study()
    {
        Console.WriteLine("Посещаем уроки, делаем домашние задания");
    }

    public override void GetDocument()
    {
        Console.WriteLine("Получаем аттестат о среднем образовании");
    }
}

class University : Education
{
    public override void Enter()
    {
        Console.WriteLine("Сдаем вступительные экзамены и поступаем в ВУЗ");
    }

    public override void Study()
    {
```

```
        Console.WriteLine("Посещаем лекции");
        Console.WriteLine("Проходим практику");
    }

    public override void PassExams()
    {
        Console.WriteLine("Сдаем экзамен по специальности");
    }

    public override void GetDocument()
    {
        Console.WriteLine("Получаем диплом о высшем образовании");
    }
}
```

При этом в базовом классе необязательно определять все методы алгоритма как абстрактные. Можно определить реализацию методов по умолчанию, как в случае с методом PassExams().

В данном случае надо отметить, что несмотря на то, что мы не можем в производном классе переопределить шаблонный метод Learn(), тем не менее мы можем скрыть реализацию этого метода в классе-наследнике:

```
class School : Education
{
    public new void Learn()
    {
        Console.WriteLine("Не хочу учиться");
    }
    //.....................................
}
```

В итоге реализация шаблонного метода не будет иметь смысла.

Также надо отметить ситуацию с наследованием базового класса. Например, у нас может быть ситуация, когда базовый класс Education сам наследует метод Learn от другого класса:

```
abstract class Learning
{
    public abstract void Learn();
}
abstract class Education : Learning
{
    public sealed override void Learn()
    {
        Enter();
        Study();
        PassExams();
        GetDocument();
    }
    //  остальные методы класса
}
```

Чтобы сокрыть алгоритм от изменения в классах наследниках, метод Learn объявляется с ключевым словом sealed.

3.5. Итератор (Iterator)

Паттерн Итератор (Iterator) предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры.

Наверное, всем программистам, работающим с языком C#, приходилось иметь дело с циклом foreach, который перебирает объекты в массиве или коллекции. При этом встроенных классов коллекций существует множество, и каждая из них отличается по своей структуре и поведению.

Ключевым моментом, который позволяет осуществить перебор коллекций с помощью foreach, является применения этими классами коллекций паттерна итератор или, проще говоря, пары интерфейсов IEnumerable / IEnumerator. Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

```
public interface IEnumerator
{
    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов
    object Current { get; }  // текущий элемент в контейнере
    void Reset(); // перемещение в начало контейнера
}
```

А интерфейс IEnumerable использует IEnumerator для получения итератора для конкретного типа объекта:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Используя данные интерфейсы, мы можем свести к одному шаблону – с помощью цикла foreach – любые составные объекты.

Когда использовать итераторы?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры.
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора.
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта.

Описание паттерна на языке UML приведено на рисунке 12.

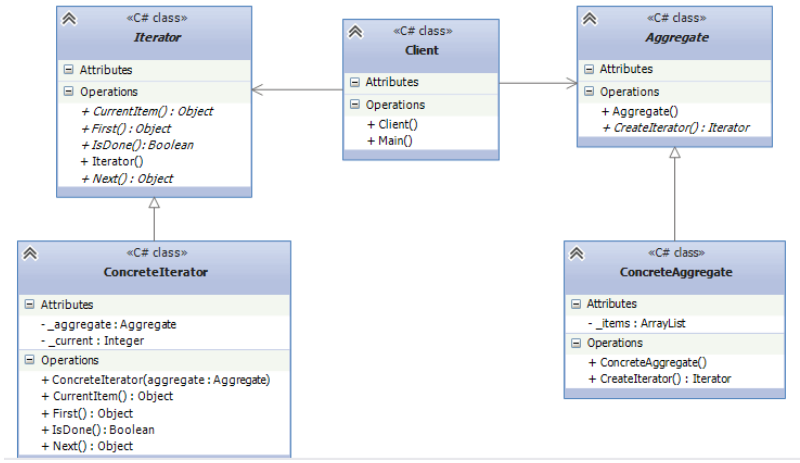


Рис. 12. Итератор (Iterator)

Формальное определение паттерна на C# может выглядеть следующим образом:

```
class Client
{
    public void Main()
    {
        Aggregate a = new ConcreteAggregate();

        Iterator i = a.CreateIterator();

        object item = i.First();
        while (!i.IsDone())
        {
            item = i.Next();
        }
    }
}

abstract class Aggregate
{
```

```
    public abstract Iterator CreateIterator();
    public abstract int Count { get; protected set; }
    public abstract object this[int index] { get; set; }
}

class ConcreteAggregate : Aggregate
{
    private readonly ArrayList _items = new ArrayList();

    public override Iterator CreateIterator()
    {
        return new ConcreteIterator(this);
    }

    public override int Count
    {
        get { return _items.Count; }
        protected set { }
    }

    public override object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

class ConcreteIterator : Iterator
{
    private readonly Aggregate _aggregate;
    private int _current;

    public ConcreteIterator(Aggregate aggregate)
    {
        this._aggregate = aggregate;
    }

    public override object First()
    {
        return _aggregate[0];
    }

    public override object Next()
    {
        object ret = null;

        _current++;

        if (_current < _aggregate.Count)
        {
            ret = _aggregate[_current];
        }

        return ret;
    }
```

```
    public override object CurrentItem()
    {
        return _aggregate[_current];
    }

    public override bool IsDone()
    {
        return _current >= _aggregate.Count;
    }
}
```

Участники:

- Iterator: определяет интерфейс для обхода составных объектов.
- Aggregate: определяет интерфейс для создания объекта-итератора.
- ConcreteIterator: конкретная реализация итератора для обхода объекта Aggregate. Для фиксации индекса текущего перебираемого элемента использует целочисленную переменную _current.
- ConcreteAggregate: конкретная реализация Aggregate. Хранит элементы, которые надо будет перебирать.
- Client: использует объект Aggregate и итератор для его обхода.

Теперь рассмотрим конкретный пример. Допустим, у нас есть классы книги и библиотеки:

```
class Book
{
    public string Name { get; set; }
}
class Library
{
    private Book[] books;
}
```

И, допустим, у нас есть класс читателя, который хочет получить информацию о книгах, которые находятся в библиотеке. И для этого надо осуществить перебор объектов с помощью итератора:

```
class Program
{
    static void Main(string[] args)
    {
        Library library = new Library();
        Reader reader = new Reader();
        reader.SeeBooks(library);

        Console.Read();
    }
}
```

```
class Reader
{
    public void SeeBooks(Library library)
    {
        IBookIterator iterator = library.CreateNumerator();
        while (iterator.HasNext())
        {
            Book book = iterator.Next();
            Console.WriteLine(book.Name);
        }
    }
}

interface IBookIterator
{
    bool HasNext();
    Book Next();
}
interface IBookNumerable
{
    IBookIterator CreateNumerator();
    int Count { get; }
    Book this[int index] { get; }
}
class Book
{
    public string Name { get; set; }
}

class Library : IBookNumerable
{
    private Book[] books;
    public Library()
    {
        books = new Book[]
        {
            new Book{Name="Война и мир"},
            new Book {Name="Отцы и дети"},
            new Book {Name="Вишневый сад"}
        };
    }
    public int Count
    {
        get { return books.Length; }
    }

    public Book this[int index]
    {
        get { return books[index]; }
    }
    public IBookIterator CreateNumerator()
    {
        return new LibraryNumerator(this);
    }
}
class LibraryNumerator : IBookIterator
{
    IBookNumerable aggregate;
    int index = 0;
    public LibraryNumerator(IBookNumerable a)
    {
        aggregate = a;
```

```
    }
    public bool HasNext()
    {
        return index < aggregate.Count;
    }

    public Book Next()
    {
        return aggregate[index++];
    }
}
```

Интерфейс IBookIterator представляет итератор наподобие интерфейса IEnumerator. Роль интерфейса составного агрегата представляет тип IBookNumerable. Клиентом здесь является класс Reader, который использует итератор для обхода объекта библиотеки.

4. СТРУКТУРНЫЕ ПАТТЕРНЫ

4.1. Декоратор (Decorator)

Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляют наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

Когда следует использовать декораторы?

- Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта.
- Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реа-

лизующие все возможные сочетания добавляемых функциональностей.

Описание паттерна на языке UML приведено на рисунке 13.

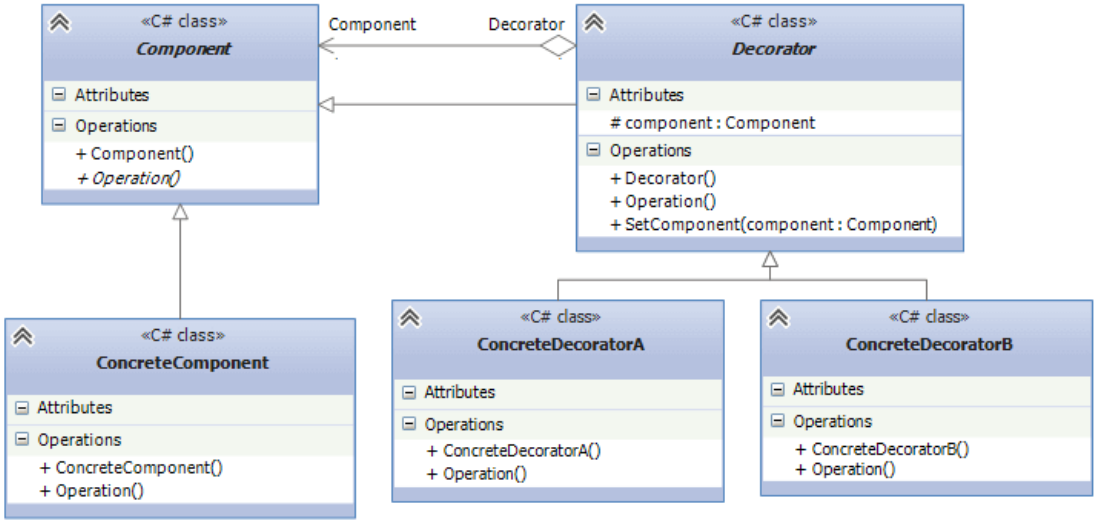


Рис. 13. Декоратор (Decorator)

Формальная организация паттерна в C# могла бы выглядеть следующим образом:

```
abstract class Component
{
    public abstract void Operation();
}
class ConcreteComponent : Component
{
    public override void Operation()
    { }
}
abstract class Decorator : Component
{
    protected Component component;

    public void SetComponent(Component component)
    {
        this.component = component;
    }

    public override void Operation()
    {
        if (component != null)
            component.Operation();
    }
}
class ConcreteDecoratorA : Decorator
{
    public override void Operation()
    {
        base.Operation();
    }
```

```
}
class ConcreteDecoratorB : Decorator
{
    public override void Operation()
    {
        base.Operation();
    }
}
```

Участники:

- Component: абстрактный класс, который определяет интерфейс для наследуемых объектов.
- ConcreteComponent: конкретная реализация компонента, в которую с помощью декоратора добавляется новая функциональность.
- Decorator: собственно декоратор, реализуется в виде абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Поэтому базовый класс Component должен быть по возможности легким и определять только базовый интерфейс.
- Класс декоратора также хранит ссылку на декорируемый объект в виде объекта базового класса Component и реализует связь с базовым классом как через наследование, так и через отношение агрегации.
- Классы ConcreteDecoratorA и ConcreteDecoratorB представляют дополнительные функциональности, которыми должен быть расширен объект ConcreteComponent.

Рассмотрим пример. Допустим, у нас есть пиццерия, которая готовит различные типы пицц с различными добавками. Есть итальянская, болгарская пиццы. К ним могут добавляться помидоры, сыр и т.д. И в зависимости от типа пицц и комбинаций добавок пицца может иметь разную стоимость. Теперь посмотрим, как это изобразить в программе на C#:

```
class Program
{
    static void Main(string[] args)
    {
        Pizza pizza1 = new ItalianPizza();
        pizza1 = new TomatoPizza(pizza1); // итальянская пицца с томатами
        Console.WriteLine("Название: {0}", pizza1.Name);
        Console.WriteLine("Цена: {0}", pizza1.GetCost());
```

```
        Pizza pizza2 = new ItalianPizza();
        pizza2 = new CheesePizza(pizza2);// итальянская пиццы с сыром
        Console.WriteLine("Название: {0}", pizza2.Name);
        Console.WriteLine("Цена: {0}", pizza2.GetCost());

        Pizza pizza3 = new BulgerianPizza();
        pizza3 = new TomatoPizza(pizza3);
        pizza3 = new CheesePizza(pizza3);// болгарская пиццы с томатами и сыром
        Console.WriteLine("Название: {0}", pizza3.Name);
        Console.WriteLine("Цена: {0}", pizza3.GetCost());

        Console.ReadLine();
    }
}

abstract class Pizza
{
    public Pizza(string n)
    {
        this.Name = n;
    }
    public string Name { get; protected set; }
    public abstract int GetCost();
}

class ItalianPizza : Pizza
{
    public ItalianPizza() : base("Итальянская пицца")
    { }
    public override int GetCost()
    {
        return 10;
    }
}
class BulgerianPizza : Pizza
{
    public BulgerianPizza()
        : base("Болгарская пицца")
    { }
    public override int GetCost()
    {
        return 8;
    }
}

abstract class PizzaDecorator : Pizza
{
    protected Pizza pizza;
    public PizzaDecorator(string n, Pizza pizza) : base(n)
    {
        this.pizza = pizza;
    }
}

class TomatoPizza : PizzaDecorator
{
    public TomatoPizza(Pizza p)
        : base(p.Name + ", с томатами", p)
    { }

    public override int GetCost()
```

```
    {
        return pizza.GetCost() + 3;
    }
}

class CheesePizza : PizzaDecorator
{
    public CheesePizza(Pizza p)
        : base(p.Name + ", с сыром", p)
    { }

    public override int GetCost()
    {
        return pizza.GetCost() + 5;
    }
}
```

В качестве компонента здесь выступает абстрактный класс Pizza, который определяет базовую функциональность в виде свойства Name и метода GetCost(). Эта функциональность реализуется двумя подклассами ItalianPizza и BulgerianPizza, в которых жестко закодированы название пиццы и ее цена.

Декоратором является абстрактный класс PizzaDecorator, который унаследован от класса Pizza и содержит ссылку на декорируемый объект Pizza. В отличие от формальной схемы здесь установка декорируемого объекта происходит не в методе SetComponent, а в конструкторе.

Отдельные функциональности – добавление томатов и сыра к пиццам реализованы через классы TomatoPizza и CheesePizza, которые обертывают объект Pizza и добавляют к его имени название добавки, а к цене – стоимость добавки, то есть переопределяя метод GetCost и изменяя значение свойства Name.

Благодаря этому при создании пиццы с добавками произойдет ее обертывание декоратором:

```
Pizza pizza3 = new BulgerianPizza();
pizza3 = new TomatoPizza(pizza3);
pizza3 = new CheesePizza(pizza3);
```

Сначала объект BulgerianPizza обертывается декоратором TomatoPizza, а затем CheesePizza. И таких обертываний мы можем сделать множество. Просто достаточно унаследовать от декоратора класс, который будет определять дополнительный функционал.

А если бы мы использовали наследование, то в данном случае только для двух видов пицц с двумя добавками нам бы пришлось создать восемь различных классов, которые бы описывали все возможные комбинации. Поэтому декораторы являются более предпочтительным в данном случае методом.

4.2. Адаптер (Adapter)

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

Когда надо использовать Адаптер?

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям.
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы.

Описание паттерна на языке UML приведено на рисунке 14.

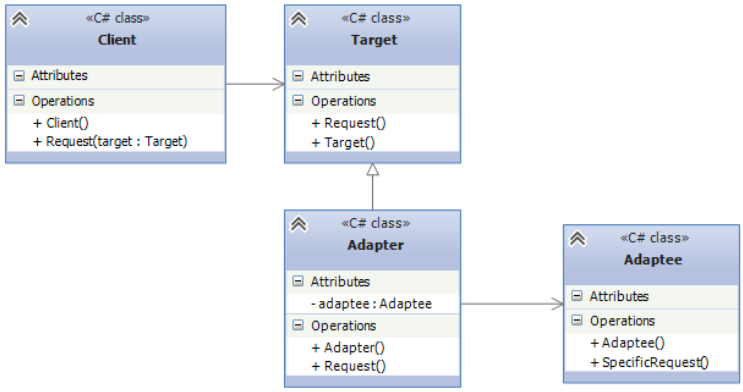


Рис. 14. Адаптер (Adapter)

Формальное описание адаптера объектов на C# выглядит таким образом:

```
class Client
{
    public void Request(Target target)
    {
        target.Request();
```

```
    }
}
// класс, к которому надо адаптировать другой класс
class Target
{
    public virtual void Request()
    { }
}

// Адаптер
class Adapter : Target
{
    private Adaptee adaptee = new Adaptee();

    public override void Request()
    {
        adaptee.SpecificRequest();
    }
}

// Адаптируемый класс
class Adaptee
{
    public void SpecificRequest()
    { }
}
```

Участники:

- Target: представляет объекты, которые используются клиентом.
- Client: использует объекты Target для реализации своих задач.
- Adaptee: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target.
- Adapter: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target.

Теперь разберем реальный пример. Допустим, у нас есть путешественник, который путешествует на машине. Но в какой-то момент ему приходится передвигаться по пескам пустыни, где он не может ехать на машине. Зато он может использовать для передвижения верблюда. Однако в классе путешественника использование класса верблюда не предусмотрено, поэтому нам надо использовать адаптер:

```
class Program
```

```
{
    static void Main(string[] args)
    {
        // путешественник
        Driver driver = new Driver();
        // машина
        Auto auto = new Auto();
        // отправляемся в путешествие
        driver.Travel(auto);
        // встретились пески, надо использовать верблюда
        Camel camel = new Camel();
        // используем адаптер
        ITransport camelTransport = new CamelToTransportAdapter(camel);
        // продолжаем путь по пескам пустыни
        driver.Travel(camelTransport);

        Console.Read();
    }
}
interface ITransport
{
    void Drive();
}
// класс машины
class Auto : ITransport
{
    public void Drive()
    {
        Console.WriteLine("Машина едет по дороге");
    }
}
class Driver
{
    public void Travel(ITransport transport)
    {
        transport.Drive();
    }
}
// интерфейс животного
interface IAnimal
{
    void Move();
}
// класс верблюда
class Camel : IAnimal
{
    public void Move()
    {
        Console.WriteLine("Верблюд идет по пескам пустыни");
    }
}
// Адаптер от Camel к ITransport
class CamelToTransportAdapter : ITransport
{
    Camel camel;
    public CamelToTransportAdapter(Camel c)
    {
        camel = c;
    }

    public void Drive()
    {
```

```
        camel.Move();
    }
}
```

В данном случае в качестве клиента применяется класс Driver, который использует объект ITransport. Адаптируемым является класс верблюда Camel, который нужно использовать в качестве объекта ITransport. И адптером служит класс CamelToTransportAdapter.

Возможно, кому-то покажется надуманной проблема использования адаптеров особенно в данном случае, так как мы могли бы применить интерфейс ITransport к классу Camel и реализовать его метод Drive(). Однако в данном случае может случиться дублирование функциональностей: интерфейс IAnimal имеет метод Move(), реализация которого в классе верблюда могла бы быть похожей на реализацию метода Drive() из интерфейса ITransport. Кроме того, нередко бывает, что классы спроектированы кем-то другим, и мы никак не можем на них повлиять. Мы только используем их. В результате чего адаптеры довольно широко распространены в .NET. В частности, многочисленные встроенные классы, которые используются для подключения к различным системам баз данных, как раз и реализуют паттерн адаптер (например, класс System.Data.SqlClient.SqlDataAdapter).

4.3. Фасад (Facade)

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

Когда использовать фасад?

- Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.

- Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
- Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

Описание паттерна на языке UML приведено на рисунке 15.

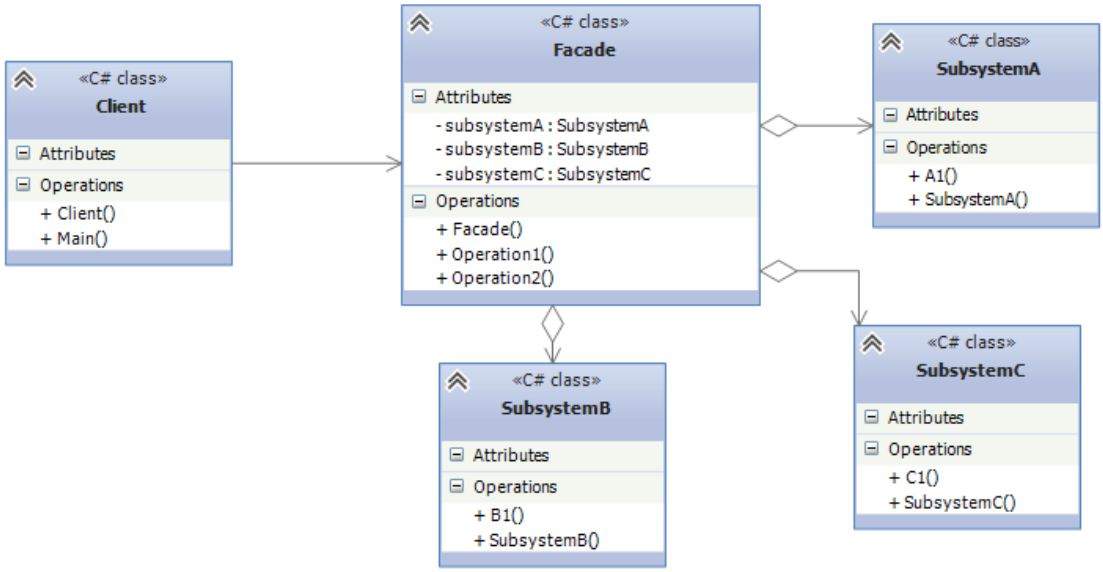


Рис. 15. Фасад (Facade)

Формальное определение программы в C# может выглядеть так:

```
class SubsystemA
{
    public void A1()
    { }
}
class SubsystemB
{
    public void B1()
    { }
}
class SubsystemC
{
    public void C1()
    { }
}

public class Facade
{
```

```
    SubsystemA subsystemA;
    SubsystemB subsystemB;
    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc)
    {
        subsystemA = sa;
        subsystemB = sb;
        subsystemC = sc;
    }
    public void Operation1()
    {
        subsystemA.A1();
        subsystemB.B1();
        subsystemC.C1();
    }
    public void Operation2()
    {
        subsystemB.B1();
        subsystemC.C1();
    }
}

class Client
{
    public void Main()
    {
        Facade facade = new Facade(new SubsystemA(), new SubsystemB(), new Subsys-
temC());
        facade.Operation1();
        facade.Operation2();
    }
}
```

Участники:

- Классы SubsystemA, SubsystemB, SubsystemC и т.д. являются компонентами сложной подсистемы, с которыми должен взаимодействовать клиент.
- Client взаимодействует с компонентами подсистемы.
- Facade – непосредственно фасад, который предоставляет интерфейс клиенту для работы с компонентами.

Рассмотрим применение паттерна в реальной задаче. Думаю, большинство программистов согласятся со мной, что писать в Visual Studio код одно удовольствие по сравнению с тем, как писался код ранее до появления интегрированных сред разработки. Мы просто пишем код, нажимаем на кнопку, и все – приложение готово. В данном случае интегрированная среда разработки представляет собой фасад, который скрывает всю слож-

ность процесса компиляции и запуска приложения. Теперь опишем этот фасад в программе на C#:

```
class Program
{
    static void Main(string[] args)
    {
        TextEditor textEditor = new TextEditor();
        Compiller compiller = new Compiller();
        CLR clr = new CLR();

        VisualStudioFacade ide = new VisualStudioFacade(textEditor, compiller, clr);

        Programmer programmer = new Programmer();
        programmer.CreateApplication(ide);

        Console.Read();
    }
}
// текстовый редактор
class TextEditor
{
    public void CreateCode()
    {
        Console.WriteLine("Написание кода");
    }
    public void Save()
    {
        Console.WriteLine("Сохранение кода");
    }
}
class Compiller
{
    public void Compile()
    {
        Console.WriteLine("Компиляция приложения");
    }
}
class CLR
{
    public void Execute()
    {
        Console.WriteLine("Выполнение приложения");
    }
    public void Finish()
    {
        Console.WriteLine("Завершение работы приложения");
    }
}

class VisualStudioFacade
{
    TextEditor textEditor;
    Compiller compiller;
    CLR clr;
    public VisualStudioFacade(TextEditor te, Compiller compil, CLR clr)
    {
        this.textEditor = te;
        this.compiller = compil;
        this.clr = clr;
    }
```

```
    public void Start()
    {
        textEditor.CreateCode();
        textEditor.Save();
        compiller.Compile();
        clr.Execute();
    }
    public void Stop()
    {
        clr.Finish();
    }
}

class Programmer
{
    public void CreateApplication(VisualStudioFacade facade)
    {
        facade.Start();
        facade.Stop();
    }
}
```

В данном случае компонентами системы являются: класс текстового редактора TextEditor, класс компилятора Compiller и класс общеязыковой среды выполнения CLR. Клиентом выступает класс программиста, фасадом – класс VisualStudioFacade, который через свои методы делегирует выполнение работы компонентам и их методам.

При этом надо учитывать, что клиент может при необходимости обращаться напрямую к компонентам, например, отдельно от других компонентов использовать текстовый редактор. Но в виду сложности процесса создания приложения лучше использовать фасад. Также это не единственный возможный фасад для работы с данными компонентами. При необходимости можно создавать альтернативные фасады так же, как в реальной жизни мы можем использовать альтернативные среды разработки.

4.4. Компоновщик (Composite)

Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое" и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит, с одной стороны, частью меню, а с другой стороны, еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.

Когда использовать компоновщик?

- Когда объекты должны быть реализованы в виде иерархической древовидной структуры.

- Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс.

Описание паттерна на языке UML приведено на рисунке 16.

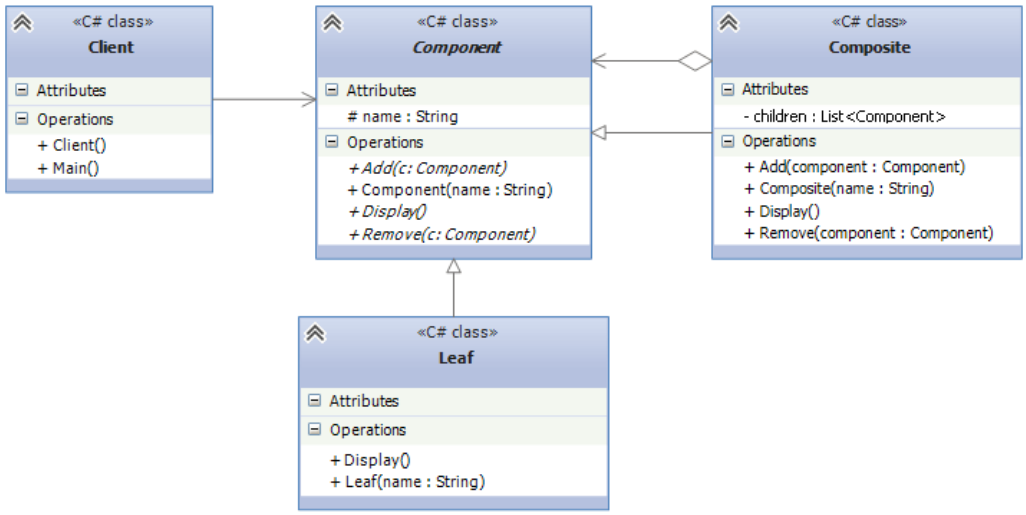


Рис. 16. Компоновщик (Composite)

Формальное определение паттерна на C# могло бы выглядеть так:

```
class Client
{
    public void Main()
    {
        Component root = new Composite("Root");
        Component leaf = new Leaf("Leaf");
        Composite subtree = new Composite("Subtree");
        root.Add(leaf);
        root.Add(subtree);
```

```
        root.Display();
    }
}
abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Display();
    public abstract void Add(Component c);
    public abstract void Remove(Component c);
}
class Composite : Component
{
    List<Component> children = new List<Component>();

    public Composite(string name)
        : base(name)
    { }

    public override void Add(Component component)
    {
        children.Add(component);
    }

    public override void Remove(Component component)
    {
        children.Remove(component);
    }

    public override void Display()
    {
        Console.WriteLine(name);

        foreach (Component component in children)
        {
            component.Display();
        }
    }
}
class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    { }

    public override void Display()
    {
        Console.WriteLine(name);
    }

    public override void Add(Component component)
    {
        throw new NotImplementedException();
    }

    public override void Remove(Component component)
    {
```

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

Участники:

- Component: определяет интерфейс для всех компонентов в древовидной структуре.
- Composite: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления.
- Leaf: представляет отдельный компонент, который не может содержать другие компоненты.
- Client: клиент, который использует компоненты.

Рассмотрим простейший пример. Допустим, нам надо создать объект файловой системы. Файловую систему составляют папки и файлы. Каждая папка также может включать в себя папки и файлы. То есть получается древовидная иерархическая структура, где с вложенными папками нам надо работать также, как и с папками, которые их содержат. Для реализации данной задачи и воспользуемся паттерном Компоновщик:

```
class Program
{
    static void Main(string[] args)
    {
        Component fileSystem = new Directory("Файловая система");
        // определяем новый диск
        Component diskC = new Directory("Диск C");
        // новые файлы
        Component pngFile = new File("12345.png");
        Component docxFile = new File("Document.docx");
        // добавляем файлы на диск C
        diskC.Add(pngFile);
        diskC.Add(docxFile);
        // добавляем диск C в файловую систему
        fileSystem.Add(diskC);
        // выводим все данные
        fileSystem.Print();
        Console.WriteLine();
        // удаляем с диска C файл
        diskC.Remove(pngFile);
        // создаем новую папку
        Component docsFolder = new Directory("Мои Документы");
        // добавляем в нее файлы
        Component txtFile = new File("readme.txt");
        Component csFile = new File("Program.cs");
        docsFolder.Add(txtFile);
```

```
        docsFolder.Add(csFile);
        diskC.Add(docsFolder);

        fileSystem.Print();

        Console.Read();
    }
}

abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
    }

    public virtual void Add(Component component) { }

    public virtual void Remove(Component component) { }

    public virtual void Print()
    {
        Console.WriteLine(name);
    }
}
class Directory : Component
{
    private List<Component> components = new List<Component>();

    public Directory(string name)
        : base(name)
    {
    }

    public override void Add(Component component)
    {
        components.Add(component);
    }

    public override void Remove(Component component)
    {
        components.Remove(component);
    }

    public override void Print()
    {
        Console.WriteLine("Узел " + name);
        Console.WriteLine("Подузлы:");
        for (int i = 0; i < components.Count; i++)
        {
            components[i].Print();
        }
    }
}

class File : Component
{
    public File(string name)
            : base(name)
    { }
```

```
}
```

В итоге подобная система обладает неплохой гибкостью: если мы захотим добавить новый вид компонентов, нам достаточно унаследовать новый класс от Component.

И также применяя компоновщик, мы легко можем обойти все узлы древовидной структуры.

5. ПРИНЦИПЫ SOLID

Термин "SOLID" представляет собой акроним для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен известным американским специалистом в области программирования Робертом Мартином (Robert Martin), более известным как "дядюшка Боб" или Uncle Bob.

Сам акроним образован по первым буквам названий SOLID-принципов:

- Single Responsibility Principle (Принцип единственной обязанности);
- Open/Closed Principle (Принцип открытости/закрытости);
- Liskov Substitution Principle (Принцип подстановки Лисков);
- Interface Segregation Principle (Принцип разделения интерфейсов);
- Dependency Inversion Principle (Принцип инверсии зависимостей).

Принципы SOLID – это не паттерны, их нельзя назвать какими-то определенными догмами, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.

5.1. Принцип единственной обязанности

Принцип единственной обязанности (Single Responsibility Principle) можно сформулировать так: каждый компонент должен иметь одну и только одну причину для изменения.

В C# в качестве компонента может выступать класс, структура, метод. А под обязанностью здесь понимается набор действий, которые выполняют единую задачу. То есть суть принципа заключается в том, что класс/структура/метод должны выполнять одну единственную задачу. Весь функционал компонента должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется компонент. Если он выполняет несколько различных действий, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у компонента несколько причин для изменения.

Допустим, нам надо определить класс отчета, по которому мы можем перемещаться по страницам и который можно выводить на печать. На первый взгляд мы могли бы определить следующий класс:

```
class Report
{
    public string Text { get; set; } = "";
    public void GoToFirstPage() =>
        Console.WriteLine("Переход к первой странице");

    public void GoToLastPage() =>
        Console.WriteLine("Переход к последней странице");

    public void GoToPage(int pageNumber) =>
        Console.WriteLine($"Переход к странице {pageNumber}");


    public void Print()
    {
        Console.WriteLine("Печать отчета");
        Console.WriteLine(Text);
    }
}
```

Ключевым понятием применительно к данному принципу является cohesion или связность/согласованность. Это понятие описывает,

насколько близко связаны компоненты. Чем больше связность между компонентами, тем больше программа соответствует принципу единой ответственности

Например, первые три метода класса относятся к навигации по отчету и представляют одно единое функциональное целое, обладают высокой связностью. От них отличается метод Print, который производит печать. Что если нам понадобится печатать отчет на консоль или передать его на принтер для физической печати на бумаге? Или вывести в файл? Сохранить в формате html, txt, rtf и т.д.? Очевидно, что мы можем для этого поменять нужным образом метод Print(). Однако это вряд ли затронет остальные методы, которые относятся к навигации страницы.

Также верно и обратное – изменение методов постраничной навигации вряд ли повлияет на возможность вывода текста отчета на принтер или на консоль. Таким образом, у нас здесь прослеживаются две причины для изменения, значит, класс Report обладает двумя обязанностями, и от одной из них этот класс надо освободить. Решением было бы вынести каждую обязанность в отдельный компонент (в данном случае в отдельный класс):

```
class Report
{
    public string Text { get; set; } = "";
    public void GoToFirstPage() =>
        Console.WriteLine("Переход к первой странице");

    public void GoToLastPage() =>
        Console.WriteLine("Переход к последней странице");

    public void GoToPage(int pageNumber) =>
        Console.WriteLine($"Переход к странице {pageNumber}");
}
//  обязанность - печать отчета
class Printer
{
    public void PrintReport(Report report)
    {
        Console.WriteLine("Печать отчета");
        Console.WriteLine(report.Text);
    }
}
```

Теперь печать вынесена в отдельный класс Printer, который через метод Print получает объект отчета и выводит его текст на консоль.

Распространенные случаи отхода от принципа SRP. Нередко принцип единственной обязанности нарушает при смешивании в одном классе функциональности разных уровней. Например, класс производит вычисления и выводит их пользователю, то есть соединяет в себя бизнес-логику и работу с пользовательским интерфейсом. Либо класс управляет сохранением/получением данных и выполнением над ними вычислений, что также нежелательно. Класс следует применять только для одной задачи – либо бизнес-логика, либо вычисления, либо работа с данными.

Другой распространенный случай – наличие в классе или его методах абсолютно несвязанного между собой функционала.

Распространенные сценарии выделения компонентов. Есть ряд распространенных сценариев, которые обычно выносятся в отдельные компоненты:

- Логика хранения данных;
- Валидация;
- Механизм уведомлений пользователя;
- Обработка ошибок;
- Логирование;
- Выбор класса или создание его объекта;
- Форматирование;
- Парсинг;
- Маппинг данных.

5.2. Принцип открытости/закрытости

Принцип открытости/закрытости (Open/Closed Principle) можно сформулировать так:

Сущности программы должны быть открыты для расширения, но закрыты для изменения.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

Рассмотрим простейший пример – класс повара:

```
class Cook
{
    public string Name { get; set; }
    public Cook(string name)
    {
        this.Name = name;
    }

    public void MakeDinner()
    {
        Console.WriteLine("Чистим картошку");
        Console.WriteLine("Ставим почищенную картошку на огонь");
        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пю-
ре");
        Console.WriteLine("Посыпаем пюре специями и зеленью");
        Console.WriteLine("Картофельное пюре готово");
    }
}
```

И с помощью метода MakeDinner любой объект данного класса сможет сделать картофельного пюре:

```
Cook bob = new Cook("Bob");
bob.MakeDinner();
```

Однако одного умения готовить картофельное пюре для повара вряд ли достаточно. Хотелось бы, чтобы повар мог приготовить еще что-то. И в этом случае мы подходим к необходимости изменения функционала класса, а именно метода MakeDinner. Но в соответствии с рассматриваемым нами принципом классы должны быть открыты для расширения, но закрыты для изменения. То есть, нам надо сделать класс Cook отрытым для расширения, но при этом не изменять.

Для решения этой задачи мы можем воспользоваться паттерном Стратегия. В первую очередь нам надо вынести из класса и

инкапсулировать всю ту часть, которая представляет изменяющееся поведение. В нашем случае это метод MakeDinner. Однако это не всегда бывает просто сделать. Возможно, в классе много методов, но на начальном этапе сложно определить, какие из них будут изменять свое поведение и как изменять. В этом случае, конечно, надо анализировать возможные способы изменения и уже на основании анализа делать выводы. То есть все, что поддается изменению, выносится из класса и инкапсулируется во вне – во внешних сущностях.

Итак, изменим класс Cook следующим образом:

```
class Cook
{
    public string Name { get; set; }

    public Cook(string name)
    {
        this.Name = name;
    }

    public void MakeDinner(IMeal meal)
    {
        meal.Make();
    }
}

interface IMeal
{
    void Make();
}
class PotatoMeal : IMeal
{
    public void Make()
    {
        Console.WriteLine("Чистим картошку");
        Console.WriteLine("Ставим почищенную картошку на огонь");
        Console.WriteLine("Сливаем остатки воды, разминаем вареный картофель в пю-
ре");
        Console.WriteLine("Посыпаем пюре специями и зеленью");
        Console.WriteLine("Картофельное пюре готово");
    }
}
class SaladMeal : IMeal
{
    public void Make()
    {
        Console.WriteLine("Нарезаем помидоры и огурцы");
        Console.WriteLine("Посыпаем зеленью, солью и специями");
        Console.WriteLine("Поливаем подсолнечным маслом");
        Console.WriteLine("Салат готов");
    }
}
```

Теперь приготовление еды абстрагировано в интерфейсе IMeal, а конкретные способы приготовления определены в реализациях этого интерфейса. А класс Cook делегирует приготовление еды методу Make объекта IMeal.

Использование класса:

```
Cook bob = new Cook("Bob");
bob.MakeDinner(new PotatoMeal());
Console.WriteLine();
bob.MakeDinner(new SaladMeal());
```

Теперь класс Cook закрыт от изменений, зато мы можем легко расширить его функциональность, определив дополнительные реализации интерфейса IMeal.

Другим распространенным способом применения принципа открытости/закрытости представляет паттерн Шаблонный метод. Переделаем предыдущую задачу с помощью этого паттерна:

```
abstract class MealBase
{
    public void Make()
    {
        Prepare();
        Cook();
        FinalSteps();
    }
    protected abstract void Prepare();
    protected abstract void Cook();
    protected abstract void FinalSteps();
}

class PotatoMeal : MealBase
{
    protected override void Cook()
    {
        Console.WriteLine("Ставим почищенную картошку на огонь");
        Console.WriteLine("Варим около 30 минут");
        Console.WriteLine("Сливаем остатки воды, разминаем вареный картофель в пю-
ре");
    }

    protected override void FinalSteps()
    {
        Console.WriteLine("Посыпаем пюре специями и зеленью");
        Console.WriteLine("Картофельное пюре готово");
    }

    protected override void Prepare()
```

```
    {
        Console.WriteLine("Чистим и моем картошку");
    }
}

class SaladMeal : MealBase
{
    protected override void Cook()
    {
        Console.WriteLine("Нарезаем помидоры и огурцы");
        Console.WriteLine("Посыпаем зеленью, солью и специями");
    }

    protected override void FinalSteps()
    {
        Console.WriteLine("Поливаем подсолнечным маслом");
        Console.WriteLine("Салат готов");
    }

    protected override void Prepare()
    {
        Console.WriteLine("Моем помидоры и огурцы");
    }
}
```

Теперь абстрактный класс MealBase определяет шаблонный метод Make, отдельные части которого реализуются классами наследниками.

Пусть класс Cook теперь принимает набор MealBase в виде меню:

```
class Cook
{
    public string Name { get; set; }

    public Cook(string name, )
    {
        this.Name = name;
    }

    public void MakeDinner(MealBase[] menu)
    {
        foreach (MealBase meal in menu)
            meal.Make();
    }
}
```

В данном случае расширение класса опять же производится за счет наследования классов, которые определяют требуемый функционал. Применим классы:

```
MealBase[] menu = new MealBase[] { new PotatoMeal(), new SaladMeal() };

Cook bob = new Cook("Bob");
bob.MakeDinner(menu);
```

5.3. Принцип подстановки Лисков

Принцип подстановки Лисков (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом: если для каждого объекта o_1 типа S существует объект o_2 типа T, такой, что для любой программы P, определенной в терминах T, поведение P не изменяется при замене o_2 на o_1, то S является подтипом T.

То есть иными словами, класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так: должна быть возможность вместо базового типа подставить любой его подтип.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

Проблему, с который связан принцип Лисков, наглядно можно продемонстрировать на примере двух классов Прямоугольника и Квадрата. Пусть они будут выглядеть следующим образом:

```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle
{
    public override int Width
    {
        get
        {
            return base.Width;
        }
```

```
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override int Height
    {
        get
        {
            return base.Height;
        }

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

Как правило, квадрат представляют как частный случай прямоугольника – те же прямые углы, четыре стороны, только ширина обязательно равна высоте. Поэтому в классе Квадрат у одного свойства устанавливаются сразу и ширина, и высота:

```
set
{
    base.Height = value;
    base.Width = value;
}
```

На первый взгляд вроде все правильно, классы предельно простые, всего два свойства, и, казалось бы, сложно где-то ошибиться. Однако представим ситуацию, что в главной программе у нас следующий код:

```
class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = new Square();
        TestRectangleArea(rect);

        Console.Read();
    }

    public static void TestRectangleArea(Rectangle rect)
    {
        rect.Height = 5;
        rect.Width = 10;
        if (rect.GetArea() != 50)
            throw new Exception("Некорректная площадь!");
    }
}
```

С точки зрения прямоугольника метод TestRectangleArea выглядит нормально, но не с точки зрения квадрата. Мы ожидаем, что переданный в метод TestRectangleArea объект будет вести себя как стандартный прямоугольник. Однако квадрат, будучи в иерархии наследования прямоугольником, все же ведет себя не как прямоугольник. В итоге программа вывалится в ошибку.

Иногда для выхода из подобных ситуаций прибегают к специальному хаку, который заключается в проверке объекта на соответствие типам:

```
public static void TestRectangleArea(Rectangle rect)
{
    if (rect is Square)
    {
        rect.Height = 5;
        if (rect.GetArea() != 25)
            throw new Exception("Неправильная площадь!");
    }
    else if (rect is Rectangle)
    {
        rect.Height = 5;
        rect.Width = 10;
        if (rect.GetArea() != 50)
            throw new Exception("Неправильная площадь!");
    }
}
```

Но такая проверка не отменяет того факта, что с архитектурой классов что-то не так. Более того, такие решения только больше подчеркивают проблему несовершенства архитектуры. И проблема заключается в том, что производный класс Square не ведет себя как базовый класс Rectangle, и поэтому его не следует наследовать от данного базового класса. В этом и есть практический смысл принципа Лисков. Производный класс, который может делать меньше, чем базовый, обычно нельзя подставить вместо базового, и поэтому он нарушает принцип подстановки Лисков.

Существует несколько типов правил, которые должны быть соблюдены для выполнения принципа подстановки Лисков. Прежде всего, это правила контракта.

Контракт представляет собой некоторый интерфейс базового класса, некоторые соглашения по его использованию, которым должен следовать класс-наследник. Контракт задает ряд ограничений или правил, и производный класс должен выполнять эти правила:

• **Предусловия (Preconditions)** не могут быть усилены в подклассе. Другими словами подклассы не должны создавать больше предусловий, чем это определено в базовом классе, для выполнения некоторого поведения. Предусловия представляют набор условий, необходимых для безошибочного выполнения метода. Например:

```
public virtual void SetCapital(int money)
{
    if (money < 0)
        throw new Exception("Нельзя положить на счет меньше 0");
    this.Capital = money;
}
```

Здесь условное выражение служит предусловием – без его выполнения не будут выполняться остальные действия, а метод завершится с ошибкой.

Причем объектом предусловий могут быть только общедоступные свойства или поля класса или параметры метода, как в данном случае. Приватное поле не может быть объектом для предусловия, так как оно не может быть установлено из вызывающего кода. Например, в следующем случае условное выражение не является предусловием:

```
private bool isValid = false
public virtual void SetCapital(int money)
{
    if (isValid == false)
        throw new Exception("Валидация не пройдена");
    this.Capital = money;
}
```

Теперь, допустим, есть два класса: Account (общий счет) и MicroAccount (мини-счет с ограничениями). И второй класс переопределяет метод SetCapital:

```
class Account
{
    public int Capital { get; protected set; }

    public virtual void SetCapital(int money)
```

```
    {
        if (money < 0)
            throw new Exception("Нельзя положить на счет меньше 0");
        this.Capital = money;
    }
}

class MicroAccount : Account
{
    public override void SetCapital(int money)
    {
        if (money < 0)
            throw new Exception("Нельзя положить на счет меньше 0");

        if (money > 100)
            throw new Exception("Нельзя положить на счет больше 100");

        this.Capital = money;
    }
}
```

В этом случае подкласс MicroAccount добавляет дополнительное предусловие, то есть усиливает его, что недопустимо. Поэтому в реальной задаче мы можем столкнуться с проблемой:

```
class Program
{
    static void Main(string[] args)
    {
        Account acc = new MicroAccount();
        InitializeAccount(acc);

        Console.Read();
    }

    public static void InitializeAccount(Account account)
    {
        account.SetCapital(200);
        Console.WriteLine(account.Capital);
    }
}
```

С точки зрения класса Account метод InitializeAccount() вполне является работоспособным. Однако при передаче в него объекта MicroAccount мы столкнемся с ошибкой. В итоге принцип Лисков будет нарушен.

- **Постусловия (Postconditions)** не могут быть ослаблены в подклассе. То есть подклассы должны выполнять все постусловия, которые определены в базовом классе. Постусловия проверяют состояние возвращаемого объекта на выходе из функции. Например:

```
public static float GetMedium(float[] numbers)
```

```
{
    if (numbers.Length == 0)
        throw new Exception("длина массива равна нулю");

    float result = numbers.Sum() / numbers.Length;

    if (result < 0)
        throw new Exception("Результат меньше нуля");
    return result;
}
```

Второе условное выражение здесь является постусловием. Рассмотрим пример нарушения принципа Лисков при ослаблении постусловия:

```
class Account
{
    public virtual decimal GetInterest(decimal sum, int month, int rate)
    {
        // предусловие
        if (sum < 0 || month > 12 || month < 1 || rate < 0)
            throw new Exception("Некорректные данные");

        decimal result = sum;
        for (int i = 0; i < month; i++)
            result += result * rate / 100;

        // постусловие
        if (sum >= 1000)
            result += 100; // добавляем бонус

        return result;
    }
}

class MicroAccount : Account
{
    public override decimal GetInterest(decimal sum, int month, int rate)
    {
        if (sum < 0 || month > 12 || month < 1 || rate < 0)
            throw new Exception("Некорректные данные");

        decimal result = sum;
        for (int i = 0; i < month; i++)
            result += result * rate / 100;

        return result;
    }
}
```

В качестве постусловия в классе Account используется начисление бонусов в 100 единиц к финальной сумме, если начальная сумма от 1000 и более. В классе MicroAccount это условие не используется.

Теперь посмотрим, с какой проблемой мы можем столкнуться с данными классами:

```
class Program
{
```

```
    public static void CalculateInterest(Account account)
    {
        decimal sum = account.GetInterest(1000, 1, 10); // 1000 + 1000 * 10 /
100 + 100 (бонус)
        if (sum != 1200) // ожидаем 1200
        {
            throw new Exception("Неожиданная сумма при вычислениях");
        }
    }
    static void Main(string[] args)
    {
        Account acc = new MicroAccount();
        CalculateInterest(acc); // получаем 1100 без бонуса 100

        Console.Read();
    }
}
```

Исходя из логики класса Account, в методе CalculateInterest мы ожидаем получить в качестве результата числа 1200. Однако логика класса MicroAccount показывает другой результат. В итоге мы приходим к нарушению принципа Лисков, хотя формально мы просто применили стандартные принципы ООП – полиморфизм и наследование.

• **Инварианты (Invariants)** – все условия базового класса – также должны быть сохранены и в подклассе. Инварианты – это некоторые условия, которые остаются истинными на протяжении всей жизни объекта. Как правило, инварианты передают внутреннее состояние объекта. Например:

```
class User
{
    protected int age;
    public User(int age)
    {
        if (age < 0)
            throw new Exception("возраст меньше нуля");

        this.age = age;
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value < 0)
                throw new Exception("возраст меньше нуля");
            this.age = value;
        }
    }
}
```

Поле age выступает инвариантом. И поскольку его установка возможна только через конструктор или свойство, то в любом случае выполнение предусловия и в конструкторе, и в свойстве гарантирует, что возраст не будет меньше 0. И данное объектоятельство сохранит свою истинность на протяжении всей жизни объекта User. Теперь рассмотрим, как здесь может быть нарушен принцип Лисков. Пусть у нас будут следующие два класса:

```
class Account
{
    protected int capital;
    public Account(int sum)
    {
        if (sum < 100)
            throw new Exception("Некорректная сумма");
        this.capital = sum;
    }

    public virtual int Capital
    {
        get { return capital; }
        set
        {
            if (value < 100)
                throw new Exception("Некорректная сумма");
            capital = value;
        }
    }
}

class MicroAccount : Account
{
    public MicroAccount(int sum) : base(sum)
    {
    }

    public override int Capital
    {
        get { return capital; }
        set
        {
            capital = value;
        }
    }
}
```

С точки зрения класса Account, поле не может быть меньше 100, и в обоих случаях, где идет присвоение – в конструкторе и свойстве, это гарантируется. А вот производный класс MicroAccount, переопределяя свойство Capital, этого уже не гарантирует. Поэтому инвариант класса Account нарушается.

Во всех трех вышеперечисленных случаях проблема решается в общем случае с помощью абстрагирования и выделения общего функционала, который уже наследуют классы Account и MicroAccount. То есть не один из них наследуется от другого, а оба они наследуются от одного общего класса.

Таким образом, принцип подстановки Лисков заставляет задуматься над правильностью построения иерархий классов и применения полиморфизма, позволяя уйти от ложных иерархий наследования и делая всю систему классом более стройной и непротиворечивой.

5.4. Принцип разделения интерфейсов

Принцип разделения интерфейсов (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным". Принцип разделения интерфейсов можно сформулировать так: клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

Техники для выявления нарушения этого принципа: слишком большие интерфейсы; компоненты в интерфейсах слабо согласованы (перекликается с принципом единой ответственности); методы без реализации (перекликается с принципом Лисков).

В этом случае интерфейс класса разделяется на отдельные части, которые составляют раздельные интерфейсы. Затем эти интерфейсы

независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять.

Рассмотрим на примере. Допустим у нас есть интерфейс отправки сообщения:

```
interface IMessage
{
    void Send();
    string Text { get; set; }
    string Subject { get; set; }
    string ToAddress { get; set; }
    string FromAddress { get; set; }
}
```

Интерфейс определяет все основное, что нужно для отправки сообщения: само сообщение, его тему, адрес отправителя и получателя и, конечно, сам метод отправки. И пусть есть класс EmailMessage, который реализует этот интерфейс:

```
class EmailMessage : IMessage
{
    public string Subject { get; set; } = "";
    public string Text { get; set; } = "";
    public string FromAddress { get; set; } = "";
    public string ToAddress { get; set; } = "";

    public void Send() => Console.WriteLine($"Отправляем Email-сообщение: {Text}");
}
```

Надо отметить, что класс EmailMessage выглядит целостно, вполне удовлетворяя принципу единственной ответственности. То есть с точки зрения связности (cohesion) здесь проблем нет. Теперь определим класс, который бы отправлял данные по смс:

```
class SmsMessage : IMessage
{
    public string Text { get; set; } = "";
    public string FromAddress { get; set; } = "";
    public string ToAddress { get; set; } = "";

    public string Subject
    {
        get
        {
            throw new NotImplementedException();
        }

        set
        {
            throw new NotImplementedException();
```

```
        }
    }

    public void Send() => Console.WriteLine($"Отправляем Sms-сообщение: {Text}");
}
```

Здесь мы уже сталкиваемся с небольшой проблемой: свойство Subject, которое определяет тему сообщения, при отправке смс не указывается, поэтому оно в данном классе не нужно. Таким образом, в классе SmsMessage появляется избыточная функциональность, от которой класс SmsMessage начинает зависеть.

Это не очень хорошо, но посмотрим дальше. Допустим, нам надо создать класс для отправки голосовых сообщений.

Класс голосовой почты также имеет отправителя и получателя, только само сообщение передается в виде звука, что на уровне C# можно выразить в виде массива байтов. И в этом случае было бы неплохо, если бы интерфейс IMessage включал бы в себя дополнительные свойства и методы для этого, например:

```
interface IMessage
{
    void Send();
    string Text { get; set; }
    string ToAddress { get; set; }
    string Subject { get; set; }
    string FromAddress { get; set; }

    byte[] Voice { get; set; }
}
```

Тогда класс голосовой почты VoiceMessage мог бы выглядеть следующим образом:

```
class VoiceMessage : IMessage
{
    public string ToAddress { get; set; } = "";
    public string FromAddress { get; set; } = "";
    public byte[] Voice { get; set; } = new byte[] { };

    public string Text
    {
        get
        {
            throw new NotImplementedException();
        }

        set
        {
            throw new NotImplementedException();
```

```
        }
    }

    public string Subject
    {
        get
        {
            throw new NotImplementedException();
        }

        set
        {
            throw new NotImplementedException();
        }
    }

    public void Send() => Console.WriteLine("Передача голосовой почты");
}
```

И здесь опять же мы сталкиваемся с ненужными свойствами. Плюс нам надо добавить новое свойство в предыдущие классы SmsMessage и EmailMessage, причем этим классам свойство Voice в принципе-то не нужно. В итоге здесь мы сталкиваемся с явным нарушением принципа разделения интерфейсов. Для решения возникшей проблемы нам надо выделить из классов группы связанных методов и свойств и определить для каждой группы свой интерфейс:

```
interface IMessage
{
    void Send();
    string ToAddress { get; set; }
    string FromAddress { get; set; }
}
interface IVoiceMessage : IMessage
{
    byte[] Voice { get; set; }
}
interface ITextMessage : IMessage
{
    string Text { get; set; }
}

interface IEmailMessage : ITextMessage
{
    string Subject { get; set; }
}

class VoiceMessage : IVoiceMessage
{
    public string ToAddress { get; set; } = "";
    public string FromAddress { get; set; } = "";

    public byte[] Voice { get; set; } = Array.Empty<byte>();
    public void Send() => Console.WriteLine("Передача голосовой почты");
}
class EmailMessage : IEmailMessage
```

```
{
    public string Text { get; set; } = "";
    public string Subject { get; set; } = "";
    public string FromAddress { get; set; } = "";
    public string ToAddress { get; set; } = "";

    public void Send() => Console.WriteLine("Отправляем по Email сообщение: {Text}");
}

class SmsMessage : ITextMessage
{
    public string Text { get; set; } = "";
    public string FromAddress { get; set; } = "";
    public string ToAddress { get; set; } = "";
    public void Send() => Console.WriteLine("Отправляем по Sms сообщение: {Text}");
}
```

Теперь классы больше не содержат неиспользуемые методы. Чтобы избежать дублирование кода, применяется наследование интерфейсов. В итоге структура классов получается проще, чище и яснее.

5.5. Принцип инверсии зависимостей

Принцип инверсии зависимостей (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом: 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. Чтобы понять принцип, рассмотрим следующий пример:

```
class Book
{
    public string Text { get; set; }
    public ConsolePrinter Printer { get; set; }

    public void Print()
    {
        Printer.Print(Text);
    }
}

class ConsolePrinter
{
    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

Класс Book, представляющий книгу, использует для печати класс ConsolePrinter. При подобном определении класс Book зависит от класса ConsolePrinter. Более того мы жестко определили, что печать книгу можно только на консоли с помощью класса ConsolePrinter. Другие же варианты, например, вывод на принтер, вывод в файл или с использованием каких-то элементов графического интерфейса – все это в данном случае исключено. Абстракция печати книги не отделена от деталей класса ConsolePrinter. Все это является нарушением принципа инверсии зависимостей. Теперь попробуем привести наши классы в соответствие с принципом инверсии зависимостей, отделив абстракции от низкоуровневой реализации:

```
interface IPrinter
{
    void Print(string text);
}

class Book
{
    public string Text { get; set; }
    public IPrinter Printer { get; set; }

    public Book(IPrinter printer)
    {
        this.Printer = printer;
    }

    public void Print()
    {
        Printer.Print(Text);
    }
}

class ConsolePrinter : IPrinter
{
    public void Print(string text)
    {
        Console.WriteLine("Печать на консоли");
    }
}

class HtmlPrinter : IPrinter
{
    public void Print(string text)
    {
        Console.WriteLine("Печать в html");
    }
}
```

Теперь абстракция печати книги отделена от конкретных реализаций. В итоге и класс Book и класс ConsolePrinter зависят от абстракции

IPrinter. Кроме того, теперь мы также можем создать дополнительные низкоуровневые реализации абстракции IPrinter и динамически применять их в программе:

```
Book book = new Book(new ConsolePrinter());
book.Print();
book.Printer = new HtmlPrinter();
book.Print();
```

ЛИТЕРАТУРА

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. П75 Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021. — 448 с.

2. Фаулер, Мартин. Архитектура корпоративных программных приложений.: Пер. с англ. - М.: Издательский дом "Вильямс", 2006. - 544 с.

3. Орлов, С. А. Программная инженерия: Учебник для вузов. 5-е изд. обнов. и доп. Стандарт третьего поколения. – СанктПетербург: Питер, 2017. – 640 с.

4. Чернышев, С. А. Принципы, паттерны и методологии разработки программного обеспечения : учебное пособие для вузов / С. А. Чернышев. — Москва : Издательство Юрайт, 2023. — 176 с

СОДЕРЖАНИЕ

Электронное учебное издание

Александр Александрович **Рыбанов**

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ НА C#

Учебное пособие

Электронное издание сетевого распространения

Редактор Матвеева Н.И.

Темплан 2023 г. Поз. № 2.
Подписано к использованию 05.06.2023. Формат 60x84 1/16.
Гарнитура Times. Усл. печ. л. 6,0.

Волгоградский государственный технический университет.
400005, г. Волгоград, пр. Ленина, 28, корп. 1.

ВПИ (филиал) ВолгГТУ.
404121, г. Волжский, ул. Энгельса, 42а.