Рыбанов А.А.

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Волжский
2024

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.

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Электронное учебное пособие

2024

УДК 004.45(07)
ББК 32.973я73
Р 931

Рецензенты:
Волгоградский государственный социально-педагогический университет, заведующий кафедрой методики преподавания математики и физики, ИКТ, профессор, д.п.н.
Смыковская Т.А.
Волгоградский государственный медицинский университет Минздрава РФ, доцент кафедры физики, математики и информатики к.п.н.
Филиппова Е.М.

Издается по решению редакционно-издательского совета
Волгоградского технического университета

Рыбанов, А.А.
Порождающие паттерны проектирования [Электронный ресурс] : учебное пособие / А. А. Рыбанов ; Министерство науки и высшего образования Российской Федерации, ВПИ (филиал) ФГБОУ ВО ВолгГТУ. – Электрон. текстовые дан. (1 файл: 8,12 МБ). – Волжский, 2023. – Режим доступа: http://lib.volpi.ru. – Загл. с титул. экрана.
ISBN 978-5-9948-4803-6

В учебном пособии рассматриваются порождающие паттерны проектирования: фабричный метод, абстрактная фабрика, одиночка, прототип, строитель. Проанализированы принципы разработки программных продуктов, таких как SOLID. 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».

Ил. 16, библиограф.: 4 назв.

ISBN 978-5-9948-4803-6

© Волгоградский государственный технический университет, 2024
© Волжский политехнический институт, 2024

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 наиболее основные и 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).

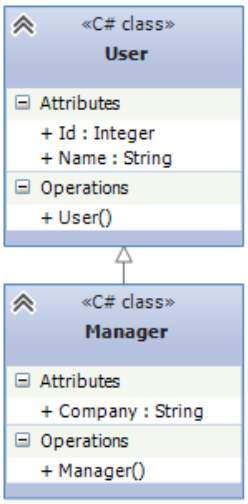

Рисунок 1. Отношение наследования

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).

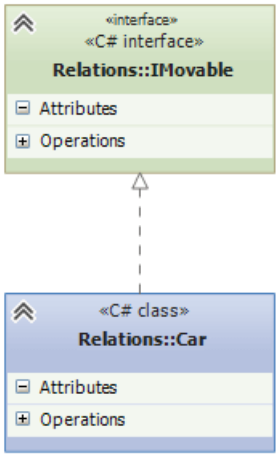

Рисунок 2. Отношение реализации

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 класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки (рис. 3).

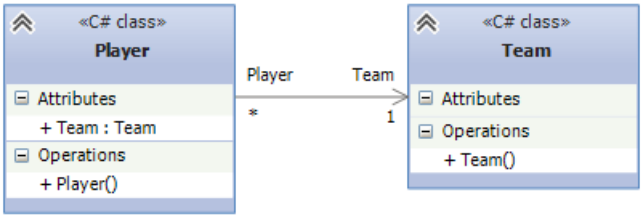

Рисунок 3. Отношение ассоциации

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).

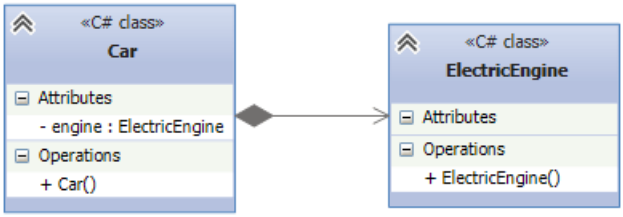

Рисунок 4. Отношение композиции

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).

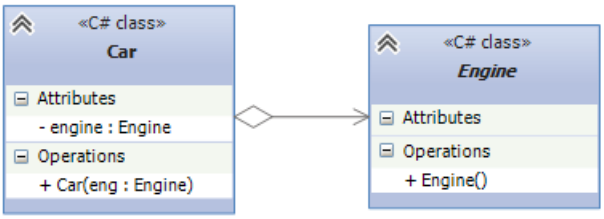

Рисунок 5. Отношение агрегации

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 также во все 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 тоже может 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. Техника использования объекта-фабрики

В основу категоризации каталога паттернов легли три простейшие объектно-ориентированные техники:

- техника использования объектов-фабрик, порождающих объекты-продукты;
- техника использования объекта-фасада;
- техника диспетчеризации.

В основу всех порождающих паттернов положена техника использования объекта-фабрики для порождения объектов-продуктов (рис. 6).

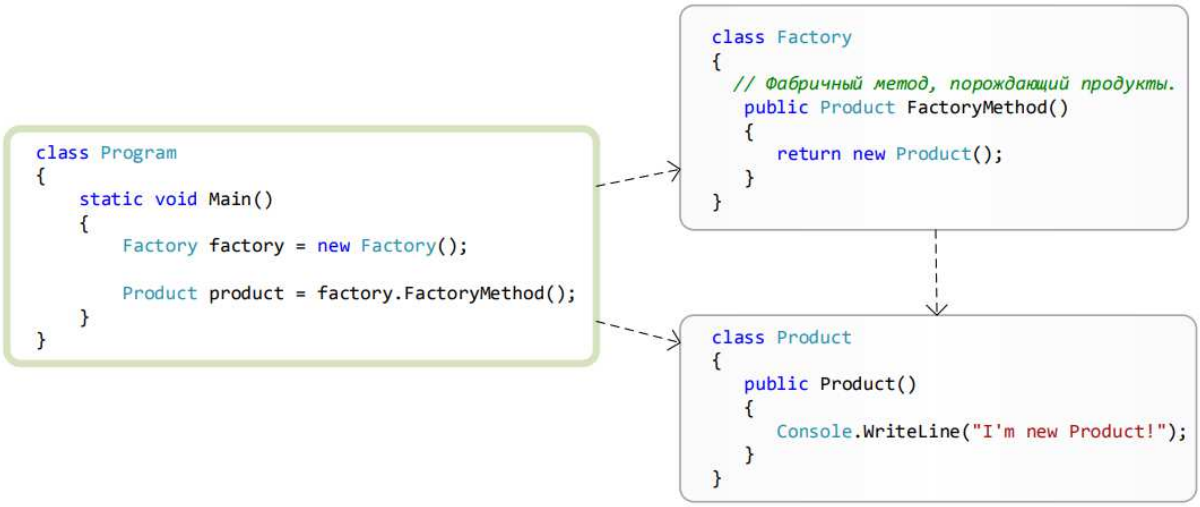

Рисунок 6. Техника использования объектов-фабрик

Методы, принадлежащие объекту-фабрике, которые порождают и возвращают объекты-продукты, принято называть фабричными-методами (или виртуальными конструкторами).

На диаграмме последовательностей (рис. 7) можно отследить работу фабричной техники.

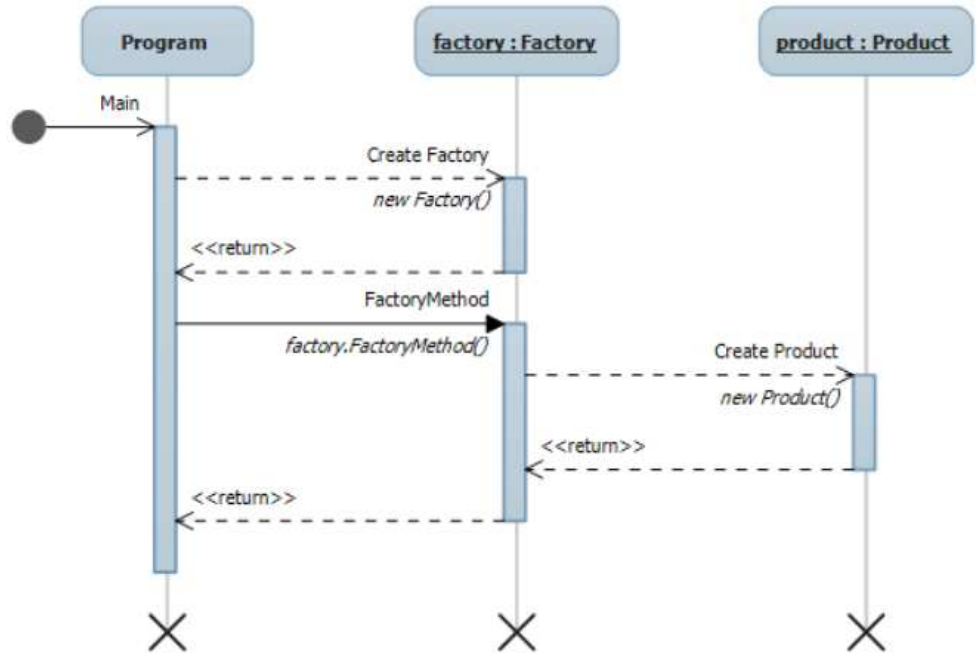

Рисунок 7. Техника использования объектов-фабрик: диаграмма последовательностей

Порождающие паттерны – группа шаблонов проектирования, которые берут на себя ответственность за логику создания объектов. Они упрощают интерфейс создания объектов и позволяют создавать гибкие, расширяемые объекты, в т.ч. сложные, независимо от способа их создания и взаимодействия.

Основная идея заключается в том, чтобы создать иерархию классов, которые будут отвечать за создание объектов, то есть базовый класс будет иметь виртуальный метод, создающий объект, а последующие "создатели" будут этот метод реализовывать.

2.2. 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 8.

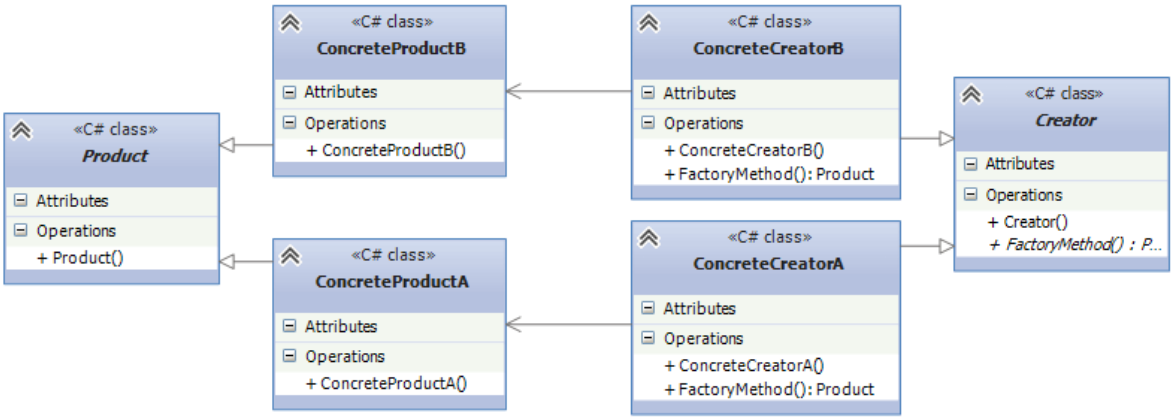

Рисунок 8. Фабричный метод (Factory Method)

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 захо-

чется 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.

2.3. 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.

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()
    { }
}
```

Описание паттерна на языке UML приведено на рисунке 9.

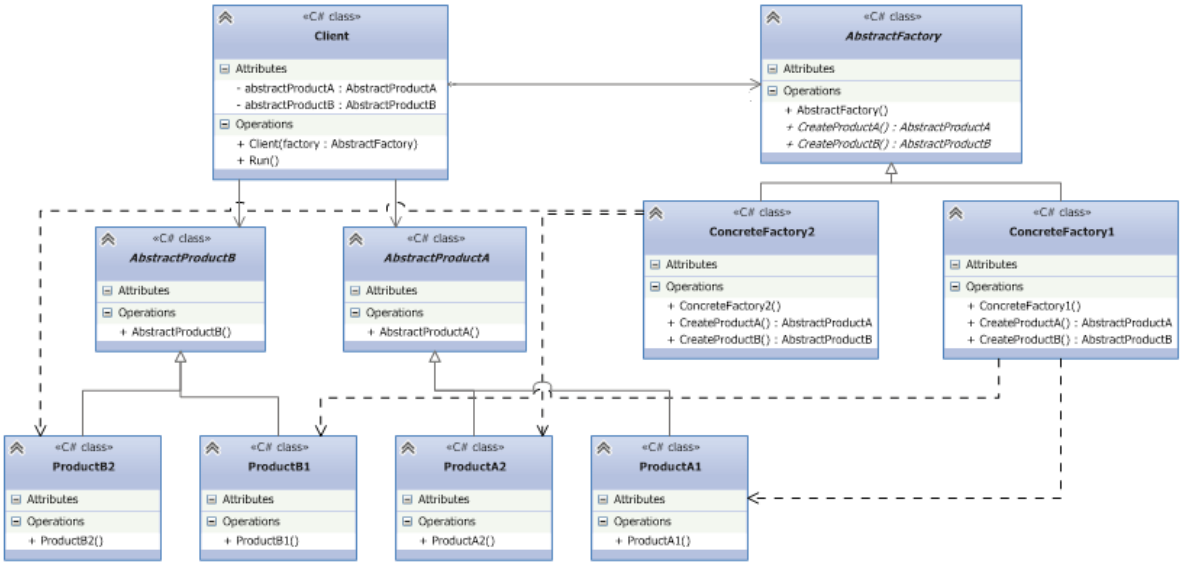

Рисунок 9. Абстрактная фабрика (Abstract Factory)

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.

2.4. 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;
    }
}
```

2.4.1. Синглтон и многопоточность

При применении паттерна синглтон в многопоточным программах мы можем столкнуться с проблемой, которую можно описать следующим образом:

```
static void Main(string[] args)
{
    (new Thread(() =>
    {
        Computer comp2 = new Computer();
        comp2.OS = OS.getInstance("Windows 10");
        Console.WriteLine(comp2.OS.Name);

    })).Start();

    Computer comp = new Computer();
    comp.Launch("Windows 8.1");
    Console.WriteLine(comp.OS.Name);
    Console.ReadLine();
}
```

Здесь запускается дополнительный поток, который получает доступ к синглтону. Параллельно выполняется тот код, который идет запуска потока и который также обращается к синглтону. Таким образом, и главный, и дополнительный поток пытаются инициализровать синглтон нужным значением – "Windows 10" либо "Windows 8.1". Какое значение сиглтон получит в итоге, предсказать в данном случае невозможно.

Вывод программы может быть таким, как на рисунке 10.

```
Windows 8.1
Windows 10
```

Рисунок 10. Консольный вывод № 1

Или таким, как на рисунке 11.

```
Windows 8.1
Windows 8.1
```

Рисунок 11. Консольный вывод № 2

В итоге мы сталкиваемся с проблемой инициализации синглтона, когда оба потока одновременно обращаются к коду:

```
if (instance == null)
    instance = new OS(name);
```

Чтобы решить эту проблему, перепишем класс синглтона следующим образом:

```
class OS
{
    private static OS instance;

    public string Name { get; private set; }
    private static object syncRoot = new Object();

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

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

Чтобы избежать одновременного доступа к коду из разных потоков критическая секция заключается в блок lock.

2.4.2. Другие реализации синглтона

Выше были рассмотрены общие стандартные реализации: потоконебезопасная и потокобезопасная реализации паттерна. Но есть еще ряд дополнительных реализаций, которые можно рассмотреть.

2.4.2.1. Потокобезопасная реализация без использования lock

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

    public string Date { get; private set; }

    private Singleton()
    {
        Date = System.DateTime.Now.TimeOfDay.ToString();
    }

    public static Singleton GetInstance()
    {
        return instance;
    }
}
```

Данная реализация также потокобезопасная, то есть мы можем использовать ее в потоках так:

```
(new Thread(() =>
{
    Singleton singleton1 = Singleton.GetInstance();
    Console.WriteLine(singleton1.Date);
})).Start();

Singleton singleton2 = Singleton.GetInstance();
Console.WriteLine(singleton2.Date);
```

2.4.2.2. Lazy-реализация

Определение объекта синглтона в виде статического поля класса открывает нам дорогу к созданию Lazy-реализации паттерна Синглтон, то есть такой реализации, где данные будут инициализироваться только перед непосредственным использованием. Поскольку статические поля инициализируются перед первым доступом к статическому членам класса и перед вызовом статического конструктора (при его наличии). Однако здесь мы можем столкнуться с двумя трудностями.

Во-первых, класс синглтона может иметь множество статических переменных. Возможно, мы вообще не будем обращаться к объекту синглтона, а будем использовать какие-то другие статические переменные:

```
public class Singleton
{
    private static readonly Singleton instance = new Singleton();
    public static string text = "hello";
    public string Date { get; private set; }

    private Singleton()
    {
        Console.WriteLine($"Singleton ctor {DateTime.Now.TimeOfDay}");
        Date = System.DateTime.Now.TimeOfDay.ToString();
    }

    public static Singleton GetInstance()
    {
        Console.WriteLine($"GetInstance {DateTime.Now.TimeOfDay}");
        Thread.Sleep(500);
        return instance;
    }
}
class Program
{
    static void Main(string[] args)
    {
```

```
        Console.WriteLine($"Main {DateTime.Now.TimeOfDay}");
        Console.WriteLine(Singleton.text);
    }
}
```

В данном случае идет только обращение к переменной text, однако статическое поле instance также будет инициализировано. Например, консольный вывод в данном случае мог бы таким, как представлено на рисунке 12.

```
Main 16:11:40.1320873
hello
```

Рисунок 12. Lazy-реализация консольный вывод

В данном случае мы видим, что статическое поле instance инициализировано.

Для решения этой проблемы выделим отдельный внутренний класс в рамках класса синглтона:

```
public class Singleton
{
    public string Date { get; private set; }
    public static string text = "hello";
    private Singleton()
    {
        Console.WriteLine($"Singleton ctor {DateTime.Now.TimeOfDay}");
        Date = DateTime.Now.TimeOfDay.ToString();
    }

    public static Singleton GetInstance()
    {
        Console.WriteLine($"GetInstance {DateTime.Now.TimeOfDay}");
        return Nested.instance;
    }

    private class Nested
    {
        internal static readonly Singleton instance = new Singleton();
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine($"Main {DateTime.Now.TimeOfDay}");
        Console.WriteLine(Singleton.text);
    }
}
```

Теперь статическая переменная, которая представляет объект синглтона, определена во вложенном классе Nested. Чтобы к этой переменной можно было обращаться из класса синглтона, она имеет модификатор

internal, в то же время сам класс Nested имеет модификатор private, что позволяет гарантировать, что данный класс будет доступен только из класса Singleton.

Консольный вывод в данном случае мог бы выглядеть так, как показано на рисунке 13.

Рисунок 13. Lazy-реализация консольный вывод

2.4.2.3. Реализация через класс Lazy<T>

Еще один способ создания синглтона представляет использование класса Lazy<T>:

```
public class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public string Name { get; private set; }

    private Singleton()
    {
        Name = System.Guid.NewGuid().ToString();
    }

    public static Singleton GetInstance()
    {
        return lazy.Value;
    }
}
```

2.5. Прототип (Prototype)

Паттерн Прототип (Prototype) позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть, по сути, данный паттерн предлагает технику клонирования объектов.

Когда использовать Прототип?

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения.
- Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии

классов (как это делается, например, при использовании паттерна Абстрактная фабрика).

- Когда клонирование объекта является более предпочтительным вариантом, нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

Формальная структура паттерна на C# могла бы выглядеть следующим образом:

```
class Client
{
    void Operation()
    {
        Prototype prototype = new ConcretePrototype1(1);
        Prototype clone = prototype.Clone();
        prototype = new ConcretePrototype2(2);
        clone = prototype.Clone();
    }
}

abstract class Prototype
{
    public int Id { get; private set; }
    public Prototype(int id)
    {
        this.Id = id;
    }
    public abstract Prototype Clone();
}

class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(int id)
        : base(id)
    { }
    public override Prototype Clone()
    {
        return new ConcretePrototype1(Id);
    }
}

class ConcretePrototype2 : Prototype
{
    public ConcretePrototype2(int id)
        : base(id)
    { }
    public override Prototype Clone()
    {
        return new ConcretePrototype2(Id);
    }
}
```

Описание паттерна на языке UML приведено на рисунке 14.

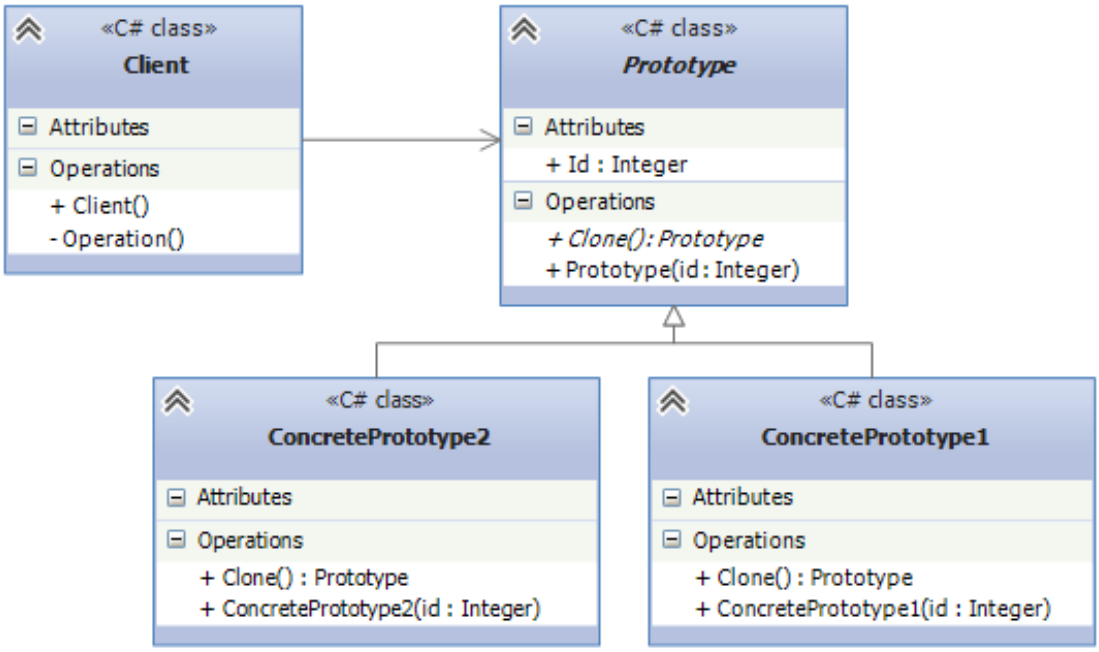

Рисунок 14. Прототип (Prototype)

Паттерн определяет следующих участников:

- Prototype: определяет интерфейс для клонирования самого себя, который, как правило, представляет метод Clone();
- ConcretePrototype1 и ConcretePrototype2: конкретные реализации прототипа. Реализуют метод Clone();
- Client: создает объекты прототипов с помощью метода Clone().

Рассмотрим клонирование на примере фигур-прямоугольников и кругов:

```
class Program
{
    static void Main(string[] args)
    {
        IFigure figure = new Rectangle(30, 40);
        IFigure clonedFigure = figure.Clone();
        figure.GetInfo();
        clonedFigure.GetInfo();

        figure = new Circle(30);
        clonedFigure = figure.Clone();
        figure.GetInfo();
        clonedFigure.GetInfo();

        Console.Read();
    }
}

interface IFigure
{
```

```
    IFigure Clone();
    void GetInfo();
}

class Rectangle : IFigure
{
    int width;
    int height;
    public Rectangle(int w, int h)
    {
        width = w;
        height = h;
    }

    public IFigure Clone()
    {
        return new Rectangle(this.width, this.height);
    }
    public void GetInfo()
    {
        Console.WriteLine("Прямоугольник длиной {0} и шириной {1}", height, width);
    }
}

class Circle : IFigure
{
    int radius;
    public Circle(int r)
    {
        radius = r;
    }

    public IFigure Clone()
    {
        return new Circle(this.radius);
    }
    public void GetInfo()
    {
        Console.WriteLine("Круг радиусом {0}", radius);
    }
}
```

Здесь в качестве прототипа используется интерфейс IFigure, который реализуется классами Circle и Rectangle.

Но в данном случае надо заметить, что фреймворк .NET предлагает функционал для копирования в виде метода MemberwiseClone(). Например, мы могли бы изменить реализацию метода Clone() в классах прямоугольника и круга следующим образом:

```
public IFigure Clone()
{
    return this.MemberwiseClone() as IFigure;
}
```

Причем данный метод был бы общим для обоих классов. И работа программы никак бы не изменилась.

В то же время надо учитывать, что метод MemberwiseClone() осуществляет неполное копирование – то есть копирование значимых типов. Если же класс фигуры содержал бы объекты ссылочных типов, то оба объекта после клонирования содержали бы ссылку на один и тот же ссылочный объект. Например, пусть фигура круг имеет свойство ссылочного типа:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}
class Circle : IFigure
{
    int radius;
    public Point Point { get; set; }
    public Circle(int r, int x, int y)
    {
        radius = r;
        this.Point = new Point { X = x, Y = y };
    }

    public IFigure Clone()
    {
        return this.MemberwiseClone() as IFigure;
    }
    public void GetInfo()
    {
        Console.WriteLine("Круг радиусом {0} и центром в точке ({1}, {2})", radius,
Point.X, Point.Y);
    }
}
```

В этом случае при изменении значений в свойстве Point начальной фигуры автоматически бы изменилось соответствующее значение и у клонированной фигуры:

```
Circle figure = new Circle(30, 50, 60);
Circle clonedFigure = figure.Clone() as Circle;
figure.Point.X = 100; // изменяем координаты начальной фигуры
figure.GetInfo(); // figure.Point.X = 100
clonedFigure.GetInfo(); // clonedFigure.Point.X = 100
```

Чтобы избежать подобной ситуации, надо применить полное копирование:

```
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

//.....................
class Program
{
    static void Main(string[] args)
```

```
    {
        Circle figure = new Circle(30, 50, 60);
        // применяем глубокое копирование
        Circle clonedFigure = figure.DeepCopy() as Circle;
        figure.Point.X = 100;
        figure.GetInfo();
        clonedFigure.GetInfo();

        Console.Read();
    }
}
//.......................

[Serializable]
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}
[Serializable]
class Circle : IFigure
{
    int radius;
    public Point Point { get; set; }
    public Circle(int r, int x, int y)
    {
        radius = r;
        this.Point = new Point { X = x, Y = y };
    }

    public IFigure Clone()
    {
        return this.MemberwiseClone() as IFigure;
    }

    public object DeepCopy()
    {
        object figure = null;
        using (MemoryStream tempStream = new MemoryStream())
        {
            BinaryFormatter binFormatter = new BinaryFormatter(null,
                new StreamingContext(StreamingContextStates.Clone));

            binFormatter.Serialize(tempStream, this);
            tempStream.Seek(0, SeekOrigin.Begin);

            figure = binFormatter.Deserialize(tempStream);
        }
        return figure;
    }
    public void GetInfo()
    {
        Console.WriteLine("Круг радиусом {0} и центром в точке ({1}, {2})", radius,
Point.X, Point.Y);
    }
}
```

Чтобы вручную не создавать у клонированного объекта вложенный объект Point, здесь используются механизмы бинарной сериализации. И в

этом случае все классы, объекты которых подлежат копированию, должны быть помечены атрибутом Serializable.

2.6. Строитель (Builder)

Строитель (Builder) – шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Когда использовать паттерн Строитель?

- Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой.
- Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания.

Описание паттерна на языке UML приведено на рисунке 15.

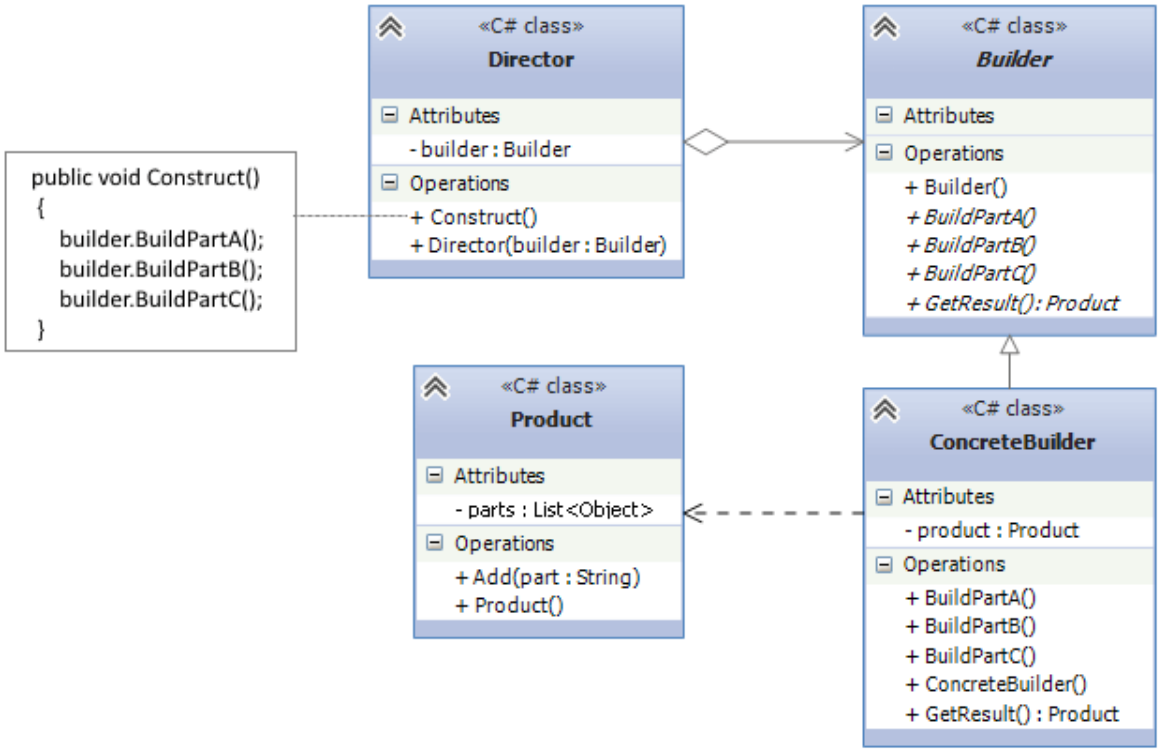

Рисунок 15. Строитель (Builder)

Формальная структура паттерна на C# могла бы выглядеть следующим образом:

```
class Client
{
    void Main()
    {
```

```
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.Construct();
        Product product = builder.GetResult();
    }
}
class Director
{
    Builder builder;
    public Director(Builder builder)
    {
        this.builder = builder;
    }
    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}

abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract void BuildPartC();
    public abstract Product GetResult();
}

class Product
{
    List<object> parts = new List<object>();
    public void Add(string part)
    {
        parts.Add(part);
    }
}

class ConcreteBuilder : Builder
{
    Product product = new Product();
    public override void BuildPartA()
    {
        product.Add("Part A");
    }
    public override void BuildPartB()
    {
        product.Add("Part B");
    }
    public override void BuildPartC()
    {
        product.Add("Part C");
    }
    public override Product GetResult()
    {
        return product;
    }
}
```

Паттерн определяет следующих участников:

- Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.
- Builder: определяет интерфейс для создания различных частей объекта Product.
- ConcreteBuilder: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему.
- Director: распорядитель – создает объект, используя объекты Builder.

Рассмотрим применение паттерна на примере выпечки хлеба. Как известно, даже обычный хлеб включает множество компонентов. Мы можем использовать для представления хлеба и его компонентов следующие классы:

```
//мука
class Flour
{
    // какого сорта мука
    public string Sort { get; set; }
}
// соль
class Salt
{ }
// пищевые добавки
class Additives
{
    public string Name { get; set; }
}

class Bread
{
    // мука
    public Flour Flour { get; set; }
    // соль
    public Salt Salt { get; set; }
    // пищевые добавки
    public Additives Additives { get; set; }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();

        if (Flour != null)
            sb.Append(Flour.Sort + "\n");
        if (Salt != null)
            sb.Append("Соль \n");
        if (Additives != null)
            sb.Append("Добавки: " + Additives.Name + " \n");
        return sb.ToString();
    }
}
```

Кстати, в данном случае для построения строки используется класс StringBuilder.

Хлеб может иметь различную комбинацию компонентов: ржаной и пшеничной муки, соли, пищевых добавок. И нам надо обеспечить выпечку разных сортов хлеба. Для разных сортов хлеба может варьироваться конкретный набор компонентов, не все компоненты могут использоваться. И для этой задачи применим паттерн Builder:

```
class Program
{
    static void Main(string[] args)
    {
        // содаем объект пекаря
        Baker baker = new Baker();
        // создаем билдер для ржаного хлеба
        BreadBuilder builder = new RyeBreadBuilder();
        // выпекаем
        Bread ryeBread = baker.Bake(builder);
        Console.WriteLine(ryeBread.ToString());
        // оздаем билдер для пшеничного хлеба
        builder = new WheatBreadBuilder();
        Bread wheatBread = baker.Bake(builder);
        Console.WriteLine(wheatBread.ToString());

        Console.Read();
    }
}
// абстрактный класс строителя
abstract class BreadBuilder
{
    public Bread Bread { get; private set; }
    public void CreateBread()
    {
        Bread = new Bread();
    }
    public abstract void SetFlour();
    public abstract void SetSalt();
    public abstract void SetAdditives();
}
// пекарь
class Baker
{
    public Bread Bake(BreadBuilder breadBuilder)
    {
        breadBuilder.CreateBread();
        breadBuilder.SetFlour();
        breadBuilder.SetSalt();
        breadBuilder.SetAdditives();
        return breadBuilder.Bread;
    }
}
// строитель для ржаного хлеба
class RyeBreadBuilder : BreadBuilder
{
```

```
    public override void SetFlour()
    {
        this.Bread.Flour = new Flour { Sort = "Ржаная мука 1 сорт" };
    }

    public override void SetSalt()
    {
        this.Bread.Salt = new Salt();
    }

    public override void SetAdditives()
    {
        // не используется
    }
}
// строитель для пшеничного хлеба
class WheatBreadBuilder : BreadBuilder
{
    public override void SetFlour()
    {
        this.Bread.Flour = new Flour { Sort = "Пшеничная мука высший сорт" };
    }

    public override void SetSalt()
    {
        this.Bread.Salt = new Salt();
    }

    public override void SetAdditives()
    {
        this.Bread.Additives = new Additives { Name = "улучшитель хлебопекарный" };
    }
}
```

Консольный вывод программы представлен на рисунке 16.

```
Ржаная мука 1 сорт
Соль

Пшеничная мука высший сорт
Соль
Добавки: улучшитель хлебопекарный
```

Рисунок 16. Строитель (Builder): консольный вывод

В данном случае с помощью конкретных строителей RyeBreadBuilder и WheatBreadBuilder создаются объекты Bread с определенным набором. В роли распорядителя выступает класс пекаря Baker, который вызывает методы конкретных строителей для построения нового объекта.

3. 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.

3.1. 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.

3.2. 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 вареный 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 подается 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 варенный 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 варенный 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);
```

3.3. 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.

3.4. 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 ин-

терфейс 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 разде-

ления 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.

3.5. 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 печатать 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**

ПОРОЖДАЮШИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Учебное пособие

Электронное издание сетевого распространения

Редактор Матвеева Н.И.

Темплан 2024 г. Поз. № 1.
Подписано к использованию 26.02.2024. Формат 60x84 1/16.
Гарнитура Times. Усл. печ. л. 4,0.

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

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