Лабораторная работа №2.6

Цель: ознакомится с понятием «шаблон проектирования», изучить порождающие шаблоны проектирования.

**Введение в паттерны проектирования**

Что представляют собой 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 пявилась 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 первом масштабной попыткой описать распространенные способы проектирования программ. И со временем применение паттернов стало считать хорошей практикой программирования.

Что же дает нам применение паттернов? 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**

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.

И в данном руководстве мы рассмотрим наиболее основные и распространенные паттерны и принципы их использования применительно к языку C#.

**Как выбрать нужный паттерн?**

Прежде всего 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 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.

## 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

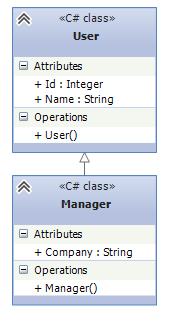
{

    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:



### 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()

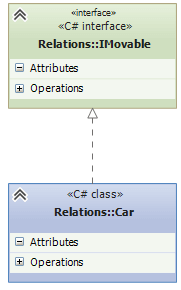
    {

        Console.WriteLine("Машина едет");

    }

}

С помощью диаграмм UML отношение реализации также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная:



### 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

{

}

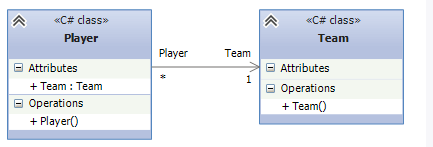
class Player

{

    public Team Team { get; set; }

}

Класс Player связан отношением ассоциации с класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки:



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()

    {

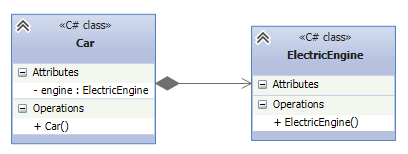
        engine = new ElectricEngine();

    }

}

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:



### 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)

    {

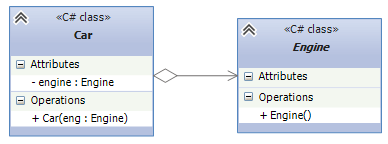
        engine = eng;

    }

}

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 также, как и отношение композиции, только теперь ромбик будет незакрашенным:



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.

## 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 функционал
* 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. Вобщем 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

{}

public class Car : Vehicle, IMovable

{

    public void Move()

    {

        Console.WriteLine("Машина едет");

    }

}

public class Bus : Vehicle, IMovable

{

    public 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 астрактного 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 электронагреатель. И так, для каждой разноплановой сущности можно было бы составить свою систему классов, исходяющую 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.

# Порождающие паттерны

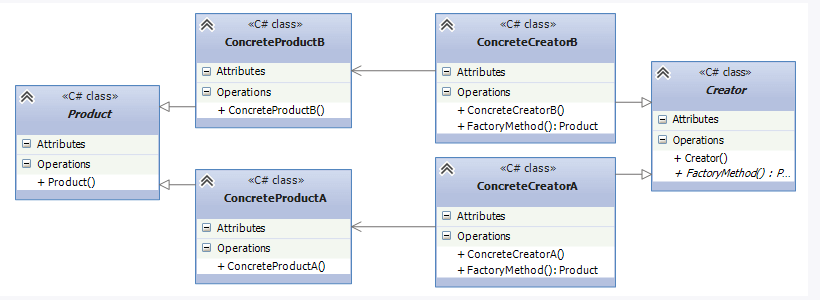
## Фабричный метод (Factory Method)

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

На языке UML паттерн можно описать следующим образом:



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.

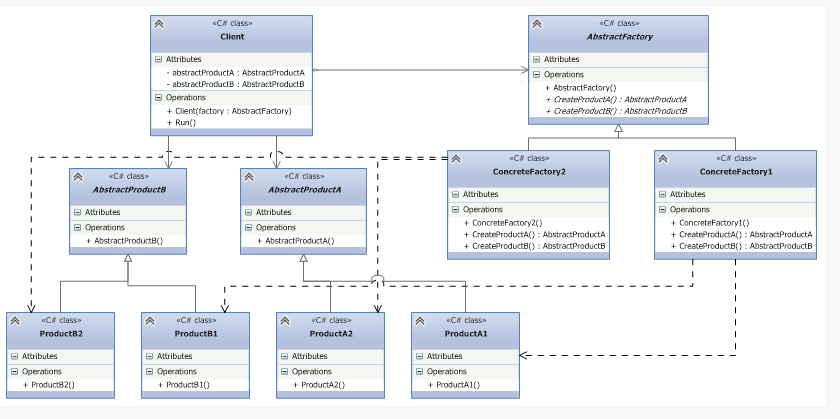
## Абстрактная фабрика (Abstract Factory)

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

С помощью UML абстрактную фабрику можно представить следующим образом:



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 сделать способ создание 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.

## Одиночка

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;

    }

}

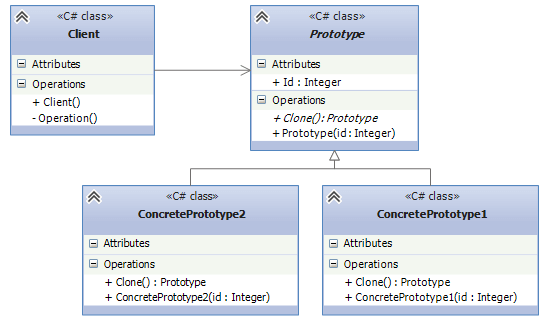
## Прототип (Prototype)

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.

На языке UML отношения между классами при применении данного паттерна можно описать следующим образом:



Формальная структура паттерна на в 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);

    }

}

### Участники

* **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.

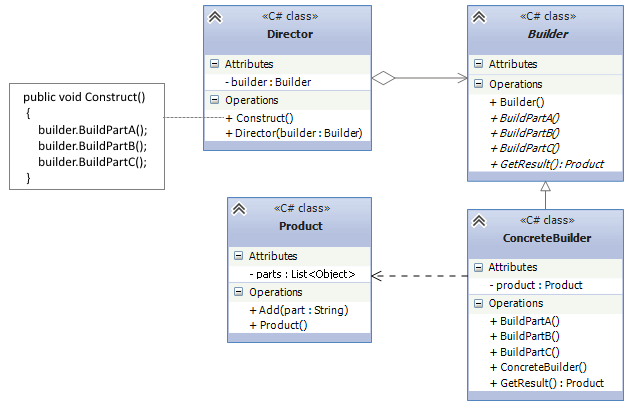
## Строитель (Builder)

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

Формально в UML паттерн мог бы выглядеть следующим образом:



Формальное определение на C# могло бы выглядеть так:

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;

    }

}

### Участники

* **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

{

    // пшеничная мука

    public Flour WheatFlour { get; set; }

    // ржаная мука

    public Flour RyeFlour { get; set; }

    // соль

    public Salt Salt { get; set; }

    // пищевые добавки

    public Additives Additives { get; set; }

    public override string ToString()

    {

        StringBuilder sb = new StringBuilder();

        if (WheatFlour != null)

            sb.Append("Пшеничная мука " + WheatFlour.Sort + "\n");

        if (RyeFlour != null)

            sb.Append("Ржаная мука " + RyeFlour.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 одна из встроенных реализаций паттерна Builder в .NET - 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

        baker.Bake(builder);

        Bread ryeBread = builder.Bread;

        Console.WriteLine(ryeBread.ToString());

        // оздаем билдер для пшеничного хлеба

        builder = new WheatBreadBuilder();

        baker.Bake(builder);

        Bread wheatBread = builder.Bread;

        Console.WriteLine(wheatBread.ToString());

        Console.Read();

    }

}

// абстрактный класс строителя

abstract class BreadBuilder

{

    public Bread Bread { get; private set; }

    public BreadBuilder()

    {

        Bread = new Bread();

    }

    public abstract void SetWheatFlour();

    public abstract void SetRyeFlour();

    public abstract void SetSalt();

    public abstract void SetAdditives();

}

// пекарь

class Baker

{

    public void Bake(BreadBuilder breadBuilder)

    {

        breadBuilder.SetWheatFlour();

        breadBuilder.SetRyeFlour();

        breadBuilder.SetSalt();

        breadBuilder.SetAdditives();

    }

}

// строитель для ржаного хлеба

class RyeBreadBuilder : BreadBuilder

{

    public override void SetWheatFlour()

    {

        // не используется

    }

    public override void SetRyeFlour()

    {

        this.Bread.RyeFlour = new Flour { Sort = "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 SetWheatFlour()

    {

        this.Bread.WheatFlour = new Flour { Sort = "высший сорт" };

    }

    public override void SetRyeFlour()

    {

        // не используется

    }

    public override void SetSalt()

    {

        this.Bread.Salt = new Salt();

    }

    public override void SetAdditives()

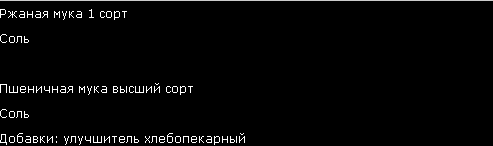
    {

        this.Bread.Additives = new Additives { Name = "улучшитель хлебопекарный" };

    }

}

Консольный вывод программы:



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.

**Варианты заданий:**

* 1. Производство конфет (Строитель, Одиночка)
  2. Кредитная компания (Фабричный метод, Одиночка)
  3. Животный мир (Абстрактная фабрика)
  4. Суши-шоп (Строитель)
  5. Пиццерия (Строитель)
  6. Ремонтная организация (Фабричный метод, Одиночка)
  7. Текстовая игра «Гонки» (Абстрактная фабрика)
  8. Школа танцев (Фабричный метод)
  9. Текстовая игра «Битва героев» (Абстрактная фабрика)
  10. Кофейня (Строитель)
  11. Производство пирожков (Строитель)
  12. Производство мебели (Фабричный метод)
  13. Такси (Абстрактная фабрика)
  14. Рекламное агентство (Фабричный метод)
  15. Турфирма (Абстрактная фабрика)
  16. Ресторан (Строитель)

Ход работы:

1. Реализовать программу согласно варианту.
2. Предоставить интерактивный интерфейс взаимодействия.
3. Подготовиться к защите.

Защита. В ходе защиты лабораторной работы могут быть следующие задания:

1. Расширение функционала.
2. Теоретический вопрос про любой шаблон описанный в лабораторной работе.