Паттерны проектирования

Жилина Е.В., 2019

Классификация паттернов:

- Порождающие паттерны
- Структурные паттерны
- Поведенческие паттерны

Порождающие паттерны – это паттерны, которые абстрагируют процесс порождения классов и объектов.

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

Поведенческие паттерны - определяют алгоритмы и взаимодействие между классами и объектами.

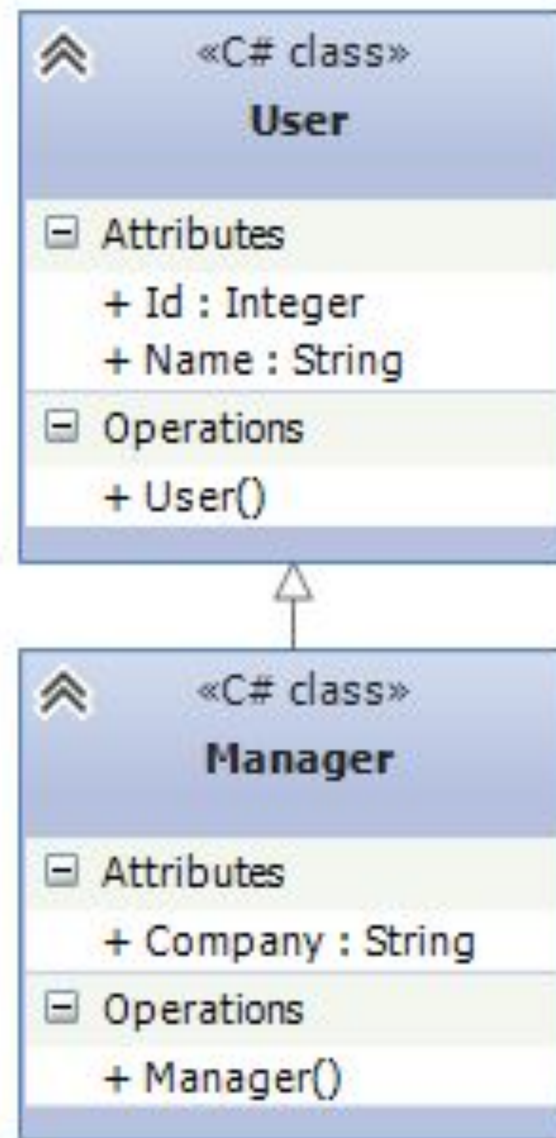
- Наблюдатель (Observer)
- Стратегия (Strategy)
- Команда (Command)
- Шаблонный метод (Template method)
- Итератор (Iterator)
- Состояние (State)
- Цепочка обязанностей (Chain of responsibility)
- Интерпретатор (Interpreter)
- Посредник (Mediator)
- Хранитель (Memento)
- Посетитель (Visitor)

Структурные паттерны - рассматривают, как классы и объекты образуют более крупные структуры, более сложные по характеру классы и объекты.

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

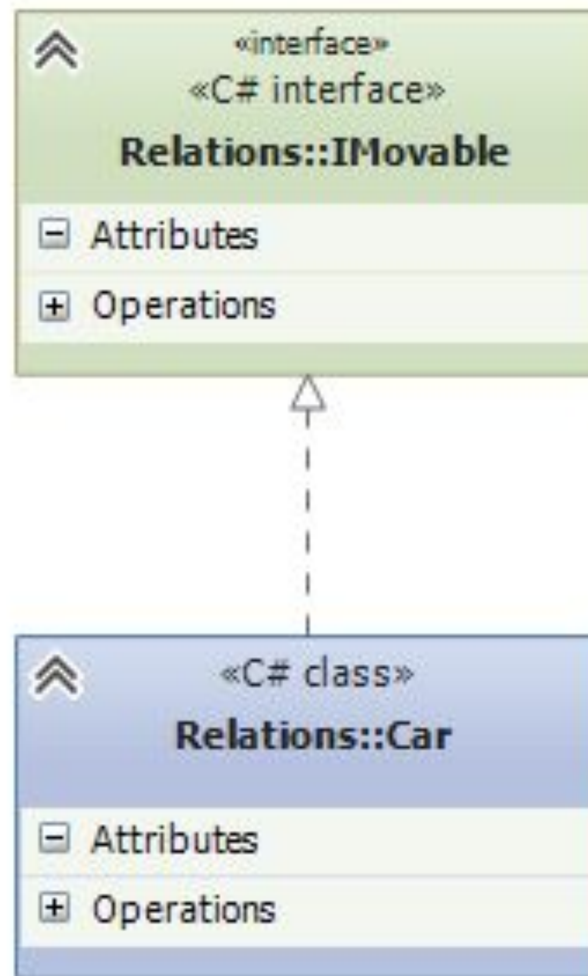
Наследование

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

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

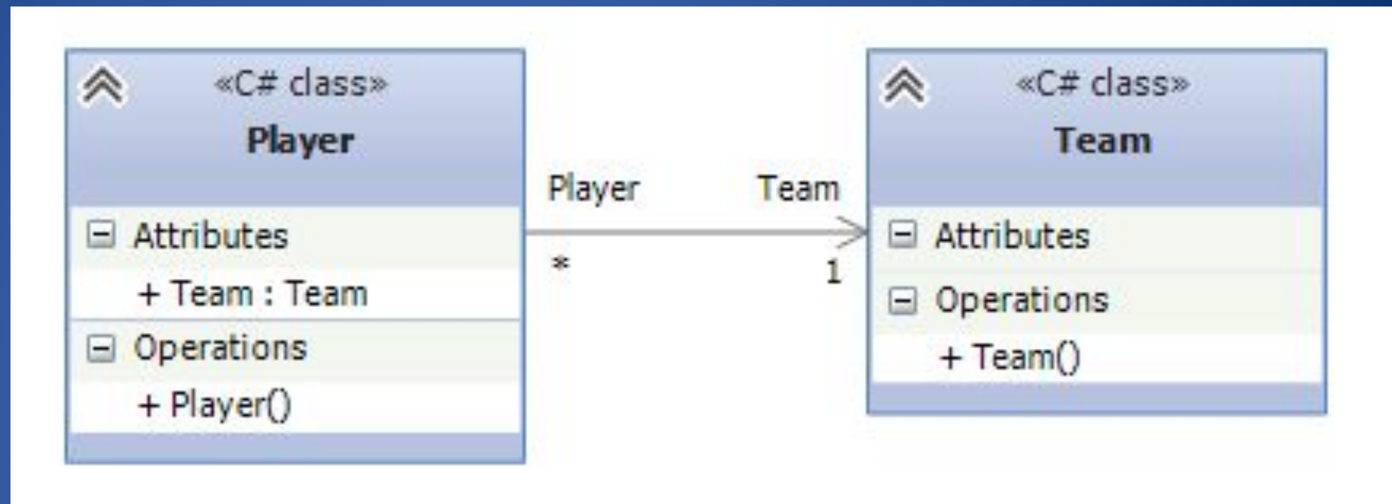


Реализация

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

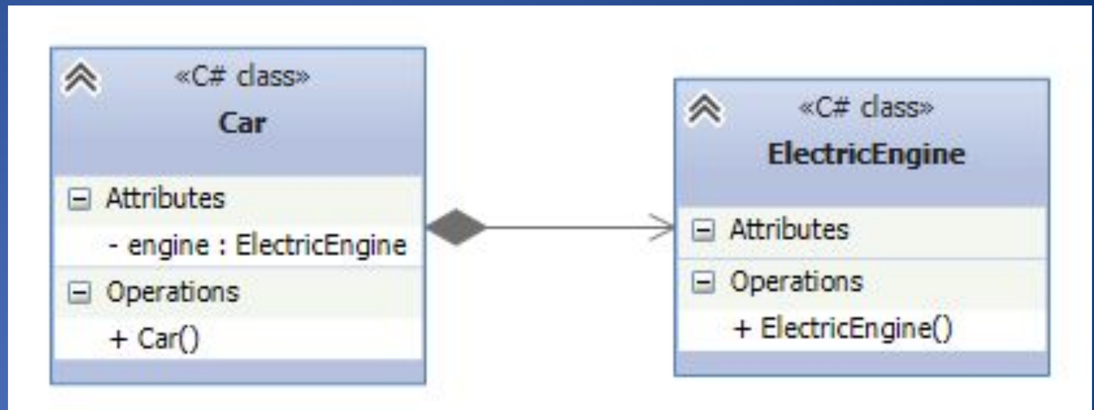


Ассоциация



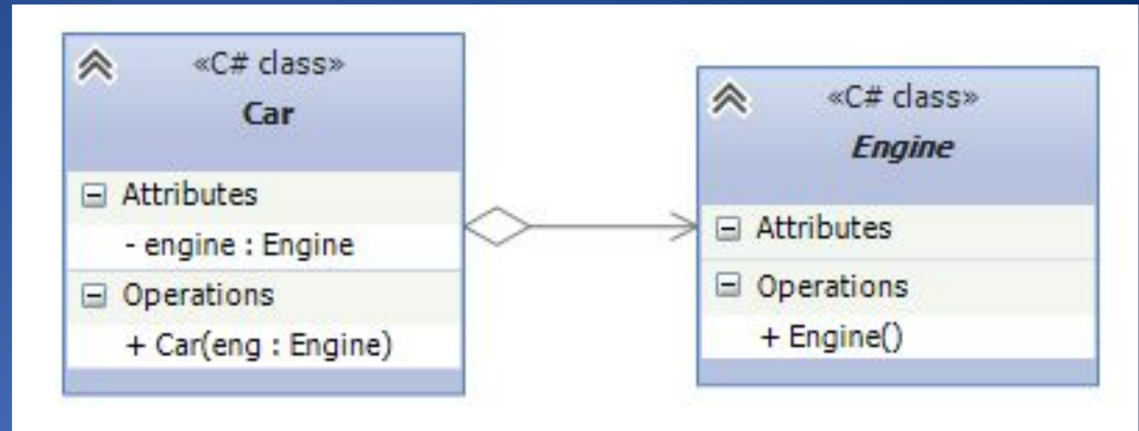

```
class Team
{

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

Композиция




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

Агрегация




```
public abstract class Engine
{ }

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

Объекты Car и Engine будут равноправны

Абстрактные классы. Когда следует их использовать:

- Если надо определить общий функционал для родственных объектов.
- Если проектируется большая функциональная единица, которая содержит много базового функционала.
- Если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию.

При использовании абстрактных классов, если необходимо изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.

```
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");
    }
}
```

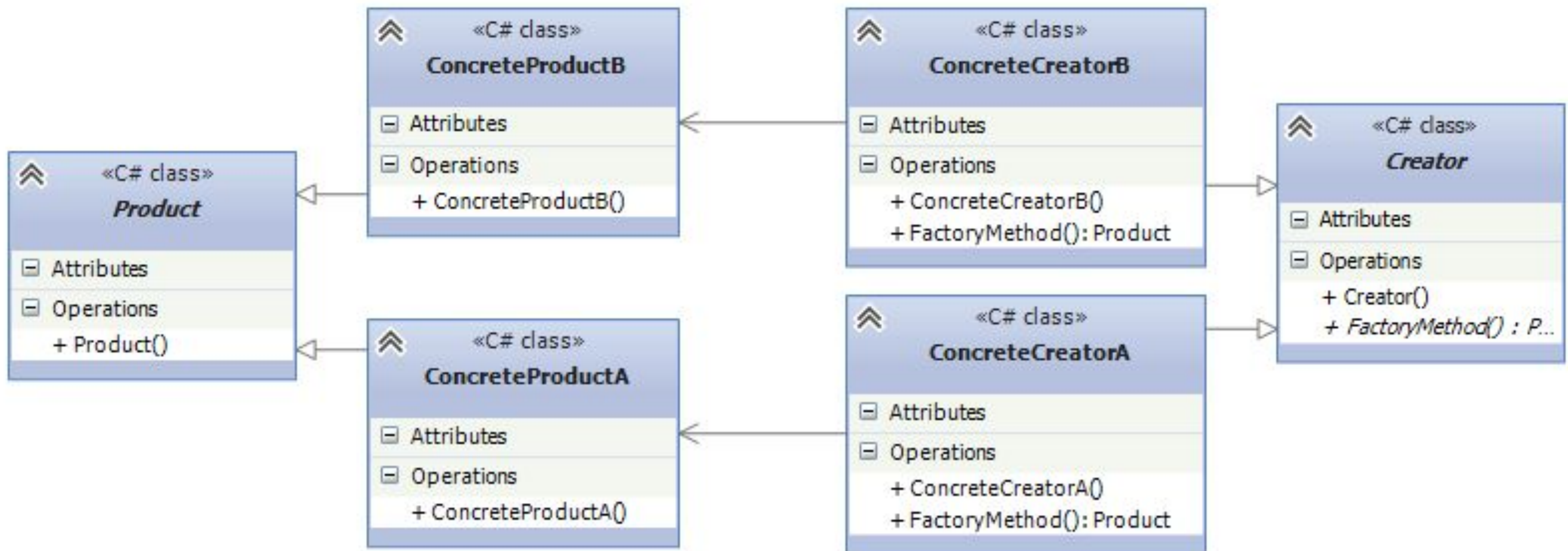
Интерфейсы. Когда следует их использовать:

- Если надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если проектируется небольшой функциональный тип.

```
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");
    }
}
```

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. Паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.



Когда надо применять паттерн Фабричный метод :

- 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.

```
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(); }
}
```

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, и предоставляет к этому объекту точку доступа.

```
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() конструктор вызывается для создания объекта (если, конечно, объект отсутствует и равен null).

Одиночка (Singleton, 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;
    }
}
```