**Лабораторная работа № 3**

**Разработка программ с использованием порождающих паттернов**

*Цель работы*: научиться разрабатывать программы с использованием таких структурных шаблонов проектирования из каталога GoF, как *Строитель, Абстрактная фабрика, Фабричный метод*.

## 1. Теоретические сведения

**Абстрактная фабрика** – порождающий шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы.

Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение.

Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс. Этот шаблон предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Как и в реальной жизни фабрика имеет некую специализацию, создавая товары или устройства какого-либо определенного типа. Фабрика, которая выпускает, например, мебель, не может производить, например, еще и компоненты для смартфонов.

*В программировании фабрика объектов может создавать только объекты определенного типа, которые используют единый интерфейс.*

Самыми главными преимуществами данного паттерна, является упрощение создания объектов различных классов, использующих единый интерфейс.

Паттерн предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

От класса «абстрактная фабрика» наследуются классы конкретных фабрик, которые содержат методы создания конкретных объектов-продуктов, являющихся наследниками класса «абстрактный продукт», объявляющего интерфейс для их создания.

Клиент пользуется только интерфейсами, заданными в классах «абстрактная фабрика» и «абстрактный продукт» (рисунок 1).

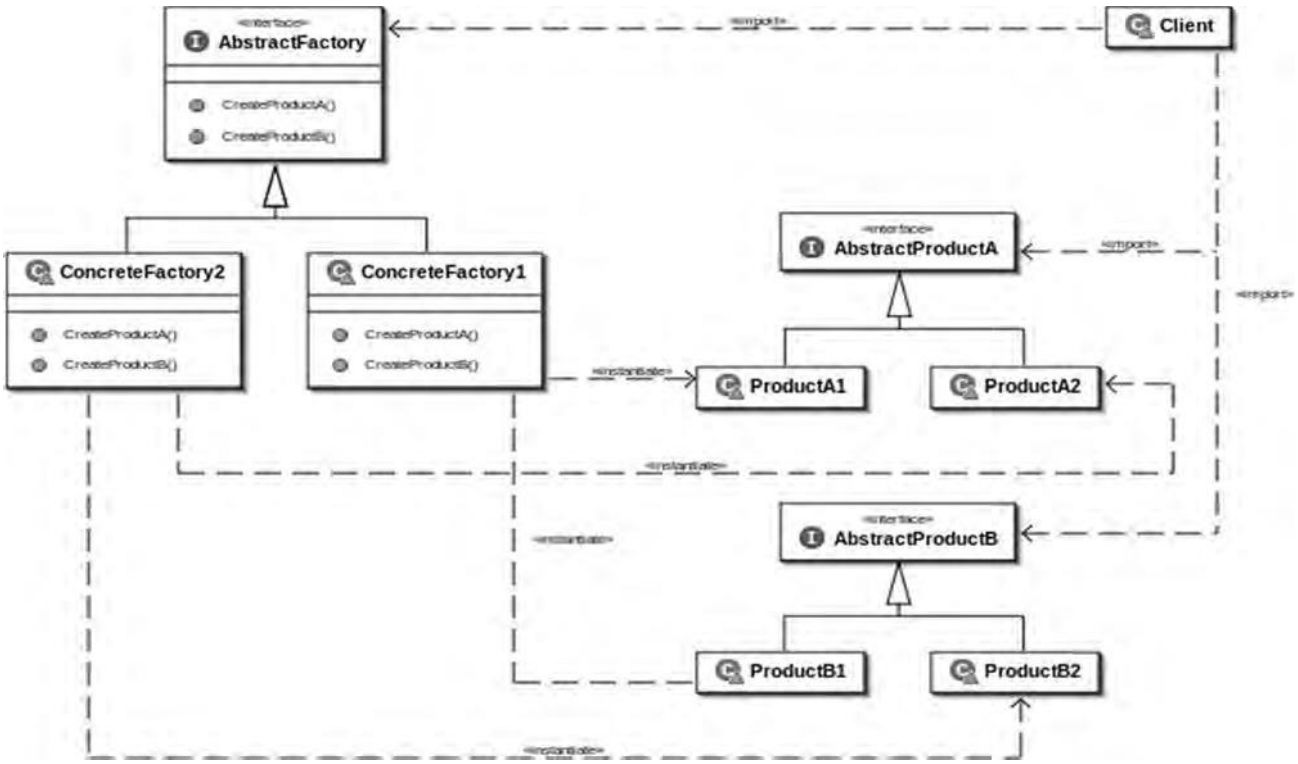



Рисунок 1 – Диаграмма классов паттерна *Абстрактный продукт*

Ниже приведена формальная реализация паттерна на языке C#.

```
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()
        { }
    }
```

**Фабричный метод** – порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса (рисунок 2).

В момент создания наследники могут определить, какой класс создавать. Иными словами, Фабрика делегирует создание объектов наследникам родительского класса.

Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Шаблон определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

*Структура*:

`Product` – продукт; определяет интерфейс объектов, создаваемых абстрактным методом.

`ConcreteProduct` – конкретный продукт, реализует интерфейс `Product`.

`Creator` – создатель; объявляет фабричный метод, который возвращаетобъект типа `Product`.

Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа *`Product`*.

`ConcreteCreator` – конкретный создатель; переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса `ConcreteProduct`.

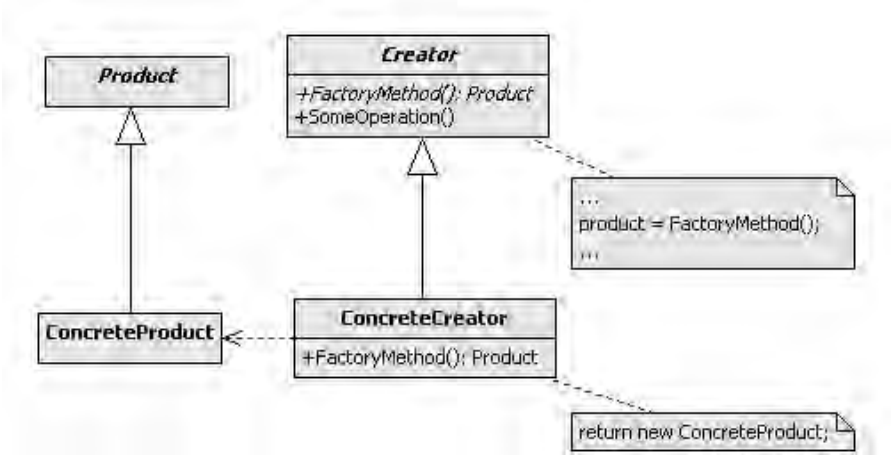



Рисунок 2 – Диаграмма классов паттерна *Фабричный метод*

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 newConcreteProductA(); }
}

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

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

Паттерн Строитель рекомендуется использовать в следующих случаях:

- 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.

Диаграмма классов паттерна показана на рисунке 3.

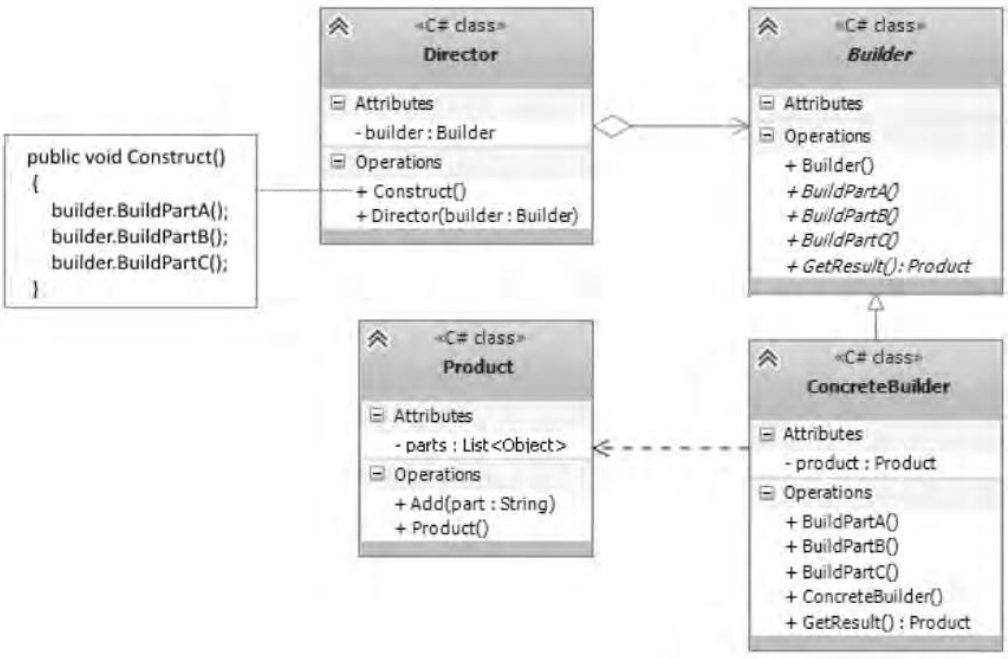



Рисунок 3 –Диаграмма классов паттерна *Строитель*

Формальное описание паттерна на языке C#.

```
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 паттерна:

- `Product`: представляет объект, который должен быть создан. В данномслучае 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, используяобъекты `Builder`.

## 2. Задания к лабораторной работе

Для заданного варианта задания разработать UML-диаграмму классов и диаграмму последовательности.

Разработать консольное приложение (C++, C#). Допускается вводить дополнительные понятия предметной области. В программе предусмотреть тестирование функциональности созданных объектов классов.

Отчет по лабораторной работе – файл формата pdf. Формат имени файла отчета: <НомерГруппы >_<ФамилияСтудента>.pdf. В отчет включить построенные диаграммы и исходный код программы (при необходимости и заголовочные файлы). Формат отчета см. в Приложении. Отчет загрузить в LMS.

**При защите лабораторной работы:** уметь объяснить логику и детали работы программы; реализацию паттернов проектирования на примере разработанной программы.

**Варианты заданий**

1. Паттерн **Builder**. Имеется текст статьи в формате TXT. Статья состоит из заголовка (первая строка), фамилий авторов (вторая строка), самого текста статьи и хеш-кода текста статьи (последняя строка). Написать приложение, позволяющее конвертировать документ в формате TXT в документ формата XML. Необходимо также проверять корректность хеш-кода статьи.
2. Паттерн **Abstract Factory**. Разработать систему Кинопрокат. Пользователь может выбрать определённую киноленту, при заказе киноленты указывается язык звуковой дорожки, который совпадает с языком файла субтитров. Система должна поставлять фильм с требуемыми характеристиками, причём при смене языка звуковой дорожки должен меняться и язык файла субтитров и наоборот.
3. Паттерн **Factory Method**. Фигуры игры «тетрис». Реализовать процесс случайного выбора фигуры из конечного набора фигур. Предусмотреть появление супер-фигур с большим числом клеток, чем обычные.

Распределение вариантов заданий

| *Варианты заданий*<br>*студенты выбирают самостоятельно* | |
|---|---|
| Количество<br>реализованных вариантов | Оценка |
| 1<br>2<br>3 | «удовлетворительно»<br>«хорошо»<br>«отлично» |

**Технологии конструирования программного обеспечения**

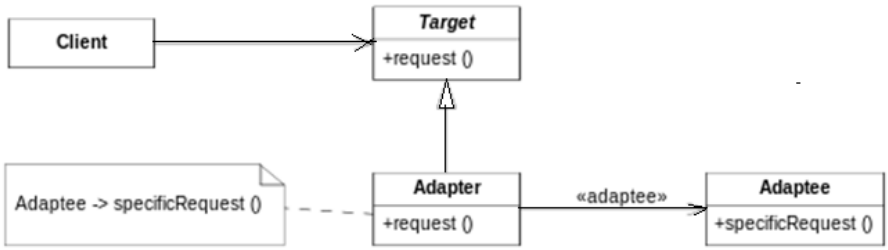
**Отчет по лабораторной работе № 00**

**Группа**: 000-000 **Студент**: Иванов Иван Иванович

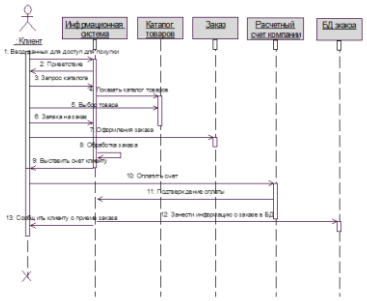
## Задание на лабораторную работу

<Формулировка задания согласно варианту …….
……..>

## Диаграмма классов




## Диаграмма последовательности



## Исходный код программы

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Add/Add.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestMethod1)
		{
			Assert::AreEqual(2, add(1, 1));
		}
```

……