

Санкт-Петербургский национальный исследовательский  
университет информационных технологий, механики и оптики

Факультет прикладной информатики  
Направление подготовки 11.04.02

Лабораторная работа №4  
«Проектирование объектной модели»

Выполнил:  
Швалов Даниил Андреевич К4112с  
Проверила:  
Болгова Екатерина Владимировна

Санкт-Петербург  
2025

## СОДЕРЖАНИЕ

1 Введение.....	3
2 Ход работы.....	3
2.1 Разработка IDEF4 диаграмм.....	3
2.2 Разработка UML диаграмм.....	5
2.3 Применение паттернов проектирования GoF.....	8
2.3.1 Адаптер.....	8
2.3.2 Абстрактная фабрика.....	10
2.3.3 Фабричный метод.....	16
2.3.4 Одиночка.....	17
2.3.5 Стратегия.....	18
2.3.6 Шаблонный метод.....	22
2.3.7 Фасад.....	25
2.3.8 Цепочка обязанностей.....	26
2.3.9 Команда.....	27
2.3.10 Декоратор.....	31
3 Вывод.....	34

## **1 Введение**

Цель работы: изучить методику построения объектно-ориентированных систем IDEF4, основы разработки объектных моделей с использованием шаблонов GRASP для распределения обязанностей между классами, освоить применение шаблонов проектирования GoF.

Задачи:

- 1) разработать статические, динамические и поведенческие IDEF4 диаграммы системы;
- 2) разработать в нотации UML диаграмму последовательностей и диаграмму классов на основе шаблонов для распределения обязанностей между классами;
- 3) освоить применение паттернов проектирования GoF.

## **2 Ход работы**

### **2.1 Разработка IDEF4 диаграмм**

Для ГИС поиска точек общественного питания были разработаны статические, динамические и поведенческие IDEF4 диаграммы. На рисунке 1 представлена статическая IDEF4 диаграмма. На ней основные модели данных, которые присутствуют в системе, а именно:

- 1) Point — это сущность, которая представляет точку общественного питания, хранит информацию о расположении;
- 2) PointTag — это сущность, которая представляет собой мета-информацию о точке;
- 3) Menu — это сущность, которая представляет собой меню и информацию о нем;
- 4) MenuCategory — это сущность, которая представляет собой категорию меню и содержит имя категории;
- 5) MenuItem — это сущность, которая содержит себе всю информацию о позиции в меню, например, название, описание, стоимость и т. п.;
- 6) MenuItemIngredient — это сущность, которая содержит в себе

информацию об ингредиентах.

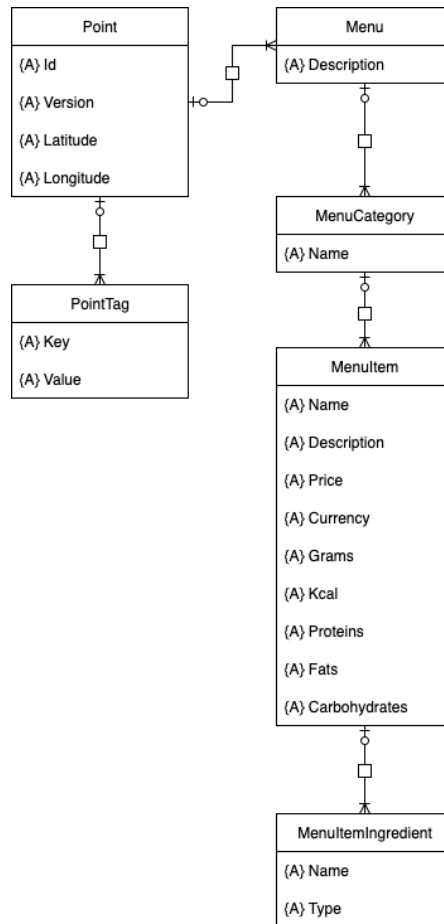


Рисунок 1 — Статическая IDEF4 диаграмма системы

На рисунке 2 представлена динамическая IDEF4 диаграмма системы. На ней изображен процесс перехода из одного состояния в другое состояние сущности меню. Как видно на диаграмме, любое изменение меню приводит к изменению номера версии меню. Благодаря этому в системе появляется возможность посмотреть историю изменений меню.

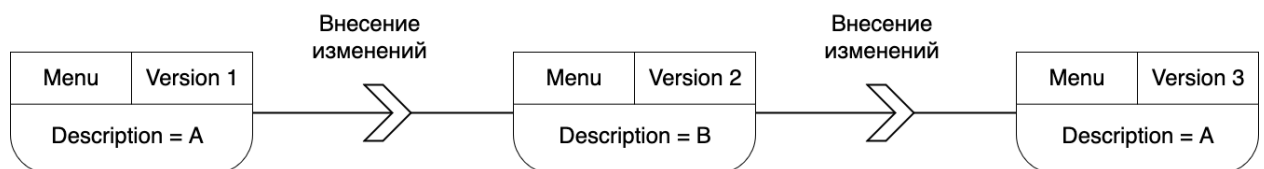


Рисунок 2 — Динамическая IDEF4 диаграмма системы

На рисунке 3 представлена поведенческая IDEF4 диаграмма системы. На ней изображен процесс поиска точки, который состоит из поиска по

имени, по расположению, по характеристикам точки и по ингредиентам из меню.

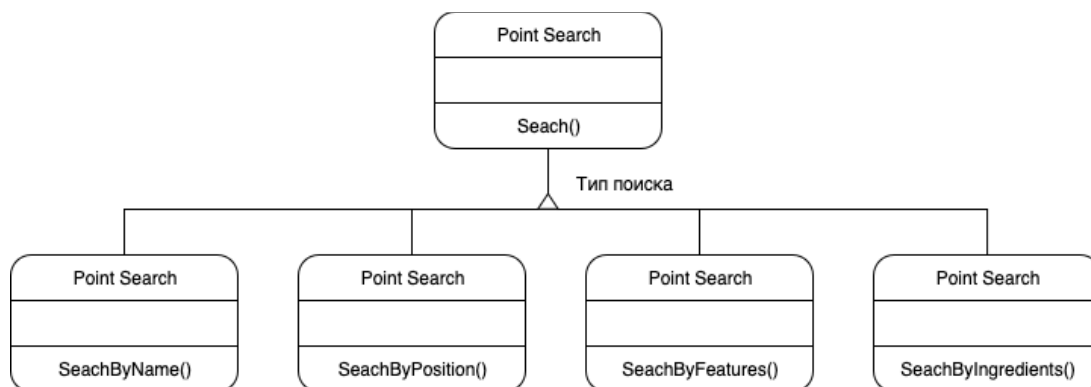


Рисунок 3 — Поведенческая IDEF4 диаграмма системы

## 2.2 Разработка UML диаграмм

Для системы была разработана диаграмма классов, представленная на рисунке 4. На ней представлены все основные классы, присутствующие в системе:

- 1) Point — это класс, который представляет информацию о точке (например, расположение, теги и т. п.);
- 2) MenuItemIngredient — это класс, который содержит информацию об ингредиентах блюд (например, название, тип);
- 3) MenuItem — это класс, который содержит информацию о позиции в меню (например, название, стоимость, вес и т. п.). Включает в себя MenuItemIntgredient;
- 4) MenuCategory — это класс, который содержит информацию о категориях в меню. Включает в себя MenuItem;
- 5) Menu — это класс, который представляет информацию о меню. Включает в себя MenuCategory;
- 6) Image — это класс, который представляет из себя информацию о изображении (например, содержимое изображения в байтах, мета информация);
- 7) Analytics — это класс, который представляет из себя аналитическую

информацию (например, название, значения);

8) `PointsRepository`, `MenuRepository`, `ImagesRepository`, `AnatylicsRepository` — это классы, который предоставляют методы для доступа к базе данных, а также преобразуют данные из базы данных в соответствующие классы;

9) `PointsManager`, `MenuManager`, `ImagesManager`, `AnalyticsManager` — это классы, которые реализуют бизнес-логику операций доступа и изменения к объектам (например, к меню или точкам). Данные классы используют соответствующие классы-репозитории для доступа к данным.

10) `ApiManager` — это класс, который обрабатывает все входные запросы, преобразует их в понятное для всех остальных классов представление, проверяет данные на корректность.

При проектировании классов были применены следующие шаблоны GRASP:

1) `Information Expert` — согласно этому шаблону обязанности по выполнению бизнес-логики были распределены по соответствующи классам-менеджерам (например, `PointsManager`);

2) `Creator` — согласно этому шаблону обязанность по созданию классов была распределена по соответствующим классам-репозиториям (например, `PointsRepository`);

3) `Controller` и `Redirection` — согласно этим шаблонам, все запросы пользователя проходят через и распределяются с помощью `ApiManager`;

4) `Low Coupling` — согласно этому шаблону логически разные сущности были отделены в отдельные классы (например, у `Point` и `Menu` разные классы-менеджеры и классы-репозитории);

5) `High Cohesion` — согласно этому шаблону разные элементы системы были разбиты на разные классы;

6) `Protected Variations` — согласно этому шаблону вся бизнес-логика и все взаимодействие с данными происходит через соответствующие классы.

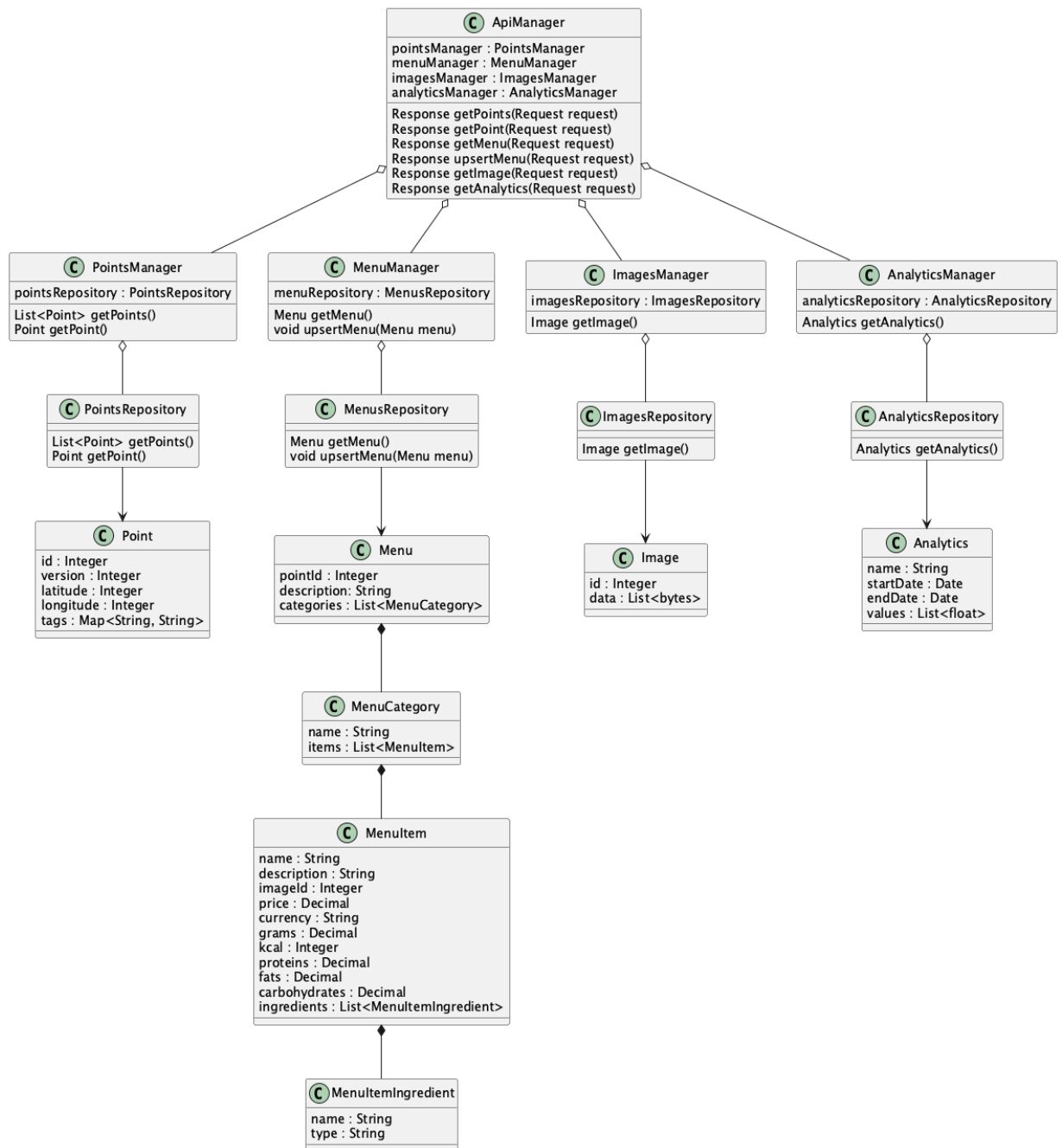


Рисунок 4 — Диаграмма классов системы

Также для системы была построена диаграмма последовательности, представленная на рисунке 5. На ней показано взаимодействие между классами при различных запросах пользователя к системе.

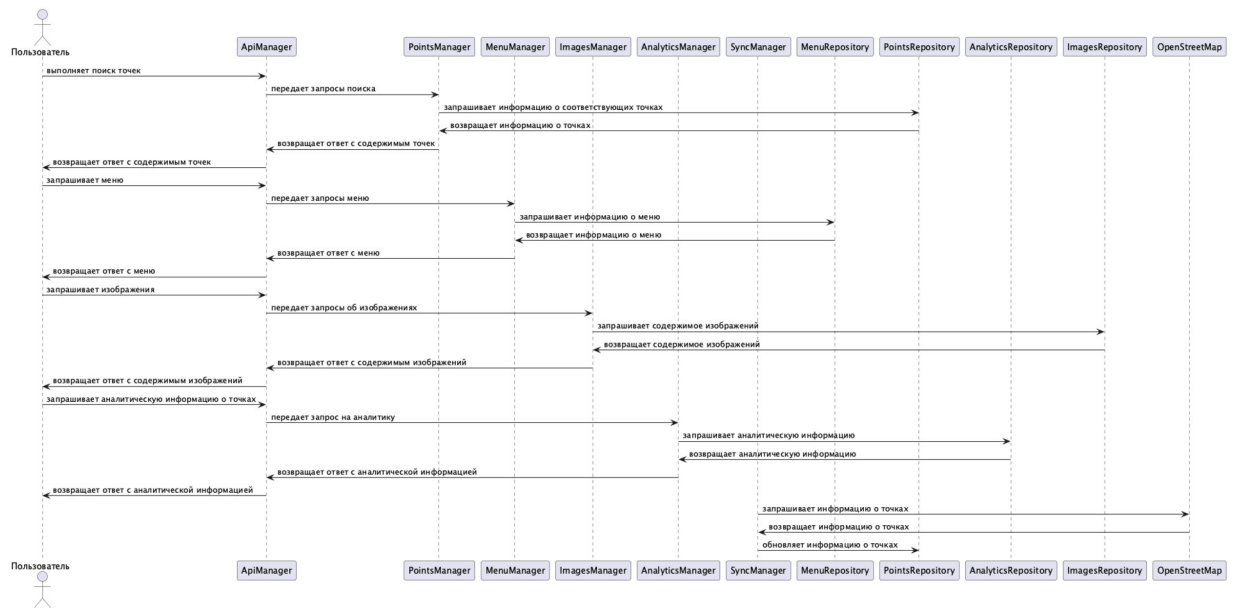


Рисунок 5 — Диаграмма последовательности классов системы

## 2.3 Применение паттернов проектирования GoF

### 2.3.1 Адаптер

В данном упражнении было необходимо разработать адаптер, преобразующий температуру в шкалу Цельсия, для системы климат-контроля.

Для реализации адаптер был выделен интерфейс `ICelciusTemperatureSensor`, который имеет метод `GetValue`, возвращающее температуру в цельсиях. Также был создан класс `FahrenheitTemperatureSensor`, который представляет из себя датчик, предоставляющий информацию о температуре в фаренгейтах. Для класса `FahrenheitTemperatureSensor` был написан адаптер `AdapterFahrenheitTemperatureSensor`, реализующий интерфейс `ICelciusTemperatureSensor`. Исходный код данных классов представлен на рисунке 6.



```

interface ICelsiusTemperatureSensor
{
    double GetValue();
}

class FahrenheitTemperatureSensor
{
    Random random;

    public FahrenheitTemperatureSensor()
    {
        random = new Random();
    }

    public double GetValue()
    {
        // Случайное число от 32 до 95.
        return random.Next(95) + 32;
    }
}

class AdapterFahrenheitTemperatureSensor : ICelsiusTemperatureSensor
{
    FahrenheitTemperatureSensor sensor;

    public AdapterFahrenheitTemperatureSensor()
    {
        sensor = new FahrenheitTemperatureSensor();
    }

    public double GetValue()
    {
        return (sensor.GetValue() - 32) * 5.0 / 9.0;
    }
}

```

Рисунок 6 — Исходный код классов ICelsiusTemperatureSensor, FahrenheitTemperatureSensor и AdapterFahrenheitTemperatureSensor

После этого был разработан класс Controller, который использует объект, реализующий интерфейс ICelsiusTemperatureSensor, для получения информации о температуре. Экземпляр класса Controller создается и вызывается в функции Main. Исходный код класса Controller и функции Main представлен на рисунке 7.

```

class Controller
{
    ICelsiusTemperatureSensor sensor;

    public Controller()
    {
        sensor = new AdapterFahrenheitTemperatureSensor();
    }

    public void CheckSensor()
    {
        Console.WriteLine("Сенсор показал {0:f2} градусов", sensor.GetValue());
    }
}

class Program
{
    static void Main(string[] args)
    {
        Controller controller = new Controller();
        controller.CheckSensor();
    }
}

```

Рисунок 7 — Исходный код класса Controller и функции Main

### 2.3.2 Абстрактная фабрика

В данном упражнении было необходимо добавить классы для фабрики, создающей автомобиль Audi. Для этого было созданы три класса:

- 1) AudiFactory — класс, реализующий интерфейс CarFactory и создающий экземпляры класса AudiCar;
- 2) AudiCar — класс, реализующий интерфейс AbstractCar и представляющий автомобиль марки Audi;
- 3) AudiEngine — класс, реализующий интерфейс AbstractEngine и представляющий двигатель марки Audi.

Исходный код вышеперечисленных классов представлен на рисунке 8.

```

class AudiFactory : CarFactory
{
    public override AbstractCar CreateCar()
    {
        return new AudiCar("Ауди");
    }

    public override AbstractEngine CreateEngine()
    {
        return new AudiEngine();
    }
}

class AudiCar : AbstractCar
{
    public AudiCar(string name)
    {
        Name = name;
    }

    public override int MaxSpeed(AbstractEngine engine)
    {
        int ms = engine.max_speed;
        return ms;
    }

    public override string ToString()
    {
        return "Автомобиль " + Name;
    }
}

class AudiEngine : AbstractEngine
{
    public AudiEngine()
    {
        max_speed = 300;
    }
}

```

Рисунок 8 — Исходный код классов AudiFactory, AudiCar и AudiEngine

После этого в функцию Main было добавлено создание и использование экземпляра класса AudiFactory. Исходный код получившейся функции Main представлен на рисунке 9.

```
abstract class AbstractBody
{
    public string name { get; set; }
}

class SedanBody : AbstractBody
{
    public SedanBody()
    {
        name = "Седан";
    }
}

class CrossoverBody : AbstractBody
{
    public CrossoverBody()
    {
        name = "Кроссовер";
    }
}
```

```
abstract class CarFactory
{
    public abstract AbstractCar CreateCar();
    public abstract AbstractEngine CreateEngine();
    public abstract AbstractBody CreateBody();
}
```

```

class Program
{
    public static void Main()
    {
        CarFactory ford_car = new FordFactory();
        Client c1 = new Client(ford_car);
        Console.WriteLine(
            "Максимальная скорость {0} составляет {1} км/час",
            c1.ToString(),
            c1.RunMaxSpeed()
        );

        CarFactory audi_car = new AudiFactory();
        Client c2 = new Client(audi_car);
        Console.WriteLine(
            "Максимальная скорость {0} составляет {1} км/час",
            c2.ToString(),
            c2.RunMaxSpeed()
        );
    }
}

```

Рисунок 9 — Исходный код функции Main

После запуска программы на экран была выведена информация, представленная на рисунке 10. Как видно из вывода, скорость и название соответствует указанным в коде выше.

```

Максимальная скорость Автомобиль Форд составляет 220 км/час
Максимальная скорость Автомобиль Ауди составляет 300 км/час

```

Рисунок 10 — Вывод программы

После этого в программу было добавлено новое свойство — тип кузова. Для этого были созданы абстрактный класс `AbstractBody` с полем `name` и два класса, реализующих этот абстрактный класс — `SedanBody` и `CrossoverBody`. Исходный код этих классов представлен на рисунке 11.

```

abstract class AbstractBody
{
    public string name { get; set; }
}

class SedanBody : AbstractBody
{
    public SedanBody()
    {
        name = "Седан";
    }
}

class CrossoverBody : AbstractBody
{
    public CrossoverBody()
    {
        name = "Кроссовер";
    }
}

```

Рисунок 11 — Исходный код классов AbstractBody, SedanBody и CrossoverBody

После этого были внесены изменения в классы CarFactory, FordFactory и AudiFactory: в них были добавлены методы CreateBody. В CarFactory метод CreateBody был помечен как абстрактный, в FordFactory метод создает экземпляр класса SedanBody, а в AudiFactory — CrossoverBody. Исходный код вышеперечисленных классов представлен на рисунках 12-14.

```

abstract class CarFactory
{
    public abstract AbstractCar CreateCar();
    public abstract AbstractEngine CreateEngine();
    public abstract AbstractBody CreateBody();
}

```

Рисунок 12 — Исходный код класса CarFactory

```

class FordFactory : CarFactory
{
    public override AbstractCar CreateCar()
    {
        return new FordCar("Форд");
    }

    public override AbstractEngine CreateEngine()
    {
        return new FordEngine();
    }

    public override AbstractBody CreateBody()
    {
        return new SedanBody();
    }
}

```

Рисунок 13 — Исходный код класса FordFactory

```

class AudiFactory : CarFactory
{
    public override AbstractCar CreateCar()
    {
        return new AudiCar("Ауди");
    }

    public override AbstractEngine CreateEngine()
    {
        return new AudiEngine();
    }

    public override AbstractBody CreateBody()
    {
        return new CrossoverBody();
    }
}

```

Рисунок 14 — Исходный код класса AudiFactory

Внесение вышеперечисленных изменений было простым: оказалось достаточно добавить несколько классов и методов, как-то изменять

существующую логику не пришлось.

### 2.3.3 Фабричный метод

В данном упражнении необходимо было добавить поддержку слуги «трезвый водитель». Для этого был создан класс `SoberDriver`, который реализует интерфейс `TransportService`. Исходный код класса `SoberDriver` представлен на рисунке 15.

```
class SoberDriver : TransportService
{
    public int Category { get; set; }

    public SoberDriver(string name, int category)
        : base(name)
    {
        Category = category;
    }

    public override double CostTransportation(double distance)
    {
        return distance * (Category + 1);
    }

    public override string ToString()
    {
        string s = String.Format(
            "Фирма {0}, услуга трезвый водитель категории {1}",
            Name,
            Category
        );
        return s;
    }
}
```

Рисунок 15 — Исходный код `SoberDriver`

После этого был создан класс `SoberDriverCompany`, реализующий интерфейс `TransportCompany`. Данный класс создает экземпляры ранее вышеописанного класса `SoberDriver`. Исходный код класса `SoberDriverCompany` представлен на рисунке 16



```

class SoberDriverCom : TransportCompany
{
    public SoberDriverCom(string name)
        : base(name) { }

    public override TransportService Create(string n, int t)
    {
        return new SoberDriver(Name, t);
    }
}

```

Рисунок 16 — Исходный код класса SoberDriverCompany

Для добавления новой услуги не пришлось изменять существующий код, достаточно было добавить несколько новых классов, реализующих уже существующий интерфейс.

#### 2.3.4 Одиночка

В данном упражнении необходимо было изменить исходный код из упражнения №2 таким образом, чтобы в конкретной фабрике использовался паттерн «Одиночка». В качестве изменяемой фабрики был выбран класс FordFactory.

В классе FordFactory было создано статическое поле factory с использованием класса Lazy для реализации отложенной инициализации. После этого был создан атрибут Factory, с помощью которого пользовательский код может получить доступ к полю factory. Также конструктор по умолчанию был помечен как приватный для того, чтобы его нельзя было использовать напрямую. Вместо этого пользовательский код должен использовать поле Factory. Исходный код класса FordFactory представлен на рисунке 17.

```

class FordFactory : CarFactory
{
    private FordFactory() { }

    static Lazy<CarFactory> factory = new Lazy<CarFactory>(() => new FordFactory());

    public static CarFactory Factory
    {
        get { return factory.Value; }
    }

    public override AbstractCar CreateCar()
    {
        return new FordCar("Форд");
    }

    public override AbstractEngine CreateEngine()
    {
        return new FordEngine();
    }
}

```

Рисунок 17 — Исходный код класса FordFactory

После этого для использования FordFactory в качестве одиночки был изменен исходный код функции Main: теперь вместо создания экземпляров класса FordFactory используется статический атрибут Factory. Исходный код функции Main представлен на рисунке 18.

```

class Program
{
    public static void Main()
    {
        Client c1 = new Client(FordFactory.Factory);
        Console.WriteLine(
            "Максимальная скорость {0} составляет {1} км/час",
            c1.ToString(),
            c1.RunMaxSpeed()
        );
    }
}

```

Рисунок 18 — Исходный код класса Main

### 2.3.5 Стратегия

В данном упражнении было необходимо спроектировать классы для

приложения навигатора, имеющего возможность показывать карту, реализовывать поиск и прокладку маршрута по автодорогам, пешим маршрутов, маршрутов по велодорожкам, на общественном транспорте, а также маршруты посещения достопримечательностей. Для этого были спроектированы следующие классы:

1) `RouteStrategy` — абстрактный класс стратегии построения маршрутов, имеющий абстрактный метод `FindRoute`;

2) `AutoRouteStrategy` — класс, реализующий интерфейс `RouteStrategy`, использующийся для построения автомобильных маршрутов;

3) `FootRouteStrategy` — класс, реализующий интерфейс `RouteStrategy`, использующийся для построения пеших маршрутов;

4) `CycleRouteStrategy` — класс, реализующий интерфейс `RouteStrategy`, использующийся для построения пеших маршрутов;

5) `PublicTransportRouteStrategy` — класс, реализующий интерфейс `RouteStrategy`, использующийся для построения маршрутов на общественном транспорте;

6) `AttractionRouteStrategy` — класс, реализующий интерфейс `RouteStrategy`, использующийся для построения маршрутов посещения достопримечательностей;

7) `Navigator` — класс, представляющий навигатор, включающий в себя методы отображения карты, поиска, а также построения маршрутов с помощью вышеописанных стратегий.

Исходный код вышеперечисленных классов представлен на рисунках 19-20.

```

abstract class RouteStrategy
{
    public abstract void FindRoute();
};

class AutoRouteStrategy : RouteStrategy
{
    public override void FindRoute()
    {
        Console.WriteLine("Поиск автомобильного маршрута");
    }
};

class FootRouteStrategy : RouteStrategy
{
    public override void FindRoute()
    {
        Console.WriteLine("Поиск пешего маршрута");
    }
};

class CycleRouteStrategy : RouteStrategy
{
    public override void FindRoute()
    {
        Console.WriteLine("Поиск велосипедного маршрута");
    }
};

class PublicTransportRouteStrategy : RouteStrategy
{
    public override void FindRoute()
    {
        Console.WriteLine("Поиск маршрута на общественном транспорте");
    }
};

```

Рисунок 19 — Исходный код классов RouteStrategy, AutoRouteStrategy, FootRouteStrategy, CycleRouteStrategy и PublicTransportRouteStrategy

```

class AttractionRouteStrategy : RouteStrategy
{
    public override void FindRoute()
    {
        Console.WriteLine("Поиск маршрута с достопримечательностями");
    }
};

class Navigator
{
    public void ShowMap()
    {
        Console.WriteLine("Показ карты");
    }

    public void Search()
    {
        Console.WriteLine("Поиск точек");
    }

    public void FindRoute(RouteStrategy strategy)
    {
        strategy.FindRoute();
    }
}

```

Рисунок 20 — Исходный код классов AttractionRouteStrategy и Navigator

В функции Main был создан экземпляр класса Navigator для демонстрации работы различных стратегий построения маршрутов. Исходный код функции Main представлен на рисунке 21.

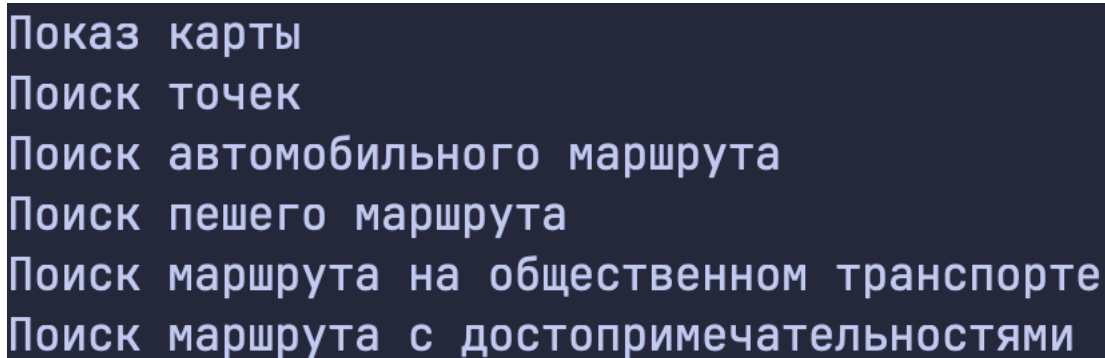
```

class Program
{
    public static void Main()
    {
        Navigator navigator = new Navigator();
        navigator.ShowMap();
        navigator.Search();
        navigator.FindRoute(new AutoRouteStrategy());
        navigator.FindRoute(new FootRouteStrategy());
        navigator.FindRoute(new PublicTransportRouteStrategy());
        navigator.FindRoute(new AttractionRouteStrategy());
    }
}

```

Рисунок 21 — Исходный код функции Main

В выводе программы на рисунке 22 видно, что при построении маршрутов используются различные алгоритмы маршрутизации.



```
Показ карты
Поиск точек
Поиск автомобильного маршрута
Поиск пешего маршрута
Поиск маршрута на общественном транспорте
Поиск маршрута с достопримечательностями
```

Рисунок 22 — Вывод программы

### 2.3.6 Шаблонный метод

В данном упражнении было необходимо применить паттерн «Шаблонный метод» для проектирования процесса стрижки. В процессе стрижки были выделены следующие этапы:

- 1) мытье головы;
- 2) настройка кресла;
- 3) стрижка;
- 4) сушка волос;
- 5) уборка.

У разных стрижек отличается только этап №3. В соответствии с этим, был создан абстрактный класс `Haircut`, который реализует все методы, кроме №3. Метод стрижки, названная `MakeHaircut`, был помечен как абстрактный. Исходный код класса `Haircut` представлен на рисунке 23.

```

abstract class Haircut
{
    public void TemplateMethod()
    {
        WashHair();
        SetupSeat();
        MakeHaircut();
        DryHair();
        CleanUp();
    }

    public void WashHair()
    {
        Console.WriteLine("Мытье головы");
    }

    public void SetupSeat()
    {
        Console.WriteLine("Настройка кресла");
    }

    public abstract void MakeHaircut();

    public void DryHair()
    {
        Console.WriteLine("Сушка волос");
    }

    public void CleanUp()
    {
        Console.WriteLine("Уборка");
    }
}

```

Рисунок 23 — Исходный код класса Haircut

После этого были добавлены два класса ShortHaircut и LongHaircut, наследующие класс Haircut. В данных классах был переопределен метод MakeHaircut, остальные методы остались нетронутыми.

В функции Main была протестирована работа классов. Как видно на рисунках 24-25, при вызове одного и того же метода меняется лишь одна строка в выводе. Все остальные строки вывода остаются такими же.

```

class ShortHaircut : Haircut
{
    public override void MakeHaircut()
    {
        Console.WriteLine("Стрижка волос с помощью машинки");
    }
}

class LongHaircut : Haircut
{
    public override void MakeHaircut()
    {
        Console.WriteLine("Стрижка волос с помощью ножниц");
    }
}

class Program
{
    public static void Main()
    {
        Haircut shortHaircut = new ShortHaircut();
        shortHaircut.TemplateMethod();
        Console.WriteLine("---");
        Haircut longHaircut = new LongHaircut();
        longHaircut.TemplateMethod();
    }
}

```

Рисунок 24 — Исходный код классов ShortHaircut и LongHaircut и функции Main

```

Мытье головы
Настройка кресла
Стрижка волос с помощью машинки
Сушка волос
Уборка
---
Мытье головы
Настройка кресла
Стрижка волос с помощью ножниц
Сушка волос
Уборка

```

Рисунок 25 — Вывод программы



### 2.3.7 Фасад

В данном упражнении было необходимо добавить метод, реализующий приготовление продукта в микроволновой печи. Для этого был добавлен метод Cook, который меняет мощность и направление движения привода. Исходный код метода Cook представлен на рисунке 26.

```
public void Cook()
{
    _notification.StartNotification();
    _power.MicrowavePower = 500;
    _drive.TurLRight();
    _drive.TurLRight();
    _drive.TurLRight();
    _drive.Stop();
    _drive.TurLLeft();
    _drive.TurLLeft();
    _drive.TurLLeft();
    _power.MicrowavePower = 200;
    _drive.Stop();
    _drive.TurLRight();
    _drive.TurLLeft();
    _drive.TurLRight();
    _drive.TurLLeft();
    _drive.Stop();
    _power.MicrowavePower = 0;
    _notification.StopNotification();
}
```

Рисунок 26 — Исходный код метода Cook

После этого в код функции Main был добавлен вызов метода Cook, что видно на рисунке 27. Как видно на рисунке 28, вывод программы соответствует вышеописанному алгоритму приготовления.

```

public static void Main(string[] args)
{
    var drive = new Drive();
    var power = new Power();
    var notification = new Notification();
    var microwave = new Microwave(drive, power, notification);

    power.powerevent += power_powerevent;
    drive.driveevent += drive_driveevent;
    notification.notificationevent += notification_notificationevent;

    Console.WriteLine("Приготовление");
    microwave.Cook();
}

```

Рисунок 27 — Исходный код Main

```

Приготовление
Информация: Операция началась
Задана мощность 500w
Привод: Поверот направо
Привод: Поверот направо
Привод: Поверот направо
Привод: Стоп
Привод: Поверот налево
Привод: Поверот налево
Привод: Поверот налево
Задана мощность 200w
Привод: Стоп
Привод: Поверот направо
Привод: Поверот налево
Привод: Поверот направо
Привод: Поверот налево
Привод: Стоп
Задана мощность 0w
Информация: Операция завершена

```

Рисунок 28 — Вывод программы

### 2.3.8 Цепочка обязанностей

В данном упражнении необходимо было изменить последовательность объектов-обработчиков цепочки, а также внести изменение в значения передаваемых параметров при инициализации. Для этого в функции Main

были внесены изменения так, чтобы был реализован следующий приоритет способов оплаты:

- 1) денежные переводы;
- 2) банковские переводы;
- 3) переводы с помощью PayPal.

Также были изменены параметры таким образом, чтобы выполнялся только перевод с помощью PayPal. Исходный код функции Main представлен на рисунке 29.

```
class Program
{
    public static void Main()
    {
        Receiver receiver = new Receiver(false, false, true);
        PaymentHandler bankPaymentHandler = new BankPaymentHandler();
        PaymentHandler moneyPaymentHandler = new MoneyPaymentHandler();
        PaymentHandler paypalPaymentHandler = new PayPalPaymentHandler();
        moneyPaymentHandler.Successor = bankPaymentHandler;
        bankPaymentHandler.Successor = paypalPaymentHandler;
        moneyPaymentHandler.Handle(receiver);
    }
}
```

Рисунок 29 — Исходный код функции Main

Как видно на рисунке 30, в качестве способа оплаты был выбран PayPal, как и ожидалось.

**Выполняем перевод через PayPal**

Рисунок 30 — Вывод программы

### 2.3.9 Команда

В данном упражнении было необходимо добавить новые типы операций с помощью добавления соответствующих команд. Для этого были созданы классы Sub, Mul и Div, реализующие функции вычитания, умножения и деления соответственно. Исходный код классов представлен на рисунках 31-33.

```

class Sub : Command
{
    public Sub(ArithmeticUnit unit, double operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('-', operand);
    }

    public override void UnExecute()
    {
        unit.Run('+', operand);
    }
}

```

Рисунок 31 — Исходный код класса Sub

```

class Mul : Command
{
    public Mul(ArithmeticUnit unit, double operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('*', operand);
    }

    public override void UnExecute()
    {
        unit.Run('/', operand);
    }
}

```

Рисунок 32 — Исходный код класса Mul

```

class Div : Command
{
    public Div(ArithmeticUnit unit, double operand)
    {
        this.unit = unit;
        this.operand = operand;
    }

    public override void Execute()
    {
        unit.Run('/', operand);
    }

    public override void UnExecute()
    {
        unit.Run('*', operand);
    }
}

```

Рисунок 33 — Исходный код класса Div

После этого в класс Calculator были добавлены методы вычитания, умножения и деления, использующие соответствующие экземпляры классов. Исходный код добавленных методов представлен на рисунке 34.

```

public double Add(double operand)
{
    return Run(new Add(arithmeticUnit, operand));
}

public double Sub(double operand)
{
    return Run(new Sub(arithmeticUnit, operand));
}

public double Mul(double operand)
{
    return Run(new Mul(arithmeticUnit, operand));
}

public double Div(double operand)
{
    return Run(new Div(arithmeticUnit, operand));
}

```

Рисунок 34 — Исходный код методов вычитания, умножения и деления

После этого было необходимо реализовать поддержку многоуровневой отмены и повтора операций. Для этого были реализованы перегруженные методы Undo и Redo в классе Calculator. Для реализации многоуровневости были использованы циклы. Исходный код получившихся методов представлен на рисунке 35.

```
public void Undo()
{
    Undo(1);
}

public void Undo(int count)
{
    for (int i = 1; i <= count; ++i)
    {
        commands[current - i].UnExecute();
    }
}

public void Redo()
{
    Redo(1);
}

public void Redo(int count)
{
    for (int i = count; i >= 1; --i)
    {
        commands[current - i].Execute();
    }
}
```

Рисунок 35 — Исходный код методов Undo и Redo

После этого были внесены доработки в функцию Main для тестирования вышеописанных изменений. Как видно на рисунках 36-37, операции вычитания, умножения, деления, а также многоуровневая отмена и повтор работают корректно.

```

class Program
{
    public static void Main()
    {
        var calculator = new Calculator();
        double result = 0;
        result = calculator.Add(5);
        Console.WriteLine(result);
        result = calculator.Sub(3);
        Console.WriteLine(result);
        result = calculator.Mul(6);
        Console.WriteLine(result);
        result = calculator.Undo(2);
        Console.WriteLine(result);
        result = calculator.Redo(2);
        Console.WriteLine(result);
    }
}

```

Рисунок 36 — Исходный код функции Main

```

5
2
12
5
12

```

Рисунок 37 — Вывод программы

### 2.3.10 Декоратор

В данном упражнении было необходимо добавить класс нового автомобиля и новые функциональные возможности. Для этого были добавлены следующие классы:

- 1) PressureSensors — класс, реализующий интерфейс DecoratorOptions и представляющий функции датчиков давления в шинах;
- 2) CruiseControl — класс, реализующий интерфейс DecoratorOptions и представляющий функции круиз-контроля;
- 3) Toyota — класс, реализующий интерфейс AutoBase и представляющий собой автомобиль марки Toyota.

Исходный код данных классов представлен на рисунках 38-39.

```

class PressureSensors : DecoratorOptions
{
    public PressureSensors(AutoBase p, string t)
        : base(p, t)
    {
        AutoProperty = p;
        Name = p.Name + ". С датчиками давления в шинах";
        Description =
            p.Description
            + ". "
            + this.Title
            + ". Датчики давления в шинах, система обнаружения пониженного давления в шинах";
    }

    public override double GetCost()
    {
        return AutoProperty.GetCost() + 5.99;
    }
}

class CruiseControl : DecoratorOptions
{
    public CruiseControl(AutoBase p, string t)
        : base(p, t)
    {
        AutoProperty = p;
        Name = p.Name + ". С круизом-контролем";
        Description =
            p.Description
            + ". "
            + this.Title
            + ". Круиз-контроль — система автоматической поддержки скорости";
    }

    public override double GetCost()
    {
        return AutoProperty.GetCost() + 15.99;
    }
}

```

Рисунок 38 — Исходный код классов PressureSensors и CruiseControl

```

class Toyota : AutoBase
{
    public Toyota(string name, string info, double costbase)
    {
        Name = name;
        Description = info;
        CostBase = costbase;
    }

    public override double GetCost()
    {
        return CostBase * 2.05;
    }
}

```

Рисунок 39 — Исходный код класса Toyota



После этого была изменена функция Main для тестирования новых классов. Исходный код функции Main представлен на рисунке 40.

```
public static void Main(string[] args)
{
    Toyota toyota = new Toyota("Toyota", "Toyota Camry", 799.0);
    Print(toyota);
    AutoBase toyota1 = new PressureSensors(
        new MediaNAV(toyota, "Навигация"),
        "Датчики давления в шинах"
    );
    Print(toyota1);
    AutoBase toyota2 = new CruiseControl(new MediaNAV(toyota, "Навигация"), "Круиз-контроль");
    Print(toyota2);
}
```

Рисунок 40 — Исходный код функции Main

Как видно на рисунке 41, новые функции работают так, как ожидается: описание функций изменяется, равно как и стоимость автомобиля.

```
Ваш автомобиль:
Toyota
Описание: Toyota Camry
Стоимость 1637.9499999999998

Ваш автомобиль:
Toyota. Современный. С датчиками давления в шинах
Описание: Toyota Camry. Навигация. Обновленная мультимедийная навигационная
система. Датчики давления в шинах. Датчики давления в шинах, система обнару
жения пониженного давления в шинах
Стоимость 1659.9299999999998

Ваш автомобиль:
Toyota. Современный. С круизом-контролем
Описание: Toyota Camry. Навигация. Обновленная мультимедийная навигационная
система. Круиз-контроль. Круиз-контроль – система автоматической поддержки
скорости
Стоимость 1669.9299999999998
```

Рисунок 41 — Вывод программы

При использовании декораторов нет необходимости в создании дополнительных классов для получения требуемой функциональности: достаточно создать экземпляр класса, скомпоновав уже существующие классы. Это сильно упрощает написание, читаемость и поддержку кодовой базы.

### **3 Вывод**

В ходе выполнения данной лабораторной работы были изучены методика построения объектно-ориентированных систем IDEF4, основы разработки объектных моделей с использованием шаблонов GRASP для распределения обязанностей между классами, освоено применение шаблонов проектирования GoF.

При применении IDEF4 и GRASP с UML были выявлены следующие отличия данных методологий:

- 1) использование GRASP с UML позволяет более гибко описать систему;
- 2) использование GRASP с UML позволяет использовать только те элементы, которые нужны для конкретного проекта;
- 3) IDEF4 помогает достаточно просто визуализировать структуру и поведение объектов;
- 4) для UML гораздо больше готовых инструментов, чем для IDEF4.

Исходя из этого, можно сделать вывод, что в большинстве случаев имеет смысл использовать GRASP с UML вместо IDEF4.