**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 10 Структурные паттерны. Фасад. Компоновщик

**Цель:**

Ознакомиться с применением структурного паттерна **Фасад (Facade)** для упрощения взаимодействия с подсистемами сложных систем.

**Теоретические сведения:**

**Фасад (Facade)** — это структурный паттерн, который предоставляет упрощённый интерфейс для работы с подсистемами. Он позволяет скрыть сложную логику и взаимодействие между несколькими классами, предоставив пользователю единый интерфейс.

**Пример:**

В рамках данной лабораторной работы рассмотрим пример домашнего кинотеатра, который состоит из нескольких подсистем: аудиосистемы, видеопроектора и системы освещения. Управление этими системами может быть сложным для пользователя, поэтому реализуем класс **Facade**, который будет упрощать взаимодействие с этими системами.

**Задачи:**

1. Реализовать подсистемы:
   * **AudioSystem** для управления звуком
   * **VideoProjector** для управления видео
   * **LightingSystem** для управления освещением
2. Создать класс **HomeTheaterFacade**, который будет предоставлять упрощённый интерфейс для управления домашним кинотеатром.
3. Написать клиентский код для демонстрации работы паттерна **Facade**.

**Ход работы**

1. Реализация подсистем

using System;

public class AudioSystem

{

public void TurnOn()

{

Console.WriteLine("Audio system is turned on.");

}

public void SetVolume(int level)

{

Console.WriteLine($"Audio volume is set to {level}.");

}

public void TurnOff()

{

Console.WriteLine("Audio system is turned off.");

}

}

public class VideoProjector

{

public void TurnOn()

{

Console.WriteLine("Video projector is turned on.");

}

public void SetResolution(string resolution)

{

Console.WriteLine($"Video resolution is set to {resolution}.");

}

public void TurnOff()

{

Console.WriteLine("Video projector is turned off.");

}

}

public class LightingSystem

{

public void TurnOn()

{

Console.WriteLine("Lights are turned on.");

}

public void SetBrightness(int level)

{

Console.WriteLine($"Lights brightness is set to {level}.");

}

public void TurnOff()

{

Console.WriteLine("Lights are turned off.");

}

}

1. Реализация класса фасада

public class HomeTheaterFacade

{

private AudioSystem \_audioSystem;

private VideoProjector \_videoProjector;

private LightingSystem \_lightingSystem;

public HomeTheaterFacade(AudioSystem audioSystem, VideoProjector videoProjector, LightingSystem lightingSystem)

{

\_audioSystem = audioSystem;

\_videoProjector = videoProjector;

\_lightingSystem = lightingSystem;

}

public void StartMovie()

{

Console.WriteLine("Preparing to start the movie...");

\_lightingSystem.TurnOn();

\_lightingSystem.SetBrightness(5);

\_audioSystem.TurnOn();

\_audioSystem.SetVolume(8);

\_videoProjector.TurnOn();

\_videoProjector.SetResolution("HD");

Console.WriteLine("Movie started.");

}

public void EndMovie()

{

Console.WriteLine("Shutting down movie...");

\_videoProjector.TurnOff();

\_audioSystem.TurnOff();

\_lightingSystem.TurnOff();

Console.WriteLine("Movie ended.");

}

}

1. Клиентский код

class Program

{

static void Main(string[] args)

{

// Create subsystems

AudioSystem audio = new AudioSystem();

VideoProjector video = new VideoProjector();

LightingSystem lights = new LightingSystem();

// Create facade

HomeTheaterFacade homeTheater = new HomeTheaterFacade(audio, video, lights);

// Start the movie

homeTheater.StartMovie();

Console.WriteLine();

// End the movie

homeTheater.EndMovie();

}

}

**Вопросы для закрепления:**

1. Какую задачу решает паттерн **Facade**?
2. В каких ситуациях наиболее эффективно применение паттерна **Facade**?
3. Какие преимущества и недостатки имеет паттерн **Facade**?

**Цель:**

Изучить и реализовать структурный паттерн **Компоновщик (Composite)** на языке C# для организации древовидной структуры объектов.

**Теоретическая справка:**

**Компоновщик (Composite)** — это структурный паттерн проектирования, который позволяет объединить объекты в древовидные структуры для представления иерархии "часть-целое". Компоновщик позволяет клиентам работать с отдельными объектами и группами объектов одинаково.

Пример: Паттерн **Компоновщик** можно использовать для создания системы управления файлами и папками, где как файлы, так и папки могут рассматриваться как единые объекты. Папка может содержать как файлы, так и другие папки, и с любым из этих объектов можно работать одинаково.

**Задачи:**

1. Реализовать абстрактный класс **Component**, представляющий как листовые элементы (например, файлы), так и контейнеры (например, папки).
2. Создать класс **File**, представляющий конкретный листовой элемент (файл).
3. Создать класс **Directory**, представляющий контейнер, который может содержать файлы и другие директории.
4. Написать клиентский код, демонстрирующий работу паттерна **Компоновщик**.

**Ход работы**

1. Реализация абстрактного компонента

using System;

using System.Collections.Generic;

public abstract class FileSystemComponent

{

protected string \_name;

public FileSystemComponent(string name)

{

\_name = name;

}

public abstract void Display(int depth);

public virtual void Add(FileSystemComponent component)

{

throw new NotImplementedException();

}

public virtual void Remove(FileSystemComponent component)

{

throw new NotImplementedException();

}

public virtual FileSystemComponent GetChild(int index)

{

throw new NotImplementedException();

}

}

1. Реализация класса **File**

public class File : FileSystemComponent

{

public File(string name) : base(name)

{

}

public override void Display(int depth)

{

Console.WriteLine(new string('-', depth) + " File: " + \_name);

}

}

1. Реализация класса **Directory**

public class Directory : FileSystemComponent

{

private List<FileSystemComponent> \_children = new List<FileSystemComponent>();

public Directory(string name) : base(name)

{

}

public override void Add(FileSystemComponent component)

{

\_children.Add(component);

}

public override void Remove(FileSystemComponent component)

{

\_children.Remove(component);

}

public override FileSystemComponent GetChild(int index)

{

return \_children[index];

}

public override void Display(int depth)

{

Console.WriteLine(new string('-', depth) + " Directory: " + \_name);

foreach (var component in \_children)

{

component.Display(depth + 2);

}

}

}

1. Клиентский код

class Program

{

static void Main(string[] args)

{

// Создание файлов и директорий

Directory root = new Directory("Root");

File file1 = new File("File1.txt");

File file2 = new File("File2.txt");

Directory subDir = new Directory("SubDirectory");

File subFile1 = new File("SubFile1.txt");

// Формирование структуры компоновщика

root.Add(file1);

root.Add(file2);

subDir.Add(subFile1);

root.Add(subDir);

// Отображение структуры файловой системы

root.Display(1);

}

}

**Вопросы для закрепления:**

1. В чем заключается суть паттерна **Компоновщик**?
2. В каких случаях удобно использовать паттерн **Компоновщик**?
3. Какие преимущества и недостатки имеет паттерн **Компоновщик**?
4. Какие другие паттерны структурного проектирования могут использоваться вместе с паттерном **Компоновщик**?