**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 09 Структурные паттерны. Декоратор. Адаптер

**Цель:**

Освоить применение структурного паттерна "Декоратор" для добавления новой функциональности объектам в динамическом порядке, не изменяя их основной структуры.

**Задание:**

Вам предстоит разработать систему оформления заказа в кафе. Основная задача — создать базовый класс для напитков и реализовать возможность динамически добавлять к ним различные добавки (декорации), такие как молоко, сахар, шоколад, и др. Каждая добавка должна изменять итоговую стоимость напитка и его описание.

**Структура работы:**

**1. Основные компоненты:**

* **Интерфейс IBeverage:** Базовый интерфейс для всех напитков, включающий метод для получения стоимости напитка (GetCost) и его описания (GetDescription).
* **Класс BaseBeverage:** Класс для базового напитка (например, кофе), реализующий интерфейс IBeverage.

**2. Декораторы:**

* Реализуйте декораторы для добавления дополнительных ингредиентов:
  + **MilkDecorator:** Добавляет молоко к напитку.
  + **SugarDecorator:** Добавляет сахар к напитку.
  + **ChocolateDecorator:** Добавляет шоколад.

**3. Дополнительные требования:**

* Напитки и декораторы должны быть взаимозаменяемы, то есть возможность добавления нескольких декораторов к одному напитку.
* Каждый декоратор должен изменять стоимость и описание напитка.

**Детали реализации:**

1. **Интерфейс для напитков:**

public interface IBeverage

{

double GetCost(); // Получить стоимость напитка

string GetDescription(); // Получить описание напитка

}

1. Базовый напиток:

public class Coffee : IBeverage

{

public double GetCost()

{

return 50.0; // Стоимость кофе

}

public string GetDescription()

{

return "Coffee";

}

}

1. Абстрактный декоратор:

public abstract class BeverageDecorator : IBeverage

{

protected IBeverage \_beverage;

public BeverageDecorator(IBeverage beverage)

{

\_beverage = beverage;

}

public virtual double GetCost()

{

return \_beverage.GetCost(); // Стоимость основного напитка

}

public virtual string GetDescription()

{

return \_beverage.GetDescription(); // Описание основного напитка

}

}

1. Декораторы для добавок:

**Декоратор для молока:**

public class MilkDecorator : BeverageDecorator

{

public MilkDecorator(IBeverage beverage) : base(beverage) { }

public override double GetCost()

{

return base.GetCost() + 10.0; // Стоимость с добавлением молока

}

public override string GetDescription()

{

return base.GetDescription() + ", Milk";

}

}

Декоратор для сахара:

public class SugarDecorator : BeverageDecorator

{

public SugarDecorator(IBeverage beverage) : base(beverage) { }

public override double GetCost()

{

return base.GetCost() + 5.0; // Стоимость с добавлением сахара

}

public override string GetDescription()

{

return base.GetDescription() + ", Sugar";

}

}

Декоратор для шоколада:

public class ChocolateDecorator : BeverageDecorator

{

public ChocolateDecorator(IBeverage beverage) : base(beverage) { }

public override double GetCost()

{

return base.GetCost() + 15.0; // Стоимость с добавлением шоколада

}

public override string GetDescription()

{

return base.GetDescription() + ", Chocolate";

}

}

1. Клиентский код:

class Program

{

static void Main(string[] args)

{

// Создаем базовый напиток — кофе

IBeverage beverage = new Coffee();

Console.WriteLine($"{beverage.GetDescription()} : {beverage.GetCost()}");

// Добавляем молоко

beverage = new MilkDecorator(beverage);

Console.WriteLine($"{beverage.GetDescription()} : {beverage.GetCost()}");

// Добавляем сахар

beverage = new SugarDecorator(beverage);

Console.WriteLine($"{beverage.GetDescription()} : {beverage.GetCost()}");

// Добавляем шоколад

beverage = new ChocolateDecorator(beverage);

Console.WriteLine($"{beverage.GetDescription()} : {beverage.GetCost()}");

}

}

**Задания:**

1. Реализуйте предложенные классы и интерфейсы.
2. Создайте дополнительные декораторы для других ингредиентов, таких как ваниль, корица, и другие.
3. Напишите клиентский код, который позволяет динамически добавлять декораторы к базовому напитку и выводит итоговую стоимость и описание напитка на экран.
4. Протестируйте работу системы с разными комбинациями напитков и добавок.

**Цель:**

Изучить применение структурного паттерна "Адаптер" для преобразования интерфейсов классов, которые не могут быть использованы совместно, но должны взаимодействовать в одном приложении.

**Задание:**

Необходимо разработать систему управления оплатой заказов в интернет-магазине. Интернет-магазин использует несколько платежных систем, каждая из которых имеет собственный API. Ваша задача — создать адаптеры для различных платежных систем, чтобы они могли использоваться через единый интерфейс.

**Описание:**

1. Система уже имеет интерфейс для работы с внутренней платежной системой:
   * **IPaymentProcessor** — интерфейс для обработки платежей, с методами:
     + ProcessPayment(double amount) — обработка платежа.
     + RefundPayment(double amount) — возврат средств.
2. Вам необходимо интегрировать сторонние платежные системы:
   * **ExternalPaymentSystemA** — внешняя платежная система, которая принимает платежи через метод MakePayment(double amount) и возвращает деньги через MakeRefund(double amount).
   * **ExternalPaymentSystemB** — другая платежная система, которая принимает платежи через метод SendPayment(double amount) и делает возврат через ProcessRefund(double amount).
3. Необходимо реализовать адаптеры для интеграции этих систем, чтобы они могли использоваться через интерфейс IPaymentProcessor.

**Структура работы:**

**1. Основные компоненты:**

* **Интерфейс IPaymentProcessor:** Интерфейс для всех платежных систем.
* **Класс InternalPaymentProcessor:** Класс, который реализует интерфейс для внутренней платежной системы.

**2. Сторонние платежные системы:**

* **ExternalPaymentSystemA:** Сторонняя платежная система с методами для оплаты и возврата.
* **ExternalPaymentSystemB:** Другая сторонняя платежная система с собственными методами для оплаты и возврата.

**3. Адаптеры:**

* **PaymentAdapterA:** Адаптер для работы с ExternalPaymentSystemA.
* **PaymentAdapterB:** Адаптер для работы с ExternalPaymentSystemB.

**4. Клиентский код:**

* Реализуйте клиентский код, который выбирает систему платежей и проводит транзакции.

**Детали реализации:**

1. **Интерфейс для работы с платежной системой:**

public interface IPaymentProcessor

{

void ProcessPayment(double amount);

void RefundPayment(double amount);

}

1. Класс для внутренней платежной системы:

public class InternalPaymentProcessor : IPaymentProcessor

{

public void ProcessPayment(double amount)

{

Console.WriteLine($"Processing payment of {amount} via internal system.");

}

public void RefundPayment(double amount)

{

Console.WriteLine($"Refunding payment of {amount} via internal system.");

}

}

1. Сторонние платежные системы:

Сторонняя платежная система A:

public class ExternalPaymentSystemA

{

public void MakePayment(double amount)

{

Console.WriteLine($"Making payment of {amount} via External Payment System A.");

}

public void MakeRefund(double amount)

{

Console.WriteLine($"Making refund of {amount} via External Payment System A.");

}

}

Сторонняя платежная система B:

public class ExternalPaymentSystemB

{

public void SendPayment(double amount)

{

Console.WriteLine($"Sending payment of {amount} via External Payment System B.");

}

public void ProcessRefund(double amount)

{

Console.WriteLine($"Processing refund of {amount} via External Payment System B.");

}

}

1. Адаптеры для сторонних платежных систем:

Адаптер для ExternalPaymentSystemA:

public class PaymentAdapterA : IPaymentProcessor

{

private ExternalPaymentSystemA \_externalSystemA;

public PaymentAdapterA(ExternalPaymentSystemA externalSystemA)

{

\_externalSystemA = externalSystemA;

}

public void ProcessPayment(double amount)

{

\_externalSystemA.MakePayment(amount);

}

public void RefundPayment(double amount)

{

\_externalSystemA.MakeRefund(amount);

}

}

Адаптер для ExternalPaymentSystemB:

public class PaymentAdapterB : IPaymentProcessor

{

private ExternalPaymentSystemB \_externalSystemB;

public PaymentAdapterB(ExternalPaymentSystemB externalSystemB)

{

\_externalSystemB = externalSystemB;

}

public void ProcessPayment(double amount)

{

\_externalSystemB.SendPayment(amount);

}

public void RefundPayment(double amount)

{

\_externalSystemB.ProcessRefund(amount);

}

}

1. Клиентский код:

class Program

{

static void Main(string[] args)

{

// Используем внутреннюю платежную систему

IPaymentProcessor internalProcessor = new InternalPaymentProcessor();

internalProcessor.ProcessPayment(100.0);

internalProcessor.RefundPayment(50.0);

// Используем внешнюю платежную систему A через адаптер

ExternalPaymentSystemA externalSystemA = new ExternalPaymentSystemA();

IPaymentProcessor adapterA = new PaymentAdapterA(externalSystemA);

adapterA.ProcessPayment(200.0);

adapterA.RefundPayment(100.0);

// Используем внешнюю платежную систему B через адаптер

ExternalPaymentSystemB externalSystemB = new ExternalPaymentSystemB();

IPaymentProcessor adapterB = new PaymentAdapterB(externalSystemB);

adapterB.ProcessPayment(300.0);

adapterB.RefundPayment(150.0);

}

}

**Задания:**

1. Реализуйте предложенные интерфейсы и классы для внутренней и внешних платежных систем.
2. Создайте адаптеры для двух внешних систем, чтобы их можно было использовать через единый интерфейс.
3. Дополнительно реализуйте логику выбора платежной системы в зависимости от условий, например, валюты или региона пользователя.
4. Проверьте работу адаптеров в клиентском коде, создав несколько различных сценариев для оплаты и возврата средств через разные платежные системы.