**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 07 Паттерны поведения. Стратегия. Наблюдатель

**Цель:**

Изучить и реализовать паттерн **Стратегия (Strategy)** на языке C#. Вы разработаете систему для расчета стоимости доставки товаров, которая будет использовать разные стратегии в зависимости от выбранного типа доставки (например, стандартная доставка, экспресс-доставка, международная доставка).

**Описание задачи:**

Ваша задача — реализовать программу, которая будет рассчитывать стоимость доставки товара на основе выбранной стратегии. Программа должна поддерживать различные стратегии доставки, такие как:

1. **Стандартная доставка** — расчет стоимости на основе веса и расстояния.
2. **Экспресс-доставка** — расчет стоимости на основе ускоренного времени доставки.
3. **Международная доставка** — расчет с учетом таможенных сборов и расстояния.

**Структура программы:**

1. **Интерфейс IShippingStrategy** — описывает общий метод расчета стоимости доставки.
2. **Реализации стратегий**:
   * **StandardShippingStrategy** — стандартная доставка.
   * **ExpressShippingStrategy** — экспресс-доставка.
   * **InternationalShippingStrategy** — международная доставка.
3. **Класс DeliveryContext** — контекст, который использует выбранную стратегию для расчета стоимости доставки.
4. **Клиентский код** — пользователю предлагается выбрать тип доставки, после чего система рассчитывает стоимость доставки на основе выбранной стратегии.

**Шаги выполнения:**

**1. Создайте интерфейс IShippingStrategy для расчета стоимости доставки:**

*public interface IShippingStrategy*

*{*

*decimal CalculateShippingCost(decimal weight, decimal distance);*

*}*

2. Реализуйте класс для **Стандартной доставки**:

*public class StandardShippingStrategy : IShippingStrategy*

*{*

*public decimal CalculateShippingCost(decimal weight, decimal distance)*

*{*

*return weight \* 0.5m + distance \* 0.1m;*

*}*

*}*

3. Реализуйте класс для **Экспресс-доставки**:

*public class ExpressShippingStrategy : IShippingStrategy*

*{*

*public decimal CalculateShippingCost(decimal weight, decimal distance)*

*{*

*return (weight \* 0.75m + distance \* 0.2m) + 10; // Дополнительная плата за скорость*

*}*

*}*

4. Реализуйте класс для **Международной доставки**:

*public class InternationalShippingStrategy : IShippingStrategy*

*{*

*public decimal CalculateShippingCost(decimal weight, decimal distance)*

*{*

*return weight \* 1.0m + distance \* 0.5m + 15; // Дополнительные сборы за международную доставку*

*}*

*}*

5. Создайте класс **DeliveryContext**, который будет использовать стратегии для расчета стоимости:

*public class DeliveryContext*

*{*

*private IShippingStrategy \_shippingStrategy;*

*// Метод для установки стратегии доставки*

*public void SetShippingStrategy(IShippingStrategy strategy)*

*{*

*\_shippingStrategy = strategy;*

*}*

*// Метод для расчета стоимости доставки*

*public decimal CalculateCost(decimal weight, decimal distance)*

*{*

*if (\_shippingStrategy == null)*

*{*

*throw new InvalidOperationException("Стратегия доставки не установлена.");*

*}*

*return \_shippingStrategy.CalculateShippingCost(weight, distance);*

*}*

*}*

**6. Реализуйте клиентский код:**

*class Program*

*{*

*static void Main(string[] args)*

*{*

*DeliveryContext deliveryContext = new DeliveryContext();*

*Console.WriteLine("Выберите тип доставки: 1 - Стандартная, 2 - Экспресс, 3 - Международная");*

*string choice = Console.ReadLine();*

*switch (choice)*

*{*

*case "1":*

*deliveryContext.SetShippingStrategy(new StandardShippingStrategy());*

*break;*

*case "2":*

*deliveryContext.SetShippingStrategy(new ExpressShippingStrategy());*

*break;*

*case "3":*

*deliveryContext.SetShippingStrategy(new InternationalShippingStrategy());*

*break;*

*default:*

*Console.WriteLine("Неверный выбор.");*

*return;*

*}*

*Console.WriteLine("Введите вес посылки (кг):");*

*decimal weight = Convert.ToDecimal(Console.ReadLine());*

*Console.WriteLine("Введите расстояние доставки (км):");*

*decimal distance = Convert.ToDecimal(Console.ReadLine());*

*decimal cost = deliveryContext.CalculateCost(weight, distance);*

*Console.WriteLine($"Стоимость доставки: {cost:C}");*

*}*

*}*

**Задания:**

1. Реализуйте код по шагам выше.
2. **Тестирование**:
   * Запустите программу и протестируйте все три типа доставки с различными весами и расстояниями.
3. **Расширение функционала**:
   * Добавьте новую стратегию для **ночной доставки**, которая увеличивает стоимость на фиксированную сумму за срочность.
4. **Обработка ошибок**:
   * Добавьте проверку на некорректные входные данные (например, отрицательный вес или расстояние).

**Вопросы для самопроверки:**

1. Какие преимущества дает использование паттерна "Стратегия" в этом проекте?
2. Как можно изменить поведение программы без модификации существующего кода?
3. Почему важно, чтобы каждый метод расчета был независимым и реализован в отдельном классе?

**Цель:**

Изучить и реализовать паттерн **Наблюдатель (Observer)** на языке C#. Ваша задача — разработать программу для мониторинга погодных данных, которая уведомляет различных наблюдателей (например, мобильные приложения, электронные табло) о любых изменениях температуры.

**Описание задачи:**

Необходимо реализовать программу, в которой:

1. **Субъект (WeatherStation)** — станция, отслеживающая изменения температуры.
2. **Наблюдатели (WeatherDisplay)** — объекты, которые подписаны на получение обновлений данных о температуре. При изменении температуры они должны получать уведомление от станции и отображать обновленные данные.

Программа должна поддерживать добавление и удаление наблюдателей, а также отправку уведомлений всем подписанным наблюдателям при изменении температуры.

**Структура программы:**

1. **Интерфейс IObserver** — интерфейс для наблюдателей, которые будут получать обновления от субъекта.
2. **Интерфейс ISubject** — интерфейс для субъекта (WeatherStation), который управляет наблюдателями.
3. **Классы WeatherStation и WeatherDisplay** — реализация субъекта и наблюдателей.
4. **Клиентский код** — демонстрация работы паттерна Наблюдатель.

**Шаги выполнения:**

**1. Создайте интерфейс IObserver для наблюдателей:**

*public interface IObserver*

*{*

*void Update(float temperature);*

*}*

2. Создайте интерфейс ISubject для субъекта:

*public interface ISubject*

*{*

*void RegisterObserver(IObserver observer); // Регистрация наблюдателя*

*void RemoveObserver(IObserver observer); // Удаление наблюдателя*

*void NotifyObservers(); // Уведомление всех наблюдателей*

*}*

3. Реализуйте класс **WeatherStation**, который будет хранить температуру и уведомлять наблюдателей при ее изменении:

*using System;*

*using System.Collections.Generic;*

*public class WeatherStation : ISubject*

*{*

*private List<IObserver> observers;*

*private float temperature;*

*public WeatherStation()*

*{*

*observers = new List<IObserver>();*

*}*

*public void RegisterObserver(IObserver observer)*

*{*

*observers.Add(observer);*

*}*

*public void RemoveObserver(IObserver observer)*

*{*

*observers.Remove(observer);*

*}*

*public void NotifyObservers()*

*{*

*foreach (var observer in observers)*

*{*

*observer.Update(temperature);*

*}*

*}*

*public void SetTemperature(float newTemperature)*

*{*

*Console.WriteLine($"Изменение температуры: {newTemperature}°C");*

*temperature = newTemperature;*

*NotifyObservers();*

*}*

*}*

4. Реализуйте класс **WeatherDisplay**, который будет играть роль наблюдателя и выводить обновленные данные о температуре:

*public class WeatherDisplay : IObserver*

*{*

*private string \_name;*

*public WeatherDisplay(string name)*

*{*

*\_name = name;*

*}*

*public void Update(float temperature)*

*{*

*Console.WriteLine($"{\_name} показывает новую температуру: {temperature}°C");*

*}*

*}*

5. Напишите клиентский код для демонстрации работы программы:

*class Program*

*{*

*static void Main(string[] args)*

*{*

*WeatherStation weatherStation = new WeatherStation();*

*// Создаем несколько наблюдателей*

*WeatherDisplay mobileApp = new WeatherDisplay("Мобильное приложение");*

*WeatherDisplay digitalBillboard = new WeatherDisplay("Электронное табло");*

*// Регистрируем наблюдателей в системе*

*weatherStation.RegisterObserver(mobileApp);*

*weatherStation.RegisterObserver(digitalBillboard);*

*// Изменяем температуру на станции, что приводит к уведомлению наблюдателей*

*weatherStation.SetTemperature(25.0f);*

*weatherStation.SetTemperature(30.0f);*

*// Убираем один из дисплеев и снова меняем температуру*

*weatherStation.RemoveObserver(digitalBillboard);*

*weatherStation.SetTemperature(28.0f);*

*}*

*}*

**Задания:**

1. Реализуйте код по шагам выше.
2. **Тестирование**:
   * Создайте несколько наблюдателей и протестируйте систему с изменениями температуры.
   * Проверьте корректность работы при добавлении и удалении наблюдателей.
3. **Расширение функционала**:
   * Добавьте новый тип наблюдателя, например, систему оповещения через email или звуковое уведомление.
4. **Обработка ошибок**:
   * Добавьте проверку на попытку удалить несуществующего наблюдателя или передать некорректные данные о температуре.

**Вопросы для самопроверки:**

1. Какие преимущества дает использование паттерна "Наблюдатель" в данной системе?
2. Как можно изменить список наблюдателей, не изменяя код субъекта?
3. Какие изменения можно внести в реализацию, чтобы сделать систему асинхронной?