**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 08 Паттерны поведения. Команда. Шаблонный метод. Посредник

**Цель:**

Изучить и реализовать паттерн **Команда (Command)** на языке C#. Ваша задача — создать систему управления умным домом, где различные устройства могут управляться через команды, отправляемые с пульта дистанционного управления. Каждый тип команды должен быть инкапсулирован в отдельный класс.

**Описание задачи:**

Необходимо реализовать систему управления умным домом, которая позволяет контролировать различные устройства (например, свет, телевизор, кондиционер) через команды. Каждое устройство должно поддерживать выполнение команд включения, выключения и, возможно, других действий. Система должна быть гибкой, чтобы легко добавлять новые команды и устройства.

**Структура программы:**

1. **Интерфейс ICommand** — описывает контракт для всех команд.
2. **Классы конкретных команд** — реализуют различные действия для устройств (включение, выключение и т.д.).
3. **Класс устройства** — представляет конкретные устройства (например, свет, телевизор и т.д.).
4. **Класс пульта дистанционного управления** — хранит команды и позволяет выполнять их.
5. **Клиентский код** — демонстрирует работу системы управления умным домом.

**Шаги выполнения:**

1. Создайте интерфейс ICommand, который будет содержать метод для выполнения команды

*public interface ICommand*

*{*

*void Execute();*

*void Undo(); // Метод для отмены команды*

*}*

2. Реализуйте классы команд для конкретных действий. Например, команды для включения и выключения света.

**Команда для включения света:**

*public class LightOnCommand : ICommand*

*{*

*private Light \_light;*

*public LightOnCommand(Light light)*

*{*

*\_light = light;*

*}*

*public void Execute()*

*{*

*\_light.On();*

*}*

*public void Undo()*

*{*

*\_light.Off();*

*}*

*}*

**Команда для выключения света:**

*public class LightOffCommand : ICommand*

*{*

*private Light \_light;*

*public LightOffCommand(Light light)*

*{*

*\_light = light;*

*}*

*public void Execute()*

*{*

*\_light.Off();*

*}*

*public void Undo()*

*{*

*\_light.On();*

*}*

*}*

*3. Создайте класс устройства, например,* ***Light****:*

*public class Light*

*{*

*public void On()*

*{*

*Console.WriteLine("Свет включен.");*

*}*

*public void Off()*

*{*

*Console.WriteLine("Свет выключен.");*

*}*

*}*

4. Реализуйте другие устройства, такие как **Television** и **AirConditioner**, аналогично классу **Light**. Пример устройства **Television**:

*public class Television*

*{*

*public void On()*

*{*

*Console.WriteLine("Телевизор включен.");*

*}*

*public void Off()*

*{*

*Console.WriteLine("Телевизор выключен.");*

*}*

*}*

5. Реализуйте класс пульта дистанционного управления, который будет принимать команды и их выполнять:

*public class RemoteControl*

*{*

*private ICommand \_onCommand;*

*private ICommand \_offCommand;*

*public void SetCommands(ICommand onCommand, ICommand offCommand)*

*{*

*\_onCommand = onCommand;*

*\_offCommand = offCommand;*

*}*

*public void PressOnButton()*

*{*

*\_onCommand.Execute();*

*}*

*public void PressOffButton()*

*{*

*\_offCommand.Execute();*

*}*

*public void PressUndoButton()*

*{*

*\_onCommand.Undo();*

*}*

*}*

6. Напишите клиентский код для демонстрации работы пульта с командами:

*class Program*

*{*

*static void Main(string[] args)*

*{*

*// Создаем устройства*

*Light livingRoomLight = new Light();*

*Television tv = new Television();*

*// Создаем команды для управления светом*

*ICommand lightOn = new LightOnCommand(livingRoomLight);*

*ICommand lightOff = new LightOffCommand(livingRoomLight);*

*// Создаем команды для управления телевизором*

*ICommand tvOn = new TelevisionOnCommand(tv);*

*ICommand tvOff = new TelevisionOffCommand(tv);*

*// Создаем пульт и привязываем команды к кнопкам*

*RemoteControl remote = new RemoteControl();*

*// Управляем светом*

*remote.SetCommands(lightOn, lightOff);*

*Console.WriteLine("Управление светом:");*

*remote.PressOnButton();*

*remote.PressOffButton();*

*remote.PressUndoButton();*

*// Управляем телевизором*

*remote.SetCommands(tvOn, tvOff);*

*Console.WriteLine("\nУправление телевизором:");*

*remote.PressOnButton();*

*remote.PressOffButton();*

*}*

*}*

**Задания:**

1. **Реализуйте код по шагам выше.**
2. **Тестирование**:
   * Проверьте работу команд для включения и выключения света.
   * Проверьте работу команд для включения и выключения телевизора.
   * Проверьте работу кнопки отмены действия (Undo).
3. **Расширение функционала**:
   * Добавьте новые устройства (например, кондиционер, аудиосистему) и реализуйте команды для их управления.
   * Реализуйте макрокоманды (MacroCommand), которые будут выполнять несколько команд подряд.
   * Добавьте возможность переключения на различные режимы работы (например, режим экономии энергии для кондиционера).
4. **Обработка ошибок**:
   * Добавьте обработку ситуации, когда на кнопке пульта не назначена команда (например, вывод сообщения об ошибке).
   * Добавьте логирование всех выполненных команд для отслеживания действий пользователя.

**Вопросы для самопроверки:**

1. В чем преимущество использования паттерна "Команда" для управления устройствами?
2. Как вы можете расширить функционал пульта, добавив новые кнопки для других устройств, не изменяя существующий код?
3. В чем разница между командами и макрокомандами, и как они помогают улучшить гибкость системы?

**Цель:**

Изучить и реализовать паттерн **Шаблонный метод (Template Method)** на языке C#. Ваша задача — создать систему для приготовления различных напитков (например, чай и кофе), где общий алгоритм приготовления будет описан в базовом классе, а конкретные шаги будут определяться в наследниках.

**Описание задачи:**

Необходимо реализовать систему для приготовления напитков, таких как чай и кофе. Общий процесс приготовления напитков включает несколько шагов (например, кипячение воды, добавление ингредиентов и т.д.). Паттерн "Шаблонный метод" используется для того, чтобы общий алгоритм был определён в базовом классе, а конкретные шаги (например, добавление чая или кофе) реализовались в подклассах.

**Структура программы:**

1. **Базовый класс** — определяет общий шаблонный метод приготовления напитка, включающий конкретные и абстрактные шаги.
2. **Классы-наследники** — реализуют конкретные шаги для приготовления различных напитков (например, чая и кофе).
3. **Клиентский код** — демонстрирует процесс приготовления различных напитков с использованием шаблонного метода.

**Шаги выполнения:**

1. Создайте базовый класс Beverage, который будет содержать шаблонный метод и общие шаги приготовления напитков:

*public abstract class Beverage*

*{*

*// Шаблонный метод*

*public void PrepareRecipe()*

*{*

*BoilWater();*

*Brew();*

*PourInCup();*

*AddCondiments();*

*}*

*// Общий шаг для всех напитков*

*private void BoilWater()*

*{*

*Console.WriteLine("Кипячение воды...");*

*}*

*// Общий шаг для всех напитков*

*private void PourInCup()*

*{*

*Console.WriteLine("Наливание в чашку...");*

*}*

*// Абстрактные методы для шагов, которые будут реализованы в подклассах*

*protected abstract void Brew();*

*protected abstract void AddCondiments();*

*}*

2. Создайте класс для приготовления чая, наследующий от Beverage, и реализуйте конкретные шаги:

*public class Tea : Beverage*

*{*

*protected override void Brew()*

*{*

*Console.WriteLine("Заваривание чая...");*

*}*

*protected override void AddCondiments()*

*{*

*Console.WriteLine("Добавление лимона...");*

*}*

*}*

3. Создайте класс для приготовления кофе, также наследующий от Beverage, и реализуйте конкретные шаги:

*public class Coffee : Beverage*

*{*

*protected override void Brew()*

*{*

*Console.WriteLine("Заваривание кофе...");*

*}*

*protected override void AddCondiments()*

*{*

*Console.WriteLine("Добавление сахара и молока...");*

*}*

*}*

4. Напишите клиентский код, который будет демонстрировать процесс приготовления различных напитков:

*class Program*

*{*

*static void Main(string[] args)*

*{*

*// Приготовление чая*

*Beverage tea = new Tea();*

*Console.WriteLine("Приготовление чая:");*

*tea.PrepareRecipe();*

*Console.WriteLine();*

*// Приготовление кофе*

*Beverage coffee = new Coffee();*

*Console.WriteLine("Приготовление кофе:");*

*coffee.PrepareRecipe();*

*}*

*}*

**Задания:**

1. **Реализуйте код по шагам выше.**
2. **Тестирование**:
   * Проверьте процесс приготовления чая и кофе с выводом на экран.
   * Убедитесь, что каждый напиток выполняет свои специфические шаги (например, добавление лимона в чай и сахара в кофе).
3. **Расширение функционала**:
   * Добавьте новый напиток, например, горячий шоколад, и реализуйте соответствующие шаги.
   * Добавьте в базовый класс возможность задать шаги, которые пользователь может пропускать (например, не добавлять сахар в кофе).
   * Реализуйте возможность кастомизации шагов (например, добавление разных типов молока в кофе).
4. **Обработка ошибок**:
   * Реализуйте обработку ситуации, когда шаг не может быть выполнен (например, отсутствие ингредиентов).

**Вопросы для самопроверки:**

1. Как работает шаблонный метод, и в чем его преимущество перед обычным наследованием?
2. Как можно использовать хуки для настройки поведения подклассов?
3. Какие еще ситуации могут потребовать использования паттерна "Шаблонный метод" в реальных проектах?

**Цель:**

Изучить и реализовать паттерн **Посредник (Mediator)** на языке C#. Ваша задача — создать систему управления диалогом между несколькими участниками, где взаимодействие между участниками организовано через объект-посредник.

**Описание задачи:**

Необходимо реализовать чат-систему, где несколько участников могут отправлять сообщения друг другу, но вместо прямого взаимодействия между участниками будет использоваться посредник. Посредник отвечает за передачу сообщений между участниками и управляет всей коммуникацией.

**Структура программы:**

1. **Интерфейс Посредник (IMediator)** — определяет контракт для посредника.
2. **Конкретный посредник (ConcreteMediator)** — реализует интерфейс посредника и управляет взаимодействием участников.
3. **Интерфейс участника (IColleague)** — определяет интерфейс для участников, которые будут взаимодействовать через посредника.
4. **Конкретные участники (ConcreteColleague)** — участники чата, которые обмениваются сообщениями через посредника.
5. **Клиентский код** — демонстрирует работу системы.

**Шаги выполнения:**

1. Создайте интерфейс IMediator, который будет описывать метод для отправки сообщений:

*public interface IMediator*

*{*

*void SendMessage(string message, Colleague colleague);*

*}*

2. Создайте интерфейс или базовый класс для участников чата (коллег):

*public abstract class Colleague*

*{*

*protected IMediator \_mediator;*

*public Colleague(IMediator mediator)*

*{*

*\_mediator = mediator;*

*}*

*public abstract void ReceiveMessage(string message);*

*}*

3. Реализуйте конкретный класс посредника, который будет управлять обменом сообщений между участниками:

*public class ChatMediator : IMediator*

*{*

*private List<Colleague> \_colleagues;*

*public ChatMediator()*

*{*

*\_colleagues = new List<Colleague>();*

*}*

*public void RegisterColleague(Colleague colleague)*

*{*

*\_colleagues.Add(colleague);*

*}*

*public void SendMessage(string message, Colleague sender)*

*{*

*foreach (var colleague in \_colleagues)*

*{*

*if (colleague != sender)*

*{*

*colleague.ReceiveMessage(message);*

*}*

*}*

*}*

*}*

4. Реализуйте конкретных участников чата, наследующихся от класса Colleague:

*public class User : Colleague*

*{*

*private string \_name;*

*public User(IMediator mediator, string name) : base(mediator)*

*{*

*\_name = name;*

*}*

*public void Send(string message)*

*{*

*Console.WriteLine($"{\_name} отправляет сообщение: {message}");*

*\_mediator.SendMessage(message, this);*

*}*

*public override void ReceiveMessage(string message)*

*{*

*Console.WriteLine($"{\_name} получил сообщение: {message}");*

*}*

*}*

5. Напишите клиентский код для демонстрации работы чата:

*class Program*

*{*

*static void Main(string[] args)*

*{*

*// Создаем посредника*

*ChatMediator chatMediator = new ChatMediator();*

*// Создаем участников*

*User user1 = new User(chatMediator, "Алиса");*

*User user2 = new User(chatMediator, "Боб");*

*User user3 = new User(chatMediator, "Чарли");*

*// Регистрируем участников в чате*

*chatMediator.RegisterColleague(user1);*

*chatMediator.RegisterColleague(user2);*

*chatMediator.RegisterColleague(user3);*

*// Участники обмениваются сообщениями*

*user1.Send("Привет всем!");*

*user2.Send("Привет, Алиса!");*

*user3.Send("Всем привет!");*

*}*

*}*

**Задания:**

1. **Реализуйте код по шагам выше.**
2. **Тестирование**:
   * Проверьте работу чата с тремя участниками. Убедитесь, что сообщение отправляется всем, кроме отправителя.
   * Добавьте возможность отправки приватных сообщений одному из участников.
3. **Расширение функционала**:
   * Реализуйте возможность создания групп чатов, где посредник управляет несколькими чатами одновременно.
   * Добавьте логирование сообщений, чтобы посредник сохранял историю переписки.
4. **Обработка ошибок**:
   * Добавьте обработку ошибок, например, если участник пытается отправить сообщение, не зарегистрировавшись в чате.
   * Реализуйте механизм отключения участника от чата.

**Вопросы для самопроверки:**

1. В чем преимущество использования паттерна "Посредник" в системе с большим количеством участников?
2. Как изменить посредника, чтобы он поддерживал отправку сообщений определённой группе участников?
3. Какие задачи в реальной жизни могут быть решены с использованием паттерна "Посредник"?