Лабораторная работа №2.7

Цель: ознакомится с понятием «шаблон проектирования», изучить шаблоны поведения.

# Паттерны поведения

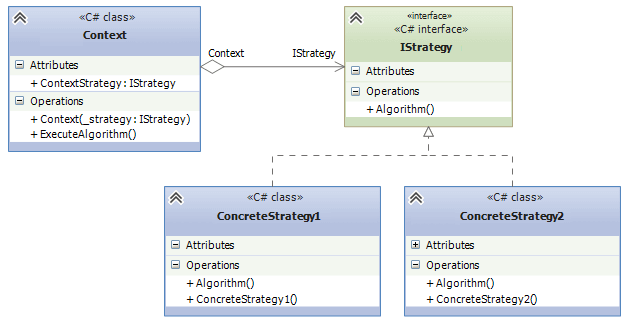
## Стратегия (Strategy)

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

### Когда использовать стратегию?

* Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
* Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
* Когда необходимо менять поведение объектов на стадии выполнения программы
* Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Формально паттерн Стратегия можно выразить следующей схемой UML:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

public interface IStrategy

{

    void Algorithm();

}

public class ConcreteStrategy1 : IStrategy

{

    public void Algorithm()

    {}

}

public class ConcreteStrategy2 : IStrategy

{

    public void Algorithm()

    {}

}

public class Context

{

    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy \_strategy)

    {

        ContextStrategy = \_strategy;

    }

    public void ExecuteAlgorithm()

    {

        ContextStrategy.Algorithm();

    }

}

## Наблюдатель (Observer)

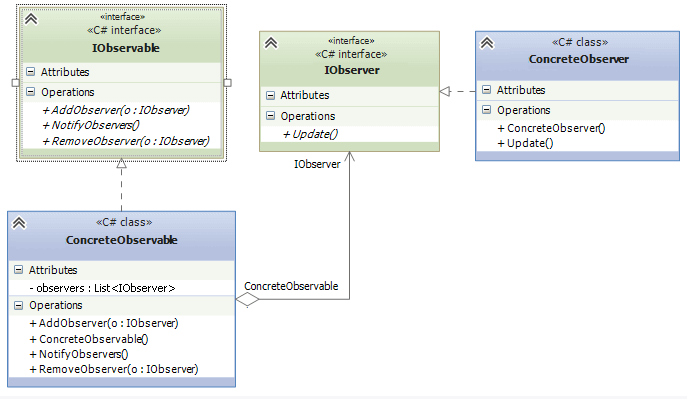
Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

### Когда использовать паттерн Наблюдатель?

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

С помощью диаграмм UML данный шаблон можно выразить следующим образом:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

interface IObservable

{

    void AddObserver(IObserver o);

    void RemoveObserver(IObserver o);

    void NotifyObservers();

}

class ConcreteObservable : IObservable

{

    private List<IObserver> observers;

    public void AddObserver(IObserver o)

    {

        observers.Add(o);

    }

    public void RemoveObserver(IObserver o)

    {

        observers.Remove(o);

    }

    public void NotifyObservers()

    {

        foreach (IObserver observer in observers)

            observer.Update();

    }

}

interface IObserver

{

    void Update();

}

class ConcreteObserver :IObserver

{

    public void Update()

    {

    }

}

### Участники

* **IObservable**: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя),RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)
* **ConcreteObservable**: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.
* **IObserver**: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет методUpdate(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
* **ConcreteObserver**: конкретная реализация интерфейса IObserver.

При этом наблюдаемому объекту не надо ничего знать о наблюдателе кроме того, что тот реализует метод Update(). С помощью отношения агрегации реализуется слабосвязанность обоих компонентов. Изменения в наблюдаемом объекте не виляют на наблюдателя и наоборот.

В определенный момент наблюдатель может прекратить наблюдение. И после этого оба объекта - наблюдатель и наблюдаемый могут продолжать существовать в системе независимо друг от друга.

Рассмотрим реальный пример применения шаблона. Допустим, у нас есть биржа, где проходят торги, и есть брокеры и банки, которые следят за поступающей информацией и в зависимости от поступившей информации производят определенные действия:

class Program

{

    static void Main(string[] args)

    {

        Stock stock = new Stock();

        Bank bank = new Bank("ЮнитБанк", stock);

        Broker broker = new Broker("Иван Иваныч", stock);

        // имитация торгов

        stock.Market();

        // брокер прекращает наблюдать за торгами

        broker.StopTrade();

        // имитация торгов

        stock.Market();

        Console.Read();

    }

}

interface IObserver

{

    void Update(Object ob);

}

interface IObservable

{

    void RegisterObserver(IObserver o);

    void RemoveObserver(IObserver o);

    void NotifyObservers();

}

class Stock : IObservable

{

    StockInfo sInfo; // информация о торгах

    List<IObserver> observers;

    public Stock()

    {

        observers = new List<IObserver>();

        sInfo= new StockInfo();

    }

    public void RegisterObserver(IObserver o)

    {

        observers.Add(o);

    }

    public void RemoveObserver(IObserver o)

    {

        observers.Remove(o);

    }

    public void NotifyObservers()

    {

        foreach(IObserver o in observers)

        {

            o.Update(sInfo);

        }

    }

    public void Market()

    {

        Random rnd = new Random();

        sInfo.USD = rnd.Next(20, 40);

        sInfo.Euro = rnd.Next(30, 50);

        NotifyObservers();

    }

}

class StockInfo

{

    public int USD { get; set; }

    public int Euro { get; set; }

}

class Broker : IObserver

{

    public string Name { get; set; }

    IObservable stock;

    public Broker(string name, IObservable obs)

    {

        this.Name = name;

        stock = obs;

        stock.RegisterObserver(this);

    }

    public void Update(object ob)

    {

        StockInfo sInfo = (StockInfo)ob;

        if(sInfo.USD>30)

            Console.WriteLine("Брокер {0} продает доллары;  Курс доллара: {1}", this.Name, sInfo.USD);

        else

            Console.WriteLine("Брокер {0} покупает доллары;  Курс доллара: {1}", this.Name, sInfo.USD);

    }

    public void StopTrade()

    {

        stock.RemoveObserver(this);

        stock=null;

    }

}

class Bank : IObserver

{

    public string Name { get; set; }

    IObservable stock;

    public Bank(string name, IObservable obs)

    {

        this.Name = name;

        stock = obs;

        stock.RegisterObserver(this);

    }

    public void Update(object ob)

    {

        StockInfo sInfo = (StockInfo)ob;

        if (sInfo.Euro > 40)

            Console.WriteLine("Банк {0} продает евро;  Курс евро: {1}", this.Name, sInfo.Euro);

        else

            Console.WriteLine("Банк {0} покупает евро;  Курс евро: {1}", this.Name, sInfo.Euro);

    }

}

Итак, здесь наблюдаемый объект представлен интерфейсом IObservable, а наблюдатель - интерфейсом IObserver. Реализацией интерфейса IObservable является класс Stock, который символизирует валютную биржу. В этом классе определен метод Market(), который имитирует торги и инкапсулирует всю информацию о валютных курсах в объекте StockInfo. После проведения торгов производится уведомление всех наблюдателей.

Реализациями интерфейса IObserver являются классы Broker, представляющий брокера, и Bank, представляющий банк. При этом метод Update() интерфейса IObserver принимает в качестве параметра некоторый объект. Реализация этого метода подразумевает получение через данный параметр объекта StockInfo с текущей информацией о торгах и произведение некоторых действий: покупка или продажа долларов и евро. Дело в том, что часто необходимо информировать наблюдателя об изменении состояния наблюдаемого объекта. В данном случае состояние заключено в объекте StockInfo. И одним из вариантом информирования наблюдателя о состоянии является push-модель, при которой наблюдаемый объект передает (иначе говоря толкает - push) данные о своем состоянии, то есть передаем в виде параметра метода Update().

Альтернативой push-модели является pull-модель, когда наблюдатель вытягивает (pull) из наблюдаемого объекта данные о состоянии с помощью дополнительных методов.

Также в классе брокера определен дополнительный метод StopTrade(), с помощью которого брокер может отписаться от уведомлений биржи и перестать быть наблюдателем.

## Команда (Command)

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

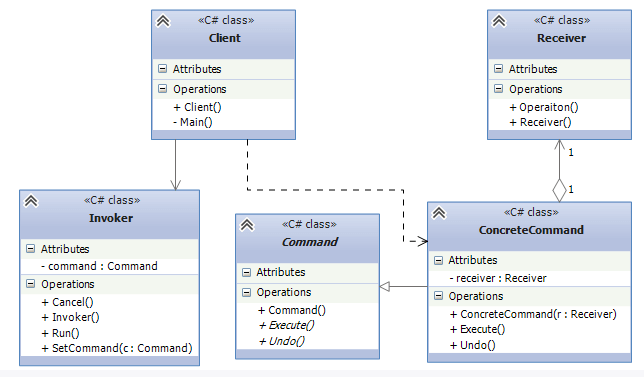
### Когда использовать команды?

Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.

Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.

Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.

Схематично в UML паттерн Команда представляется следующим образом:



Формальное определение на языке C# может выглядеть следующим образом:

abstract class Command

{

    public abstract void Execute();

    public abstract void Undo();

}

// конкретная команда

class ConcreteCommand : Command

{

    Receiver receiver;

    public ConcreteCommand(Receiver r)

    {

        receiver = r;

    }

    public override void Execute()

    {

        receiver.Operaiton();

    }

    public override void Undo()

    {}

}

// получатель команды

class Receiver

{

    public void Operaiton()

    { }

}

// инициатор команды

class Invoker

{

    Command command;

    public void SetCommand(Command c)

    {

        command = c;

    }

    public void Run()

    {

        command.Execute();

    }

    public void Cancel()

    {

        command.Undo();

    }

}

class Client

{

    void Main()

    {

        Invoker invoker = new Invoker();

        Receiver receiver = new Receiver();

        ConcreteCommand command=new ConcreteCommand(receiver);

        invoker.SetCommand(command);

        invoker.Run();

    }

}

### Участники

* **Command**: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды
* **ConcreteCommand**: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver
* **Receiver**: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
* **Invoker**: инициатор команды - вызывает команду для выполнения определенного запроса
* **Client**: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

Таким образом, инициатор, отправляющий запрос, ничего не знает о получателе, который и будет выполнять команду. Кроме того, если нам потребуется применить какие-то новые команды, мы можем просто унаследовать классы от абстрактного класса Command и реализовать его методы Execute и Undo.

В программах на C# команды находят довольно широкое применение. Так, в технологии WPF и других технологиях, которые используют XAML и подход MVVM, на командах во многом базируется взаимодействие с пользователем. В некоторых архитектурах, например, в архитектуре CQRS, команды являются одним из ключевых компонентов.

Нередко в роли инициатора команд выступают панели управления или кнопки интерфейса. Самая простая ситуация - надо программно организовать включение и выключение прибора, например, телевизора. Решение данной задачи с помощью команд могло бы выглядеть так:

class Program

{

    static void Main(string[] args)

    {

        Pult pult = new Pult();

        TV tv = new TV();

        pult.SetCommand(new TVOnCommand(tv));

        pult.PressButton();

        pult.PressUndo();

        Console.Read();

    }

}

interface ICommand

{

    void Execute();

    void Undo();

}

// Receiver - Получатель

class TV

{

    public void On()

    {

        Console.WriteLine("Телевизор включен!");

    }

    public void Off()

    {

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

    }

}

class TVOnCommand : ICommand

{

    TV tv;

    public TVOnCommand(TV tvSet)

    {

        tv = tvSet;

    }

    public void Execute()

    {

        tv.On();

    }

    public void Undo()

    {

        tv.Off();

    }

}

// Invoker - инициатор

class Pult

{

    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)

    {

        command = com;

    }

    public void PressButton()

    {

        command.Execute();

    }

    public void PressUndo()

    {

        command.Undo();

    }

}

Итак, в этой программе есть интерфейс команды - ICommand, есть ее реализация в виде класса TVOnCommand, есть инициатор команды - класс Pult, некий прибор - пульт, управляющий телевизором. И есть получатель команды - класс TV, представляющий телевизор. В качестве клиента используется класс Program.

При этом пульт ничего не знает об объекте TV. Он только знает, как отправить команду. В итоге мы получаем гибкую систему, в которой мы легко можем заменять одни команды на другие, создавать последовательности команд. Например, в нашей программе кроме телевизора появилась микроволновка, которой тоже неплохо было бы управлять с помощью одного интерфейса. Для этого достаточно добавить соответствующие классы и установить команду:

class Program

{

    static void Main(string[] args)

    {

        Pult pult = new Pult();

        TV tv = new TV();

        pult.SetCommand(new TVOnCommand(tv));

        pult.PressButton();

        pult.PressUndo();

        Microwave microwave = new Microwave

        // 5000 - время нагрева пищи

        pult.SetCommand(new MicrowaveCommand(microwave, 5000));

        pult.PressButton();

        Console.Read();

    }

}

//.....ранее описанные классы

class Microwave

{

    public void StartCooking(int time)

    {

        Console.WriteLine("Подогреваем еду");

        // имитация работы с помощью асинхронного метода Task.Delay

        Task.Delay(time).GetAwaiter().GetResult();

    }

    public void StopCooking()

    {

        Console.WriteLine("Еда подогрета!");

    }

}

class MicrowaveCommand : ICommand

{

    Microwave microwave;

    int time;

    public MicrowaveCommand(Microwave m, int t)

    {

        microwave = m;

        time = t;

    }

    public void Execute()

    {

        microwave.StartCooking(time);

        microwave.StopCooking();

    }

    public void Undo()

    {

        microwave.StopCooking();

    }

}

Теперь еще одним получателем запроса является класс Microwave, функциональностью которого можно управлять через команды MicrowaveCommand.

Правда, в вышеописанной системе есть один изъян: если мы попытаемся выполнить команду до ее назначения, то программа выдаст исключение, так как команда будет не установлена. Эту проблему мы могли бы решить, проверяя команду на значение null в классе инициатора:

class Pult

{

    ICommand command;

    public Pult() { }

    public void SetCommand(ICommand com)

    {

        command = com;

    }

    public void PressButton()

    {

        if(command!=null)

            command.Execute();

    }

    public void PressUndo()

    {

        if(command!=null)

            command.Undo();

    }

}

Либо можно определить класс пустой команды, которая будет устанавливаться по умолчанию:

class NoCommand : ICommand

{

    public void Execute()

    {

    }

    public void Undo()

    {

    }

}

class Pult

{

    ICommand command;

    public Pult()

    {

        command = new NoCommand();

    }

    public void SetCommand(ICommand com)

    {

        command = com;

    }

    public void PressButton()

    {

        command.Execute();

    }

    public void PressUndo()

    {

        command.Undo();

    }

}

При этом инициатор необязательно указывает на одну команду. Он может управлять множеством команд. Например, на пульте от телевизора есть как кнопка для включения, так и кнопки для регулировки звука:

class Program

{

    static void Main(string[] args)

    {

        TV tv = new TV();

        Volume volume = new Volume();

        MultiPult mPult = new MultiPult();

        mPult.SetCommand(0, new TVOnCommand(tv));

        mPult.SetCommand(1, new VolumeCommand(volume));

        // включаем телевизор

        mPult.PressButton(0);

        // увеличиваем громкость

        mPult.PressButton(1);

        mPult.PressButton(1);

        mPult.PressButton(1);

        // действия отмены

        mPult.PressUndoButton();

        mPult.PressUndoButton();

        mPult.PressUndoButton();

        mPult.PressUndoButton();

        Console.Read();

    }

}

interface Command

{

    void Execute();

    void Undo();

}

class TV

{

    public void On()

    {

        Console.WriteLine("Телевизор включен!");

    }

    public void Off()

    {

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

    }

}

class TVOnCommand : ICommand

{

    TV tv;

    public TVOnCommand(TV tvSet)

    {

        tv = tvSet;

    }

    public void Execute()

    {

        tv.On();

    }

    public void Undo()

    {

        tv.Off();

    }

}

class Volume

{

    public const int OFF = 0;

    public const int HIGH = 20;

    private int level;

    public Volume()

    {

        level = OFF;

    }

    public void RaiseLevel()

    {

        if (level < HIGH)

            level++;

        Console.WriteLine("Уровень звука {0}", level);

    }

    public void DropLevel()

    {

        if (level > OFF)

            level--;

        Console.WriteLine("Уровень звука {0}", level);

    }

}

class VolumeCommand : ICommand

{

    Volume volume;

    public VolumeCommand(Volume v)

    {

        volume = v;

    }

    public void Execute()

    {

        volume.RaiseLevel();

    }

    public void Undo()

    {

        volume.DropLevel();

    }

}

class NoCommand : ICommand

{

    public void Execute()

    {

    }

    public void Undo()

    {

    }

}

class MultiPult

{

    ICommand[] buttons;

    Stack<ICommand> commandsHistory;

    public MultiPult()

    {

        buttons = new ICommand[2];

        for (int i = 0; i < buttons.Length; i++)

        {

            buttons[i] = new NoCommand();

        }

        commandsHistory = new Stack<ICommand>();

    }

    public void SetCommand(int number, ICommand com)

    {

        buttons[number] = com;

    }

    public void PressButton(int number)

    {

        buttons[number].Execute();

        // добавляем выполненную команду в историю команд

        commandsHistory.Push(buttons[number]);

    }

    public void PressUndoButton()

    {

        if(commandsHistory.Count>0)

        {

            ICommand undoCommand = commandsHistory.Pop();

            undoCommand.Undo();

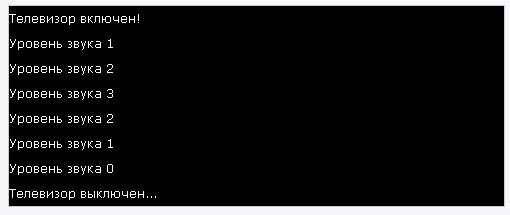
        }

    }

}

Здесь два получателя команд - классы TV и Volume. Volume управляет уровнем звука и сохраняет текущий уровень в переменной level. Также есть две команды TVOnCommand и VolumeCommand.

Инициатор - MultiPult имеет две кнопки в виде массива buttons: первая предназначена для TV, а вторая - для увеличения уровня звука. Чтобы сохранить историю команд используется стек. При отправке команды в стек добавляется новый элемент, а при ее отмене, наоборот, происходит удаление из стека. В данном случае стек выполняет роль примитивного лога команд.



### Макрокоманды

Для управления набором команд используются макрокоманды. Макрокоманда должна реализовать тот же интерфейс, что и другие команды, при этом макрокоманда инкапсулирует в одной из своих переменных весь набор используемых команд. Рассмотрим на примере.

Для создания и развития программного продукта необходимо несколько исполнителей, выполняющих различные функции: программист пишет код, тестировщик выполняет тестирование продукта, а маркетолог пишет рекламные материалы и проводит кампании по рекламированию продукта. Управляет всем процессом менеджер. Программа на C#, описывающая создание программного продукта с помощью паттерна команд, могла бы выглядеть следующим образом:

class Program

{

    static void Main(string[] args)

    {

        Programmer programmer = new Programmer();

        Tester tester = new Tester();

        Marketolog marketolog = new Marketolog();

        List<ICommand> commands = new List<ICommand>

        {

            new CodeCommand(programmer),

            new TestCommand(tester),

            new AdvertizeCommand(marketolog)

        };

        Manager manager = new Manager();

        manager.SetCommand(new MacroCommand(commands));

        manager.StartProject();

        manager.StopProject();

        Console.Read();

    }

}

interface ICommand

{

    void Execute();

    void Undo();

}

// Класс макрокоманды

class MacroCommand : ICommand

{

    List<ICommand> commands;

    public MacroCommand(List<ICommand> coms)

    {

        commands = coms;

    }

    public void Execute()

    {

        foreach(ICommand c in commands)

            c.Execute();

    }

    public void Undo()

    {

        foreach (ICommand c in commands)

            c.Undo();

    }

}

class Programmer

{

    public void StartCoding()

    {

        Console.WriteLine("Программист начинает писать код");

    }

    public void StopCoding()

    {

        Console.WriteLine("Программист завершает писать код");

    }

}

class Tester

{

    public void StartTest()

    {

        Console.WriteLine("Тестировщик начинает тестирование");

    }

    public void StopTest()

    {

        Console.WriteLine("Тестировщик завершает тестирование");

    }

}

class Marketolog

{

    public void StartAdvertize()

    {

        Console.WriteLine("Маркетолог начинает рекламировать продукт");

    }

    public void StopAdvertize()

    {

        Console.WriteLine("Маркетолог прекращает рекламную кампанию");

    }

}

class CodeCommand : ICommand

{

    Programmer programmer;

    public CodeCommand(Programmer p)

    {

        programmer = p;

    }

    public void Execute()

    {

        programmer.StartCoding();

    }

    public void Undo()

    {

        programmer.StopCoding();

    }

}

class TestCommand : ICommand

{

    Tester tester;

    public TestCommand(Tester t)

    {

        tester = t;

    }

    public void Execute()

    {

        tester.StartTest();

    }

    public void Undo()

    {

        tester.StopTest();

    }

}

class AdvertizeCommand : ICommand

{

    Marketolog marketolog;

    public AdvertizeCommand(Marketolog m)

    {

        marketolog = m;

    }

    public void Execute()

    {

        marketolog.StartAdvertize();

    }

    public void Undo()

    {

        marketolog.StopAdvertize();

    }

}

class Manager

{

    ICommand command;

    public void SetCommand(ICommand com)

    {

        command = com;

    }

    public void StartProject()

    {

        if (command != null)

            command.Execute();

    }

    public void StopProject()

    {

        if (command != null)

            command.Undo();

    }

}

В роли инициатора здесь выступает менеджер, а в роли получателей запросов - программист, маркетолог и тестеровщик. Запуская проект, менеджер тем самым запускает макрокоманду, которая содержит ряд отдельных команд. Выполнение этих команд делегируется получателям.

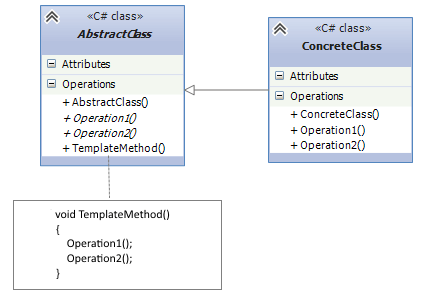
## Шаблонный метод (Template Method)

Шаблонный метод (Template Method) определяет общий алгоритм поведения подклассов, позволяя им переопределить отдельные шаги этого алгоритма без изменения его структуры.

### Когда использовать шаблонный метод?

* Когда планируется, что в будущем подклассы должны будут переопределять различные этапы алгоритма без изменения его структуры
* Когда в классах, реализующим схожий алгоритм, происходит дублирование кода. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.

Схематично в UML алгоритм можно изобразить следующим образом:



Формальная реализация паттерна на C#:

abstract class Template

{

    public abstract void TemplateMethod();

}

abstract class AbstractClass

{

    public sealed override void TemplateMethod()

    {

        Operation1();

        Operation2();

    }

    public abstract void Operation1();

    public abstract void Operation2();

}

class ConcreteClass : AbstractClass

{

    public override void Operation1()

    {

    }

    public override void Operation2()

    {

    }

}

### Участники

* **AbstractClass**: определяет шаблонный метод TemplateMethod(), который реализует алгоритм. Алгоритм может состоять из последовательности вызовов других методов, часть из которых может быть абстрактными и должны быть переопределены в классах-наследниках. При этом сам метод TemplateMethod(), представляющий структуру алгоритма, переопределяться не должен. Поэтому в данном случае мы можем объявить его с модификатором sealed для закрытия от переопределения в подклассах.
* **ConcreteClass**: подкласс, который может переопределять различные методы родительского класса.

Правда тут надо отметить, что чтобы сокрыть алгоритм от изменения в классах наследниках, метод объявляется с ключевым словом**sealed**. Но так как sealed используется только в связке с другим ключевым словом override, то необходимо абстрактный класс унаследовать от еще одного класса, в котором объявлен шаблонный метод.

Рассмотрим применение на конкретном примере. Допустим, в нашей программе используются объекты, представляющие учебу в школе и в вузе:

class School

{

    // идем в первый класс

    public void Enter() { }

    // обучение

    public void Study() { }

    // сдаем выпускные экзамены

    public void PassExams() { }

    // получение аттестата об окончании

    public void GetAttestat() { }

}

class University

{

    // поступление в университет

    public void Enter() { }

    // обучение

    public void Study() { }

    // проходим практику

    public void Practice() { }

    // сдаем выпускные экзамены

    public void PassExams() { }

    // получение диплома

     public void GetDiploma() { }

}

Как видно, эти классы очень похожи, и самое главное, реализуют примерно общий алгоритм. Да, где-то будет отличаться реализация методов, где-то чуть больше методов, но в целом мы имеем общий алгоритм, а функциональность обоих классов по большому счету дублируется. Поэтому для улучшения структуры классов мы могли бы применить шаблонный метод:

class Program

{

    static void Main(string[] args)

    {

        School school = new School();

        University university = new University();

        school.Learn();

        university.Learn();

        Console.Read();

    }

}

abstract class Learning

{

    public abstract void Learn();

}

abstract class Education :Learning

{

    public sealed override void Learn()

    {

        Enter();

        Study();

        PassExams();

        GetDocument();

    }

    public abstract void Enter();

    public abstract void Study();

    public virtual void PassExams()

    {

        Console.WriteLine("Сдаем выпускные экзамены");

    }

    public abstract void GetDocument();

}

class School : Education

{

    public override void Enter()

    {

        Console.WriteLine("Идем в первый класс");

    }

    public override void Study()

    {

        Console.WriteLine("Посещаем уроки, делаем домашние задания");

    }

    public override void GetDocument()

    {

        Console.WriteLine("Получаем аттестат о среднем образовании");

    }

}

class University : Education

{

    public override void Enter()

    {

        Console.WriteLine("Сдаем вступительные экзамены и поступаем в ВУЗ");

    }

    public override void Study()

    {

        Console.WriteLine("Посещаем лекции");

        Console.WriteLine("Проходим практику");

    }

    public override void PassExams()

    {

        Console.WriteLine("Сдаем экзамен по специальности");

    }

    public override void GetDocument()

    {

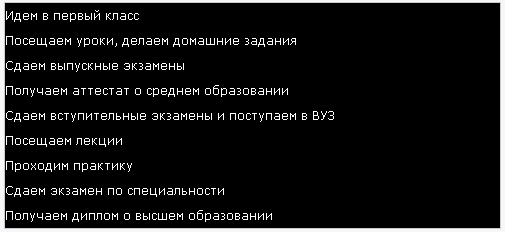
        Console.WriteLine("Получаем диплом о высшем образовании");

    }

}

При этом в базовом классе необязательно определять все методы алгоритма как абстрактные. Можно определить реализацию методов по умолчанию, как в случае с методом PassExams().

И в результате работы программы консоль выведет:



## Итератор (Iterator)

Паттерн Итератор (Iterator) предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры.

Наверное, всем программистам, работающим с языком C#, приходилось иметь дело с циклом foreach, который перебирает объекты в массиве или коллекции. При этом встроенных классов коллекций существует множество, и каждая из них отличается по своей структуре и поведению.

Ключевым моментом, который позволяет осуществить перебор коллекций с помощью foreach, является применения этими классами коллекций паттерна итератор, или проще говоря пары интерфейсов IEnumerable / IEnumerator. Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

public interface IEnumerator

{

    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов

    object Current {get;}  // текущий элемент в контейнере

    void Reset(); // перемещение в начало контейнера

}

А интерфейс IEnumerable использует IEnumerator для получения итератора для конкретного типа объекта:

public interface IEnumerable

{

    IEnumerator GetEnumerator();

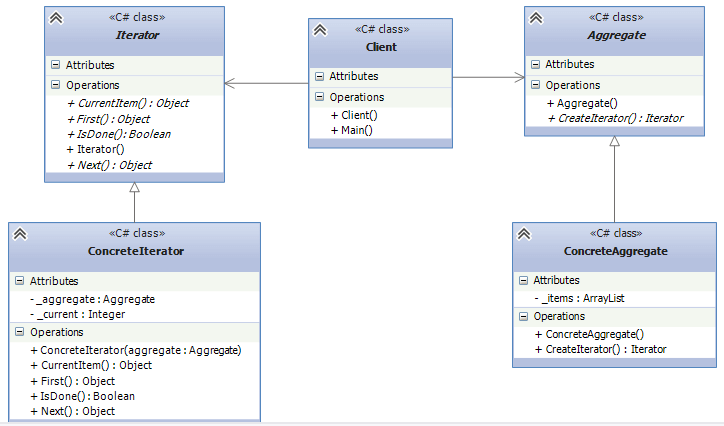
}

Используя данные интерфейсы, мы можем свести к одному шаблону - с помощью цикла foreach - любые составные объекты.

### Когда использовать итераторы?

* Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
* Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
* Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

С помощью схем UML итераторы можно описать так:

Формальное определение паттерна на C# может выглядеть следующим образом:

class Client

{

    public void Main()

    {

        Aggregate a = new ConcreteAggregate();

        Iterator i = new ConcreteIterator(a);

        object item = i.First();

        while (!i.IsDone())

        {

            item = i.Next();

        }

    }

}

abstract class Aggregate

{

    public abstract Iterator CreateIterator();

    public abstract int Count { get; protected set; }

    public abstract object this[int index] { get; set; }

}

class ConcreteAggregate : Aggregate

{

    private readonly ArrayList \_items = new ArrayList();

    public override Iterator CreateIterator()

    {

        return new ConcreteIterator(this);

    }

    public override int Count

    {

        get { return \_items.Count; }

        protected set { }

    }

    public override object this[int index]

    {

        get { return \_items[index]; }

        set { \_items.Insert(index, value); }

    }

}

abstract class Iterator

{

    public abstract object First();

    public abstract object Next();

    public abstract bool IsDone();

    public abstract object CurrentItem();

}

class ConcreteIterator : Iterator

{

    private readonly Aggregate \_aggregate;

    private int \_current;

    public ConcreteIterator(Aggregate aggregate)

    {

        this.\_aggregate = aggregate;

    }

    public override object First()

    {

        return \_aggregate[0];

    }

    public override object Next()

    {

        object ret = null;

        \_current++;

        if (\_current < \_aggregate.Count)

        {

            ret = \_aggregate[\_current];

        }

        return ret;

    }

    public override object CurrentItem()

    {

        return \_aggregate[\_current];

    }

    public override bool IsDone()

    {

        return \_current > \_aggregate.Count;

    }

}

### Участники

* **Iterator**: определяет интерфейс для обхода составных объектов
* **Aggregate**: определяет интерфейс для создания объекта-итератора
* **ConcreteIterator**: конкретная реализация итератора для обхода объекта Aggregate. Для фиксации индекса текущего перебираемого элемента использует целочисленную переменную \_current
* **ConcreteAggregate**: конкретная реализация Aggregate. Хранит элементы, которые надо будет перебирать
* **Client**: использует объект Aggregate и итератор для его обхода

Теперь рассмотрим конкретный пример. Допустим, у нас есть классы книги и библиотеки:

class Book

{

    public string Name { get; set; }

}

class Library

{

    private string[] books;

}

И, допустим, у нас есть класс читателя, который хочет получить информацию о книгах, которые находятся в библиотеке. И для этого надо осуществить перебор объектов с помощью итератора:

class Program

{

    static void Main(string[] args)

    {

        Library library = new Library();

        Reader reader = new Reader();

        reader.SeeBooks(library);

        Console.Read();

    }

}

class Reader

{

    public void SeeBooks(Library library)

    {

        IBookIterator iterator = library.CreateNumerator();

        while(iterator.HasNext())

        {

            Book book = iterator.Next();

            Console.WriteLine(book.Name);

        }

    }

}

interface IBookIterator

{

    bool HasNext();

    Book Next();

}

interface IBookNumerable

{

    IBookIterator CreateNumerator();

    int Count { get; }

    Book this[int index] { get;}

}

class Book

{

    public string Name { get; set; }

}

class Library : IBookNumerable

{

    private Book[] books;

    public Library()

    {

        books = new Book[]

        {

            new Book{Name="Война и мир"},

            new Book {Name="Отцы и дети"},

            new Book {Name="Вишневый сад"}

        };

    }

    public int Count

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get { return books[index]; }

    }

    public IBookIterator CreateNumerator()

    {

        return new LibraryNumerator(this);

    }

}

class LibraryNumerator : IBookIterator

{

    IBookNumerable aggregate;

    int index=0;

    public LibraryNumerator(IBookNumerable a)

    {

        aggregate = a;

    }

    public bool HasNext()

    {

        return index<aggregate.Count;

    }

    public Book Next()

    {

        return aggregate[index++];

    }

}

Интерфейс IBookIterator представляет итератор наподобие интерфейса IEnumerator. Роль интерфейса составного агрегата представляет тип IBookNumerable. Клиентом здесь является класс Reader, который использует итератор для обхода объекта библиотеки.

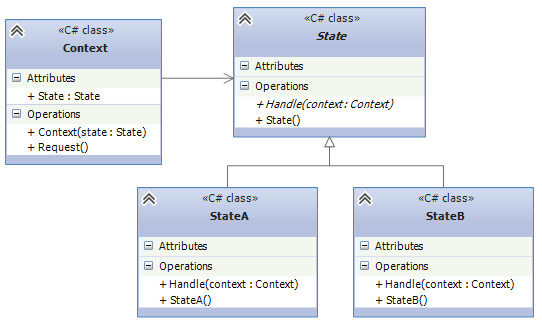
## Состояние (State)

Состояние (State) - шаблон проектирования, который позволяет объекту изменять свое поведение в зависимости от внутреннего состояния.

### Когда применяется данный паттерн?

* Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения
* Когда в коде методов объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта

UML-диаграмма данного шаблона проектирования предлагает следующую систему:



Формальное определение паттерна на C#:

class Program

{

    static void Main()

    {

        Context context = new Context(new StateA());

        context.Request();

        context.State = new StateB();

        context.Request();

    }

}

abstract class State

{

    public abstract void Handle(Context context);

}

class StateA : State

{

    public override void Handle(Context context)

    {

        context.State = new StateB();

    }

}

class StateB : State

{

    public override void Handle(Context context)

    {

        context.State = new StateA();

    }

}

class Context

{

    public State State { get; set; }

    public Context(State state)

    {

        this.State = state;

    }

    public void Request()

    {

        this.State.Handle(this);

    }

}

### Участники паттерна

* **State**: определяет интерфейс состояния
* Классы **StateA** и **StateB** - конкретные реализации состояний
* **Context**: представляет объект, поведение которого должно динамически изменяться в соответствии с состоянием. Выполнение же конкретных действий делегируется объекту состояния

Например, вода может находиться в ряде состояний: твердое, жидкое, парообразное. Допустим, нам надо определить класс Вода, у которого бы имелись методы для нагревания и заморозки воды. Без использования паттерна Состояние мы могли бы написать следующую программу:

class Program

{

    static void Main(string[] args)

    {

        Water water = new Water(WaterState.LIQUID);

        water.Heat();

        water.Frost();

        water.Frost();

        Console.Read();

    }

}

enum WaterState

{

    SOLID,

    LIQUID,

    GAS

}

class Water

{

    public WaterState State { get; set; }

    public Water(WaterState ws)

    {

        State = ws;

    }

    public void Heat()

    {

        if(State==WaterState.SOLID)

        {

            Console.WriteLine("Превращаем лед в жидкость");

            State = WaterState.LIQUID;

        }

        else if (State == WaterState.LIQUID)

        {

            Console.WriteLine("Превращаем жидкость в пар");

            State = WaterState.GAS;

        }

        else if (State == WaterState.GAS)

        {

            Console.WriteLine("Повышаем температуру водяного пара");

        }

    }

    public void Frost()

    {

        if (State == WaterState.LIQUID)

        {

            Console.WriteLine("Превращаем жидкость в лед");

            State = WaterState.SOLID;

        }

        else if (State == WaterState.GAS)

        {

            Console.WriteLine("Превращаем водяной пар в жидкость");

            State = WaterState.LIQUID;

        }

    }

}

Вода имеет три состояния, и в каждом методе нам надо смотреть на текущее состояние, чтобы произвести действия. В итоге с трех состояний уже получается нагромождение условных конструкций. Да и самим методов в классе Вода может также быть множество, где также надо будет действовать в зависимости от состояния. Поэтому, чтобы сделать программу более гибкой, в данном случае мы можем применить паттерн Состояние:

class Program

{

    static void Main(string[] args)

    {

        Water water = new Water(new LiquidWaterState());

        water.Heat();

        water.Frost();

        water.Frost();

        Console.Read();

    }

}

 class Water

    {

        public IWaterState State { get; set; }

        public Water(IWaterState ws)

        {

            State = ws;

        }

        public void Heat()

        {

            State.Heat(this);

        }

        public void Frost()

        {

            State.Frost(this);

        }

    }

interface IWaterState

{

    void Heat(Water water);

    void Frost(Water water);

}

class SolidWaterState : IWaterState

{

    public void Heat(Water water)

    {

        Console.WriteLine("Превращаем лед в жидкость");

        water.State = new LiquidWaterState();

    }

    public void Frost(Water water)

    {

        Console.WriteLine("Продолжаем заморозку льда");

    }

}

class LiquidWaterState : IWaterState

{

    public void Heat(Water water)

    {

        Console.WriteLine("Превращаем жидкость в пар");

        water.State = new GasWaterState();

    }

    public void Frost(Water water)

    {

        Console.WriteLine("Превращаем жидкость в лед");

        water.State = new SolidWaterState();

    }

}

class GasWaterState : IWaterState

{

    public void Heat(Water water)

    {

        Console.WriteLine("Повышаем температуру водяного пара");

    }

    public void Frost(Water water)

    {

        Console.WriteLine("Превращаем водяной пар в жидкость");

        water.State = new LiquidWaterState();

    }

}

Таким образом, реализация паттерна Состояние позволяет вынести поведение, зависящее от текущего состояния объекта, в отдельные классы, и избежать перегруженности методов объекта условными конструкциями, как if..else или switch. Кроме того, при необходимости мы можем ввести в систему новые классы состояний, а имеющиеся классы состояний использовать в других объектах.

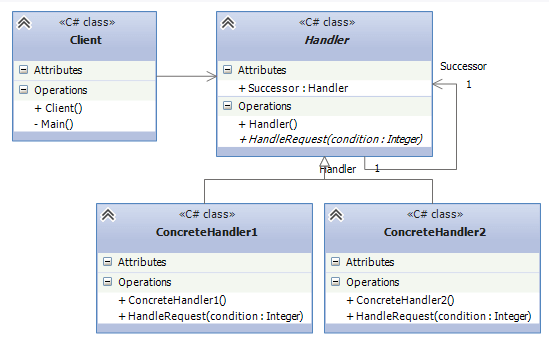
## Цепочка Обязанностей (Chain of responsibility)

Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке, пока один из ее объектов не обработает запрос. Каждый объект при получении запроса выбирает, либо обработать запрос, либо передать выполнение запроса следующему по цепочке.

### Когда применяется цепочка обязанностей?

* Когда имеется более одного объекта, который может обработать определенный запрос
* Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту
* Когда набор объектов задается динамически

UML-представление паттерна:



Формальное определение на языке C#:

class Client

{

    void Main()

    {

        Handler h1 = new ConcreteHandler1();

        Handler h2 = new ConcreteHandler2();

        h1.Successor = h2;

        h1.HandleRequest(2);

    }

}

abstract class Handler

{

    public Handler Successor { get; set; }

    public abstract void HandleRequest(int condition);

}

class ConcreteHandler1 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition == 1)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

class ConcreteHandler2 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition==2)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

### Участники

* **Handler**: определяет интерфейс для обработки запроса. Также может определять ссылку на следующий обработчик запроса
* **ConcreteHandler1** и **ConcreteHandler2**: конкретные обработчики, которые реализуют функционал для обработки запроса. При невозможности обработки и наличия ссылки на следующий обработчик, передают запрос этому обработчику

В данном случае для простоты примера в качестве параметра передается некоторое число, и в зависимости от значения данного числа обработчики и принимают решения об обработке запроса.

* **Client**: отправляет запрос объекту Handler

То есть у нас образуется небольшая цепочка обработки запроса:



Использование цепочки обязанностей дает нам следующие преимущества:

* Ослабление связанности между объектами. Отправителю и получателю запроса ничего не известно друг о друге. Клиенту неизветна цепочка объектов, какие именно объекты составляют ее, как запрос в ней передается.
* В цепочку с легкостью можно добавлять новые типы объектов, которые реализуют общий интерфейс.

В то же время у паттерна есть недостаток: никто не гарантирует, что запрос в конечном счете будет обработан. Если необходимого обработчика в цепочки не оказалось, то запрос просто выходит из цепочки и остается необработанным.

Использование паттерна довольно часто встречается в нашей жизни. Достаточно распространена ситуация, когда чиновники перекладывают друг на друга по цепочке выполнения какого-нибудь дела, и оно в конце концов оказывается не выполненным. Или когда мы идем в поликлинику, но при этом точно не знаем характер заболевания. В этом случае мы идем к терапевту, а он в зависимости от заболевания уже может либо сам лечить, либо отправить на лечение другим специализированным врачам.

Рассмотрим конкретный пример. Допустим, необходимо послать человеку определенную сумму денег. Однако мы точно не знаем, какой способ отправки может использоваться: банковский перевод, системы перевода типа WesternUnion и Unistream или система онлайн-платежей PayPal. Нам просто надо внести сумму, выбрать человека и нажать на кнопку. Подобная система может использоваться на сайтах фриланса, где все отношения между исполнителями и заказчиками происходят опосредованно через функции системы и где не надо знать точные данные получателя.

class Program

{

    static void Main(string[] args)

    {

        Receiver receiver = new Receiver(false, true, true);

        PaymentHandler bankPaymentHandler = new BankPaymentHandler();

        PaymentHandler moneyPaymentHnadler = new MoneyPaymentHandler();

        PaymentHandler paypalPaymentHandler = new PayPalPaymentHandler();

        bankPaymentHandler.Successor = paypalPaymentHandler;

        paypalPaymentHandler.Successor = moneyPaymentHnadler;

        bankPaymentHandler.Handle(receiver);

        Console.Read();

    }

}

class Receiver

{

    // банковские переводы

    public bool BankTransfer { get; set; }

    // денежные переводы - WesternUnion, Unistream

    public bool MoneyTransfer { get; set; }

    // перевод через PayPal

    public bool PayPalTransfer { get; set; }

    public Receiver(bool bt, bool mt, bool ppt)

    {

        BankTransfer = bt;

        MoneyTransfer = mt;

        PayPalTransfer = ppt;

    }

}

abstract class PaymentHandler

{

    public PaymentHandler Successor { get; set; }

    public abstract void Handle(Receiver receiver);

}

class BankPaymentHandler : PaymentHandler

{

    public override void Handle(Receiver receiver)

    {

        if (receiver.BankTransfer == true)

            Console.WriteLine("Выполняем банковский перевод");

        else if (Successor != null)

            Successor.Handle(receiver);

    }

}

class PayPalPaymentHandler : PaymentHandler

{

    public override void Handle(Receiver receiver)

    {

        if (receiver.BankTransfer == true)

            Console.WriteLine("Выполняем перевод через PayPal");

        else if (Successor != null)

            Successor.Handle(receiver);

    }

}

// переводы с помощью системы денежных переводов

class MoneyPaymentHandler : PaymentHandler

{

    public override void Handle(Receiver receiver)

    {

        if (receiver.MoneyTransfer == true)

            Console.WriteLine("Выполняем перевод через системы денежных переводов");

        else if (Successor != null)

            Successor.Handle(receiver);

    }

}

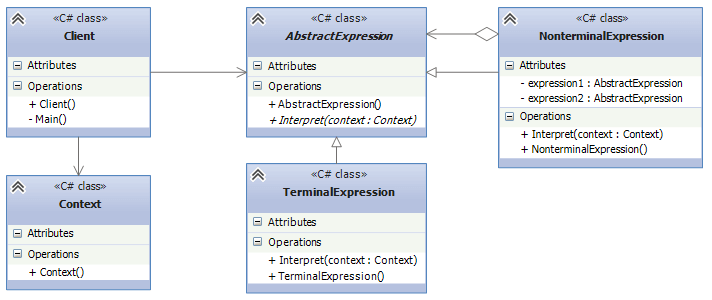
Класс Receiver с помощью конструктора и передаваемых в него параметров устанавливает возможные используемые системы платежей. При осуществлении платежа каждый отдельный объект PaymentHandler будет проверять установку у получателя определенного типа платежей. И если произойдет сопоставление типа платежей у получателя объекту PaymentHandler, то данный объект выполняет платеж. Если же необходимого способа платежей не будет определено, то деньги остаются в системе.

При этом преимуществом цепочки является и то, что она позволяет расположить последовательность объектов-обработчиков в ней в зависимости от их приоритета.

Паттерн Интерпретатор (Interpreter) определяет представление грамматики для заданного языка и интерпретатор предложений этого языка. Как правило, данный шаблон проектирования применяется для часто повторяющихся операций.

Хотя паттерн требует понимания теории формальных языков и грамматик, на самом деле он не так сложен в понимании.

С помощью диаграмм UML паттерн можно описать так:



На языке C# формальная структура программы могла бы выглядеть следующим образом:

class Client

{

    void Main()

    {

        Context context = new Context();

        var expression = new NonterminalExpression();

        expression.Interpret(context);

    }

}

class Context

{

}

abstract class AbstractExpression

{

    public abstract void Interpret(Context context);

}

class TerminalExpression : AbstractExpression

{

    public override void Interpret(Context context)

    {

    }

}

class NonterminalExpression : AbstractExpression

{

    AbstractExpression expression1;

    AbstractExpression expression2;

    public override void Interpret(Context context)

    {

    }

}

### Участники

* **AbstractExpression**: определяет интерфейс выражения, объявляет метод Interpret()
* **TerminalExpression**: терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект TerminalExpression
* **NonterminalExpression**: нетерминальное выражение, представляет правило грамматики. Для каждого отдельного правила грамматики создается свой объект NonterminalExpression.
* **Context**: содержит общую для интерпретатора информацию. Может использоваться объектами терминальных и нетерминальных выражений для сохранения состояния операций и последующего доступа к сохраненному состоянию
* **Client**: строит предложения языка с данной грамматикой в виде абстрактного синтаксического дерева, узлами которого являются объекты TerminalExpression и NonterminalExpression

Методы Interpret в нетерминальных выражениях позволяют реализовать правила грамматики. При этом мы легко может добавить новые правила грамматики, определив новые объекты NonterminalExpression со своей реализацией метода Interpret. Однако данный паттерн подходит только для тех случаев, когда правила грамматики относительно простые. В более сложных случаях следует выбирать другие способы проектирования приложения.

Например, нам надо разработать программ для вычислений простейших операций сложения и вычитания с помощью переменных: x + y - z. Для этого можно определить следующую грамматику:

class Program

{

    static void Main(string[] args)

    {

        Context context = new Context();

        // определяем набор переменных

        int x = 5;

        int y = 8;

        int z = 2;

        // добавляем переменные в контекст

        context.SetVariable("x", x);

        context.SetVariable("y", y);

        context.SetVariable("z", z);

        // создаем объект для вычисления выражения x + y - z

        IExpression expression = new SubtractExpression(

            new AddExpression(

                new NumberExpression("x"), new NumberExpression("y")

            ),

            new NumberExpression("z")

        );

        int result = expression.Interpret(context);

        Console.WriteLine("результат: {0}", result);

        Console.Read();

    }

}

class Context

{

    Dictionary<string, int> variables;

    public Context()

    {

        variables = new Dictionary<string, int>();

    }

    // получаем значение переменной по ее имени

    public int GetVariable(string name)

    {

        return variables[name];

    }

    public void SetVariable(string name, int value)

    {

        if (variables.ContainsKey(name))

            variables[name] = value;

        else

            variables.Add(name, value);

    }

}

// интерфейс интерпретатора

interface IExpression

{

    int Interpret(Context context);

}

// терминальное выражение

class NumberExpression : IExpression

{

    string name; // имя переменной

    public NumberExpression(string variableName)

    {

        name = variableName;

    }

    public int Interpret(Context context)

    {

        return context.GetVariable(name);

    }

}

// нетерминальное выражение для сложения

class AddExpression : IExpression

{

    IExpression leftExpression;

    IExpression rightExpression;

    public AddExpression(IExpression left, IExpression right)

    {

        leftExpression = left;

        rightExpression = right;

    }

    public int Interpret(Context context)

    {

        return leftExpression.Interpret(context) + rightExpression.Interpret(context);

    }

}

 // нетерминальное выражение для вычитания

class SubtractExpression : IExpression

{

    IExpression leftExpression;

    IExpression rightExpression;

    public SubtractExpression(IExpression left, IExpression right)

    {

        leftExpression = left;

        rightExpression = right;

    }

    public int Interpret(Context context)

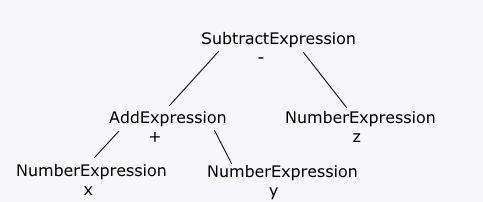
    {

        return leftExpression.Interpret(context) - rightExpression.Interpret(context);

    }

}

В данном случае все действия интерпретатора можно описать следующим деревом:



Класс Context определяет методы для установки значений переменных и для получения их значений.

В качестве интерпретатора используется интерфейс IExpression. Его реализация - класс NumberExpression предназначен для выражения отдельных переменных - это терминальные объекты.

Другие реализации интерфейса - классы AddExpression и SubtractExpression представляют нетерминальные объекты. Они реализуют простейшие операции сложения и вычитания и могут рекурсивно обращаться к методам Interpret используемых терминальных и нетерминальных объектов.

Клиент, в роли которого выступает класс Program, инициализирует контекст и для вычисления выражения x + y - z создается объект SubtractExpression, который в качестве параметров принимает другие объекты IExpression.

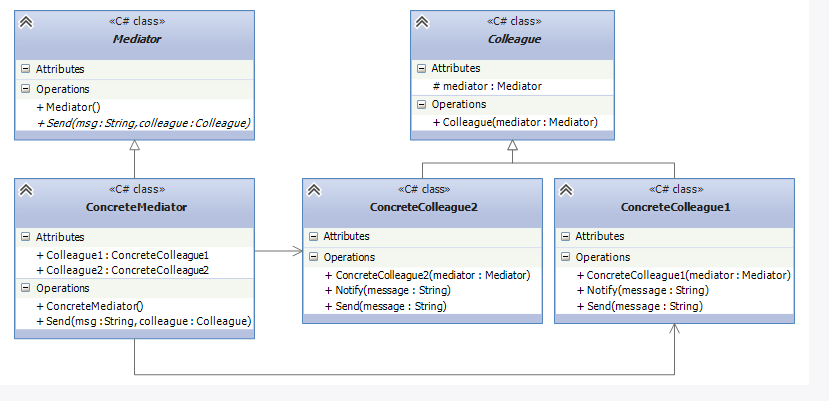
## Посредник (Mediator)

Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда используется паттерн Посредник?

* Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
* Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

Схематично с помощью UML паттерн можно описать следующим образом:



Формальная структура классов и связей между ними с применением паттерна на языке C#:

abstract class Mediator

{

    public abstract void Send(string msg, Colleague colleague);

}

abstract class Colleague

{

    protected Mediator mediator;

    public Colleague(Mediator mediator)

    {

        this.mediator = mediator;

    }

}

class ConcreteColleague1 : Colleague

{

    public ConcreteColleague1(Mediator mediator)

        : base(mediator)

    { }

    public void Send(string message)

    {

        mediator.Send(message, this);

    }

    public void Notify(string message)

    { }

}

class ConcreteColleague2 : Colleague

{

    public ConcreteColleague2(Mediator mediator)

        : base(mediator)

    { }

    public void Send(string message)

    {

        mediator.Send(message, this);

    }

    public void Notify(string message)

    { }

}

class ConcreteMediator : Mediator

{

    public ConcreteColleague1 Colleague1 { get; set; }

    public ConcreteColleague2 Colleague2 { get; set; }

    public override void Send(string msg, Colleague colleague)

    {

        if (Colleague1 == colleague)

            Colleague2.Notify(msg);

        else

            Colleague1.Notify(msg);

    }

}

### Участники

* **Mediator**: представляет интерфейс для взаимодействия с объектами Colleague
* **Colleague**: представляет интерфейс для взаимодействия с объектом Mediator
* **ConcreteColleague1** и **ConcreteColleague2**: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
* **ConcreteMediator**: конкретный посредник, реализующий интерфейс типа Mediator

Рассмотрим реальный пример. Система создания программных продуктов включает ряд акторов: заказчики, программисты, тестировщики и так далее. Но нередко все эти акторы взаимодействуют между собой не непосредственно, а опосредованно через менеджера проектов. То есть менеджер проектов выполняет роль посредника. В этом случае процесс взаимодействия между объектами мы могли бы описать так:

class Program

{

    static void Main(string[] args)

    {

        ManagerMediator mediator = new ManagerMediator();

        Colleague customer = new CustomerColleague(mediator);

        Colleague programmer = new ProgrammerColleague(mediator);

        Colleague tester = new TesterColleague(mediator);

        mediator.Customer = customer;

        mediator.Programmer = programmer;

        mediator.Tester = tester;

        customer.Send("Есть заказ, надо сделать программу");

        programmer.Send("Программа готова, надо протестировать");

        tester.Send("Программа протестирована и готова к продаже");

        Console.Read();

    }

}

abstract class Mediator

{

    public abstract void Send(string msg, Colleague colleague);

}

abstract class Colleague

{

    protected Mediator mediator;

    public Colleague(Mediator mediator)

    {

        this.mediator = mediator;

    }

    public virtual void Send(string message)

    {

        mediator.Send(message, this);

    }

    public abstract void Notify(string message);

}

// класс заказчика

class CustomerColleague : Colleague

{

    public CustomerColleague(Mediator mediator)

        : base(mediator)

    { }

    public override void Notify(string message)

    {

        Console.WriteLine("Сообщение заказчику: " + message);

    }

}

// класс программиста

class ProgrammerColleague : Colleague

{

    public ProgrammerColleague(Mediator mediator)

        : base(mediator)

    { }

    public override void Notify(string message)

    {

        Console.WriteLine("Сообщение программисту: " + message);

    }

}

// класс тестера

class TesterColleague : Colleague

{

    public TesterColleague(Mediator mediator)

        : base(mediator)

    { }

    public override void Notify(string message)

    {

        Console.WriteLine("Сообщение тестеру: " + message);

    }

}

class ManagerMediator : Mediator

{

    public Colleague Customer { get; set; }

    public Colleague Programmer { get; set; }

    public Colleague Tester { get; set; }

    public override void Send(string msg, Colleague colleague)

    {

        // если отправитель - заказчик, значит есть новый заказ

        // отправляем сообщение программисту - выполнить заказ

        if (Customer == colleague)

            Programmer.Notify(msg);

        // если отправитель - программист, то можно приступать к тестированию

        // отправляем сообщение тестеру

        else if (Programmer == colleague)

            Tester.Notify(msg);

        // если отправитель - тест, значит продукт готов

        // отправляем сообщение заказчику

        else if (Tester == colleague)

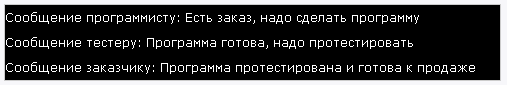
            Customer.Notify(msg);

    }

}

Класс менеджера - ManagerMediator в методе Send() проверяет, от кого пришло сообщение, и в зависимости от отправителя перенаправляет его другому объекту с помощью методов Notify(), определенных в классе Colleague.

Консольный вывод программы:



В итоге применение паттерна Посредник дает нам следующие преимущества:

* Устраняется сильная связанность между объектами Colleague
* Упрощается взаимодействие между объектами: вместо связей по типу "все-ко-всем" применяется связь "один-ко-всем"
* Взаимодействие между объектами абстрагируется и выносится в отдельный интерфейс
* Централизуется управления отношениями между объектами

## Хранитель (Memento)

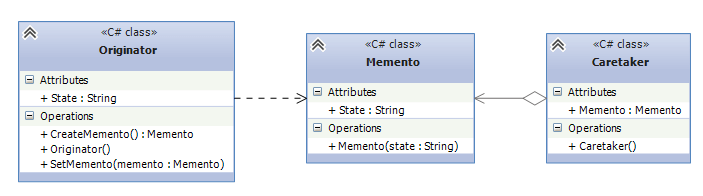
Паттерн Хранитель (Memento) позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

Когда использовать Memento?

* Когда нужно охранить его состояние объекта для возможного последующего восстановления
* Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции

То есть ключевыми понятиями для данного паттерна являются **сохранение внутреннего состояния** и **инкапсуляция**, и важно соблюсти баланс между ними. Ведь, как правило, если мы не нарушаем инкапсуляцию, то состояние объекта хранится в объекте в приватных переменных. И не всегда для доступа к этим переменным есть методы или свойства с сеттерами и геттерами. Например, в игре происходит управление героем, все состояние которого заключено в нем самом - оружие героя, показатель жизней, силы, какие-то другие показатели. И нередко может возникнуть ситуация, сохранить все эти показатели во вне, чтобы в будущем можно было откатиться к предыдущему уровню и начать игру заново. В этом случае как раз и может помочь паттерн Хранитель.

С помощью диаграмм структуру паттерна можно изобразить следующим образом:



Формальная структура паттерна на языке C#:

class Memento

{

    public string State { get; private set;}

    public Memento(string state)

    {

        this.State = state;

    }

}

class Caretaker

{

    public Memento Memento { get; set; }

}

class Originator

{

    public string State { get; set; }

    public void SetMemento(Memento memento)

    {

        State = memento.State;

    }

    public Memento CreateMemento()

    {

        return new Memento(State);

    }

}

### Участники

* **Memento**: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
* **Originator**: создает объект хранителя для сохранения своего состояния
* **Caretaker**: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

Теперь рассмотрим реальный пример: нам надо сохранять состояние игрового персонажа в игре:

class Program

{

    static void Main(string[] args)

    {

        Hero hero = new Hero();

        hero.Shout(); // делаем выстрел, осталось 9 патронов

        GameHistory game = new GameHistory();

        game.History.Push(hero.SaveState()); // сохраняем игру

        hero.Shout(); //делаем выстрел, осталось 8 патронов

        hero.RestoreState(game.History.Pop());

        hero.Shout(); //делаем выстрел, осталось 8 патронов

        Console.Read();

    }

}

// Originator

class Hero

{

    private int patrons=10; // кол-во патронов

    private int lives=5; // кол-во жизней

    public void Shout()

    {

        if (patrons > 0)

        {

            patrons--;

            Console.WriteLine("Производим выстрел. Осталось {0} патронов", patrons);

        }

        else

            Console.WriteLine("Патронов больше нет");

    }

    // сохранение состояния

    public HeroMemento SaveState()

    {

        Console.WriteLine("Сохранение игры. Параметры: {0} патронов, {1} жизней", patrons, lives);

        return new HeroMemento(patrons, lives);

    }

    // восстановление состояния

    public void RestoreState(HeroMemento memento)

    {

        this.patrons=memento.Patrons;

        this.lives = memento.Lives;

        Console.WriteLine("Восстановление игры. Параметры: {0} патронов, {1} жизней", patrons, lives);

    }

}

// Memento

class HeroMemento

{

    public int Patrons { get; private set; }

    public int Lives { get; private set; }

    public HeroMemento(int patrons, int lives)

    {

        this.Patrons = patrons;

        this.Lives = lives;

    }

}

// Caretaker

class GameHistory

{

    public Stack<HeroMemento> History { get; private set; }

    public GameHistory()

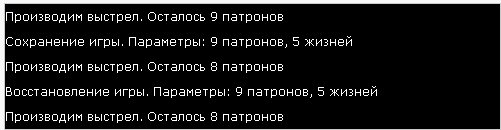
    {

        History = new Stack<HeroMemento>();

    }

}

Консольный вывод программы:



Здесь в роли Originator выступает класс Hero, состояние которого описывается количество патронов и жизней. Для хранения состояния игрового персонажа предназначен класс HeroMemento. С помощью метода SaveState() объект Hero может сохранить свое состояние в HeroMemento, а с помощью метода RestoreState() - восстановить.

Для хранения состояний предназначен класс GameHistory, причем все состояния хранятся в стеке, что позволяет с легкостью извлекать последнее сохраненное состояние.

Использование паттерна Memento дает нам следующие преимущества:

* Уменьшение связанности системы
* Сохранение инкапсуляции информации
* Определение простого интерфейса для сохранения и восстановления состояния

В то же время мы можем столкнуться с недостатками, в частности, если требуется сохранение большого объема информации, то возрастут издержки на хранение всего объема состояния.

## Посетитель (Visitor)

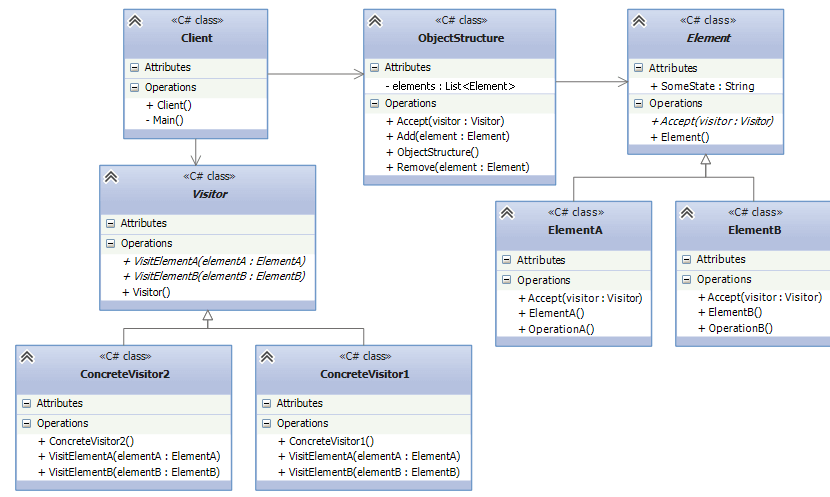
Паттерн Посетитель (Visitor) позволяет определить операцию для объектов других классов без изменения этих классов.

При использовании паттерна Посетитель определяются две иерархии классов: одна для элементов, для которых надо определить новую операцию, и вторая иерархия для посетителей, описывающих данную операцию.

Когда использовать данный паттерн?

* Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов
* Когда классам необходимо добавить одинаковый набор операций без изменения этих классов
* Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

Общая структура классов паттерна, описанная с помощью диаграмм UML:



Формальная структура на C#:

class Client

{

    void Main()

    {

        var structure = new ObjectStructure();

        structure.Add(new ElementA());

        structure.Add(new ElementB());

        structure.Accept(new ConcreteVisitor1());

        structure.Accept(new ConcreteVisitor2());

    }

}

abstract class Visitor

{

    public abstract void VisitElementA(ElementA elemA);

    public abstract void VisitElementB(ElementB elemB);

}

class ConcreteVisitor1 : Visitor

{

    public override void VisitElementA(ElementA elementA)

    {

        elementA.OperationA();

    }

    public override void VisitElementB(ElementB elementB)

    {

            elementB.OperationB();

    }

}

class ConcreteVisitor2 : Visitor

{

    public override void VisitElementA(ElementA elementA)

    {

        elementA.OperationA();

    }

    public override void VisitElementB(ElementB elementB)

    {

        elementB.OperationB();

    }

}

class ObjectStructure

{

    List<Element> elements = new List<Element>();

    public void Add(Element element)

    {

        elements.Add(element);

    }

    public void Remove(Element element)

    {

        elements.Remove(element);

    }

    public void Accept(Visitor visitor)

    {

        foreach (Element element in elements)

            element.Accept(visitor);

    }

}

abstract class Element

{

    public abstract void Accept(Visitor visitor);

    public string SomeState { get; set; }

}

class ElementA : Element

{

    public override void Accept(Visitor visitor)

    {

        visitor.VisitElementA(this);

    }

    public void OperationA()

    { }

}

class ElementB : Element

{

    public override void Accept(Visitor visitor)

    {

        visitor.VisitElementB(this);

    }

    public void OperationB()

    { }

}

### Участники

* **Visitor**: интерфейс посетителя, который определяет метод Visit() для каждого объекта Element
* **ConcreteVisitor1 / ConcreteVisitor2**: конкретные классы посетителей, реализуют интерфейс, определенный в Visitor.
* **Element**: определяет метод Accept(), в котором в качестве параметра принимается объект Visitor
* **ElementA / ElementB**: конкретные элементы, которые реализуют метод Accept()
* **ObjectStructure**: некоторая структура, которая хранит объекты Element и предоставляет к ним доступ. Это могут быть и простые списки, и сложные составные структуры в виде деревьев

Сущность работы паттерна состоит в том, что вначале создает объект посетителя, который обходит или посещает все элементы в структуре ObjectStructure, у которой вызывается метод Accept():

public void Accept(Visitor visitor)

{

    foreach (Element element in elements)

                element.Accept(visitor);

}

При посещении каждого элемента посещаемый элемент вызывает у посетителя метод, соответствующий данному элементу:

public override void Accept(Visitor visitor)

{

    visitor.VisitElementA(this);

}

В этот метод элемент передает ссылку на себя, чтобы посетитель мог получить доступ к состоянию элемента. А в самом посетителе уже могут вызываться методы элемента или производиться различные действия над элементом:

public override void VisitElementA(ElementA elementA)

{

    elementA.OperationA();

}

Данная техника еще называется **двойной диспетчеризацией (double dispatch)**, когда выполнение операции зависит от имени запроса и двух типов получателей (объект Visitor и объект Element).

Рассмотрим на примере. Как известно, нередко для разных категорий вкладчиков банки имеют свои правила: оформления вкладов, выдача кредитов, начисления процентов и т.д. Соответственно классы, описывающие данные объекты, тоже будут разными. Но что важно, как правило, правила обслуживания четко описает весь набор категорий клиентов. Например, есть физические лица, есть юридические, отдельные правила для индивидуальных или частных предпринимателей и т.д. Поэтому структура классов, представляющая клиентов будет относительно фиксированной, то есть не склонной к изменениям.

И допустим, в какой-то момент мы решили добавить в классы клиентов функционал сериализации в html. В этом случае мы могли бы построить следующую структуру классов:

interface IAccount

{

    string ToHtml();

}

// физическое лицо

class Person : IAccount

{

    public string FIO { get; set; } //Фамилия Имя Отчество

    public string AccNumber { get; set; } // номер счета

    public void ToHtml()

    {

        string result = "<table><tr><td>Свойство<td><td>Значение</td></tr>";

        result += "<tr><td>FIO<td><td>" + FIO + "</td></tr>";

        result += "<tr><td>Number<td><td>" + Number + "</td></tr></table>";

        Console.WriteLine(result);

    }

}

// юридическое лицо

class Company : IAccount

{

    public string Name { get; set; } // название

    public string RegNumber { get; set; } // гос регистрационный номер

    public string Number { get; set; } // номер счета

    public void ToHtml()

    {

        string result = "<table><tr><td>Свойство<td><td>Значение</td></tr>";

        result += "<tr><td>Name<td><td>" + Name + "</td></tr>";

        result += "<tr><td>RegNumber<td><td>" + RegNumber + "</td></tr>";

        result += "<tr><td>Number<td><td>" + Number + "</td></tr></table>";

        Console.WriteLine(result);

    }

}

Каждый класс имеет свой набор свойств и с помощью метода ToHtml() создает таблицу со значениями этих свойств. Но допустим, мы решили добавить потом еще сериализацию в формат xml. Задача относительно проста: добавить в интерфейс IAccount новый метод ToXml() и реализовать его в классах Person и Company.

Но еще через некоторое время мы захотим добавить сериализацию в формат json. Однако в будущем могут появиться новые форматы, которые мы также захотим поддерживать. Частое внесение изменение в фиксированную структуру классов в данном случае не будет оптимально. И для решения этой задачи воспользуемся паттерном Посетитель:

class Program

{

    static void Main(string[] args)

    {

        var structure = new Bank();

        structure.Add(new Person { Name = "Иван Алексеев", Number = "82184931" });

        structure.Add(new Company {Name="Microsoft", RegNumber="ewuir32141324", Number="3424131445"});

        structure.Accept(new HtmlVisitor());

        structure.Accept(new XmlVisitor());

        Console.Read();

    }

}

interface IVisitor

{

    void VisitPersonAcc(Person acc);

    void VisitCompanyAc(Company acc);

}

// сериализатор в HTML

class HtmlVisitor : IVisitor

{

    public void VisitPersonAcc(Person acc)

    {

        string result = "<table><tr><td>Свойство<td><td>Значение</td></tr>";

        result += "<tr><td>Name<td><td>" + acc.Name + "</td></tr>";

        result += "<tr><td>Number<td><td>" + acc.Number + "</td></tr></table>";

        Console.WriteLine(result);

    }

    public void VisitCompanyAc(Company acc)

    {

        string result = "<table><tr><td>Свойство<td><td>Значение</td></tr>";

        result += "<tr><td>Name<td><td>" + acc.Name + "</td></tr>";

        result += "<tr><td>RegNumber<td><td>" + acc.RegNumber + "</td></tr>";

        result += "<tr><td>Number<td><td>" + acc.Number + "</td></tr></table>";

        Console.WriteLine(result);

    }

}

// сериализатор в XML

class XmlVisitor : IVisitor

{

    public void VisitPersonAcc(Person acc)

    {

        string result = "<Person><Name>"+acc.Name+"</Name>"+

            "<Number>"+acc.Number+"</Number><Person>";

        Console.WriteLine(result);

    }

    public void VisitCompanyAc(Company acc)

    {

        string result = "<Company><Name>" + acc.Name + "</Name>" +

            "<RegNumber>" + acc.RegNumber + "</RegNumber>" +

            "<Number>" + acc.Number + "</Number><Company>";

        Console.WriteLine(result);

    }

}

class Bank

{

    List<IAccount> accounts = new List<IAccount>();

    public void Add(IAccount acc)

    {

        accounts.Add(acc);

    }

    public void Remove(IAccount acc)

    {

        accounts.Remove(acc);

    }

    public void Accept(IVisitor visitor)

    {

        foreach (IAccount acc in accounts)

            acc.Accept(visitor);

    }

}

interface IAccount

{

    void Accept(IVisitor visitor);

}

class Person : IAccount

{

    public string Name { get; set; }

    public string Number { get; set; }

    public void Accept(IVisitor visitor)

    {

        visitor.VisitPersonAcc(this);

    }

}

class Company : IAccount

{

    public string Name { get; set; }

    public string RegNumber { get; set; }

    public string Number { get; set; }

    public void Accept(IVisitor visitor)

    {

        visitor.VisitCompanyAc(this);

    }

}

В роли абстрактного класса Element здесь выступает интерфейс IAccount. Однако его реализации теперь не содержат методToHtml(), и любой другой метод для сериализации в какой-либо формат. Так как вся функциональность по сериализации вынесена в отдельные классы посетителей. В итоге классы Person и Company становятся намного чище и проще по структуре.

И если нам надо добавить новый способ сериализации, достаточно просто определить еще один класс посетителя.

Варианты заданий:

1. Танкист (различные танки/различные пушки/снаряды) (Стратегия)
2. Водитель (машины/двигатели/топливо) (Стратения)
3. Рынок драгоценных металлов (Наблюдатель)
4. Тотализатор на ипподроме (Наблюдатель)
5. Проект умного дома (Команда)
6. Командование армией (Команда)
7. Интернет магазин (Итератор)
8. Игра перемещение по лабиринту (Состояние)
9. Система для обзвона (Цепочка обязанностей)
10. Строительная компания (Посредник)
11. Салон печати (Посредник)
12. Битва 2х героев на арене (Хранитель)
13. Аукцион (Наблюдатель)
14. Система управления марсоходом (Состояние) (Придумать карту передвижения влияющую на состояние)
15. Система удаленного доступа (Команда)

Ход работы:

1. Реализовать программу согласно варианту.
2. Предоставить интерактивный интерфейс взаимодействия.
3. Подготовиться к защите.
4. Для повышения оценки используйте дополнительный шалон.

Защита. В ходе защиты лабораторной работы могут быть следующие задания:

1. Расширение функционала.
2. Теоретический вопрос про любой шаблон описанный в лабораторной работе.