Урок № 3

Паттерны поведения

Содержание

1. Понятие паттерна поведения

Паттерны поведения (поведенческие паттерны) как видно из названия служат для управления различными вариантами поведения системы объектов (классов). В этом уроке мы рассмотрим некоторые из данных паттернов.

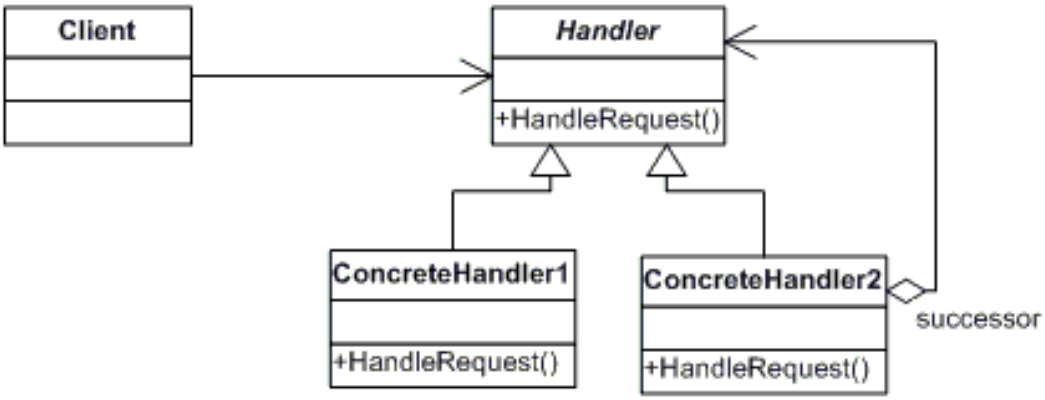
2. Паттерн Chain Of Responsibility

Данный паттерн предназначен для того чтобы позволять объекту отправлять команду, не имея информации об объекте(-ах), получающих ее. Важно отметить, что команда передается группе объектов, которая часто является частью более крупной структуры.

Каждый объект цепочки может обрабатывать, передавать

полученную команду следующему объекту в цепи или делать и то, и другое.

Рассмотрим UML диаграмму для данного паттерна:



В данной диаграмме следующие участники:

Handler

- Определяет интерфейс для обработки запросов

ConcreteHandler

- Обрабатывает запрос, предназначенный конкретному исполнителю
- Если ConcreteHandler может обработать запрос он это делает, иначе запрос переадресовывается отправителю

Client (ChainApp)

- Инициирует запрос к объекту ConcreteHandler, находящемуся в цепочке

Пример кода (показывает работу паттерна в цепи, содержащей

несколько объектов, у объектов есть возможность отвечать на запросы либо же перенаправлять следующему в цепи):

```
using System;
namespace Chain
{
    class MainApp
    {

        static void Main()
        {

            // Настраиваем цепочку

            Handler h1 = new ConcreteHandler1();

            Handler h2 = new ConcreteHandler2();

            Handler h3 = new ConcreteHandler3();

            h1.SetSuccessor(h2);

            h2.SetSuccessor(h3);



            // Генерируем запросы

            int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20
};

            foreach (int request in requests)
            {
                h1.HandleRequest(request);
            }

            // Ждем нажатия пользователя

            Console.ReadLine();
```

```
        }

    }


    //  'Handler' абстрактный класс

    abstract class Handler
    {

        protected Handler successor;
        public void SetSuccessor(Handler successor)
        {

            this.successor = successor;
        }
        public abstract void HandleRequest(int request);

    }

    //  'ConcreteHandler1' класс

    class ConcreteHandler1 : Handler
    {

        public override void HandleRequest(int request)
        {

            if (request >= 0 && request < 10)
            {
                Console.WriteLine("{0} обработан запрос
{1}",this.GetType().Name, request);
            }

            else if (successor != null)
            {
                successor.HandleRequest(request);
            }

        }
    }
```

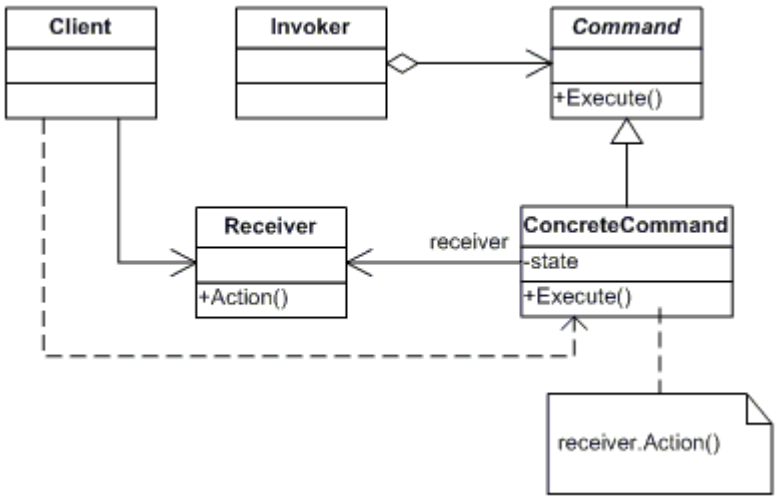
```
    // 'ConcreteHandler2' класс

    class ConcreteHandler2 : Handler
    {

        public override void HandleRequest(int request)
        {

            if (request >= 10 && request < 20)
            {
                Console.WriteLine("{0} обработан запрос
{1}", this.GetType().Name, request);
            }
            else if (successor != null)
            {
                successor.HandleRequest(request);
            }

        }

    }

    // 'ConcreteHandler3' класс

    class ConcreteHandler3 : Handler
    {

        public override void HandleRequest(int request)
        {

            if (request >= 20 && request < 30)
            {
                Console.WriteLine("{0} обработан запрос
{1}", this.GetType().Name, request);
            }
            else if (successor != null)
            {
                successor.HandleRequest(request);
            }
        }
```

```
        }

}
```

3. Паттерн Command

Паттерн Command инкапсулирует команды в некотором объекте. Инкапсулирование, таким образом, позволяет выполнять различные манипуляции, например такие как: управление выбором и последовательностью исполнения команд, возможность постановки команд в очередь, отмена команд и т.д.

Рассмотрим UML диаграмму для данного паттерна



В данной диаграмме следующие участники:

Command

- Определяет интерфейс для исполнения операции

ConcreteCommand

- Определяет связывание между объектом-получателем (Receiver) и действием
- Реализует исполнение путем вызова соответствующих операций Receiver

Client

- Создает объект **ConcreteCommand** и устанавливает его получателя

Invoker

- Запрашивает команду выполнить некоторый запрос

Receiver

- Знает, как выполнить операции связанные с обработкой запроса

Пример кода (демонстрирует работу паттерна Command)

```
using System;
namespace Command
{
    class MainApp
    {
        static void Main()
        {

            // Создаем объекты receiver, command, and
invoker

            Receiver receiver = new Receiver();

            Command command = new
ConcreteCommand(receiver);

            Invoker invoker = new Invoker();

            // Устанавливаем и запускаем команду

            invoker.SetCommand(command);

            invoker.ExecuteCommand();
```

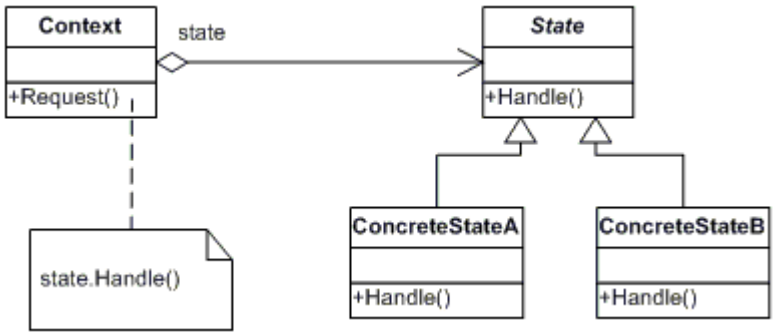
```
            Console.ReadLine();

        }

    }

    //'Command' абстрактный класс

    abstract class Command
    {
        protected Receiver receiver;
        public Command(Receiver receiver)
        {
            this.receiver = receiver;
        }
        public abstract void Execute();
    }

    // 'ConcreteCommand' класс
    class ConcreteCommand : Command
    {
        public ConcreteCommand(Receiver receiver) :

            base(receiver)
        {}
        public override void Execute()
        {
            receiver.Action();
        }

    }

    // The 'Receiver' класс

    class Receiver
    {

        public void Action()
        {

            Console.WriteLine("Вызван метод
```

```
Receiver.Action()");

        }

    }
    // 'Invoker' класс
    class Invoker
    {
        private Command _command;
        public void SetCommand(Command command)
        {
            this._command = command;
        }
        public void ExecuteCommand()
        {
            _command.Execute();
        }
    }

}
```

4. Паттерн State

Паттерн State заключает состояния объекта в отдельные объекты, каждый из которых расширяет общий суперкласс.

Рассмотрим UML диаграмму для данного паттерна



В данной диаграмме следующие участники:

Context

- Определяет интерфейс для клиентов
- Поддерживает объект наследника ConcreteState, определяющего текущее состояние

State

- Определяет интерфейс для инкапсуляции поведения, связанного с состоянием Context

ConcreteState

- Каждый наследник реализует поведение, связанное с состоянием Context

Пример кода (демонстрирует работу паттерна, в зависимости от внутреннего состояния объект будет себя вести по-разному)

```
using System;

namespace State
{

    class MainApp
    {

        static void Main()
        {

            // Создаём контекст в определенном состоянии

            Context c = new Context(new ConcreteStateA());



            // Вызываем операции, которые переключают
состояние
```

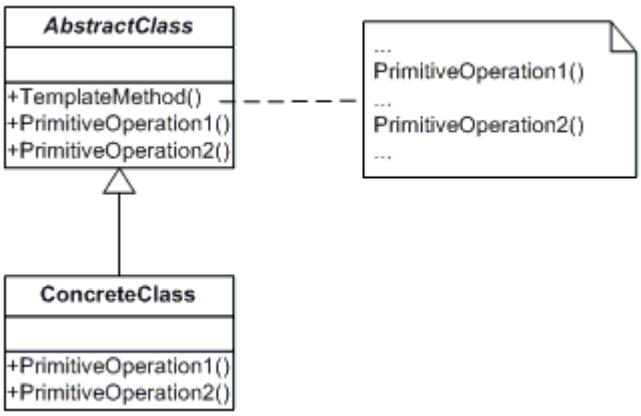
```
            c.Request();

            c.Request();

            c.Request();

            c.Request();

            Console.ReadLine();

        }

    }


    // 'State' абстрактный класс

    abstract class State
    {

        public abstract void Handle(Context context);

    }

    //  'ConcreteStateA' класс

    class ConcreteStateA : State
    {

        public override void Handle(Context context)
        {

            context.State = new ConcreteStateB();

        }

    }

    // 'ConcreteStateB' class

    class ConcreteStateB : State
```

```
    {

        public override void Handle(Context context)
        {

            context.State = new ConcreteStateA();

        }

    }

    // 'Context' класс

    class Context
    {

        private State _state;

        public Context(State state)
        {
            this.State = state;
        }
        public State State
        {

            get { return _state; }
            set
            {
                _state = value;
                Console.WriteLine("Состояние: "
+_state.GetType().Name);
            }
        }
        public void Request()
        {
            _state.Handle(this);
        }
    }
}
```

5. Паттерн Template Method

Паттерн Template Method строится на абстрактном классе, содержащем часть логики, требуемой для исполнения задачи. Оставшаяся часть логики содержится в методах классов-потомков, которые создают свою реализацию абстрактных методов.

Рассмотрим UML диаграмму для данного паттерна:



В данной диаграмме следующие участники:

AbstractClass

- Определяет абстрактные примитивные операции, которые будут определены потомками для реализации шагов алгоритма
- Реализует шаблонный метод, определяя скелет алгоритма. Шаблонный метод вызывает примитивные операции, определенные в AbstractClass или в других объектах

ConcreteClass

- Реализует примитивные операции, необходимые классам-потомкам для реализации алгоритмов

Пример кода (показывает пример работы паттерна):

```
using System;
namespace Template
{

    class MainApp
    {

        static void Main()
        {

            AbstractClass aA = new ConcreteClassA();

            aA.TemplateMethod();

            AbstractClass aB = new ConcreteClassB();

            aB.TemplateMethod();

            Console.ReadLine();

        }

    }

    // 'AbstractClass' абстрактный класс

    abstract class AbstractClass
    {

        public abstract void PrimitiveOperation1();

        public abstract void PrimitiveOperation2();



        //  "Template method"

        public void TemplateMethod()
        {

            PrimitiveOperation1();
```

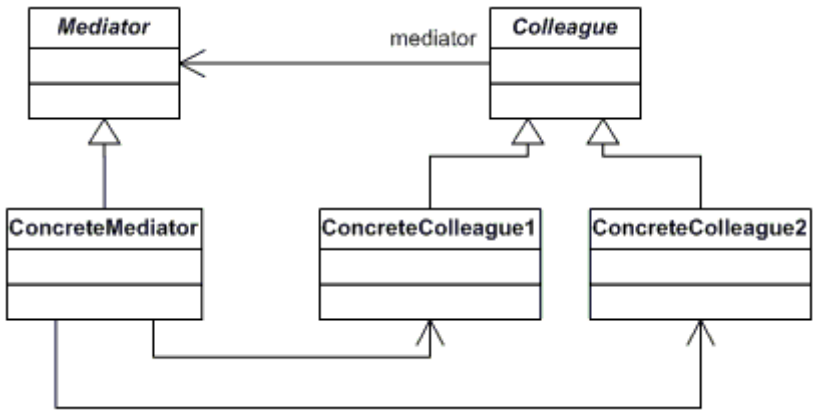
```
            PrimitiveOperation2();

            Console.WriteLine("");

        }

    }

    // 'ConcreteClassA' класс

    class ConcreteClassA : AbstractClass
    {

        public override void PrimitiveOperation1()
        {

Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");

        }

        public override void PrimitiveOperation2()
        {

Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");

        }

    }


    // 'ConcreteClassB' класс

    class ConcreteClassB : AbstractClass
    {

        public override void PrimitiveOperation1()
        {
```

```
Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");

        }

        public override void PrimitiveOperation2()
        {

Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");

        }

    }

}
```

6.Паттерн Mediator

Паттерн Mediator используется для согласования изменений состояний набора объектов с помощью одного объекта. То есть вместо раскидывания логики поведения по разным классам данный паттерн инкапсулирует логику управления изменением состояний в рамки одного класса.

Рассмотрим UML диаграмму для данного паттерна:



В данной диаграмме следующие участники:

Mediator

- Определяет интерфейс для общения с объектами Colleague

ConcreteMediator

- Реализует совместное поведение путем координирования объектов Colleague
- Знает и поддерживает своих Colleague

Colleague классы

- Каждый Colleague класс знает своего Mediator
- Каждый Colleague общается со своим медиатором

Пример кода (показывает принцип работы Mediator)

```
using System;
namespace Mediator
{

    class MainApp
    {

        static void Main()
        {

            ConcreteMediator m = new ConcreteMediator();

            ConcreteColleague1 c1 = new
ConcreteColleague1(m);

            ConcreteColleague2 c2 = new
ConcreteColleague2(m);



            m.Colleague1 = c1;

            m.Colleague2 = c2;
```

```
            c1.Send("Как дела?");

            c2.Send("Хорошо спасибо");

            Console.ReadLine();

        }

    }

    // 'Mediator' абстрактный класс

    abstract class Mediator
    {

        public abstract void Send(string message,

          Colleague colleague);

    }

    // 'ConcreteMediator' класс
    class ConcreteMediator : Mediator
    {
        private ConcreteColleague1 _colleague1;

        private ConcreteColleague2 _colleague2;

        public ConcreteColleague1 Colleague1
        {

            set { _colleague1 = value; }
        }
        public ConcreteColleague2 Colleague2
        {
            set { _colleague2 = value; }
        }
        public override void Send(string message,Colleague
```

```
colleague)
        {

            if (colleague == _colleague1)
            {

                _colleague2.Notify(message);

            }
            else
            {
                _colleague1.Notify(message);
            }

        }

    }

    // 'Colleague' абстрактный класс
    abstract class Colleague
    {

        protected Mediator mediator;

        // Конструктор

        public Colleague(Mediator mediator)
        {

            this.mediator = mediator;

        }

    }

    // 'ConcreteColleague1' класс

    class ConcreteColleague1 : Colleague
    {

        // Конструктор
```

```
        public ConcreteColleague1(Mediator mediator)

            : base(mediator)
        {

        }
        public void Send(string message)
        {

            mediator.Send(message, this);

        }
        public void Notify(string message)
        {

            Console.WriteLine("Colleague1 получил
сообщение: "+ message);

        }

    }


    // 'ConcreteColleague2' класс
    class ConcreteColleague2 : Colleague
    {

        // Конструктор
        public ConcreteColleague2(Mediator mediator)

            : base(mediator)
        {

        }
        public void Send(string message)
        {
            mediator.Send(message, this);
        }
        public void Notify(string message)
        {
```

```
            Console.WriteLine("Colleague2 получил
сообщение: " + message);
        }

    }

}
```

В данном уроке мы привели часть паттернов поведения. Оставшиеся паттерны вам предназначены для самостоятельной проработки.

7.Экзаменационное задание

Реализуйте с использованием паттернов проектирования простейший графический редактор. Должны поддерживаться операции с геометрическими объектами (вставка, вырезание, копирование и т.д.), работа с изображениями (загрузка, выделение части изображения и т.д.), другие операции.