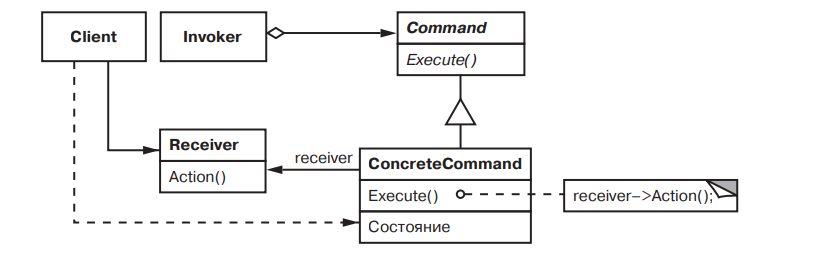
**Команда – паттерн поведения объектов**

Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Назначение – инкапсулирует запрос в объекте, позволяя тем самым параметризовать клиенты для разных запросов, ставить запросы в очередь или проконтролировать их, а также поддерживать отмену операций.

**Структура**



**Участники**

* Command – команда:
  + Объявляет интерфейс для выполнения операции;
* ConcreteCommand (PasteCommand, OpenCommand) – конкретная команда:
  + Определяет связь между объектом-получателем Receiver и действием;
  + Реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
* Client (Application) – клиент:
  + Создает объект класса ConcreteCommand и устанавливает получателя;
* Invoker (MenuItem) – инициатор:
  + Обращается к команде для выполнения запроса;
* Receiver (Document, Application) – получатель:
  + Располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получается может выступать любой класс.

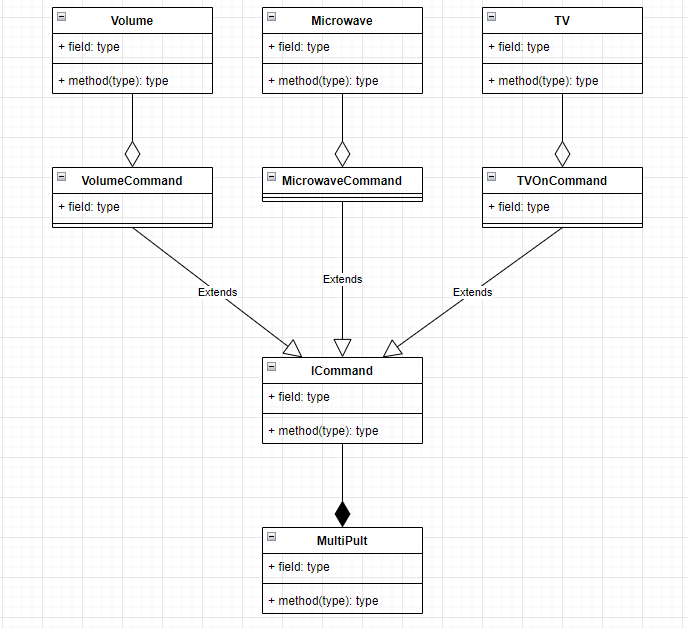
**Отношения**

* Клиент создает объект ConcreteCommand и устанавливает для него получателя;
* Инициатор Invoker сохраняет объект ConcreteCommand;
* Инициатор отправляет запрос, вызывая операцию команды Execute. Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отмены;
* Объект ConcreteCommand вызывает операции получателя для выполнения запроса.

**Результаты**

Основные результаты применения паттерна команда:

* команда отделяет объект, инициирующий операцию, от объекта, располагающего информацией о том, как ее выполнить;
* команды — это самые настоящие объекты. Их можно обрабатывать и расширять точно так же, как любые другие объекты;
* из простых команд можно собирать составные, например, класс MacroCommand, рассмотренный выше. В общем случае составные команды описываются паттерном компоновщик;
* новые команды добавляются легко, поскольку никакие существующие классы изменять не нужно.



**Участники**

* ICommand – команда:
  + Объявляет интерфейс для выполнения операции;
* ConcreteCommand (VolumeCommand, MicrowaveCommand) – конкретная команда:
  + Определяет связь между объектом-получателем Receiver и действием;
  + Реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
* Invoker (MultiPutle) – инициатор:
  + Обращается к команде для выполнения запроса;
* Receiver (Volume, Microwave, TV) – получатель:
  + Располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

/// Интерфейс команды

public interface ICommand

{

void Execute();

void Undo();

}

/// Получатель команд

class TV

{

public void On() => Console.WriteLine("Телевизор включен");

public void Off() => Console.WriteLine("Телевизор выключен");

}

/// Реализация интерфейса ICommand

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 {level}");

}

public void DropLevel()

{

if (level > OFF)

level--;

Console.WriteLine($"Уровень звука {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 = new ICommand[3];

Stack<ICommand> commandsHistory;

public MultiPult()

{

// buttons = new ICommand[3];

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();

}

}

}

class Microwave

{

public void StartCooking(int time)

{

Console.WriteLine("Подогреваем еду");

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();

}

}

class Program

{

static void Main(string[] args)

{

MultiPult mPult = new MultiPult();

TV tv = new TV();

Volume volume = new Volume();

Microwave microwave = new Microwave();

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

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

mPult.SetCommand(2, new MicrowaveCommand(microwave, 2000));

mPult.PressButton(0);

mPult.PressButton(1);

mPult.PressButton(1);

mPult.PressButton(1);

//mPult.PressButton(2);

mPult.PressUndoButton();

//mPult.PressUndoButton(1);

//mPult.PressUndoButton();

mPult.PressButton(1);

mPult.PressButton(1);

mPult.PressButton(1);

}

}