Command Pattern

The command pattern is a behavioral object design pattern. In the command pattern, a Command interface declares a method for executing a particular action. Concrete Command classes implement the execute() method of the Command interface, and this execute() method invokes the appropriate action method of a Receiver class that the Concrete Command class contains. The Receiver class performs a particular action. A Client class is responsible for creating a Concrete Command and setting the Receiver of the Concrete Command. An Invoker class contains a reference to a Command and has a method to execute the Command.

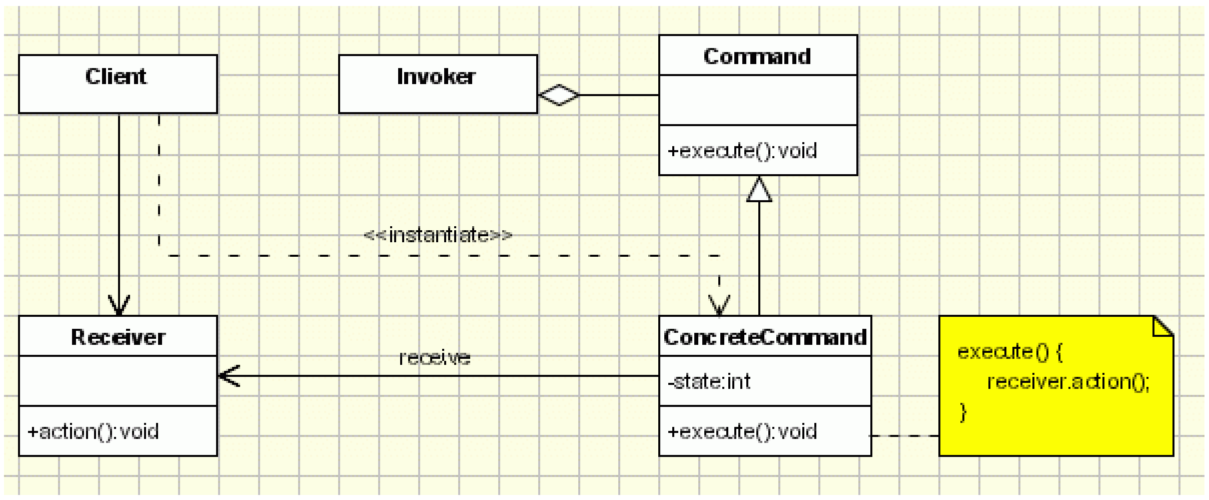
In the command pattern, the invoker is decoupled from the action performed by the receiver. The invoker has no knowledge of the receiver. The invoker invokes a command, and the command executes the appropriate action of the receiver. Thus, the invoker can invoke commands without knowing the details of the action to be performed. In addition, this decoupling means that changes to the receiver's action don't directly affect the invocation of the action.

A command is an object whose role is to **store all the information required for executing an action**, including the method to call, the method arguments, and the object (known as the receiver) that implements the method.

A receiver is an object that **performs a set of cohesive actions**. It’s the component that performs the actual action when the command’s *execute()* method is called.

An **invoker** is an object that knows how to execute a given command but doesn’t know how the command has been implemented. It only knows the command’s interface.

A client is an object that **controls the command execution process** by specifying what commands to execute and at what stages of the process to execute them.



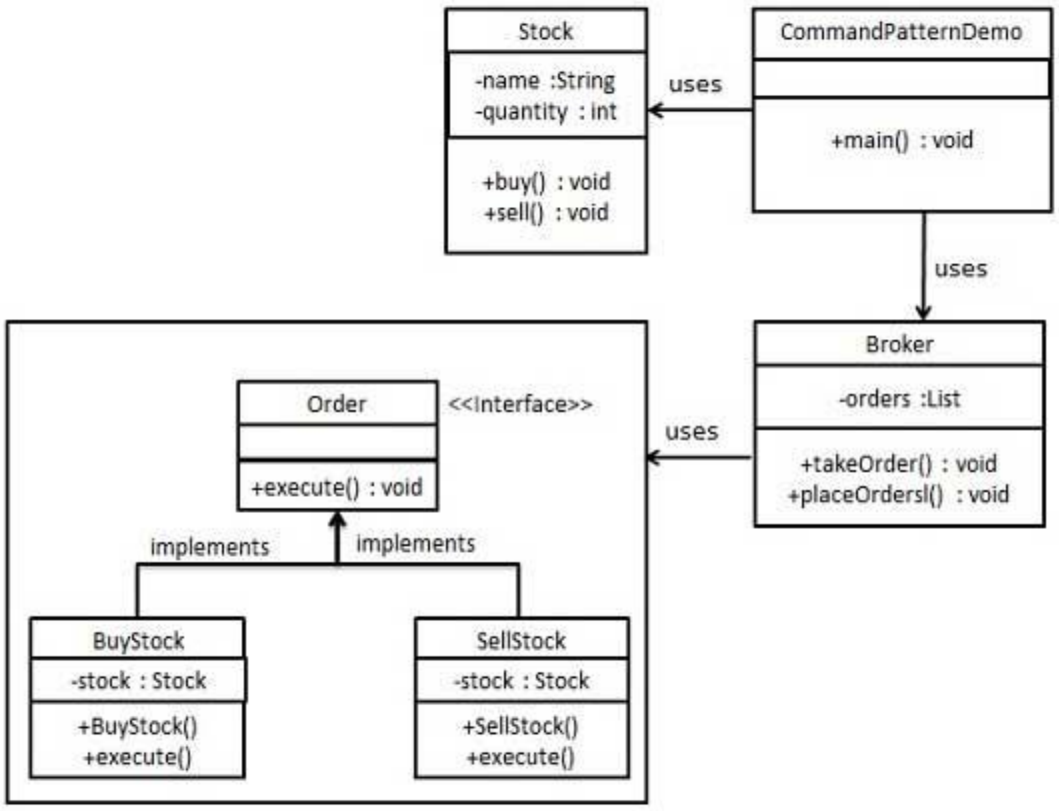
Implementation of Command Design Pattern

1. Define a Command interface having an execute method execute().
2. All command objects must implements a Command interface. The execute method delegates the request to a receiver to execute the command.
3. A receiver class holds the logic of performing any specific task requested as command. It is called from execute method of command object.
4. The client creates a set of command objects and associates receiver with it. Client passes commands to invoker to store it. Later, client calls invoker to execute the commands.

Here is a sample code of a classic implementation of this pattern for placing orders for buying and selling stocks:

We have created an interface Order which is acting as a command. We have created a Stock class which acts as a request. We have concrete command classes BuyStock and SellStock implementing Order interface which will do actual command processing. A class Broker is created which acts as an invoker object. It can take and place orders.

Broker object uses command pattern to identify which object will execute which command based on the type of command. CommandPatternDemo, our demo class, will use Broker class to demonstrate command pattern.



Step 1

Create a command interface.

Order.java

```
public interface Order {
   void execute();
}
```

Step 2

Create a request class or Receiver

Stock.java

```
public class Stock {
   private String name = "ABC";
   private int quantity = 10;
   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

Step 3

Create concrete classes implementing the *Order* interface.

BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;
   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
   private Stock abcStock;
   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.sell();
   }
}
```

Step 4

Create command invoker class.

Broker.java

```
import java.util.ArrayList;

import java.util.List;


   public class Broker {

   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){

      orderList.add(order);

   }

   public void placeOrders(){

      for (Order order : orderList) {

         order.execute();

      }

      orderList.clear();

   }

}
```

Step 5

Client class. Use the Broker class to take and execute commands.

CommandPatternDemo.java

```
public class CommandPatternDemo {

   public static void main(String[] args) {

      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);

      SellStock sellStockOrder = new SellStock(abcStock);
      Broker broker = new Broker();

      broker.takeOrder(buyStockOrder);

      broker.takeOrder(sellStockOrder);

      broker.placeOrders();

   }

}
```

Step 6

Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

Let's use a remote control as the example. Our remote is the center of home automation and can control everything. We'll just use a light as an example, that we can switch on or off, but we could add many more commands.

```
// A simple Java program to demonstrate
// implementation of Command Pattern using
// a remote control example.

// An interface for command
interface Command
{
	public void execute();
}

// Light class (Receiver class)
class Light
{
	public void on()
	{
		System.out.println("Light is on");
	}
	public void off()
	{
```

```
                System.out.println("Light is off");
        }
}
```
// and its corresponding command classes
class LightOnCommand implements Command
```
{
        Light light;

        // The constructor is passed the light it
        // is going to control.
        public LightOnCommand(Light light)
        {
        this.light = light;
        }


       public void execute()
        {
        light.on();
        }
}
```
class LightOffCommand implements Command
```
{
        Light light;
        public LightOffCommand(Light light)
        {
                this.light = light;
        }
        public void execute()
        {
                light.off();
        }
}
```

// Stereo (Receiver class)

class Stereo
```
{
        public void on()
```

```
	{
		System.out.println("Stereo is on");
	}
	public void off()
	{
		System.out.println("Stereo is off");
	}
	public void setCD()
	{
		System.out.println("Stereo is set " +
						"for CD input");
	}
	public void setDVD()
	{
		System.out.println("Stereo is set"+
						" for DVD input");
	}
	public void setRadio()
	{
		System.out.println("Stereo is set" +
						" for Radio");
	}
	public void setVolume(int volume)
	{
	// code to set the volume
	System.out.println("Stereo volume set"
						+ " to " + volume);
	}
}
class StereoOffCommand implements Command
{
	Stereo stereo;
	public StereoOffCommand(Stereo stereo)
	{
		this.stereo = stereo;
	}
	public void execute()
	{
        stereo.off();
        }
}
// and its corresponding command classes
class StereoOnWithCDCommand implements Command
{
        Stereo stereo;
        public StereoOnWithCDCommand(Stereo stereo)
        {
                this.stereo = stereo;
        }
        public void execute()
        {
                stereo.on();
                stereo.setCD();
                stereo.setVolume(11);
        }
}

// Invoker - A Simple remote control with one button
class SimpleRemoteControl
{
        Command button; // only one button

        public SimpleRemoteControl()
        {
        }

        public void setCommand(Command command)
        {
                // set the command the remote will
                // execute
                button = command;
        }

        public void buttonWasPressed()
        {
                button.execute();
        }
}
```

// Driver class or client class

```
class RemoteControlTest
{
        public static void main(String[] args)
        {
                SimpleRemoteControl remote =
                                new SimpleRemoteControl();
                Light light = new Light();
                Stereo stereo = new Stereo();

                // we can change command dynamically
                remote.setCommand(new
                                        LightOnCommand(light));
                remote.buttonWasPressed();
                remote.setCommand(new
                                StereoOnWithCDCommand(stereo));
                remote.buttonWasPressed();
                remote.setCommand(new
                                StereoOffCommand(stereo));
                remote.buttonWasPressed();
        }
}
```

Watch Out for the Downsides

This pattern ends up forcing a lot of Command classes that will make your design look cluttered - more operations being made possible leads to more command classes. Intelligence required of which Command to use and when leads to possible maintainence issues for the central controller.

Another example of command pattern

Link to another command pattern example

https://www.avajava.com/tutorials/lessons/command-pattern.html?page=2