# Command Pattern

**Command pattern** is a behavioral design pattern which is useful to **abstract business logic into discrete actions** which we call *commands*. This command object helps in loose coupling between two classes where one class (invoker) shall call a method on other class (receiver) to perform a business operation.

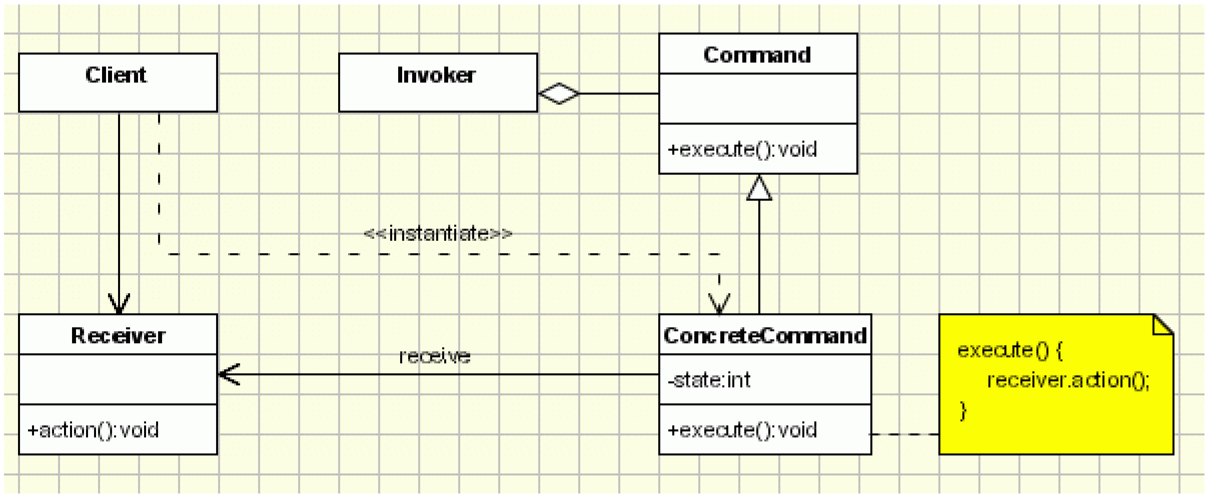
## Design Participants

Participants for command design pattern are:

- **Command interface** – for declaring an operation.
- **Concrete command classes** – which extends the `Command` interface, and has execute method for invoking business operation methods on receiver. It internally has reference of the receiver of command.
- **Invoker** – which is given the command object to carry out the operation.
- **Receiver** – which execute the operation.

In 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.

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.



## Problem Statement

Suppose we need to build a remote control for home automation system which shall control different lights/electrical units of the home. A single button in remote may be able to perform same operation on similar devices e.g. a TV ON/OFF button can be used to turn ON/OFF different TV set in different rooms.

Here this remote will be a programmable remote and it would be used to turn on and off various lights/fan etc.

First of all, let’s see how the problem can be solved with any design approach. Her the code of the remote control may look like –

```
If(buttonName.equals(“Light”))

{

    //Logic to turn on that light

}

else If(buttonName.equals(“Fan”))

{

    //Logic to turn on that Fan

}
```

But above solution apparently has many visible issues like –

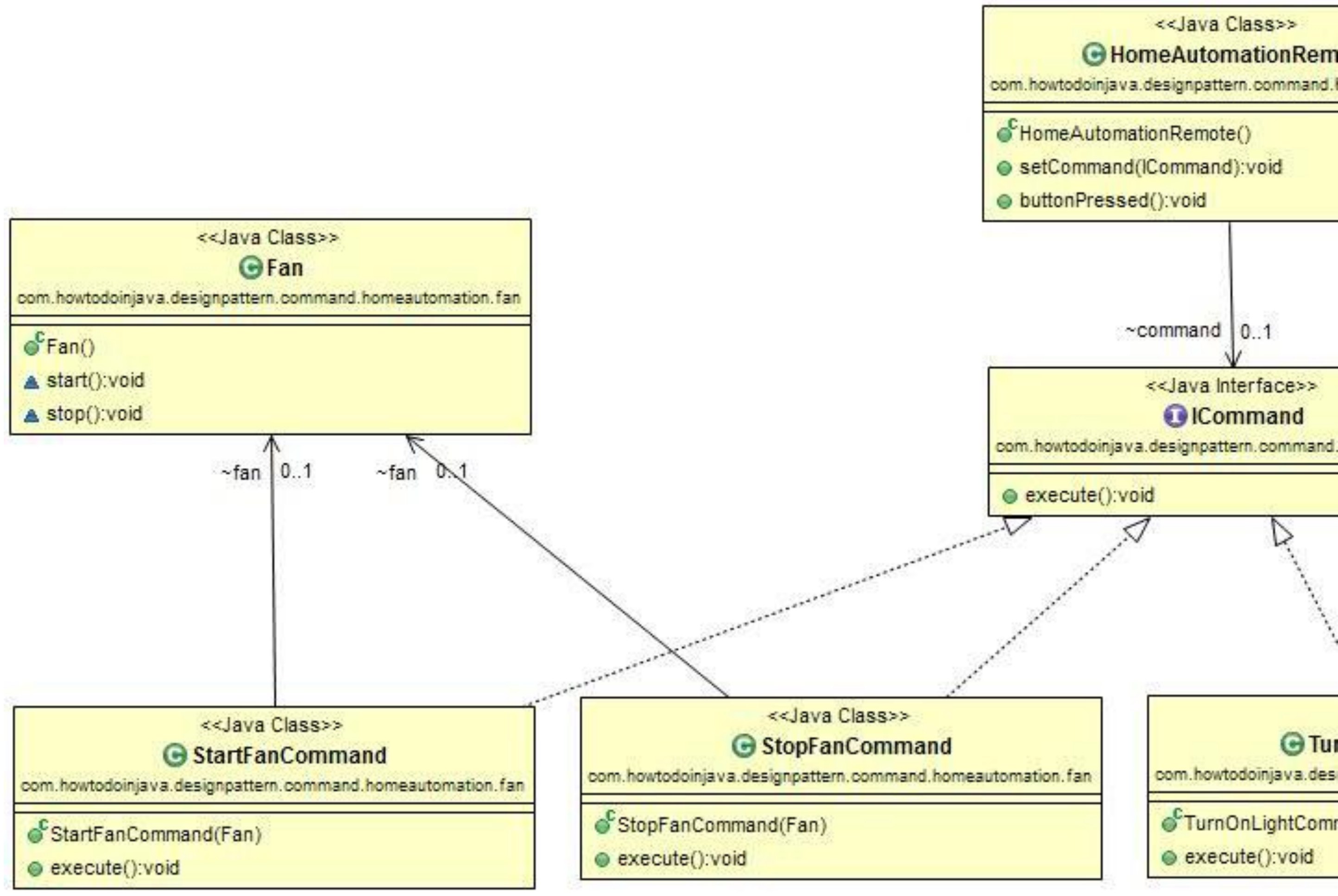
- Any new item (e.g. TubeLight) will require change in the code of the remote control. You will need to add more if-elses.
- If we want to change the button for any other purpose, then we need to change the code as well.
- On top of that, complexity and maintainability of the code will increase in case there are lots of items in the home.
- Finally, code is not clean and is tightly coupled and we are not following best practices like *coding to interfaces* etc.

## Command Pattern Implementation

Let's solve above home automation problem with command design pattern and design each component one at a time.

- `ICommand` interface which is the **command interface**
- `Light` is one of a **receiver** component. It can accept multiple commands related to Light like turn on and off
- `Fan` is also another type of a **receiver** component. It can accept multiple commands related to Fan like turn on and off
- `HomeAutomationRemote` is the **invoker** object, which asks the command to carry out the request. Here Fan on/off, Light on/off.
- `StartFanCommand`,`StopFanCommand`,`TurnOffLightCommand`,`TurnOnLightCommand` etc. are different type of **command implementations**.

## *Class Diagram*



Lets see the java source of each class and interface.

**ICommand.java**
```
/**
 * Command Interface which will be implemented by the exact commands.
 *
 */
@FunctionalInterface
public interface ICommand {
    public void execute();
}
```

**Light.java**

```
package com.cecs277.command.homeautomation.light;

/**
```

```
 * Light is a Receiver component in command pattern terminology.
 *
 */
public class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}
```

`Fan.java`

```
package com.cecs277.command.homeautomation.fan;

/**
 * Fan class is a Receiver component in command pattern terminology.
 *
 */
public class Fan {
    void start() {
        System.out.println("Fan Started..");

    }

     void stop() {
        System.out.println("Fan stopped..");

    }
}
```

`TurnOffLightCommand.java`

```
package com.cecs277.command.homeautomation.light;

import com.cecs277.command.homeautomation.ICommand;
/**
 * Light Start Command where we are encapsulating both Object[light] and
*the operation[turnOn] together as command. This is the essence of the
*command.

 **/
public class TurnOffLightCommand implements ICommand {

    Light light;
    public TurnOffLightCommand(Light light) {
        super();
        this.light = light;
    }

    public void execute() {
        System.out.println("Turning off light.");
        light.turnOff();
    }
}
```

TurnOnLightCommand.java

```
package com.cecs277.command.homeautomation.light;

import com.cecs277.command.homeautomation.ICommand;

/**
 * Light stop Command where we are encapsulating both Object[light] and
*the operation[turnOff] together as command. This is the essence of the
command.
 **/
public class TurnOnLightCommand implements ICommand {

    Light light;

    public TurnOnLightCommand(Light light) {
        super();
        this.light = light;
    }

    public void execute() {
        System.out.println("Turning on light.");
        light.turnOn();
    }
}
```

StartFanCommand.java

```
package com.cecs277.command.homeautomation.fan;

import com.cecs277.command.homeautomation.ICommand;

/**
 * Fan Start Command where we are encapsulating both Object[fan] and the
 * operation[start] together as command. This is the essence of the
command.
 *
 */
public class StartFanCommand implements ICommand {

    Fan fan;

    public StartFanCommand(Fan fan) {
        super();
        this.fan = fan;
    }

    public void execute() {
        System.out.println("starting Fan.");
        fan.start();
    }
}
```

`StopFanCommand.java`

```
package com.cecs277.command.homeautomation.fan;

import com.cecs277.command.homeautomation.ICommand;
/**
 * Fan stop Command where we are encapsulating both Object[fan] and the
 * operation[stop] together as command. This is the essence of the command.
 *
 */
public class StopFanCommand implements ICommand {

    Fan fan;

    public StopFanCommand(Fan fan) {
        super();
        this.fan = fan;
    }

    public void execute() {
        System.out.println("stopping Fan.");
        fan.stop();
    }
}
```

`HomeAutomationRemote.java`

```
package com.cecs277.command.homeautomation;

/**
 * Remote Control for Home automation where it will accept the command *and
execute. This is the invoker in terms of command pattern *terminology
```

```
 **/
public class HomeAutomationRemote {

    //Command Holder
    ICommand command;

    //Set the command in runtime to trigger.
    public void setCommand(ICommand command) {
        this.command = command;
    }

    //Will call the command interface method so that particular command
    //can be invoked.
    public void buttonPressed() {
        command.execute();
    }
}
```

## Demo

Lets code and excute the client code to see how commands are executed.

```
package com.cec277.designpattern.command.homeautomation;

import com.cecs277.command.homeautomation.fan.Fan;
import com.cecs277.command.homeautomation.fan.StartFanCommand;
import com.cecs277.command.homeautomation.fan.StopFanCommand;
import com.cecs277.command.homeautomation.light.Light;
import com.cecs277.command.homeautomation.light.TurnOnLightCommand;

/**
 * Demo class for HomeAutomation
 *
 */
public class Demo   //client
{
    public static void main(String[] args)
    {
        Light livingRoomLight = new Light();    //receiver 1

        Fan livingRoomFan = new Fan();  //receiver 2

        Light bedRoomLight = new Light();   //receiver 3

        Fan bedRoomFan = new Fan();     //receiver 4
        HomeAutomationRemote remote = new
              homeAutomationRemote();   //Invoker

        remote.setCommand(new TurnOnLightCommand( livingRoomLight ));
        remote.buttonPressed();

        remote.setCommand(new TurnOnLightCommand( bedRoomLight ));
        remote.buttonPressed();

        remote.setCommand(new StartFanCommand( livingRoomFan ));
        remote.buttonPressed();

        remote.setCommand(new StopFanCommand( livingRoomFan ));
        remote.buttonPressed();

        remote.setCommand(new StartFanCommand( bedRoomFan ));
        remote.buttonPressed();

        remote.setCommand(new StopFanCommand( bedRoomFan ));
        remote.buttonPressed();
    }
}
```

Output:

```
Turning on light.
Light is on

Turning on light.
Light is on

starting Fan.
Fan Started..

stopping Fan.
Fan stopped..

starting Fan.
Fan Started..

stopping Fan.
Fan stopped..
```

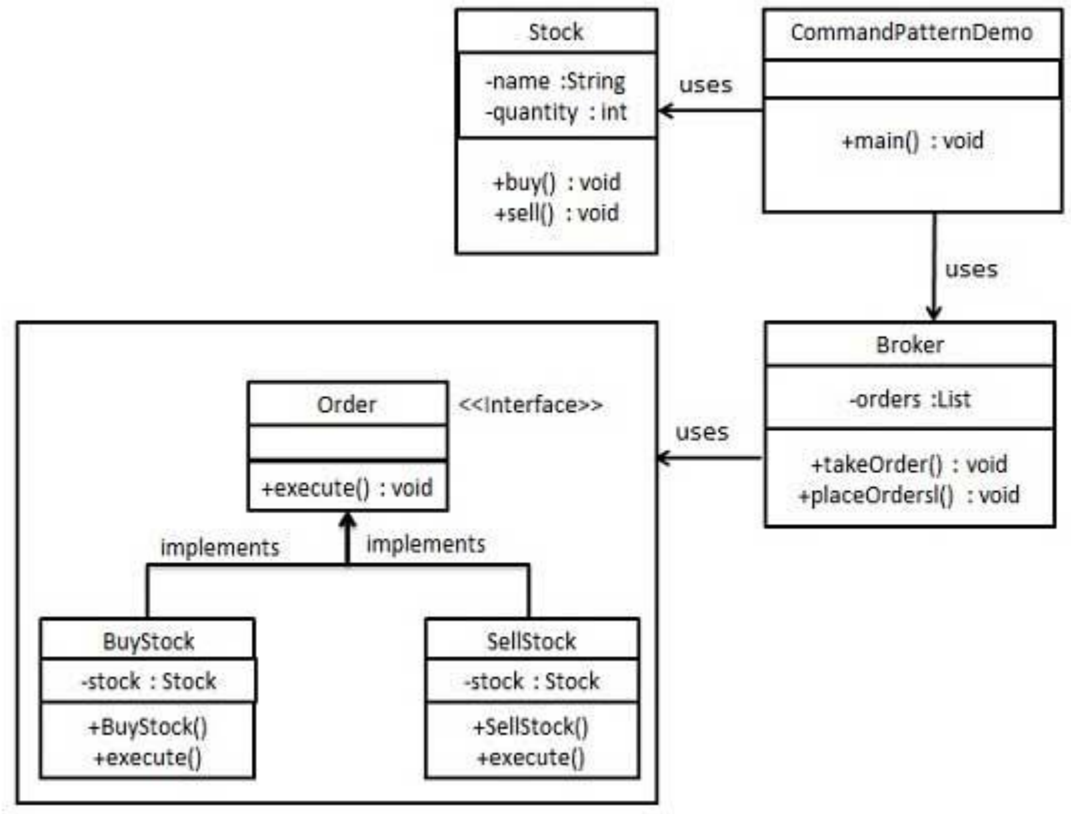
## Implementation of Command Design Pattern

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 (Receiver). 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 maintenance issues for the central controller.