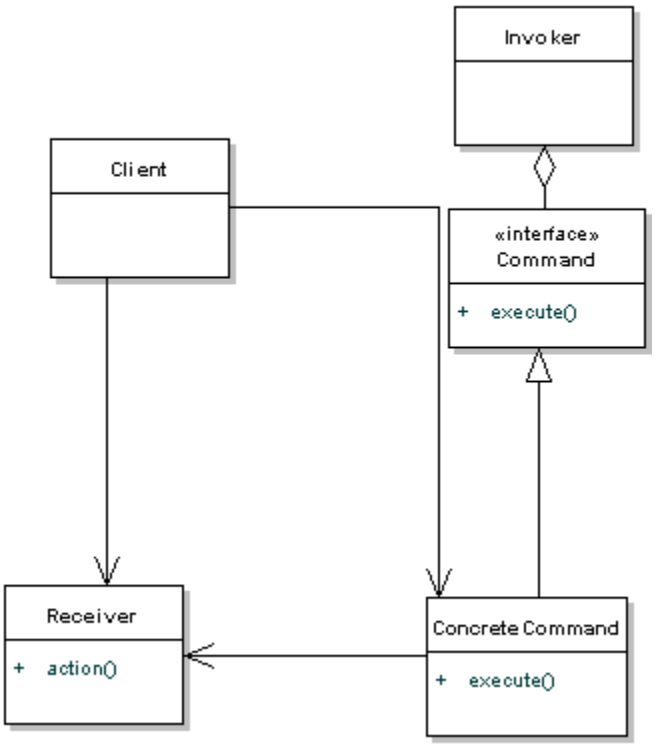
Command Pattern

The **Command Design Pattern** wraps a request in an object as command and pass it to a command invoker object. When invoker receives a request, it looks for the appropriate object which is capable of handling this request and passes the command to the corresponding object for execution. This allows clients to be parameterized with different requests.

In **command pattern**, a client just gives instructions to invoke without knowing how it is going to be executed. It supports decoupling between a client object which request a task and the receiver object which actually performs the task. This pattern is not focused on the sequence of the tasks stored, but on hiding the details/implementation of the actions performed.



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.

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.

1. First we'll create our command interface.

```
//Command
public interface Command{
  public void execute();
}
```

2. Now let's create two concrete commands. One will turn on the lights, another turns off lights.

```
//Concrete Command
public class LightOnCommand implements Command{
  //reference to the light
  Light light;
  public LightOnCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOn();
  }
}

//Concrete Command
public class LightOffCommand implements Command{
  //reference to the light
  Light light;
  public LightOffCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOff();
  }
}
```

3. Light is our receiver class, so let's set that up now.

```
//Receiver
public class Light{
  private boolean on;
  public void switchOn(){
    on = true;
  }
  public void switchOff(){
    on = false;
  }
}
```

4. Our invoker in this case is the remote control.

```
//Invoker
public class RemoteControl{
  private Command command;
  public void setCommand(Command command){
    this.command = command;
  }
  public void pressButton(){
    command.execute();
  }
}
```

5. Finally we'll set up a client to use the invoker

```
//Client
public class Client{
  public static void main(String[] args)    {
    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightsOnCommand(light);
    Command lightsOff = new LightsOffCommand(light);
    //switch on
    control.setCommand(lightsOn);
    control.pressButton();
    //switch off
    control.setCommand(lightsOff);
```

```
    control.pressButton();
  }
}
```

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:

Command.java

```
1    public interface Command {
2       public void execute();
3    }
```

DrawCircle and DrawRectangle are concrete implementation of Command interface.

DrawCircle .java

```
1
2     public class DrawCircle implements Command {
3         ShapeDrafter drafter;
4
5         public DrawCircle(ShapeDrafter drafter){
6             this.drafter = drafter;
7         }
8
9         public void execute(){
10            drafter.drawCircle();
11        }
    }
```

DrawRectangle.java

```
1     public class DrawRectangle implements Command {
2         ShapeDrafter drafter;
3
4         public DrawRectangle(ShapeDrafter drafter){
5             this.drafter = drafter;
6         }
7
8         public void execute(){
9             drafter.drawRectangle();
          }
      }
```

ShapeDrafter is the receiver class which draws circle and rectangle.

ShapeDrafter.java

```
public class ShapeDrafter {
    public void drawRectangle(){
        System.out.println("Drawing a Rectangle on Screen");
    }

    public void drawCircle(){
        System.out.println("Drawing a Circle on Screen");
    }
}
```

CommandInvoker.java

```
import java.util.List;
import java.util.ArrayList;

public class CommandInvoker {
    private List<Command> commandList = new ArrayList<Command>();

    public void addCommand(Command c){
        commandList.add(c);
    }

    public void executeCommands(){
        for(Command c : commandList){
            c.execute();
        }
    }
}
```

CommandPatternExample class creates command objects and pass it to CommandInvoker for execution.

```
public class CommandPatternExample {
    public static void main(String args[]){
        ShapeDrafter drafter = new ShapeDrafter();

        Command rectangleCommand = new DrawRectangle(drafter);
        Command circleCommand = new DrawCircle(drafter);

        CommandInvoker invoker = new CommandInvoker();
        invoker.addCommand(circleCommand);
        invoker.addCommand(rectangleCommand);
```

```
9
10              invoker.executeCommands();
11          }
12      }
13
14
```

Output

```
Drawing a Circle on Screen
Drawing a Rectangle on Screen
```