

Command Pattern

 [geeksforgeeks.org/command-pattern/](https://www.geeksforgeeks.org/command-pattern/)

May 6, 2016

Like [previous](#) articles, let us take up a design problem to understand command pattern. Suppose you are building a home automation system. There is a programmable remote which can be used to turn on and off various items in your home like lights, stereo, AC etc. It looks something like this.



You can do it with simple if-else statements like

```
if (buttonPressed == button1)
    lights.on()
```

But we need to keep in mind that turning on some devices like stereo comprises of many steps like setting cd, volume etc. Also we can reassign a button to do something else. By using simple if-else we are coding to implementation rather than interface. Also there is tight coupling.

So what we want to achieve is a design that provides loose coupling and remote control should not have much information about a particular device. The command pattern helps us do that.

Definition: The **command pattern** encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

The definition is a bit confusing at first but let's step through it. In analogy to our problem above remote control is the client and stereo, lights etc. are the receivers. In command pattern there is a Command object that *encapsulates a request* by binding together a set of actions on a specific receiver. It does so by exposing just one method `execute()` that causes some actions to be invoked on the receiver.

Parameterizing other objects with different requests in our analogy means that the button used to turn on the lights can later be used to turn on stereo or maybe open the garage door.

queue or log requests, and support undoable operations means that Command's `Execute` operation can store state for reversing its effects in the Command itself. The Command may have an added `unExecute` operation that reverses the effects of a previous call to `execute`. It may also support logging changes so that they can be reapplied in case of a system crash.

Below is the Java implementation of above mentioned remote control example:

filter_none

edit

close

play_arrow

link

brightness_4

code

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

class Light
{
    public void on()
    {
        System.out.println( "Light is on" );
    }
    public void off()
    {
        System.out.println( "Light is off" );
    }
}
class LightOnCommand implements Command
{
    Light light;

    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();
    }
}

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)
{
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();
}
}
class StereoOnWithCDCommand implements Command
{
Stereo stereo;
public StereoOnWithCDCommand(Stereo stereo)
{
this .stereo = stereo;
}
public void execute()
{
stereo.on();
stereo.setCD();
stereo.setVolume( 11 );
}
}

class SimpleRemoteControl
{
Command slot;

```

```

public SimpleRemoteControl()
{
}

public void setCommand(Command command)
{
    slot = command;
}

public void buttonWasPressed()
{
    slot.execute();
}
}

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

        remote.setCommand( new
            LightOnCommand(light));
        remote.buttonWasPressed();
        remote.setCommand( new
            StereoOnWithCDCommand(stereo));
        remote.buttonWasPressed();
        remote.setCommand( new
            StereoOffCommand(stereo));
        remote.buttonWasPressed();
    }
}

```

chevron_right

filter_none

Output:

Light is on
 Stereo is on
 Stereo is set for CD input
 Stereo volume set to 11
 Stereo is off

Notice that the remote control doesn't know anything about turning on the stereo. That information is contained in a separate command object. This reduces the coupling between them.

Advantages:

- Makes our code extensible as we can add new commands without changing existing code.
- Reduces coupling the invoker and receiver of a command.

Disadvantages:

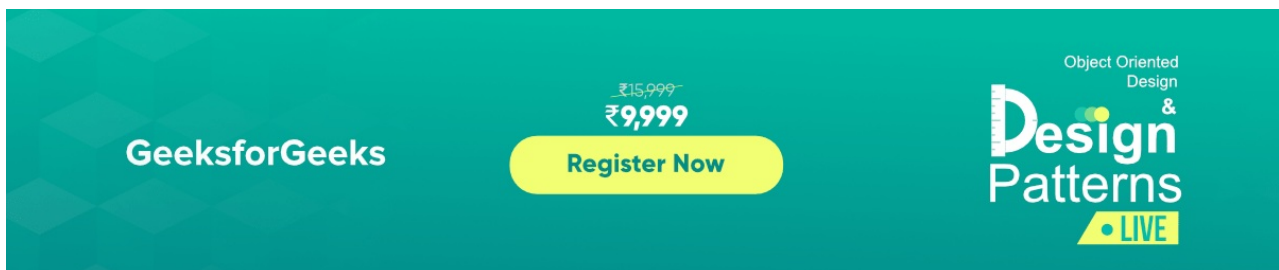
Increase in the number of classes for each individual command

References:

- Head First Design Patterns (book)
- <https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/command>

If This article is contributed by **Sulabh Kumar**. you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above



The banner features the GeeksforGeeks logo on the left. In the center, it displays a price tag of ₹15,999 with a crossed-out line and a yellow button labeled 'Register Now'. On the right, the text 'Object Oriented Design & Design Patterns' is shown, with 'Design Patterns' in a large font and a yellow 'LIVE' badge below it.

My Personal Notes *arrow_drop_up*

Save