Command Design Pattern

Last Modified: August 26, 2021

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.

Let's learn how command helps in decoupling the invoker from receiver.

Table of Contents

Introduction

In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action, a business operation or trigger an event e.g. method name, receiver object reference and method parameter values, if any. This object is called *command*.

The similar approach is adapted into chain of responsibility pattern as well. Only difference is that in command there is one request handler, and in chain of responsibility there can be many handlers for single request object.

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.

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.

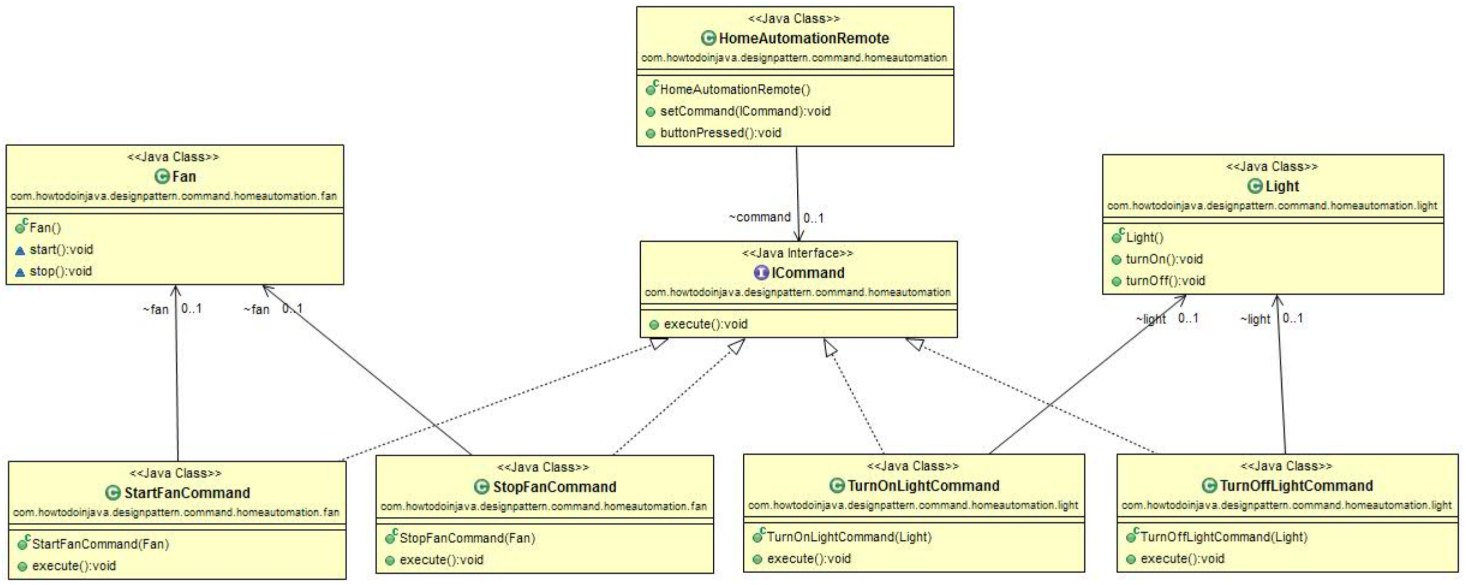
- Finally, code is not clean and is tightly coupled and we are not following best practices like *coding to interfaces* etc.

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



<<Java Class>>
HomeAutomationRemote
com.howtodoinjava.designpattern.command.homeautomation
HomeAutomationRemote()
setCommand(ICommand):void
buttonPressed():void
~command 0..1
<<Java Interface>>
ICommand
com.howtodoinjava.designpattern.command.homeautomation
execute():void
<<Java Class>>
Fan
com.howtodoinjava.designpattern.command.homeautomation.fan
Fan()
start():void
stop():void
~fan 0..1
~fan 0..1
<<Java Class>>
Light
com.howtodoinjava.designpattern.command.homeautomation.light
Light()
turnOn():void
turnOff():void
~light 0..1
~light 0..1
<<Java Class>>
StartFanCommand
com.howtodoinjava.designpattern.command.homeautomation.fan
StartFanCommand(Fan)
execute():void
<<Java Class>>
StopFanCommand
com.howtodoinjava.designpattern.command.homeautomation.fan
StopFanCommand(Fan)
execute():void
<<Java Class>>
TurnOnLightCommand
com.howtodoinjava.designpattern.command.homeautomation.light
TurnOnLightCommand(Light)
execute():void
<<Java Class>>
TurnOffLightCommand
com.howtodoinjava.designpattern.command.homeautomation.light
TurnOffLightCommand(Light)
execute():void

Command Pattern Class Diagram

Lets see the java source of each class and interface.

`ICommand.java`

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

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

Light.java

```
package com.howtodoinjava.designpattern.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.howtodoinjava.designpattern.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.howtodoinjava.designpattern.command.homeautomation.light;

import com.howtodoinjava.designpattern.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.howtodoinjava.designpattern.command.homeautomation.light;

import com.howtodoinjava.designpattern.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.howtodoinjava.designpattern.command.homeautomation.fan;

import com.howtodoinjava.designpattern.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.howtodoinjava.designpattern.command.homeautomation.fan;

import com.howtodoinjava.designpattern.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.howtodoinjava.designpattern.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.howtodoinjava.designpattern.command.homeautomation;

import com.howtodoinjava.designpattern.command.homeautomation.fan.Fan;
import com.howtodoinjava.designpattern.command.homeautomation.fan.StartFanCommand;
import com.howtodoinjava.designpattern.command.homeautomation.fan.StopFanCommand;
import com.howtodoinjava.designpattern.command.homeautomation.light.Light;
import com.howtodoinjava.designpattern.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..
```

When to Use Command Pattern

You can use command pattern for solving many design problems e.g.

- Handling actions for Java menu items and buttons.
- Providing support for macros (recording and playback of macros).
- Providing “undo” support.
- Progress bars implementations.
- Creating multi-step wizards.

Popular Command Pattern Implementations

These are some real world examples of command pattern implementations:

- Runnable interface (java.lang.Runnable)
- Swing Action (javax.swing.Action) uses command pattern
- Invocation of Struts Action class by Action Servlet uses command pattern internally.

Summary

1. Command pattern is a behavioral design pattern.
2. In command pattern, an object is used to encapsulate all the information required to perform an action or trigger an event at any point time, enabling developer to decentralize business logic. It is called command.
3. Client talks to invoker and pass the command object.
4. Each command object has reference to it’s receiver.
5. Command’s execute() method invoke actual business operation defined in receiver.
6. receiver executes the business operation.

Here are some of the pros and cons of this pattern. It may help you in taking the right decision about using command pattern.

Advantages

- Makes our code scalable as we can add new commands without changing existing code.
- Increase loose-coupling between the invoker and the receiver using the command object.

Disadvantages

- Increase in the number of classes for each individual command, in a different view. It may not be preferred in some specific scenarios.

That's all about *command design pattern*. Drop me your questions in comments section.

Download Source Code

Happy Learning !!

Share this:

Twitter Facebook LinkedIn Reddit WhatsApp

Subscribe to Blog

Feedback, Discussion and Comments

Nisal

February 11, 2020

Very Clear tutorial. Thank You Author.

Reply

Salman Khan

July 4, 2017

Hello Sir,
I read your blog regarding command pattern, which is very informative.
can you please share a link with me regarding simple way to learn spring security using different approaches like DAO etc.

Looking forward

Regards,
Salman Khan

Reply

Leave a Reply

Enter your comment here...

Search Tutorials

Type and Press ENTER...

Meta Links

About Me

Contact Us

Privacy policy

Advertise

Guest and Sponsored Posts

Recommended Reading

10 Life Lessons

Secure Hash Algorithms

How Web Servers work?

How Java I/O Works Internally?

Best Way to Learn Java

Java Best Practices Guide

Microservices Tutorial

REST API Tutorial

How to Start New Blog