# Theory: Command



42 minutes 0 / 5 problems solved



You probably have heard of **behavioral** patterns by now. Behavioral patterns are concerned about the interaction of objects. While there are about 12 design patterns that belong to behavioral patterns, the command pattern takes a special place as it is used more often than other design patterns. The purpose of the command pattern is to **decouple** the logic between command and its consumers.

The formal definition of the command pattern is **encapsulating** all the data related to command in one object. Usually, this data consists of a set of methods, their parameters, and one or more objects to which these methods belong to. We call this object **Receiver**. So the important point about decoupling is if you had to change any of these values, you only have to change one class.

In its classic version, implementing the command pattern involves five steps. Let's see what they are.

Current topic:

Command

Topic depends on:

✓ The concept of patterns

✗ Interface

Table of contents:

↑ Command

§1. The classic version

§2. Example of the command pattern

§3. Additional options

§4. Applicability

§5. Conclusion

Feedback & Comments

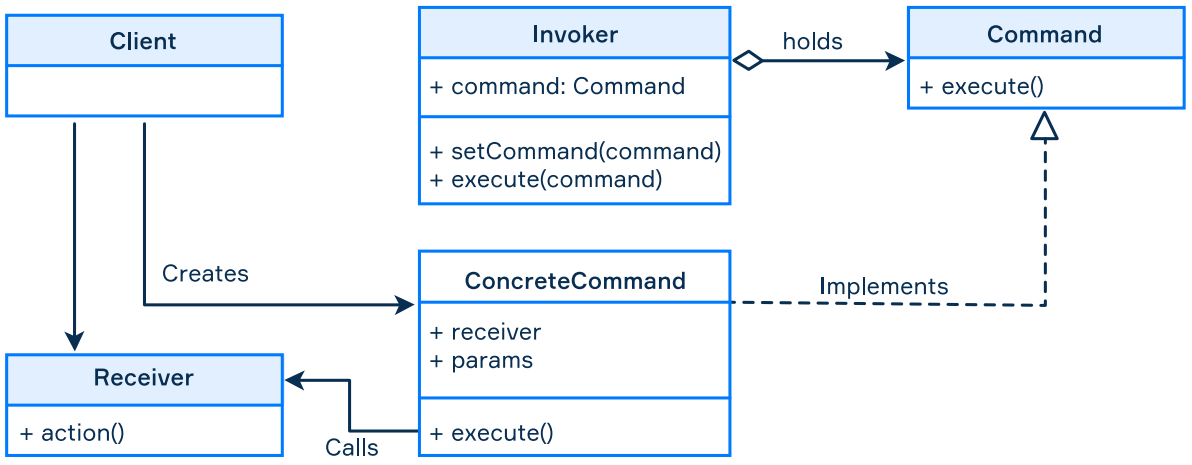
## §1. The classic version

The classic version of the **Command** pattern has the following elements:

- The **Command** interface usually declares just a single method for executing the command.
- The **ConcreteCommand** is an operation with parameters that pass the call to the receiver; In the classic approach, a command only invokes one or more methods of a ***Receiver*** rather than perform business logic.
- The **Receiver** knows how to perform the action.
- The **Invoker** asks the command to carry out the request.
- The **Client** creates a **ConcreteCommand** object and sets the ***Receiver***.

> Note that the interface ***Command*** is not necessarily an interface as a language's construct. It can be a simple or abstract class. The main thing is that it represents an abstract command that is inherited by concrete commands.

The following diagram illustrates all the elements of the pattern and their relations.




The *Client* creates an object of *Receiver* and a *ConcreteCommand* and sets up the Invoker to execute the command. Each type of *ConcreteCommand* (e.g. *CreateFileCommand*, *RemoveFileCommand*) has a set of fields which represent the params. A command calls one or more methods of the *Receiver* to execute concrete actions to change the state of the application.

You will be able to understand these concepts better with a real-world example.

## §2. Example of the command pattern

Suppose you are going to build a home automation system where you need to turn on and off a light. Here we have two commands which are quite similar. So we will create one interface as `Command` first. It will have only one method which is `execute()`.

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

Then we will create two classes that will implement the `Command` interface. These concrete classes encapsulate data needed for the command. So you have to create concrete classes for each command. We will be creating two concrete classes as our application has two commands, *Light On* and *Light Off*.

First. `LightOnCommand` will implement the `Command` interface.

```
public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override

    public void execute() {

        light.lightOn();

    }

}
```

Next, `LightOffCommand` will implement the `Command` interface. `LightOffCommand` basically has the same code that `LightOnCommand` has.

```
public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override

    public void execute() {

        light.lightOff();

    }

}
```

We haven't created the `Light` class yet which is our *Receiver*. So next we are going to create it.

```
public class Light {

    public void lightOn() {
        System.out.println("Turn on Light");
    }

    public void lightOff() {
        System.out.println("Turn off Light");
    }

}
```

Sometimes learning design patterns with simple examples is difficult because these examples don't represent the complexity of real-world applications. For example, someone may think why do we create a separate

`Light` class when only we need two methods and both of them can be implemented on the command classes itself. Well, in a real-world application, the `Light` class can be more complicated with more fields and methods in which command classes have nothing to do.

Next, we need to create the **Invoker class**. The invoker class decides how the commands are executed. For example, the invoker can keep a list of commands that need to be executed in a specific order. Please note that Invoker is just a general term we use to call this class which decides how commands are executed. You can name it as you want depending on the application you develop.

We will name the invoker class as `Controller` here.

```
public class Controller {

    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() {

        command.execute();

    }

}
```

Finally, our client or the `main` method will use the invoker to execute the command.

```
public class HomeAutomationDemo {

    public static void main(String[] args) {

        Controller controller = new Controller();
        Light light = new Light();

        Command lightsOn = new LightOnCommand(light);
        Command lightsOff = new LightOffCommand(light);



        controller.setCommand(lightsOn);

        controller.executeCommand();



        controller.setCommand(lightsOff);

        controller.executeCommand();

    }

}
```

It's quite straightforward as to what happens here. Basically, there are three significant steps in the main method.

1. Creating an object from the invoker class which is `Controller` in our application.
2. Creating objects from commands that we are going to execute.
3. Executing commands using invokers.

There could be other steps that are needed to support these three main steps. For example, this `main()` method has created a `Light` object because a `Light` object is needed to pass to create `Command` objects. When you execute this code, the following output will be produced.

```
Turn on Light
Turn off Light
```

# §3. Additional options

The Command pattern can be used together with the following options:

- adding commands to a queue to execute them later;
- supporting undo/redo operations;
- storing a history of commands;
- serializing commands to store them on a disk;
- assembling a set of commands into a single composite command known as **macros.**

These options are not essential to the pattern but are often used in practice.

Sometimes, a command performs all the work by itself instead of invoking the receiver object to do the action. This option is somewhat simpler and also used in practice.

# §4. Applicability

Possible applications of this pattern include:

- **GUI buttons and menu items.** In Swing programming, an *Action* is a command object. In addition to the ability to perform the desired command, an Action may have an associated icon, a keyboard shortcut, tooltip text, and so on.
- **Networking.** It is possible to send whole command objects across the network to be executed on the other machines: for example, player actions in computer games.
- **Transactional behavior.** Similar to *undo*, a database engine or software installer may keep a list of operations that have been or will be performed. Should one of them fail, all others can be reversed or discarded (this is usually called **rollback**).
- **Asynchronous method invocation.** In multithreading programming, this pattern makes it possible to run commands asynchronously in the background of an application. In this case, the Invoker is running in the main thread and sends the requests to the Receiver which is running in a separate thread. The invoker will keep a queue of commands and send them to the receiver while it finishes running them.

# §5. Conclusion

The main advantage of the command pattern is that it decouples the object that invokes the operation from the one that knows how to perform it. Various modifications of this pattern can be used to keep a history of requests, implement the undo functionality and create macro commands. However, keep it in mind that your application can become more complicated because this pattern adds another layer of abstraction instead of simply invoking methods.

Report a typo

**117** users liked this theory. **11** didn't like it. **What about you?**

Start practicing

Comments (16) Hints (1) Useful links (0) Show discussion