# CSC207H1: Command Design Pattern

Matthew Du

October 25, 2021

## Contents

# 1 Introduction

## 1.1 The Problem

You need to send out an request for an object to do something. However, you do not know anything (and don't want to know anything) about what operation is being requested, as well as who will receive the request. In other words, you need a way to blindly send out requests without needing to know how that request will be implemented. It should be loosely coupled, as well as abide by rules of clean architechture. [1]

## 1.2 The Solution

You use the Command Design Pattern [1]!

## 1.3 The Analogy

(**To the TA grading this:** Sorry this is rediculously long. It began to sound like a short story halfway through writing this, so I went along with it. I had way too much fun with this, so I hope you enjoy reading it! Here is the code if you want to follow along: https://github.com/matthewrhdu/CSC207Design

Suppose you are building a robot, and you it has a `Camera`, `SoundRecorder`, and `Brain`. You also have a `Robot` controller class that holds all of the objects together.

Let's say, you want to start a recoding. Doing so requires a lot of information, such as turning on the camera and sound. So you try something like this in the `CommandlessRobot` class,

```java
class CommandlessRobot {
    private final Camera camera;
    private final SoundRecorder sound;
    private final Brain brain;

    public CommandlessRobot(){
        camera = new Camera();
        sound = new SoundRecorder();
        brain = new Brain();
    }

    public void StartRecording(boolean withSound){
        this.camera.on();
        if (withSound){
            this.sound.open();
        }
        this.brain.receiveCameraData();
```

[1]Obviously. Seriously, what did you expect? This document is literally called Command Design Pattern...

```java
    }

    public void StopRecording(){
        this.camera.off();
        this.sound.close();
        this.brain.saveCameraData();
    }
    // ...
}

class RobotWithOutCommand {
    public static void main(String[] args) {
        CommandlessRobot rob = new CommandlessRobot();
        rob.StartRecording(true);
        rob.StopRecording();
    }
}
```

Now we have a functional program! Yay!

But wait! Right before you prepare to submit your code, you remember something that you learned in your software design course. You remebered that your professors exquisitely told you that code needs to be ”loosely coupled and obey clean architechture”.

”Is this implementation a good design?” You ask yourself.

After some deep thinking, you realize a few things.

- What if I need to add another device to the robot, say another camera? How would I do that? I probably will need to go into `CommandlessRobot` and modify the Start and Stop Recording methods. Oh dear. That seems like pretty tight coupling!
- What if I need to, say, control the lighting on the camera? Or I need to turn the camera off now? So now I need two more methods. What if I come up with another feature 200 years from now that I didn’t even think of? I would need go into robot and add the function as a method. Every feature would require me to add a method. How would I keep track of everything? Oh the terrany!

”Surely”, you tell yourself, ”There must be a better way to do this”.

## 2 What is the Command Design Pattern

### 2.1 Definitions and Terms

The technical definition of the Command Design Pattern is

”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." [1][2]

This means is that the program will make an object out of the request, with an `execute` method. The object will hold all execution details of the command. The object can then be used to support operations.

## 2.2 How does it work

There are four main players when it comes to the command design pattern [3]:

- **Client**: The object making the request
- **Invoker**: The object that receives the request from the client and creates the request.
- **Command**: The Request that is made
- **Receiver**: The object whose task the command is trying to execute.

Here is a diagram that represents the demonstrate the relation between the members:

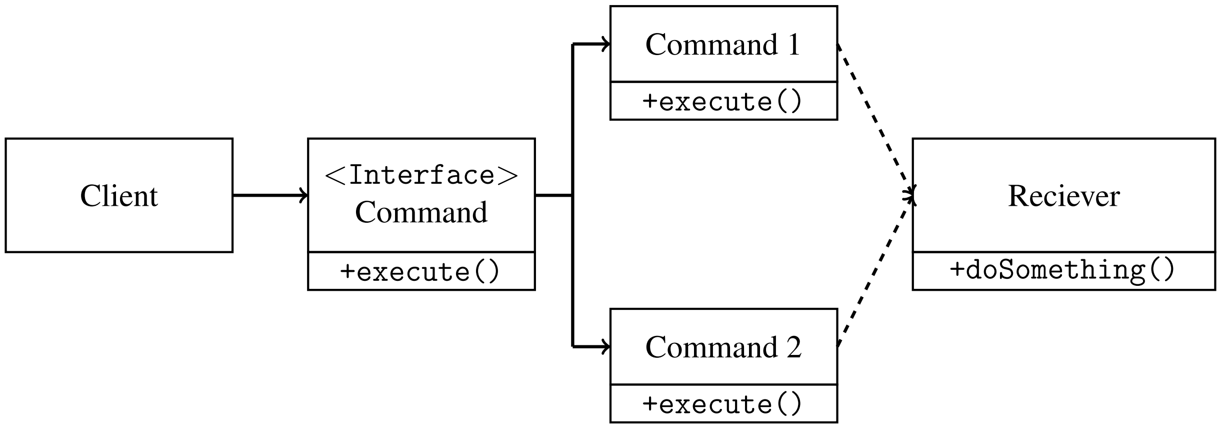

Figure 1: Command Pattern Diagram

Firstly, when an user makes a request, it goes into the client. The client then translates that request into a command object.

The client will then make a request to the invoker. This is typically done by the client calling an `add` type method to add a command to the invoker. This step is also useful as it allows the user to be able to easily implement an "Undo" command, as it provides a log of commands.

Next, the invoker will then be asked to execute the commands by the client. The invoker will call on the `execute` command of the command, which will execute the commands. Each command will then contain the set of instructions on how to execute the command, and will send that information to the correct receiver to be used.

## 2.3 How we will use it in our Project

Our group project's domain is to create a timetable application that allows for various features, such as saving, scrambling, and anything else we can think of before the due date.

One way that we can use the command pattern in our design project is to use it to control and command the various functions that we plan to implement. The Command Pattern allows for us to cleanly send and receive commands from the `UserInterface` class to various functions classes, as well as provide a log of all commands, which will allow for a rollback feature.

For instance, when the user chooses to, say schedule all the courses using a solver, there are multiple steps involved in initializing the solver. That would be a perfect candidate for the Command Pattern, since it is a command that would need to passed from the `DatabaseController` to the `function` classes. The `DatabaseController` would create the command, that will initiate the function. The members would act as:

- `UserInterface`: Client
- `DataBaseController`: Invoker
- `Commands`: Commands
- `Function`: Receiver

## 3 Code

Now going back to our story. You decide that you want to give this command pattern thing a try, so you write a new robot class:

```java
/**
* A robot Class
*/
class CommandRobot {
    private final ArrayList<Command> command;
    private final Camera camera;
    private final SoundRecorder sound;
    private final Brain brain;

    public CommandRobot(Camera theCamera, SoundRecorder theSound,
    Brain theBrain){
        this.camera = theCamera;
        this.sound = theSound;
        this.brain = theBrain;
        this.command = new ArrayList<>();
    }

    /**
    * Sets the theCommand for the robot to execute
    * @param theCommand The theCommand that the robot wants to execute
    */
    public void setCommand(Command theCommand) {
        command.add(theCommand);
    }

    /**
    * Executes the command saved in the robot
    */
    public void executeCommand() {
        for (Command cmd : this.command){
            cmd.execute();
        }
    }
}
```

This class acts as our Invoker in the command pattern, where it will store a list all the commands. Notice how this new implementation does not require us to know anything about the individual implementations of any of its components, hence, it is less coupled. Much better!

Now, instead of writing a bunch of `if / else` statements, you defined a `Command` interface to hold all the commands.

```java
    /**
    * The Command interface
    */
    interface Command {
        /**
        * Executes the command
        */
        void execute();
    }
```

This will make sure all the commands you write are executable. This could also be an abstract class, but that depends on what you use it for.

Now it's time to create some commands! The command you need is to allow the camera to start and stop recording, so you define two tasks.

```java
/**
* The command object that turns starts a recording
*/
class SoundRecordingCommand implements Command {
    private final Camera camera;
    private final SoundRecorder recorder;
    private final Brain brain;
    private final boolean onOff;

    /**
    * Constructor
    *
    * @param theCamera the Camera object associated with this
    command
    * @param theRecorder the Brain object associated with this
    command
    * @param theBrain the Brain object associated with this
    command
    */
    public SoundRecordingCommand(Camera theCamera,
   SoundRecorder theRecorder, Brain theBrain, boolean onOff) {
        this.camera = theCamera;
        this.recorder = theRecorder;
        this.brain = theBrain;
        this.onOff = onOff;
    }

    /**
    * Executes the command
    */
    @Override
    public void execute() {
        if (onOff) {
            camera.on();
            recorder.open();
            brain.receiveCameraData();
        } else {
            camera.off();
            recorder.close();
            brain.saveCameraData();
        }
    }
}

/**
* The command object that turns starts a recording
*/
class SoundlessRecordingCommand implements Command {
    private final Camera camera;
    private final Brain brain;
    private final boolean onOff;

    /**
    * Constructor
    *
    * @param theCamera the Camera object associated with this command
    * @param theBrain the Brain object associated with this command
    */
    public SoundlessRecordingCommand(Camera theCamera, Brain theBrain, boolean onOff) {
        this.camera = theCamera;
        this.brain = theBrain;
        this.onOff = onOff;
    }

    /**
    * Executes the command
    */
    @Override
    public void execute() {
        if (onOff) {
```

```java
            camera.on();
            brain.receiveCameraData();
        } else {
            camera.off();
            brain.saveCameraData();
        }
    }
}
```

Wow! That's a lot more code than before! Why did we need all this?

Well, one reason is that notice that in each of the commands, there are many tasks to do at once. A command can then be more accomplish more complex tasks much more cleanly.

In addition, all of the requests that we would need to make are encapsulated in this command object, thus we do not need to worry about the implementation details in the outer classes. Now that is what I call clean architechture!

Now, finally, a `main` method brings this all together

```java
/**
* Sample Usage
*/
class RobotWithCommand {
    public static void main(String[] args) {
        // The Components of the robot
        Camera GeoffreyCam = new Camera();
        SoundRecorder GeoffreySoundSystem = new SoundRecorder()
;
        Brain GeoffreyBrain = new Brain();

        // The Robot
        CommandRobot Geoffrey = new CommandRobot(GeoffreyCam,
        GeoffreySoundSystem, GeoffreyBrain);

        Command[] commands = {
            new SoundRecordingCommand(GeoffreyCam,
GeoffreySoundSystem, GeoffreyBrain, true),
            new SoundlessRecordingCommand(GeoffreyCam,
GeoffreyBrain, true),
            new SoundRecordingCommand(GeoffreyCam,
GeoffreySoundSystem, GeoffreyBrain, false),
            new SoundlessRecordingCommand(GeoffreyCam,
GeoffreyBrain, false)
        };

        for (Command cmd : commands){
            Geoffrey.setCommand(cmd);
        }

        Geoffrey.executeCommand();
    }
}
```

That's better! I don't see anything about how the components operate! That's a fine looking code.

And so, as you eagerly prepare your file for submission, you think about the journey you have taken today and many things that you have learned along the way about the command pattern. You now understand the usage and why we use the command pattern. As you press the scary big white submit button, you feel confident to say that you now understand why you should use the Command design pattern.

## References

[1] “Design Patterns and Refactoring.” [Online]. Available: https://sourcemaking.com/design_patterns/command. [Accessed: 18-Oct-2021]

[2] “Command Pattern,” 2016. [Online]. Available: https://www.geeksforgeeks.org/command-pattern/. [Accessed: 18-Oct-2021]

[3] “Command pattern,” Wikipedia, 29-May-2021. [Online]. Available: https://en.wikipedia.org/wiki/Command_pattern [Accessed: 24-Oct-2021].

[4] “Design patterns and refactoring,” SourceMaking. [Online]. Available: https://sourcemaking.com/design_patterns/command/java/1. [Accessed: 24-Oct-2021].