AARHUS UNIVERSITET

SWD Design Pattern - Command pattern

Group 50

Participants	
AU ID	Name
au607527	Cuong Nguyen
au556770	Danny Tran
au559497	Stanie Cheung
au546123	Nina Nguyen

Contents

1 Introduction

This report contains a thoroughly description on a design pattern we have chosen for our last assignment in I4SWD. The design pattern we have chosen to work with is the **Command pattern**.

The report will go through the following sections:

- Purpose of the design pattern
- Design patterns structure
- Related patterns
- Example -> Demonstration
- Pros. and cons.

The report shall deliver a good understanding about the design pattern.

2 Purpose

The Command Pattern is a simple pattern that allows you to decouple a class that invokes a command from a class who will know how to perform it. In other words, it encapsulates a command as an object, allowing us to issue requests without knowing the requested operation or the requesting object. It is a behavioural design pattern in which an object is used to encapsulate a request. It requires necessary information to perform the command, which are the following: the method name, the object related to the method and the parameters related to the method.

Type:
The type of this pattern is a behavioural pattern. This is because it encapsulates a command request as an object. Behavioral patterns are those patterns that are specifically concerned about the communication between objects. In Command Pattern, we are concerned about encapsulating commands as objects.

Intent:

- Encapsulate a request as an object
- Allows parameterization of clients with different requests
- Allows saving the requests in a queue
- Supports undoable operations

2.1 Reflection

Suppose we have a automation system for all the doors/gates/windows/etc in a building. There is a programmable remote which can be used to open and close these various entrances. The code for the remote control would consist of **If-else** statements. *"Simple right?"*.

Keep in mind that opening and closing various entrances can have different functionalities depending on what it is. For instance, a window doesn't necessarily need to close completely, it can still have a small gap of a hole opened for fresh air purposes *"An odd example, i know"*.
By using **If-else** statements only, we are coding to the implementation rather than the interface. Also, there is a tight coupling.

So the idea behind a Command Pattern, is to loose the coupling, whereas the remote control shouldn't have much knowledge about the various entrances.

3 Command Pattern Structure

The foundation of the Command pattern consists of five classes: Client, Invoker, Receiver, Command, ConcreteCommand.

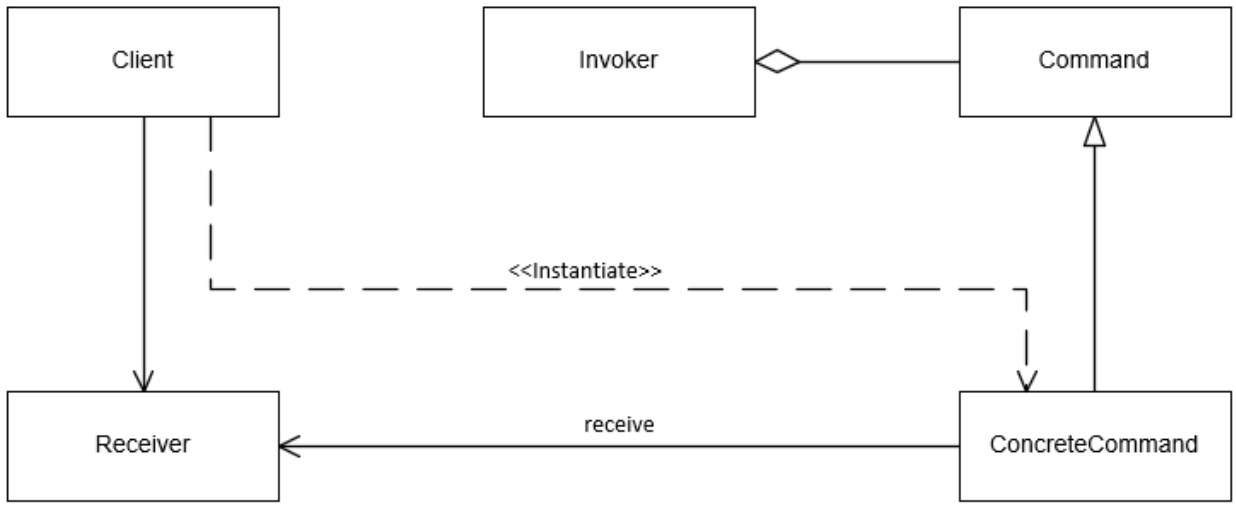

Figure 1: This figure is an overview that illustrates the associations between the classes.

Client
This class is the client, it is the one who is in charge of the actions that will be triggered depending on what kind of commands has been made. These commands are ConcreteCommand.

Invoker
This class is the invoker, the purpose behind this class is to deliver the requests coming from the Command class. These requests will be delivered to the Receiver class, which is gonna perform the actions, requested by the Client class.

Receiver
This class is the receiver, the purpose behind this class is to perform the requests, the client requested for. The action/command is delivered by the Command class, who "invoked" a command that it has to perform. The Receiver class then performs the commands. The class is considered as the "business layer", it does all the work.

Command

This class is the command, it contains all the actions/commands that can be performed whenever they are triggered by the client. These commands will be "invoked" and performed by the Receiver class.

ConcreteCommand

This class is the command performed by the client. it is an object and it sets its receiver. For instance, the client could write a mail the client wants to send out to his/her friends or co-workers. The mail(s) would be the ConcreteCommand.

3.1 Process of the Command Pattern

The process starts with the Client requesting an execution of a command. This will trigger the ConcreteCommand to say *"hey, something has been requested, lets complete the task"*, it then invokes the command. This is where the Invoker class comes in. The Invoker encapsulates the command(s) and stores it in a queue(Having a queue, keeps the commands in an order). It will then deliver the command(s) to the Receiver, where it will be performed. The Receiver receives the command(s) and executes it.

4 Examples of Command Pattern

There are many things that we can use as an example for the Command Pattern. An easy example we could come up with was a door controlled by two buttons, that can open and close the door.

4.1 Design & Implementation

In This section, a class diagram and two sequence diagrams have been made to illustrate how we made use of the Command Pattern.

4.1.1 Class Diagram

Figure 2 below is our class diagram to our example, "Door Open/Close". This solution contains five classes Client, QueueingService, IOrder, ConcreteCommand and Action.

The Client class is our Client, according the to Commander Pattern.
The QueueingService is our Invoker.
The IOrder is our Command.
The ConcreteCommand is our ConcreteCommands.
The Action is our Receiver.

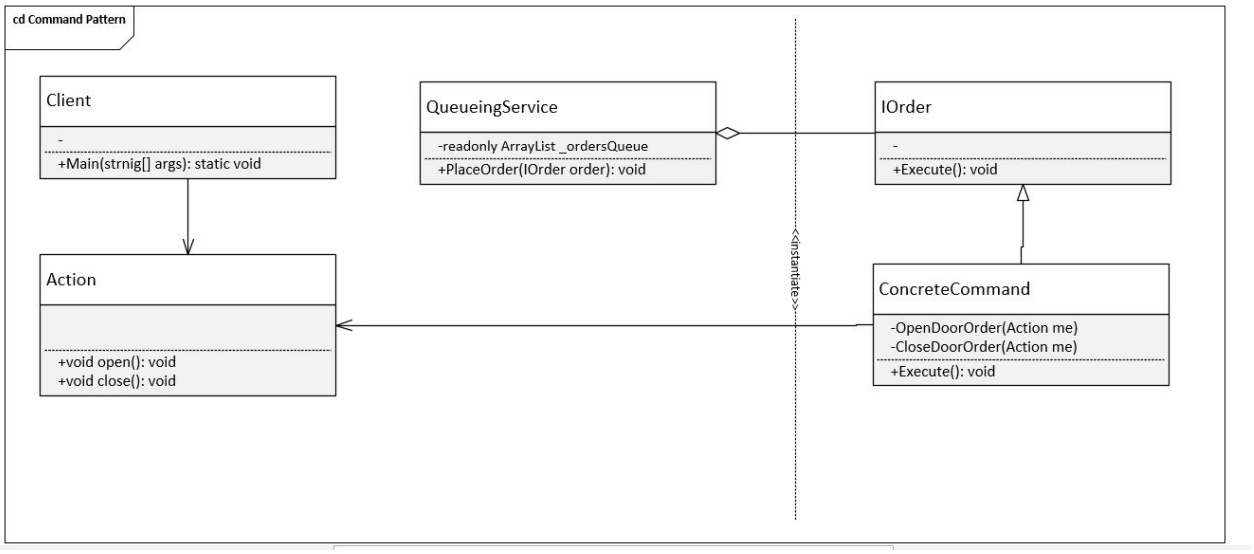

Figure 2: Class diagram for Command Pattern

4.1.2 Sequence Diagrams

We have designed two Sequence Diagrams for each button, Open() and Close(), as seen below. The Sequence Diagrams are based on the Class Diagram as seen above.

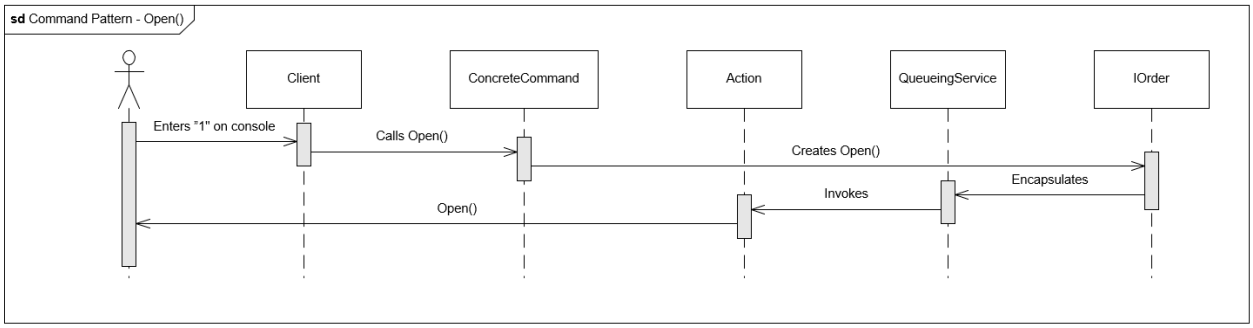

Figure 3: Sequence Diagram for Open()

As you can see on Figure 3 an User enters "1" on the console and calls the Open() in ConcreteCommand, which creates the Open() in IOrder. When IOrder receives this command, it will encapsulate the command as an object and put it in a QueueingService, which works basically like the Producer-consumer problem. When it is encapsulated, it will then invoke the Action class, which will execute the Open(). In this situation the Action class is the receiver, which shows the how the actions are implemented.
The same principle is added in the Close() Sequence Diagram as you can see below.

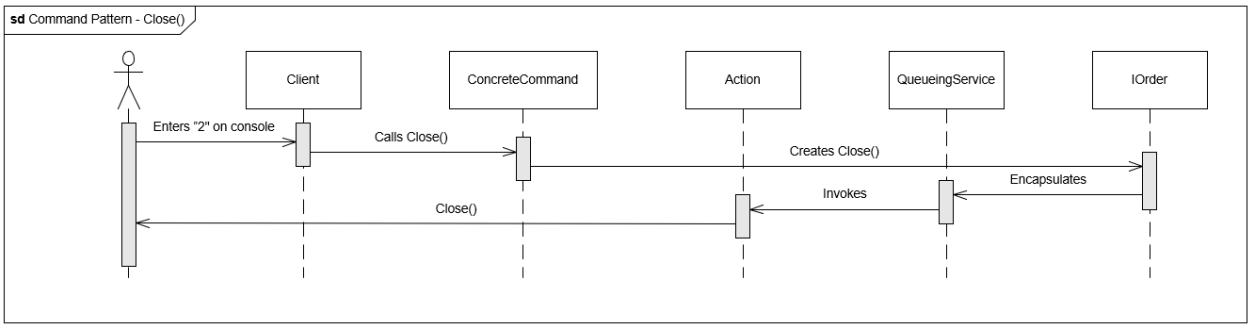

Figure 4: Sequence Diagram for Close()

The only change in Close() is that we have changed the button, which is in our case button "2". It does the same as Open().

5 Related Patterns

In this section related design patterns that shares the most resemblance to the command pattern will be explained and the similarities and differences between the related design patterns will be discussed.

5.1 Command pattern vs Strategy Pattern

The command pattern shares similarities with the strategy pattern. It is similar in the sense that while you in the command pattern encapsulates a class so it can be passed around (which provides flexibility), you can also use encapsulation in the same sense in strategy pattern.
However encapsulation in the strategy pattern focus more on storing algorithms where as the command pattern focus on encapsulating an action to send from one client to another. Therefore,it can be said that the encapsulations scale of the command pattern operates on a smaller level in terms of details, as opposed to the strategy pattern that mainly focus on hiding the details of an algorithm's implementation. So to sum it up, both design patterns use encapsulation, but they don't encapsulate one the same level.

5.2 Command pattern vs Memento pattern

Another design pattern that has resemblance to the command pattern is Memento pattern.The memento pattern has three key components consisting of a memento - which contains the basic storage of the object, an originator - that creates the new mementoes and assigns current values, and a caretaker - contains all previous mementoes that can store and retrieve. Both design patterns can be used to create the undo functionality (how the command pattern can undo actions is stated earlier in the report).
The memento pattern can be used to store a previous state of an object so that you can undo an operation, just like with how you are able to restore undo in the command pattern due to the decoupling of an object that invokes an operation.
The main difference between those two design patterns is that while the memento pattern is intended to change just a singular state, the command pattern is used to handle more than one, as in multiple states simultaneously.

6 Pros & Cons

In this section, there will be pointed out the pros. and cons. for the Command pattern.

Pros.

- Decreases the dependency/coupling of the system.
- It helps in terms of extensible as we can add a new command without changing the context of the existing code.
- Macro instructions are easier to implement now.
- Ability to undo/redo easily e.g. use Memento pattern to maintain the states.

Cons.

- Using command pattern may require more effort on implementation, since each command requires a concrete command class, which will increase the number of classes significantly.

7 Conclusion

The usability on the Command Pattern can be limited, however it is still very dependable when it is used on small systems such as air-conditions, televisions, doors/gates, windows, etc. Basically a remote control that is used for easy going devices(Devices that doesn't require many functions). Furthermore the design pattern is relatively useful within other areas, such as:

- Graphical User Interfaces(GUI)
- Macro recording
- Multi-step undo
- Networking
- Progress bar
- Transactions

For the GUI, the Command pattern is used on the button clicks. For instance, whenever a button is pressed we can read the current information of the GUI and take an action.

The structure of the pattern is very consistent, so it is really easy to understand how to create the commands/actions.

Overall the pattern is simple and easy to get used to once you have made the first command working.