Command Design Pattern

Team 7 - Eric Bai, Andy Baek, Xiaotong Chi, Yuhao Liu

Purpose

The purpose of the command design pattern is to encapsulate a request into an object (known as the command), so that the service that must make requests (known as the invoker) does not need to know any details about the request.

Intended Use Case

The command design pattern should be used when:

- You have many general components that need to execute methods without the need to know anything about the method (e.g. a GUI interface requires buttons that all do very different things, but should have a generalized "onClicked" function)
- Requests need to be handled at variant times or queued up (e.g. task schedulers)
- You need to keep track of a history of what commands were performed (e.g. when you want to be able to undo or redo the last few commands)
- You want to add logging to all of the commands that you want to perform

Vocabulary

The command design pattern contains components known as the command, receiver, client and invoker.

- **Command** - An object that has a defined execute() method. The command object also holds the receiver object that it is associated with.
- **Receiver** - When a command is executed, it calls a method on the receiver that it is holding. The receiver runs the task that the user is trying to perform.
- **Client** - Decides which commands should be run at which points. It does this by creating a command object mapped to a receiver and hands it to the invoker to execute.
- **Invoker** - The client calls the invoker with a command object that it wants to execute. The invoker can keep a history of the commands executed and perform bookkeeping. The invoker only knows the interface for a command, and does not need to know the implementation details of execute().

Structure

When a client chooses a point when some number of specific commands should be executed (e.g. immediately or in the future), it will create a new instance of the specific commands. Each specific command is passed in a receiver, and the command will know how to use the receiver to execute the desired action. The receiver is therefore the component that does the actual work that the client wants. The client will then pass this command to the invoker. At the chosen point when the client wants to execute the commands in the invoker, it can tell the invoker to do so. The invoker only knows how to use the abstract command interface to call execute(), which tells the commands to use the receivers. The receivers then perform the actions required, and the client's commands have all been completed.

UML Diagram

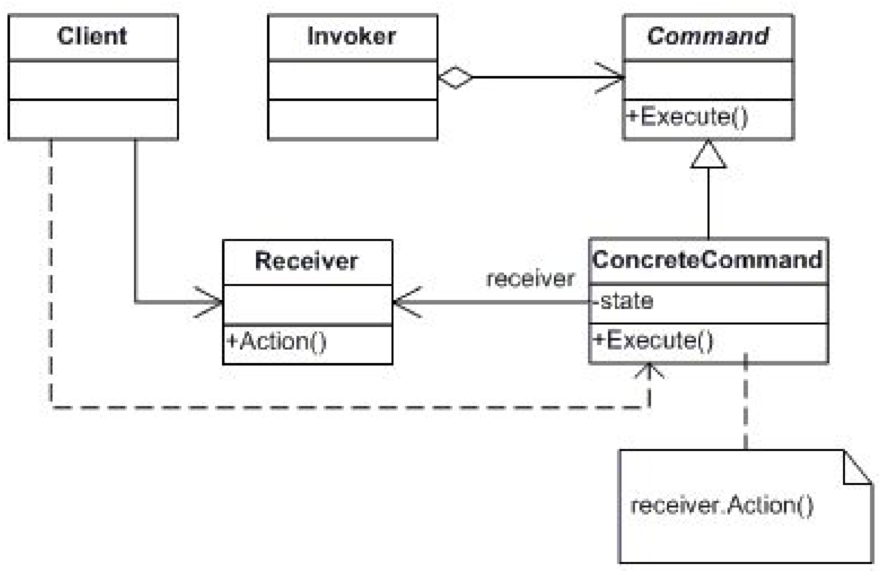


*diagram taken from http://www.dofactory.com/net/command-design-pattern

Consequences

Positive:

- Allows the invoker software to support a wide variety of commands, without having to be rewritten each time a new type of command is added
- Decouples the classes that invoke the operation from the object that knows how to execute the operation
- Sequences of Commands can be queued up or assembled into composite (or macro) commands

Negative:

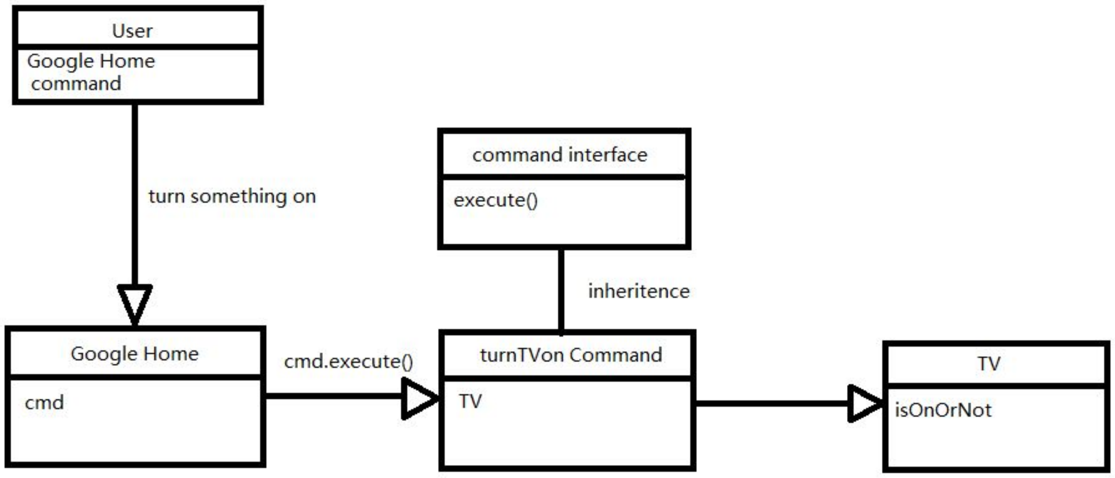
- There will be a lot of command classes if we want to support large amount of operations, which can clutter the system

Relation to NFPs

Command design property improves scalability and security, since it decouples the invoker from the command and we can have different commands inherit from same command interface.

Examples

Note: Our group tried to keep the examples simple for the class to understand the basic principles of the command design pattern. Therefore these examples do not cover all possible capabilities of the Command Design Pattern.



Example 1 - Simple Command
In the first scenario, we are going to have a user(client), a smart speaker like Google Home(invoker) and a TV(receiver). The command is represented by the procedure the smart home system takes to turn on the TV.

The user simply tell Google home "turn the living room TV on" and Google Home will invoke command to turn the TV on. The benefit here is that the user can add additional smart devices that connect to the Google Home, and the code for the Google Home does not need to know the implementation details of each smart device to support each new device the user adds.

Example 2 - Macro Command
In the second scenario, we are going to have a user(client), a smart speaker like Google Home(invoker), a TV and a light. The command will be a pre-stored macro command that turn off everything.

When the user leaves home, he will tell Google Home "Turn off everything" and Google Home will invoke a pre-stored command that turns off everything in the house, including the light and the TV. The benefit here provided by command pattern is that we can store multiple commands in one command (a macro command) and we can reuse it whenever we want.

Example 3 - Undo

In the third scenario, the user will first tell google to turn on TV, then turn on the radio. Afterwards, the user will tell Google Home “undo the last command”. Google Home, the invoker, can store a history of its commands in a stack. It can then pop the last command off that stack, and call a abstract “undo” function on that command. As a result, the radio will turn off. The benefit here is that the client and invoker do not need to know how to undo the action, and yet the code for undoing the action is incredibly simple. This undo functionality requires all commands to have a Command.undo() function defined as well as the regular Command.execute() function.

Example 4 - Add a New Receiver

In the fourth scenario, the user will be given a new smart device, such as a thermostat. The user will install the thermostat, then tell Google Home to “change temperature to 25 degrees Celsius”. Google Home will execute the command accordingly with the new thermostat device that was installed, and the temperature will change. This will show how simple it is to add a new receiver and that there is no code in the invoker that needs to be changed to handle this new smart device.

Example 5 - Schedule a Command for a Later Time

In the fifth scenario, the user will tell google to “change temperature to 20 degrees Celsius at 5:30 today”. Google Home will add this command to the scheduler. Once it hits 5:30, Google will execute the command and the temperature will change. This shows the the invoker Google Home can handle various functionalities such as scheduling commands in the future, similar to how it handled undo and marco commands in the previous examples.