Command Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the use of the Command Design Pattern
- Use and implement the Command Pattern

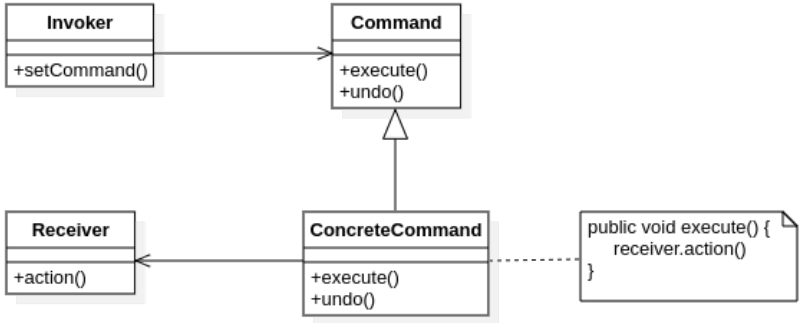
Inspiration

“A language that doesn’t have everything is actually easier to program in than some that do.” – Dennis Ritchie

Command Pattern: Definition

- The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different request, queue or log requests, and support undoable operations
- Think of a Restaurant
  - You, the Customer, give your Server an Order
  - The Server takes the Order to the kitchen and says “Order Up”
  - The Cook takes the Order and prepares your meal
    - Think of the order as making calls on the Cook like `makeBurger()`
- A request (`Order`) is given to one object (`Server`) but invoked on another (`Cook`)
  - This decouples the object making the request (`Customer`) from the object that responds to the request (`Cook`); This is good if there are potentially many objects that can respond to requests

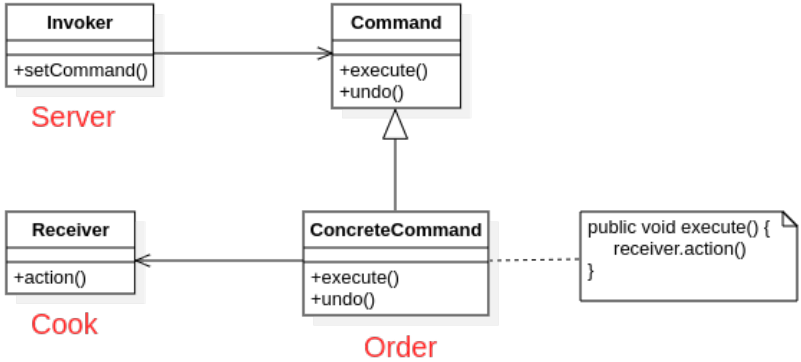
Command Pattern: Structure



I'm leaving one piece out of this diagram: the client.

In order for this pattern to work, someone needs to create a command object and set its receiver. And, someone needs to give command objects to an invoker to invoke at a later time.

Command Pattern: Structure



I'm leaving one piece out of this diagram: the client.

In order for this pattern to work, someone needs to create a command object and set its receiver. And, someone needs to give command objects to an invoker to invoke at a later time.

Example: Remote Control

The example in the textbook involves a remote control for various household devices

- Each device has a different interface (plays role of `Receiver`)
- Remote control has uniform interface (plays role of `Client`): “on” and “off”
- `Command` objects are created to “load” into the various slots of the remote control
  - Each command has an `execute()` method that allows it to emit a sequence of commands to its associated receiver
    - Light: turn light on
    - Stereo: turn Stereo on, select “CD”, `play()`

In this way, the details of each receiver are hidden from the client. The client simply says `on()` which translates to `execute()` which translates to the sequence of commands on the receiver: nice loosely-coupled system

Enabling Undo

- The command pattern is an excellent mechanism for enabling undo functionality in your application designs
  - The `execute()` method of a command performs a sequence of actions
  - The `undo()` method performs the reverse sequence of actions
- Assumption: `undo()` is being invoked right after `execute()`
  - If that assumption holds, the `undo()` command will return the system to the state it was in before the `execute()` method was invoked
- Since the `Command` class is a full-fledged object, it can track "previous values" of the system, in order to perform the `undo()` request
  - Example in book of a command "fan speed". Before `execute()` changes the speed, it records the previous speed in an instance variable

Macro Commands

- Another nice aspect of the Command pattern is that it is easy to create Macro commands
  - You simply create a command that contains an array of commands that need to be executed in a particular order
  - `execute()` on the macro command, loops through the array of commands invoking their `execute()` methods
  - `undo()` can be performed by looping through the array of commands backwards invoking their `undo()` methods
- From the standpoint of the client, a Macro command is simply a “decorator” that shares the same interface as normal `Command` objects
  - This is an example of one pattern building on another

Demonstration

The sample code demonstrates several aspects of the Command pattern

- Simple commands
- Simple Undo
- Macro Commands

Additional Uses: Queuing

- The command pattern can be used to handle the situation where there are a number of jobs to be executed but only limited resources available to do the computations
  - Make each job a `Command`
  - Put them on a `Queue`
  - Have a thread pool of computation threads
  - And one thread that pulls jobs off the queue and assigns them to threads in the thread pool
    - If all computation threads are occupied, then the job manager thread blocks and waits for one to become free

Additional Uses: Logging

- This variation involves adding `store()` and `load()` methods to command objects that allow them to be written and read to and from a persistent store
  - The idea is to use `Command` objects to support system recovery functionality
- Imagine a system that periodically saves a “checkpoint” of its state to disk
  - Between checkpoints, it executes commands and saves them to disk
  - Imagine the system crashes
  - On reboot, the system loads its most recent “checkpoint” and then looks to see if there are saved commands
    - If so, it executes those commands in order, taking the system back to the sate it was in just before the crash

Are there any questions?