Singleton and Command

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 22 — 11/08/2007

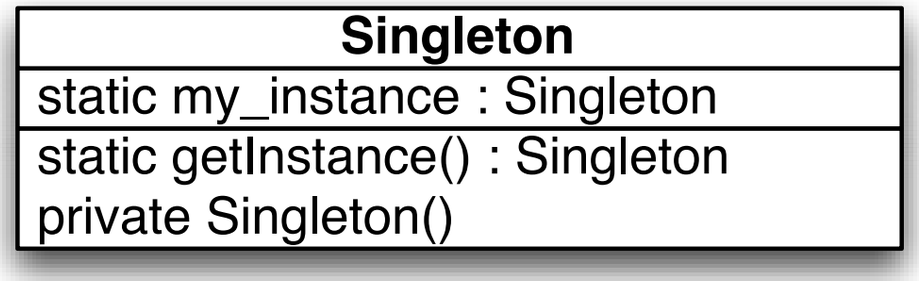
Lecture Goals

- Cover Material from Chapters 5 and 6 of the Design Patterns Textbook
  - Singleton Pattern
  - Command Pattern

Singleton Pattern: Definition

- The Singleton Pattern ensures a class has only one instance (or a constrained set of instances), and provides a global point of access to it
  - Useful for objects that represent real-world resources, such as printers, in which you want to instantiate one and only one object to represent each resource
    - Also useful for “management” code, such as a thread/connection pool
- At first, Singleton may seem difficult to achieve… typically, once you define a class, you can create as many instances as you want
  - Foo f = new Foo(); Foo f1 = new Foo(); Foo f2 = new Foo()…
- The key (in most languages) is to limit access to the class’s constructor, such that only code in the class can invoke a call to the constructor (or initializer or <insert code that creates instances here>)
  - Indeed, as you will see, different languages achieve the Singleton pattern in different ways

Singleton Pattern: Structure



Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown)

The class has a static variable that points at a single instance of the class.

The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance

World's Smallest Java-based Singleton Class

```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13 }
14
```

Meets Requirements: static var, static method, private constructor

src.zip has this class in ken/simple augmented with test code

World's Smallest Python-Based Singleton Class

```
1  class Singleton(object):
2
3      _instance = None
4
5      def __new__(cls, *args, **kwargs):
6          if not cls._instance:
7              cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
8          return cls._instance
9
10 if __name__ == '__main__':
11     a = Singleton()
12     b = Singleton()
13
14     print "a = %s" % (a)
15     print "b = %s" % (b)
16
```

Different Approach: static var, override constructor

only 8 lines of code!

src.zip has this class in ken/simple

World's Smallest Ruby-based Singleton Class

```
 1 require 'singleton'
 2
 3 class Example
 4   include Singleton
 5 end
 6
 7 a = Example.instance
 8 b = Example.instance
 9
10 puts "a = #{a}"
11 puts "b = #{b}"
12
13 c = Example.new
14
```

Yet a different approach, using a mechanism in Ruby called a "mixin"

The "include Singleton" statement causes the Example class to be modified such that its new() method becomes private and an instance() method is added to retrieve an instance. As a bonus, it will also handle hiding allocate(), overriding the clone() and dup() methods, and is thread safe!

Only 5 lines of code!

Thread Safe?

- The Java and Python code just shown is not thread safe
  - This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously
  - If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)
  - Example Next Slide

Program to Test Thread Safety

```
1  public class Creator implements Runnable {
2
3      private int id;
4
5      public Creator(int id) {
6          this.id = id;
7      }
8
9      public void run() {
10         try {
11             Thread.sleep(200L);
12         } catch (Exception e) {
13         }
14         Singleton s = Singleton.getInstance();
15         System.out.println("s" + id + " = " + s);
16     }
17
18     public static void main(String[] args) {
19         Thread[] creators = new Thread[10];
20         for (int i = 0; i < 10; i++) {
21             creators[i] = new Thread(new Creator(i));
22         }
23         for (int i = 0; i < 10; i++) {
24             creators[i].start();
25         }
26     }
27
28 }
29
```

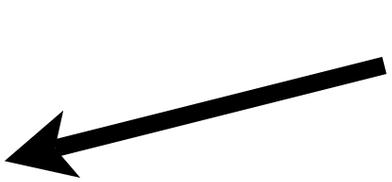
Creates a “runnable” object that can be assigned to a thread.

When its run, its sleeps for a short time, gets an instance of the Singleton, and prints out its object id.

The main routine, creates ten runnable objects, assigns them to ten threads and starts each of the threads

Output for Non Thread-Safe Singleton Code

- s9 = Singleton@45d068
- s8 = Singleton@45d068
- s3 = Singleton@45d068
- s6 = Singleton@45d068
- s1 = Singleton@45d068
- s0 = Singleton@ab50cd
- s5 = Singleton@45d068
- s4 = Singleton@45d068
- s7 = Singleton@45d068
- s2 = Singleton@45d068



Whoops!

Thread 0 created on instance of the Singleton class at memory location ab50cd at the same time that another thread (we don't know which one) created an additional instance of Singleton at memory location 45d068!

How to Fix?

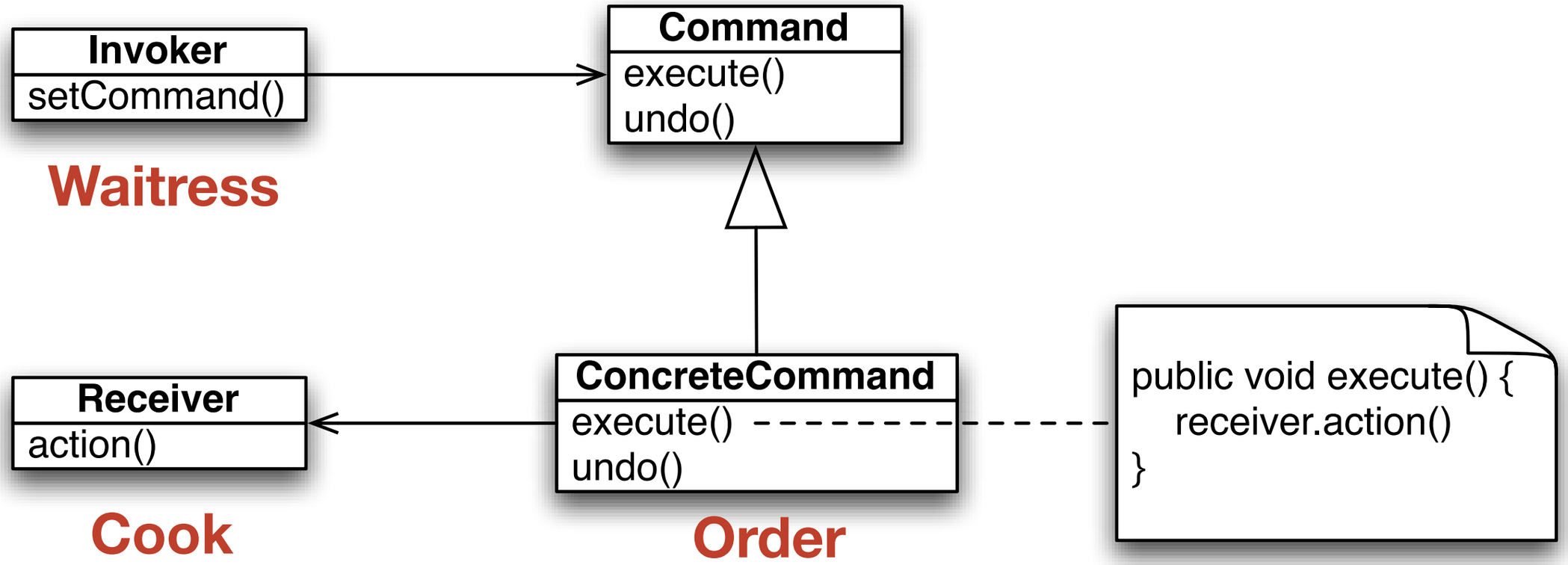
```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static synchronized Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13
14 }
15
```

In Java, the easiest fix is to add the **synchronized** keyword to the getInstance() method. The book talks about other methods that address performance-related issues. My advice: use this approach first!

Command Pattern: Definition

- The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations
- Think of a Restaurant
  - You, the Customer, give your Waitress an Order
  - The Waitress 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 (Waitress) 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.

Those "someones" may be the same object, they may be different objects

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 to control “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

- Code in src.zip 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 state it was in just before the crash

Coming Up Next

- Lecture 23: Adapters and Template Methods
    - Read Chapters 7 and 8 of the Design Patterns Textbook
- Lecture 24: Iterator, Composite, and State
    - Read Chapters 9 and 10 of the Design Patterns Textbook