Lecture 21: Design Patterns (Part 3)

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2005

Credit where Credit is Due

Some of the material for this lecture is taken from “Head First Design Patterns” by Eric and Elisabeth Freeman; as such some of this material is copyright © O'Reilly, 2004
Goals for this (short) lecture

- Cover three more useful design patterns
  - Command
  - Facade
  - Proxy
- This will bring the number of design patterns covered in this class to at least 15
  - Twelve from lectures 13, 17, and 21 plus Double Dispatch, Blackboard, and Model-View-Controller.

Command

- 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
- Consider the operation 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 prepares your meal from the Order
    - Think of the order as making calls on the Cook like “makeBurger()”
- A request is given to one object but implemented by another one
  - This decouples the object making the request from the object that responds to the request
Command’s Structure and Roles

Invoker
setCommand()

Command
execute()
undo()

Receiver
action()

ConcreteCommand
execute()
undo()

public void execute() {
    receiver.action()
}

Client

Back to the analogy...

Invoker
setCommand()

Command
execute()
undo()

Waitress

Order

Cook

Receiver
action()

ConcreteCommand
execute()
undo()

public void execute() {
    receiver.action()
}

Client

Customer
Example: Home Remote Control

Imagine a programmable remote control that can control various devices around your home
- e.g. lights, TV, DVD player, etc.

We’ll show code that has commands to turn a light on and off and an undo button to reverse the previously executed command

First, we need a light class; plays the role of Receiver

```java
public class Light {
    public Light(String name) { ... }
    public void on() { ... }
    public void off() { ... }
}
```

Command Interface; LightOn

Next, we need the Command interface

```java
public interface Command {
    public void execute();
    public void undo();
}
```

And a Command to turn the Light on

```java
public class LightOnCommand implements Command {
    Light light;
    public LightOnCommand(Light light) {this.light = light;}
    public void execute() {light.on();}
    public void undo() {light.off();}
}
```
LightOffCommand

And, a command to turn the light off

```java
public class LightOffCommand implements Command {
    Light light;
    public LightOffCommand(Light light) {this.light = light;}
    public void execute() {light.off();}
    public void undo() {light.on();}
}
```

Remote Control

The remote control stores three commands; acts as Invoker

```java
public class RemoteControl {
    Command onCommand;
    Command offCommand;
    Command undoCommand;
    public void setOnCommand(Command c) {onCommand = c;}
    public void setOffCommand(Command c) {offCommand = c;}
    public void on() { onCommand.execute(); undoCommand = onCommand;}
    public void off() { offCommand.execute(); undoCommand = offCommand;}
    public void undo() {undoCommand.undo();}
}
```
Client

- The client configures the remote control and then uses it

```java
public static void main(...) {
    RemoteControl rc = new RemoteControl();
    Light kitchenLight = new Light("Kitchen");
    LightOnCommand on = new LightOnCommand(kitchenLight);
    LightOffCommand off = new LightOffCommand(kitchenLight);
    rc.setOnCommand(on);
    rc.setOffCommand(off);
    rc.on(); // Light On
    rc.undo(); // Light Off
}
```

Facade

- The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. The Facade defines a higher level interface that makes the subsystem easier to use

- Principle of Least Knowledge
  - Talk only to your immediate friends
    - or, for any one object, try to limit its knowledge of other objects

- The principle recommends the following
  - given an object, code in one of its methods can invoke methods on
    - the object itself
    - Objects passed as a parameter to the method
    - Any object the method creates
    - Any components of the object (HAS-A relationships)
Facade’s Structure and Roles

Clients ask the Facade to perform a service; the Facade responds by making appropriate calls on the objects contained within the subsystem; the Client has no direct access or knowledge of the classes contained within the subsystem.

Example: Home Theater System

Imagine a home theater system represented as a bunch of objects.

- You might have objects like:
  - Amplifier, tuner, DVDPlayer, Projector, CDPlayer, TheaterLights, Screen, and PopcornPopper.
- To watch a DVD, you might have to:
  - Turn the popcorn popper on
  - Start making popcorn
  - Dim the lights
  - Put the screen down
  - Turn the projector on
  - Set the projector input to DVD
  - ...
Watching a DVD via Code

In code, this corresponds to manipulating a lot of different objects:

popper.on();
popper.pop();
lights.dim(10);
screen.down();
projector.on();
projector.setInput(dvd);
...

Plus, if you want to watch TV, you may need a way to undo these settings and then configure your system for TV viewing.

Facade to the Rescue

Let's create an object to simplify our interactions with the Home Theatre “sub system”:

For instance:

```
HomeTheaterFacade
watchMovie()
endMovie()
watchTV()
endTV()
playCD()
endCD()
```

We would now only call these methods and not interact directly with the individual components.
Implement `watchMovie()`

- `watchMovie()` would look something like this:
  ```java
  public void watchMovie(...) {
    popper.on(); popper.pop();
    lights.dim(10); screen.down();
    projector.on(); ...
  }
  ```

- While `endMovie()` would look something like this:
  ```java
  public void endMovie() {
    popper.off; lights.on();
    screen.up(); projector.off();
    ...
  }
  ```

Proxy

- The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

- Use the Proxy pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.

- Two common forms of the proxy pattern:
  - Remote Proxy: used in client-server programming; code on the client side interacts with a proxy object that forwards method invocations to an object on the server side.
  - Virtual proxy: some objects are expensive to create (example: large, high-resolution images); client code interacts with a proxy to avoid creating the expensive object for as long as possible.
Proxy’s Structure and Roles

- Proxy may handle request itself or forward to RealSubject.

```
Client
Proxy
RealSubject
```

```
subject
request()
request()
```

Example: Image Files

- On the class website, you can download code that implements the virtual proxy pattern.
  - An ImageProxy class is used to display an “Image Loading” message while image data is loaded in a background thread.
  - Once the image is loaded, the proxy delegates calls to the actual image.
- Note: if you compile this code on your own machine, you will need to modify the useImageProxy.java file to point to image files located on your computer.