19 Design Pattern: Observers

The next design pattern is a bit different than the last ones. It is more *architectural*, in the sense that it pertains to how classes are put together to achieve a certain goal.

The motivating scenario is as follows. Suppose we have an object in the system that is in charge of generating news of interest for the rest of the application. For instance, perhaps it is in charge of keep track of user input, and tells the rest of the application whenever the user does something of interest. Or, it is in charge of maintaining a clock, and tells the rest of the application whenever the clock ticks one time step. Is there a general approach for handling this kind of thing?

If we analyze the situation carefully, you'll notice that we have two sorts of entities around: an *observable* that is in charge of generating or advertising items of interest to the rest of the application, and the dual *observers* that are the parts of the application that are interested in getting the news.

(Sometimes, observables are called *publishers*, and observers are called *subscribers*.)

Think about the operations that we would like to support on observers, first. Well, the main thing we want an observer to be able to do is to be notified when a news item is published. Thus, this calls for an observer implementing the following interface, parameterized by a type `E` of values conveyed during the notification (e.g., the news item itself).

```
public interface Observer<E> {
   public void notify (E arg);
}
```

What about the other end? What do we want an observable to do? First off, we need to *register* (or subscribe) an observer, so that that observer can be notified when a news item is produced. The other operation, naturally enough, is to *notify* all the subscribers that a news item has been produced. When notifying a subscriber, we will also pass a value (perhaps the news item in question). This leads to the following interface that an observable should implement, parameterized over a type `E` of values to pass when notifying an observer.

```
public interface Observable<E> {
   public void registerObserver (Observer<E> ob);
   public void notifyObservers (E arg);
}
```

And that's it. These two interfaces together define the Observer design pattern.

Let's look at an example. Suppose that the observable we care about is a loop that simply queries an input string from the user, and notifies all the observers that a new string has been input, passing that string along as the notification value.

Here is the class for the input loop, implementing the `Observable<String>` interface.

```
import java.io.*;

public class InputLoop implements Observable<String> {

   private List<Observer<String>> observers;

   private InputLoop () {
      observers = List.empty();
   }

   public static InputLoop create() {
      return new InputLoop();
   }

   public void registerObserver (Observer<String> ob) {
      observers = List.cons(ob,observers);   // mutation
   }

   public void notifyObservers (String data) {
      FuncIterator<Observer<String>> it = observers.getFuncIterator();
      while (it.hasElement()) {
         it.current().notify(data);
         it = it.advance();
      }
   }

   private String getInput () {
     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      String response = "";

      try {
         response = br.readLine();
         if (response==null) {
            return "";
         }
      } catch (IOException ioe) {
```

```
            System.out.println("IO error reading from terminal\n");
            System.exit(1);
        }
        return response;
    }

    public void loop () {
        String response;
        while (true) {
            System.out.print("> ");
            response = getInput();
            notifyObservers(response);
        }
    }
}
```

Note that we are using an implementation of `List<A>` equipped with functional iterators. The code for `getInput` is boilerplate code that performs the necessary magical invocations required to read a string from the terminal. The `loop` method simply repeatedly queries a string from the user, and notifies all observers of that string. Note that there is no way built into the loop to actually terminate the loop. We'll see how to deal with that shortly. The observers are recorded in a `List<Observer<String>>`, which is initially empty. Registering a new observer is a simple matter of adding that observer to the list. Notifying the observers is a simple matter of iterating over the list, calling the `notify` method of each observer in the list.

Just to have something concrete, here is how we launch the loop.

```
InputLoop inLoop = InputLoop.create();
inLoop.loop ();
```

Of course, this does nothing useful. It simply repeatedly gets a string from the user, and does absolutely nothing with it.

Let's define some observers, than. The first observer is a simple observer that echoes the input string back to the user. Since it is an observer and we want it to work with the `InputLoop` class, it implements the `Observer<String>` interface:

```
public class EchoObserver implements Observer<String> {

  private EchoObserver () { }

  public static EchoObserver create () {
```

```
    return new EchoObserver();
  }

  public void notify (String arg) {
    System.out.println(" Input was: " + arg);
  }
}
```

All the action is in the `notify` method.

Another observer we can define is one that checks whether the input string is a specific string (in this case, the string `quit`), and does something accordingly (in this case, quit the application).

```
public class QuitObserver implements Observer<String> {

  private QuitObserver () {}

  public static QuitObserver create () {
     return new QuitObserver();
  }

  public void notify (String input) {
     if (input.equals("quit")) {
        System.exit(0);
     }
     System.out.println (" This was not a quit request");
  }
}
```

Finally, a more general observer than can print a response for any particular input.

```
public class ResponseObserver implements Observer<String> {

  private String seeThis;
  private String respondThat;

  private ResponseObserver (String si, String sa) {
     seeThis = si;
     respondThat = sa;
  }
```

```
  public static ResponseObserver create (String si, String sa) {
    return new ResponseObserver(si,sa);
  }

  public void notify (String input) {
    if (input.equals(seeThis)) {
      System.out.println(" " + respondThat);
    }
  }
}
```

Now, if we register those observers before invoking the `loop` method of a newly created `InputLoop`:

```
InputLoop inLoop = InputLoop.create ();
inLoop.registerObserver(EchoObserver.create());
inLoop.registerObserver(QuitObserver.create());
inLoop.registerObserver(ResponseObserver.create("hello",
                                                "Well, hello back to you!"));
inLoop.registerObserver(ResponseObserver.create("1+1","2"));
inLoop.loop();
```

we get the following sample output:

```
> this is a string
 Input was: this is a string
 This was not a quit request
> help
 Input was: help
 This was not a quit request
> hello
 Input was: hello
 This was not a quit request
 Well, hello back to you!
> 1+1
 Input was: 1+1
 This was not a quit request
 2
> quit
 Input was: quit
```

It is also easy to add an observer that recognizes URLs and reads off the corresponding web page. Here is such an observer, using some of the Java networking libraries:

```
import java.net.*;
import java.io.*;

public class URLObserver implements Observer<String> {

  private URLObserver () {}

  public static URLObserver create () {
    return new URLObserver();
  }

  public void notify (String input) {
    if (input.startsWith("http://")) {
      System.out.println(" Trying to read URL " + input);
      try {
        URL url = new URL(input);
        BufferedReader in =
          new BufferedReader(new InputStreamReader(url.openStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
          System.out.println(inputLine);
        in.close();
      } catch (Exception e) {
        System.out.println(" Error trying to read URL: "
                           + e.getMessage());
      }
    }
  }
}
```

Tossing it into the input loop:

```
InputLoop inLoop = InputLoop.create ();
inLoop.registerObserver(EchoObserver.create());
inLoop.registerObserver(QuitObserver.create());
inLoop.registerObserver(ResponseObserver.create("hello",
                                                "Well, hello back to you!"));
inLoop.registerObserver(ResponseObserver.create("1+1","2"));
inLoop.registerObserver(URLObserver.create());
inLoop.loop();
```

and trying it out:

```
> http://www.ccs.neu.edu/index.html
 Input was: http://www.ccs.neu.edu/index.html
 This was not a quit request
 Trying to read URL http://www.ccs.neu.edu/index.html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DT

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<title>
College of Computer and Information Science | College of Computer and Information Scienc
</title>
<link href="css/ccis.css" rel="stylesheet" type="text/css" />

<script src="/ccis/scripts/swfobject.js" type="text/javascript"></script>
<script type="text/javascript">
swfobject.embedSWF("flash/ccis_home_slideshow.swf", "flashcontent", "768", "213", "9.0.0
</script>



</head>

<body>

<div id="toplinks">
<span class="toplinks-inside"><a href="index.html" class="toplinks-link">CCIS Home</a></
</div> <!-- end toplinks -->

<div id="header">
<div id="logo">
<h1 id="header-image">
<a href="/ccis/index.html"><span></span>Northeastern University College of Computer and
</h1>
</div> <!-- end logo -->

<div id="usernav">
<ul>

<li><a href="prospectivestudents/index.html">Prospective Students</a></li><li><a href="p
```

```
</ul>
</div> <!-- end usernav -->
</div> <!-- end header -->
```

and it continues on for a long time.

Slightly harder is to implement an observer that can read a URL and extracts text that actually looks good printed out, by interpreting the HTML. (Try it if you're bored...)

The Observer pattern is central to much of GUI programming: the application is a tight loop (often called an *event loop*) that simply collects inputs from the user such as mouse movement, mouse button clicks, and key presses, and notifies its subscribers of those events. Those subscribers, which are graphical elements such as buttons and windows and the likes, react to those events that concerns them (such as a mouse click over their surface), and affect the application accordingly.

More complicated forms of observable/observer relationships can be layered on top of the basic pattern I described here. For instance, we may be interested in *unsubscribing* observers (which can have an interesting effect when this unsubscription happens in the context of a notification of another observer!), or we may be interesting in defining different categories of news that we can notify observers with, so that when an observer registers with an observable it gets to tell the observable what category of news it wants to be notified about.