21 Design Pattern: Publish-Subscribe

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 keeping 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: a *publisher* that is in charge of publishing or generating items of interest to the rest of the application, and the dual *subscribers* that are the parts of the application that are interested in getting these updates.

(The Publish-Subscribe design pattern is sometimes called the Observer design pattern, in which context publishers are called observables, and subscribers are called observers.)

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

```
public interface Subscriber<E> {
   public void getPublication (E arg);
}
```

What about the other end? What do we want a publisher to do? First off, we need to *register* (or subscribe) a subscriber, so that that subscriber can be notified when a news item is produced. The other operation, naturally enough, is to *publish* a piece of data, which should let every subscriber know that the data 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 a publisher should implement, parameterized over a type `E` of values to pass when notifying a subscriber.

```
public interface Publisher<E> {
   public void subscriber (Subscriber<E> sub);
```

```
    public void publish (E arg);
}
```

And that's it. These two interfaces together define the Publish-Subscribe design pattern.

Let's look at an example. Suppose that the publisher we care about is a loop that simply queries an input string from the user, and notifies all subscribers 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 `Publisher<String>` interface.

```
import java.io.*;

public class InputLoop implements Publisher<String> {

    private List<Subscriber<String>> subscribers;

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

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

    public void subscribe (Subscriber<String> sub) {
        subscribers = List.cons(sub,subscribers);   // mutation
    }

    public void publish (String data) {
        for (Subscriber<String> sub : subscribers)
            sub.getPublication(data);
    }

    public static 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 () {

        while (true) {
          System.out.print("> ");
          String s = getInput();
          publish(s);
        }
    }
}
```

Note that we are using an implementation of `List<A>` equipped with Java iterators — that is, a method with signature `Iterator<A> iterator ()` 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 subscribers are recorded in a `List<Subscriber<String>>`, which is initially empty. Registering a new subscriber is a simple matter of adding that subscriber to the list. Notifying the subscribers is a simple matter of iterating over the list, calling the `getPublication()` method of each subscriber in the list.

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

```
InputLoop il = InputLoop.create();
il.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 subscribers, then. The first subscriber is a simple subscriber that echoes the input string back to the user. Since it is a subscriber and we want it to work with the `InputLoop` class, it implements the `Subscriber<String>` interface:

```
public class Echo implements Subscriber<String> {
```

```
    private Echo () {}

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

    public void getPublication (String data) {
        System.out.println("Got: " + data);
    }
}
```

All the action is in the `getPublication()` method.

Another subscriber 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 Quit implements Subscriber<String> {

    private Quit () {}

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

    public void getPublication (String data) {
        if (data.equals("quit"))
            System.exit(0);
    }
}
```

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

```
public class Response implements Subscriber<String> {
    private String ifthis;
    private String thenthat;

    private Response (String it, String tt) {
        ifthis = it;
        thenthat = tt;
    }
```

```
    public static Response create (String it, String tt) {
        return new Response(it,tt);
    }

    public void getPublication (String data) {
        if (data.equals(ifthis))
            System.out.println(thenthat);
    }
}
```

Now, if we subscribe those subscribers before invoking the `loop()` method of a newly created `InputLoop`:

```
  InputLoop il = InputLoop.create();
  il.subscribe(Echo.create());
  il.subscribe(Quit.create());
  il.subscribe(Response.create("foo","bar"));
  il.subscribe(Response.create("1+1","2"));
  il.loop();
```

we get the following sample output:

```
> this is a string
Got: this is a string
> help
Got: help
> hello
Got: hello
> foo
bar
Got: foo
> 1+1
2
Got: 1+1
> quit
```

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

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

```
public class Url implements Subscriber<String> {

   private Url () {}

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

   public static void printUrlContent (String 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());
  }
   }

   public void getPublication (String data) {
      if (data.startsWith("http://"))
          printUrlContent(data);
   }
}
```

Tossing it into the input loop:

```
  InputLoop il = InputLoop.create();
  il.subscribe(Echo.create());
  il.subscribe(Quit.create());
  il.subscribe(Response.create("foo","bar"));
  il.subscribe(Response.create("1+1","2"));
  il.subscribe(Url.create());
  il.loop();
```

and trying it out:

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