The Observer Design Pattern

The last design pattern we will see this semester 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. (Not coincidentally, this is the scenario that occurs in Homework 7.) 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*, in charge of publishing 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 the news. (Other terms used for those entities are *observables* for *publishers*, and *observers* for *subscribers*. We will use those terms for consistency.)

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 (perhaps 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 *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.

```
public class InputLoop implements Observable<String> {

  public InputLoop () { }

  private LinkedList<Observer<String>> observers =
    new LinkedList<Observer<String>>();

  public void registerObserver (Observer<String> ob) {
    observers.add(ob);
  }

  public void notifyObservers (String arg) {
    for (Observer<String> item : this.observers) {
       item.notify(arg);
    }
  }

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

  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;
  }
}
```

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 `LinkedList<Observer>`, 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 = new InputLoop ();
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 PrintObserver implements Observer<String> {

  public PrintObserver () { }

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

All the action is in the `notify` method, as you can imagine.

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> {

  public QuitObserver () { }

  public void notify (String arg) {
    if (arg.equals("quit")) {
      System.exit(0);
    } else {
      System.out.println(" No, we're not quitting...");
    }
  }
}
```

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

```
InputLoop inLoop = new InputLoop ();
inLoop.registerObserver(new PrintObserver());
inLoop.registerObserver(new QuitObserver());
inLoop.loop ();
```

we get the following sample output:

```
> test
 The input was: test
 No, we're not quitting...
> this is a test
 The input was: this is a test
 No, we're not quitting...
> let's get out of here
 The input was: let's get out of here
 No, we're not quitting...
> quit
 The input was: quit
```

Now, have a look at Homework 7, especially the `Clock` class. While it does not implement the `Observable` interface, it could, by interpreting a `Callback` as an observer. (Indeed, a `Callback` is just like an observer, except it has an `action` method instead of a `notify` method, and no argument is exchanged.) The code is slightly more complicated there because we not only allow observers to subscribe to a publisher, we also allow subscribed observers to unsubscribe.

UML Class Diagrams

UML class diagrams are a convenient graphical notation to convey the relationship between various classes in a system. In particular, they are often used to represent design patterns.

Because I suck at drawing pictures in LaTeX, I will point you to the readings for this lecture on the web page.