The Observer Design Pattern

This lab exercise is worth double the others since it actually asks you to write a non-trivial amount of code.

I will give you two weeks (two lab sections - 6 hours) to finish it.

Executive Summary

In class, we have discussed the Observer design pattern. In this lab exercise, you will gain experience in using the Observer design pattern, in this case in the specific context of Java's implementation of the Observer design pattern in Swing.

Generally, as discussed in class, the Observer pattern addresses situations in which one object is dependent on another for state updates; that is the former object *observes* the latter object. The solution taken in the Observer pattern is for the *client* objects to register (i.e., *subscribe*) to be informed when the observed object changes. Recall from the end of the discussion in class, that it is possible to implement two modes of interaction in these cases: a *push* mode, in which the observed object sends the updated state information to the client object and a *pull* mode, in which the client interrogates the observed object after simply being informed of a change in state.

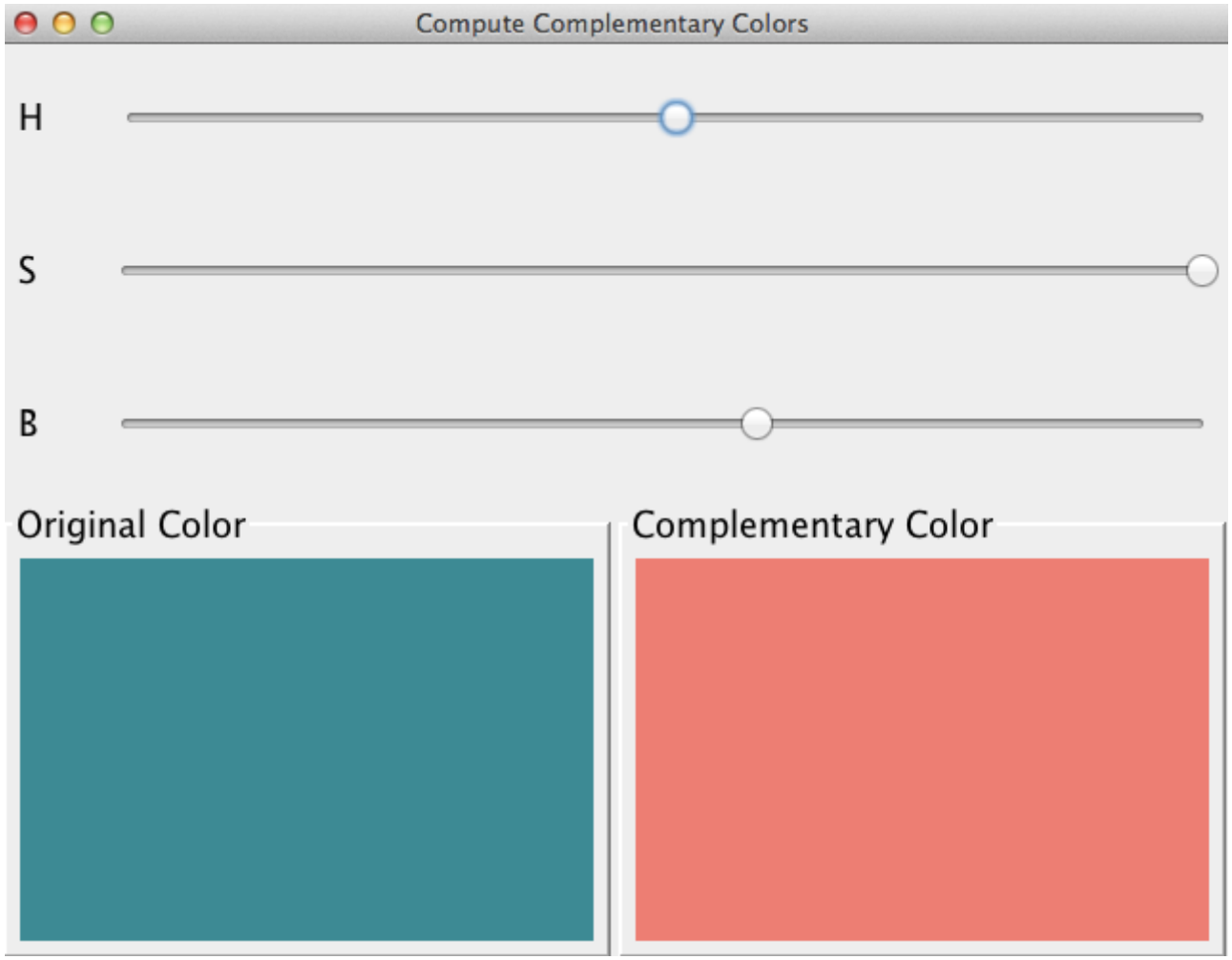
To complete this assignment, you're going to need to (1) draw some design documents (you may want to break out ArgoUML for those), (2) answer some questions (a PDF document with those answers and your UML diagrams together would be fantastic!), and (3) write some code (yeah, Java/Eclipse/whatever for that).

This tutorial is based on materials from Dr. Christine Julien.

The Observer Pattern and Swing

The Observer pattern is often used in the development of Graphical User Interfaces (GUIs). When a user of the interface interacts with some *widget* in the graphical representation of the application, various objects within the application may need to be informed of the change so that they can update the application's internal state. Swing introduces a new term for such clients: *listeners*; applications associate (register) listeners with any GUI components the user may interact with. Any component may have as many registered listeners as the application desires.

Let's build the following application:



This is a pretty simple application (and to be honest, I'm not even sure if it's right... I'm not an artist, and I gave up trying to perfect it). The application consists of three "sliders" that the user can use to adjust the hue, saturation, and brightness levels of a displayed color, which is then displayed in the left hand panel. The application then computes the "complementary" color and displays that color in the right hand panel.

Task 1

First consider a relatively naive implementation of the above. You can grab the following three java files and put them in a folder called observer1/ (or, more likely, in an Eclipse project called observer1).

ColorPanel.java

DisplayColors.java

SwingFacade.java

Examine these three classes. Create a UML class diagram that represents the classes, their internals, and the relationships between them. (You can leave out the SwingFacade class.) Your class diagram should also include the built-in Java classes ChangeListener and JPanel.

These files are also missing a small piece of functionality (albeit a key piece). Investigate DisplayColors.java, look at the comments and fill in the necessary functions. Specifically, you'll have to make use of the listener construct, and you'll have to do something with events when they fire.

Compile and run your application. If you're working in Eclipse you're on your own to make the packages and such all set up correctly. You've got this. If you're on the command line instead (like me... can't teach an old dog new tricks, apparently, don't forget that to compile classes in packages, you need to be at the directory *above* the package directory). Make sure it works (basically, make sure the colors change when you move the sliders around).

Keep this observer1/ folder/project with its correct code handy. You're going to turn it in.

A First Refactor

So this isn't terrible code, and it's actually quite common to see. But it's not at all a real implementation of the observer pattern. The problem here is that the observed object is basically observing itself and then dispatching its observations to the clients, instead of letting those clients take care of their own observations.

Task 2

First, let's create a second directory called observer2/ (or in an Eclipse project called observer2). Make a copy of the code you have in observer1 inside of observer2 (remember, we wanted to hold onto that observer1 code, so we're going to make our refactoring in its own project). Now let's think about the changes we want to make.

To be consistent with the Observer pattern, each of the interested components should register itself to receive the sliders' events. The question is, who is really the "client" in this interaction? If you answered "Aha! The ColorPanels!" you would be exactly correct.

First, let's refactor our *design*. Provide a new class diagram depicting a design that allows each of the interested observers to register themselves to receive the sliders' events. (Make sure you update the design with respect to the DisplayColors class, too... no need for it to be an observer anymore!)

There's a catch, though. I had to refactor the ColorPanel into two different extensions of the same abstract base class (I called them OriginalColorPanel and ComplementaryColorPanel if you want to follow my lead).

Alright, now fix the code. Notice that something cool happens when you do this. Before, the logic for updating the "original" color and for computing the "complementary" color were both embedded in the DisplayColors class. Where are they now?

Compile your code and run it.

A Second Refactor

So the thing I said above was cool. It actually is. The bonus can be succinctly described by the fact that the OriginalColorPanel doesn't have to know anything about the existence of the complementary color computation at all! Fantastic. But it's still kind of a bummer because both color panels are basically computing the original color (the OriginalColorPanel just displays it; the ComplementaryColorPanel uses it to compute the complementary color). This is pretty duplicative and gives the potential for introducing bugs if we forget to change it in one place when we change it in the other.

There are a couple of ways of refactoring this. Let's start with what I think is an intuitive interpretation of our components. What is the ComplementaryColorPanel doing? What's it's goal? It's basically to "mirror" the color displayed in the OriginalColorPanel, right? It actually isn't, intuitively, observing the sliders, but it's observing the OriginalColorPanel instead. Huh. That's kind of cool.

Task 3

Make it so. Refactor your design first. Then figure out a way to refactor your implementation to match your design. Do this one in an observer3 directory or project.

I used the PropertyChangeListener and the associated event (it gets fired automatically when a property of some object (in this case the background color property of an OriginalColorPanel object) changes).

This brings me to an important point. This was non-trivial for me to do. I relied heavily on the Java API. You know about that, right? Check it out here.

One last time...

I mentioned that there were a couple of ways we could consider doing that last refactoring. Another option is to employ something called the *Model-View-Controller*.

Task 4

Go research that. Use whatever resources you want, just be prepared to cite them (because we want you to). Thinking back to the result of the second refactoring, how might you refactor it differently to adhere to something more akin to the MVC "pattern"? Write a description of not more than 1/2 a page indicating how the MVC pattern might relate to our color changing problem. Draw one last UML diagram that sketches this design.

Hint: I find it useful to think about the DisplayColors as the view and the control and the Color as the model. Now, what's observable(s)? Who are the observers? What do they observe?

You don't have to implement this one (of course, you're always welcome to...)

What to Turn In

This one is going to be a little complicated. I want you to create an archive (a zip file for consistency). Call it LASTNAME_FIRSTNAME.zip.

At the top level of the archive, I want four things to appear:

- Report.pdf -- this should be a PDF version of the "reportables" in this tutorial. This includes the UML diagrams you generated, some brief descriptions to go along with them, and the final designed refactoring for the last task.
- observer1/
- observer2/
- observer3/

The last three are directories named exactly as I state above. Within each directory should be the .java files you created for that task. Each java file should have a package declaration at the top of the file that matches the directory the file is located in.

For example, if I were to turn in my current files, my directory would unpack like the following:

- Report.pdf
- observer1/ColorPanel.java
- observer1/DisplayColors.java
- observer1/SwingFacade.java
- observer2/ColorPanel.java
- observer2/ComplementaryColorPanel.java
- observer2/DisplayColors.java
- observer2/OriginalColorPanel.java
- observer2/SwingFacade.java
- observer3/ColorPanel.java
- observer3/ComplementaryColorPanel.java
- observer3/DisplayColors.java
- observer3/OriginalColorPanel.java
- observer3/SwingFacade.java