Tutorial 8: The Observer Design Pattern

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.

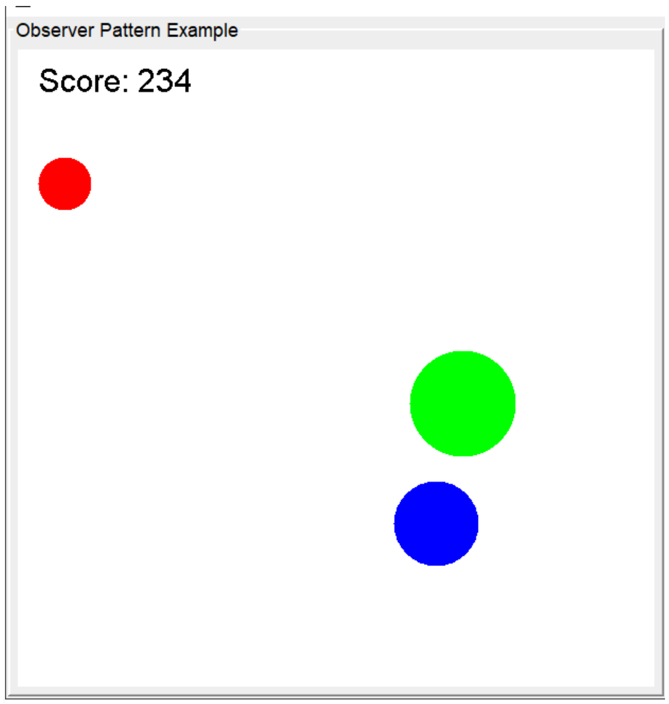
To complete this assignment, you're going to need to refactor original code to meet our requirements.

The tutorial is based on materials from Dr. Christine Julien, and redesigned by PAN Chao.

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 game with only three balls. The user controls the balls through the keyboard to prevent them from colliding. Space key to start or pause the game.

Balls	Key to action
Red ball	a or d: Swap xSpeed and ySpeed
Green ball	a: xSpeed becomes negative d: xSpeed becomes positive w: ySpeed becomes negative s: ySpeed becomes positive
Blue ball	All keys: change the direction of xSpeed and ySpeed

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 1

First, let's create a second package 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 keyboard's events. The question is, who is really the "client" in this interaction? If you answered " The Balls!" 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 keyboard's events.

There's a catch, though. I had to refactor the three balls into three different extensions of the same abstract base class Ball (I called them RedBall, GreenBall and BlueBall if you want to follow my lead).

So, we can regard the MainPanel class as the Subject class, and regard three Ball classes (RedBall, GreenBall and Blueball) that extends the Ball as the observer. So in the "Subject" class, you should design methods like "registerObserver()", "notifyObservers()", "removeObserver()" (if you need) and "measurementsChanged()" (if you need). On the whole, no matter how you design your project, you should guarantee that, **when you press the keyboard, it will notice the three ball classes and finally the xSpeed and ySpeed of three ball classes would be changed accordingly.**

All right, now fix the code. Notice that something cool happens when you do this. Before, the logic for updating the xSpeed and ySpeed were both embedded in the MainPanel class. Where are they now?

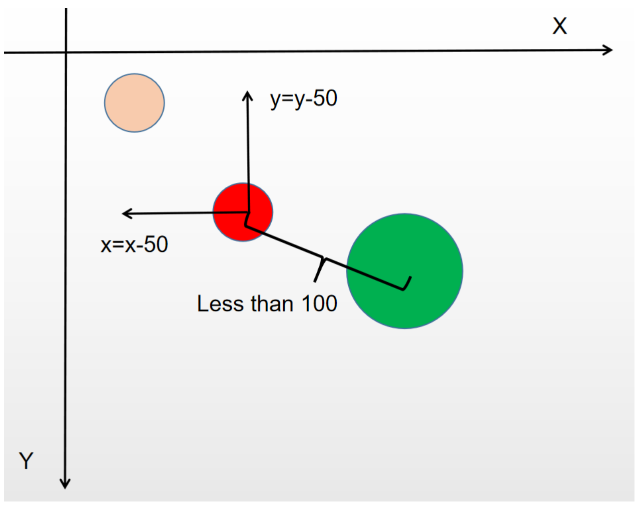
Compile your code and run it.

A Second Refactor

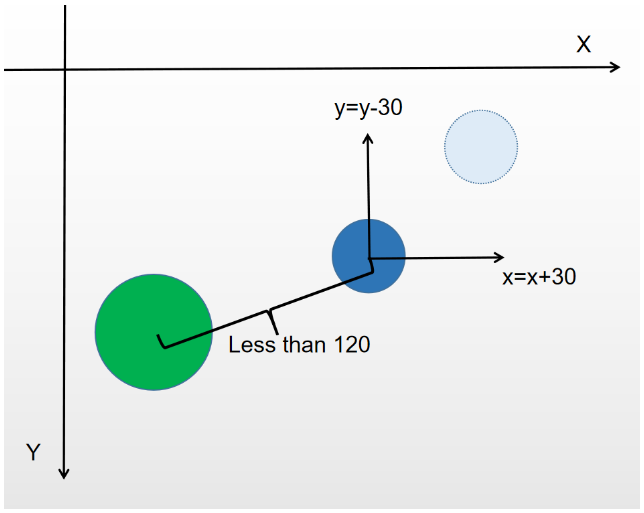
So the thing I said above was cool. It actually is. It can be succinctly described by the fact that the GreenBall doesn't have to know anything about the existence of the BlueBall and RedBall at all! Fantastic.

But it's still kind of a pity that the game is always over in a short time due to my poor game talent. I hope that BlueBall and RedBall can automatically stay away from GreenBall, **which means GreenBall should notify its location to BlueBall and RedBall when it moves,** then the BlueBall and the RedBall can **update** (shifting) their position automatically when the GreenBall is closed to them.

When the distance between the center of GreenBall and RedBall is less than 100, RedBall shifts the x and y by 50 units away from GreenBall.



When the distance between the center of GreenBall and BlueBall is less than 120, BlueBall shifts the x and y by 30 units away from GreenBall.



Task 2

Refactor your design first. Then figure out a way to refactor your implementation to match your design. In this case, the collision only be existed is between the RedBall and the BlueBall, and if the collision occurs, both of them would be set to invisible and the game is over.

Do this one in an observer3 directory or project.

What to Submit

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

The three things 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:

```
observer2/Ball.java
observer2/Main.java
observer2/MainPanel.java
observer2/RedBall.java
observer2/GreenBall.java
observer2/BlueBall.java
observer2/SwingFacade.java
observer3/Ball.java
observer3/Main.java
observer3/MainPanel.java
observer3/RedBall.java
observer3/GreenBall.java
observer3/BlueBall.java
observer3/SwingFacade.java
observer3/may be some other files
```