CUS1156 Lab 9

Design Patterns, part 1, Observer-Observable

Download the following files and make a project with them :

Bid.java, Bidder.java, Auction.java, and BidBus.java

Together, these make up a really simple simulation that uses the Java implementation of the Observer-Observable pattern.

1. Take a look at the code. Can you describe what this should do?

2. Describe, in English, not in code, what happens when a bidder makes a bid. Look at the makeBid method while answering this.

3. The BidBus class, which represents a bus layer for messages, is the Observable. When a bid is submitted, what two actions does the BidBus take to conform to the pattern? What should get called in the Observer as a result of these actions?

4. The Bidder class does not compile because a lot of code is missing. It needs a constructor and an update method. What calls the update method? Auction also does not compile.

5. Fill in those methods. The constructor needs to instantiate its instance variables and register with the Observable. The update method mainly needs to do some output. It should get the latest bid from the BidBus, call display, and then if the bidder's limit is hit, display a message to that effect. That message should only be displayed ONCE, the first time the limit is hit. What instance variable can you use to make sure that happens?

Design Patterns Part 2 : Singleton

The Singleton pattern is used when a particular class should only be instantiated ONCE, but is needed in many different methods in a software system. This is a common need for classes that manage a connection, say to a database or network. While this could be handled by creating one instance at startup and then passing the object around as a parameter, this rapidly becomes unwieldy in large systems. Also, since many developers may use the class, it Is important that it be used correctly, with only one instance permitted. The Singleton

pattern is used in this circumstance. Here are the components that turn a class into a Singleton class

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

Download the files Course.java, DeansOffice.java, FreshmenCenter.java, Registration.java, and University.java. Together, these classes make up a very simple registration system. The Course class keeps a simple count of the numbers registered. The Registration class keeps a list of courses. Since the dean's office and the freshman center both register students, it is important that both those classes are working with one shared registration object.

1. Write the rest of Registration.java. Fill in the two missing methods, withdraw and register. Run the program and write down what it does

2. Notice that a registration object is being passed to both FreshmanCenter and DeansOffice when doing scheduling. Remove those parameters and also remove the line in University.java that creates a Registration object.

3. Now, turn Registration into a Singleton. Make sure you include all the components described above. You will want to call the getInstance method right in the doScheduling methods of DeansOffice and FreshmanCenter. There are two ways to handle instantiation: eager instantiation and lazy instantiation. In eager instantiation, you create the Registration instance right where you declare it. Implement your Singleton like that and run it.

4. Now change to lazy instantiation. In this style, the instance is only created when it is first needed, which will be the first time the getInstance method is called. So in that method, test if the instance is null, and if it is, create the instance. Test the code again.

This is due next Tuesday in the usual fashion.