The follow are exercises to get you familiar with writing design patterns. We’ll post answers to these exercises on Tuesday.

1   Iterator

   BTree is a simple implementation of trees with arbitrary numbers of children, where a value is stored at each node in the tree. Write a method that returns an implementation of java.util.Iterator that returns the value at each node using a pre-order traversal. Write another method that returns an iterator doing post-order traversal instead.

2   Visitor

   Expression is an abstract superclass for arithmetic expressions on integers, where each expression can have up to two arguments. There are various subclasses of this class, including Constant, for representing integer constants; AddExpression, for representing the sum of two expressions, etc.

   Write two visitors for this class. The first will print out an expression, and the second with compute its result. To do this, you will have to write accept methods for each subclass of Expression, and write two implementations of ExpressionVisitor, one for printing expressions (called PrintVisitor), and one for computing expressions (called ComputeVisitor). Each visitor should maintain its own state that it accumulates while visiting. For printing, this will be a string, while for computation this will be an integer, holding the result of the computation. Each visitor will define a method getResult that will return its state.

   Try writing these visitors in two ways (you’ll have to copy the code into two separate directories). First, do internal traversal. That is, have the accept method in each subclass of Expression invoke the visitor on each of its children (if it has any). Second, do an external traversal, within the visitor. In this case, the accept method will simply call the visitor on itself, while the calls to visit
an Expression subclass’s children will occur as part of the visit method for that kind of expression. See the lecture notes for more description on these two kinds of traversal.

3 Decorator

The StringGenerator interface has a single method getString, which returns a String. One concrete implementation of StringGenerator is ConstString, which simply returns the same string each time it is called. Write two decorators for StringGenerator. The first, called EscapeQuotes, should, when its getString method is called, retrieve the string s from the StringGenerator it decorates, and then create a new string that is the result of replacing all occurrences of " in s with \" instead. The second, called MakeBold, should wrap prepend s with <b> and append to it </b>.

4 Singleton/Typesafe Enum

Create a type-safe enumeration class called Color that represents the six colors red, blue, green, yellow, purple, and orange. Follow the example for suits of cards given in the lecture notes, so that Color instances have a toString method. (Question: do you need to define a separate equals method? Why or why not?).

Think about adding a compareTo method having the signature int compareTo(Color c), which could implement the following ordering: red < blue < yellow < purple < green < orange. In what ways might you do this? Think about doing this by defining new subclasses to implement the colors, as opposed to just keeping the class that you have.

5 Proxy

Create a proxy for StringGenerator objects described above, called SecureGenerator. This class should have an extra method (that is, in addition to the getString method) called void authenticate(int key). By default, the SecureGenerator.getString method always return the empty string "". However, if the user calls authenticate with the “magic number,” then the proxy will from then on delegate calls to getString to the object that it’s a proxy for.

Question: can you characterize the difference between a proxy and a decorator?

6 Abstract Factory

Project 2 used abstract factory in the GraphFactory class. If you wanted to make this factory potentially return multiple implementations of Graph, how
might you recode it? If you were to change it, assuming that your JUnit tests used the factory, will they need to be modified?

7 State/Strategy

Look at the movie example from the refactoring lecture, and see how they transform the switch statement for price codes to use the Strategy pattern. In particular, define a Price interface which has a single method double getPrice(int daysRented). Then implement three subclasses of Price: RegularPrice, NewPrice, and ChildrenPrice. Each one of these will implement getPrice differently (look at the example for possible equations). Then define a Movie class which has two fields, the name of the movie, and the Price. Provide a get method for the name, and get/set methods for the price. Finally, define a Rental class with two fields, the Movie and the number of days it was rented. This class will have a single method int amountFor() that delegates its work to the Price of its Movie object.

Notice here that we have not ever used a switch statement—the control comes from the decision of which Price object to initialize our movie with. Moreover, we can change its price code later, if we wish, using the set method. This is the key benefit of strategy: the algorithm is not baked into the object, but can be switched at runtime.

Finally, is there ever a need to have more than one NewPrice, ChildrenPrice, or RegularPrice object? How can you use singleton to ensure that at most one of each is ever created?

8 Observer

You have already written an Observer, as explained in the class notes. Look over your code, and understand why it implements this pattern.