Comp 401 – Assignment 11:

Delegation, Adapter and Observer

**Date Assigned: Wed Nov 9, 2011**

**Completion Date: Fri Nov 18, 2011**

**Early Submission Date: Wed Nov 17, 2011**

In this assignment you will change your parser to delegate rather than inherit from the scanner and write observable list and Bean objects that are automatically refreshed by ObjectEditor. To write the observables, you will define adapters for existing Java collection classes. You will also be answering a couple of questions.

## *Converting an Inheriting Class to a Delegating Class*

Currently your parser is a subclass of your scanner. As a result, it has all the properties of the scanner. Change your parser so that, instead of using inheritance to use the code of the scanner, it now uses delegation. Thus, your parser must no longer be a subclass of the scanner. Instead, it should have a reference to the scanner. The interface of the parser remains the same except that the parser no longer defines the property the” list of tokens” property. Thus, it has only the input string property, and the optional error property. It will be easiest to make the parser instantiate the scanner. However, you can also define a parser constructor that takes the scanner as a parameter. In this case, you will have to change the code instantiating the parser.

## *Property Change Events*

Currently, you must do explicit refreshes of the simulation to see the results of a command processed by the parser. Use your new knowledge of observables, property change events, and ObjectEditor behavior to automatically update the avatar in response to executionof the command to move an avatar. You don’t have to create new subclasses to add this functionality – you can directly change existing classes. Try to make sure that the code you write to announce property change events is shared by as many classes as possible – ideally a single class should change to move the complete avatar (with its chat history).

Some of you have moved an avatar by creating some or all parts of the avatar in the new position. As mentioned earlier, this solution is inefficient. Moreover it will be more difficult and expensive to automatically refresh such avatar “movements”. You should move an avatar by moving all of its components and announce appropriate changes to these components. Also, you should aim for efficiency by not sending coarser-grained notifications than necessary.

## *Automatic Refresh of Element Addition/Removal*

Announce changes to the collections to automatically refresh the display when elements are added or removed from then. To implement this part, use the information in the class material about events of variable sized collections processed by ObjectEditor. The recitation should have provided you with a template for doing this part.

## *Debugging Refresh Problems*

If you feel ObjectEditor is not automatically refreshing some changed object on the screen, please contact us after going through the following check list:

1. Does the class of the object define a registration method?
2. If it does, is some ObjectEditor calling the method?
3. Is it sending the change information to ObjectEditor.

You can use print statements or break points to answer these questions.

Send us the answers to these questions along with email explaining the problem. It is possible that ObjectEditor has refresh bugs, so do not feel shy about sending us the email.

If a user command results in at least one notification, ObjectEditor does not do any refreshes, and assumes that all changes will be announced to it.

## *Constraints*

1. Property change notifications should be sent only by atomic shapes.
2. If the same object registers multiple times as an observer, do not send it multiple notifications. This means your list of observer lists should be sets.
3. As the types List, ArrayList, and Vector have now been covered in class, you are free to use them. Build appropriate *adapters* for them rather than using them directly. Ideally, you should use implement one adapter type (interface and collection) for the two kinds of observer sets.

## *Questions*

Answer these questions in the remarks section of blackboard so that the TAs can easily retrieve them.

1. Explain why the parser should use delegation rather than inheritance to reuse the code of the scanner. Give as many reasons as you can.

The parser should use delegation instead of inheritance because using inheritance does not follow the rules of encapsulation and least privilege: the Scanner only needs to return the tokens. By using inheritance, all of variables and properties of the scanner are visible to the parser. Using inheritance can cause problems when we modify the parser class, and in the process, the good-intentioned programmer can actually modify contents of the scanner code and completely break the functionality of the superclass in the subclass. By creating a has-a relationship rather than an is-a relationship in this case is also more intuitive, as the parser that is written in my program really only uses the token history created by the scanner class, and that is all that the parser really needs to access, so the parser has-a scanner that it accesses, but it is not a scanner. The scanner has also gone through many design revisions and has methods that aren't really applicable to the parser functionality. By delegating the scanner, we provide better encapsulation and prevent the programmer from accidentally calling a method from the scanner that was not intended to be called. Using delegation in this case is far more “object oriented” because it breaks up the functionality of the code, and makes the parser more reusable. Delegation also saves us from some of the confusion of overriding methods and can make the code easier to understand because we can see all of the code at once rather than running through a tree of all of the classes that the parser would otherwise inherit from. From a design point of view, it is sometimes confusing for me to have a variety of inherited classes and have to search through three different classes to find the code for a particular method.

1. Explain why the parser, unlike the new collection classes you define, is not an adapter.

The parser is not an adapter because it does not convert the token scanner class into something that is compatible with ObjectEditor. Object Editor is the interface of the parser, and the parser is the controller. The parser defines the methods to change the values of the various atomic shapes that are displayed by ObjectEditor, but because ObjectEditor does not know that these values change, the interface and the controller need an adapter so that they can be compatible and so that they can effectively talk to one another. In this case, the collection class defined that adds in listeners, and these listeners are the adapters that notifies ObjectEditor of changes is the adapter between these two that enables proper functionality.

1. What is the class of the ObectEditor observer that for a (a) property change event, and (b) a list change event? How did you determine these classes? (Hint: as mentioned in class, the getClass() method can be invoked on any object to determine its class, which can then be printed using println().
   1. The class of the object editor observer for a property change event is the bus.uigen.oadapters.TextShapeAdapter or .OvalAdapter or .LineAdapter or whatever other atomic shape I am displaying.
   2. The class of the object editor observer for a list change event is util.models.VectorChangeEvent
   3. I determined these classes by using the getClass method on the listeners added for both the PropertyChangeListener and the VectorListener and printing them out using a println statement as described in the problem. I then scanned through the console to find where they had been printed out amidst everything else I was using print lines for.