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. (When an object has only one property, the property name is now shown as it is implicit). 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 execution of 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 them. 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. Does the class follow the conventions associated with the registration method – collection conventions for `addVectorListener()` and Bean conventions for `addPropertyChangeListener()`? (A class could follow both)?
3. If it does, is some ObjectEditor calling the method? You can use print statements or break points to answer this question.
4. Is it sending the change information to ObjectEditor.

To determine if the class follows the appropriate conventions, put the following annotation before the class declaration:

```java
@util.annotations.StructurePattern(util.annotations.StructurePatternNames.VECTOR_PATTERN)
@util.annotations.StructurePattern(util.annotations.StructurePatternNames.LIST_PATTERN)
@util.annotations.StructurePattern(util.annotations.StructurePatternNames.BEAN_PATTERN)
```

depending on what convention(s) the class is following.

To print out ObjectEditor warning messages:

```java
import util.misc.Message;

Insert

    Message.showWarnings(true);

before

    ObjectEditor.edit(object)
```

and see what warnings you get. You will probably get a warning that implies you don’t have the correct conventions for the structure pattern or the registration method.
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 for atomic shapes, as discussed in class. (For example, when a knight moves, do not send ObjectEditor the whole knight – instead send it the new location of each atomic shape that moved.) Collection change events, on the other hand, can be sent for non-atomic shapes. Thus, an insert event can (and must) send the whole knight that was added.

2. If the same object registers multiple times as an observer, do not send it multiple notifications. This means your lists of observer should be sets.

3. As the types List, ArrayList, and Vector have now been covered in class, you are free to use them for this and future assignments. 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.

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

3. What is the class of the ObjectEditor 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().)