Comp 401 - Assignment 8: Observer Pattern

Early Submission Date: Wed Oct 25, 2017 (+5%)

Completion Date: Tue Nov 14, 2017

First Late Day: Fri Nov 17, 2017 (-10%)

Second Late Day: Tue Nov 21, 2017 (-25%)

In this assignment you will learn how to write observable Bean objects that are observed by both ObjectEditor and an observer object you will write. As ObjectEditor now observes them, there will be no need for you to explicitly call the ObjectEditor refresh() method. In fact, you are no longer allowed to call this method, and should *no longer get a warning of the form*:

W***Refreshing complete object: ….

As mentioned in class, you should aim for efficiency by not sending coarser-grained notifications than necessary and not creating new shapes in getters.

The following new material is relevant to this assignment. The assignment should be trivial if you read and understand this material and may be impossible if you do not.

MVC	PowerPoint PDF YouTube Mix	Docx PDF Drive		MVC Checks File	lectures.mvc.monolithic Package Git (Monolithic) lectures.mvc.interactor Package Git (Interactor) lectures.mvc Package Git (MVC)
Component Notifications	PowerPoint PDF YouTube	Docx PDF Drive			lectures.mvc.properties Package Git (Propertie) lectures.mvc.collections PackageGit

	Mix				(Collections)

Part 1: Observable Bean Announcing PropertyChangeEvent

Transform each of your atomic shape classes into an observable bean that announces property change events. Study the class material to understand what this exactly means. Some of the steps you have to take include:

1. Making sure that each atomic class implements the interface PropertyListenerRegisterer (import util.models.PropertyListenerRegisterer);.
2. Storing each registering PropertyChangeListener in a readonly property called *PropertyChangeListeners* of type *List* (import java,util.List).
3. Making each setter of a visible property announce an appropriate PropertyChangeEvent (import java.beans.PropertyChangeEvent) instance to every observer in the list.

A property is visible if ObjectEditor sees it, that is, it has not been hidden by the @Visible annotation.

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 – changing just the locatable and bounded shape classes should take care of most of the changes you have to make.

As mentioned above, you can store the list of observers presented in class, but do not reference JavaTeaching directly – your project must be self contained.

Part 2: Observing Console Scene "View" and Factory Method

Create a class, tagged "*ConsoleSceneView*," that prints on the console each property event announced by each atomic shape (such as left arm, head, right leg) in the logical structure of the scene object. Specifically, this class:

1. Implements the java.beans.PropertyChangeListener interface.
2. Provides a parameterless constructor that registers `this`, the current instance, as a listener of each atomic shape in the scene. The scene is retrieved using its factory method.
3. Uses println() to display on the console each PropertyChangeEvent it receives from the atomic objects being observed. You should simply pass the PropertyChangeEvent to a println() call, do not do your own custom printing of the event, as our tests need to process the output.

This class is not a very interesting view because it does not really display the scene in a meaningful way, but it does have the characteristics of such a view – hence the name. This class is much like the view class we saw in the praxis for this material.

Like the bridge scene, this class will be a singleton, returning only one instance. In the singletons creator factor class, define a parameterless static factory method, tagged "consoleSceneViewFactoryMethod", that gets and possibly creates (the first time it is called) an instance of the console scene view class.

Part 3: Animating Demoing Main Class

To demonstrate your observable and observables, write a main class that creates a scene object and displays an animation of it using both the console scene view and ObjectEditor. Specifically, the main class:

1. Gets the scene object using the associated factory method.
2. Displays it using ObjectEditor.
3. Displays" it using your console scene view, which it gets using the factory method. Just getting it will cause the constructor of the view to register itself as a listener. The main method does not have to do anything with the returned value and need not even assign it to a variable.
4. Creates an animation that moves an avatar, sets its text, and rotates each of its rotatable parts (if you did extra credit). You should not call the OEFrame refresh method in this assignment and thus should not get any warning about invoking this method.

If you have followed all of the constraints of this and previous assignments, the avatar should animate without the refresh method, and every change to a visible shape property should be printed by the console view.

In particular, the animation will not work if you have moved an avatar or rotated an avatar limb by creating new parts of the avatar, instead of directly moving or rotating existing avatar components. As mentioned before, such a solution is inefficient and is prohibited.

Extra Credit

You will get extra credit for rotating a limb using property notifications. The rotate() method will now result in notifications of height and width.

Debugging Refresh Problems

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

1. Is your console view printing out the event from the object you think ObjectEditor missed?

2. Does the class of the object implement the PropertyListenerRegistarar interface?
3. If it does, do the ObjectEditor object and your console view call the registration method defined by the interface? You can use print statements or break points to answer this question.
4. Is the registration method adding observers to a list?
5. Is the object sending the change information to ObjectEditor and other observers in its setters?

ObjectEditor frame may occasionally flicker – because of the amount of work it does, it cannot keep up with a high number of property changes.

Additional Constraints

Make sure that each class implements a single interface, which could extend multiple interfaces. This may mean that you will create empty interfaces that do nothing else beyond extending existing interfaces. This requirement reduces the need for casting

As mentioned above and in class, you should not be sending notifications about composite shapes. For example, when an avatar moves, do not send an observer such as ObjectEditor a notification about the avatar object– instead send it the new location of each atomic shape that moved. For the graphics to update, we require you to send notifications only about properties of atomic shapes (changes in the height or width of a line shape, etc).

You should not be sending notifications about invisible properties - you might have to override inherited code from your locatable superclass to do turn these notifications off, which is ok, it implies invisible properties require special handling. If you do not follow this rule, you will get a message of the form: "***Received notification(s) for unknown (possibly invisible) property: x of object: avatars.AnAvatar@44a8253a. Updating complete object."This message implies that you do not know if the update to the screen occurred because of some correct or incorrect notification. To convince yourself and us that you meet the requirement of announcing changes to each atomic shape, get rid of these warnings. Often there is no need really to have invisible properties as you can simply not show the main panel.

You should announce changes only to those properties that ObjectEditor knows about. It does not, of course, know about invisible properties. In addition, it does not know about non-standard properties of atomic shapes such as angle and radius. So your setAngle() method should call notifyAllListeners to announce changes in the known Width and Height properties, instead of changes in the unknown Angle property.

As mentioned above, do not call the OEFrame refresh method.

Be sure to follow the constraints of the previous assignments.

Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- Be sure to follow the conventions for the name and package of the main class so that its full name is: **main.Assignment8**.

Good luck!