Comp 401 - Assignment 9: Toolkits and Graphics

Early Submission Date: Wed Nov 1, 2017 (+5%)

Completion Date: Tue Nov 28, 2017 (After Thanksgiving)

First and Last Late Day: Fri Dec 1, 2017 (-10%)

In this assignment, you will create your user-interface to display and manipulate the scene, and thus, will no longer rely on ObjectEditor to do so. In this process, .you will learn to implement a graphics view and controller. For additional extra credit, you can embellish the user interface in many ways. Some of the extra credit, labelled (Toolkit), requires you to look at the material on MVC and toolkits, which has not been covered in class and will not be tested in the final exam.

The following new material is relevant to this assignment. Again, the key is understanding the relevant material. Once you do so, it should be straightforward.

The class material does not give details of the Graphics API. The shape parameters it assumes are slightly different from those ObjectEditor uses – for lines, images, and string. So, you will have to do some translation for these. Look at the online Java API documentation for Graphics to learn its details.

MVC and Toolkits	PowerPoint PDF YouTube Mix	Docx PDF Drive		- Toolkit and Graphics Checks File	lectures.mvc.toolkit Package Git (Toolkits)
MVC and Graphics	PowerPoint PDF YouTube Mix	Docx PDF Drive		-	lectures.mvc.graphics Package Git (Graphics)

Regular Credit: Inheriting Bridge-Scene Painter and Factory Method

Create a view class, tagged *"InheritingBridgeScenePainter,"* that displays the bridge scene and reacts to changes to models in the scene. This scene will be very similar to the console view you implemented in the last assignment, except that it will call a paint method in response to receiving a property change event. This means that the view class is a (direct or indirect) subclass of Component and implements the standard paint() method to draw all graphical objects in the logical structure of the scene model. The view object should register itself as a listener of all the model objects it paints. It should repaint the entire scene even if only part of it (say the arm) changes. This is inefficient but no worse than what OE and most applications do. (There is a way to paint only part of the scene; you override the update rather than paint method of a component. The advantage of overriding paint() is that you do not have to erase the previous contents of the component before redrawing – the whole components is automatically cleared before paint() is called. Overriding update would require you to clear the area you are redrawing before doing the redraw, which is tedious to program.)

Like the console view, it implements a parameterless constructor that registers itself as a listener of the atomic shapes in the bridge scene. It can provide other constructors and arbitrary properties.

Your view should ignore the OE annotations in the models it displays, but should render each of the required properties of a shape correctly. You are free to consider also the optional properties of the avatar such as color, stroke, and font.

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

If you sailed through the previous assignment, you may want to create the alternative extra credit implementation given below.

Alternative to above (Extra Credit): Observing Bridge-Scene Painter and Factory Methods

Instead of using the above approach to create the view, fix a problem in AWT/Swing – there is no notion of a paint listener. Define a paint listener interface*, tagged "PaintListener,"* which should include a method with the following signature:

void paint (Graphics2D g)

All classes that implement this interface should also have the tag of the interface. Now create a subclass of Component, *tagged "ObservableBridgeScenePainter,"* that provides a method, tagged "*addPaintListener*", to register instances of this interface, and a readonly property, called, *PaintListeners*, of type List (import java.util.List) that stores the list of registered paint listeners. You can use *ArrayList* to implement this property. This class will also override the inherited paint(Graphics) method, which gets called with the proper Graphics argument whenever the (inherited) repaint() method is called on this class. The paint(Graphics) method must call the paint(Graphics2D) method in each of the registered paint listeners.

You will no longer have a single monolithic view object that paints all of the objects in the scene. Instead, you will create multiple view objects –one for each avatar and one for the background. (You are free to create even finer-grained view objects). A view object will no longer be a subclass of some window class. Instead, it will be an observer or listener of the observable painter subclass you created above (it will be a PaintListener, in other words). At the same time, the view object must receive property change events from the model objects it paints and thus will also register itself as a listener with these objects. When it observes a property change event, it informs the observable painter by calling its repaint() method, and the observable painter, in turn, orders all of the paint listeners to update by calling their paint methods.

The order in which the view objects get registered with the observable painter will determine whether a drawing is on top or bottom of another because it will determine the order in which the paint methods are called. The observable painter will call the paint methods in all of the paint listeners in the order in which they are registered.

The observable painter class should define a constructor with no arguments. Define a parameterless static factory method, tagged "observableBridgeScenePainterFactoryMethod", that gets and possibly creates (the first time it is called) an instance of this class.

Finally, create a separate class, tagged, "*DelegatingBridgeSceneView*", that accesses the ObservableBridgeScenePainter, instantiates the view objects, and makes them listeners of the observable painter. The word delegating means that it delegates its tasks to other objects.

Define a parameterless static factory method, tagged "delegatingBridgeSceneViewFactoryMethod", that gets and possibly creates (the first time it is called) an instance of this class.

Bridge Scene Controller (Required)

Define a controller that listens to the mouse and key events of the window displaying the bridge scene. (This window will be the inheriting bridge scene view if you did not do the extra credit and the observable painter if you did.) The controller should keep track of the position of the last mouse click. Let us refer to this location as the last click point. (You can get this location by calling the getPoint(), getX(), getY() methods on a MouseEvent) If the user types the letter 'a', 'g', 'l', or "r' in this window, then the Arthur, Galahad, Lancelot, or Robin avatar, respectively, should move to the last click point. If the user types the letter 'o' then all avatars should return to their first positions, that is, the positions at the start of the scene simulation. Tag this class as *"BridgeSceneController."*

In case your key listener does not receive key events, then call setFocusable(true) in the constructor of your component subclass. If this also does not work, make this class a subclass of Panel instead of Component. My experience is that these two steps are not necessary if a frame has only one input component, but I may be wrong.

This class needs to access both the BridgeScene and the AWT/Swing component to which it listens. Define a single-parameter constructor that takes the component as an argument. This component is either the inheriting bridge scene view or the observable painter, both of which are available from factory methods.

In SingletonsCreator, define a parameterless static factory method, tagged "bridgeSceneControllerFactoryMethod", that gets and possibly creates (the first time it is called) an instance of this class. The factory method for this class should pass the appropriate component to the controller by calling the appropriate factory method for the component. Thus, you see that it is useful to make factory methods call other factory methods.

Sharing Observer Registration Code

You can share between the console view of the last assignment and the inheriting view of this assignment the code for registering a PropertyChangeListener with the atomic shapes in the bridge scene logical structure. Since they do not share a common super class, you may want to define some static methods. For example, in the Angle or V Shape, you can define a static method of the form:

```
public static void addPropretyChangeListener(Angle anAngle, PropertyChangeListener
aListener) { ..}
```

that calls the instance addPropertyChangeListener(aListener) method on both the left and right line. The avatar and bridge scene objects can have similar methods.

Extra Credit (Toolkit): Command Interpreter Controller and User Interface

Implement, using Swing, a user interface to manipulate the command interpreter. Use the model-view-controller design pattern to do so. This means you must write a controller object that calls the setter of the command interpreter. How the manually implemented user-interface looks is entirely up to you. You can use the widgets discussed in class or other widgets to create your user interface. The minimum requirement is that you provide a JTextField to set the string manipulated by the interpreter. The class material on the manual BMI Spreadsheet shows the use of JTextField. Tag your controller as "*CommandInterpreterController".* It should provide a readonly property, TextField, for getting the JTextField.

In SingletonsCreator, define a parameterless static factory method, tagged "commandInterpreterController FactoryMethod", that gets and possibly creates (the first time it is called) an instance of this controller class.

Extra Credit (Toolkit): Command-Interpreter View and Factory Method

If your interpreter has a read-only errors property, then for extra credit, display it in your command interpreter user interface. This means you must now make the command interpreter an observable also and include a view that observes this object. Tag the view as "*CommandInterpreterView*".

In SingletonsCreator, define a parameterless static factory method, tagged "commandInterpreterView FactoryMethod", that gets and possibly creates (the first time it is called) an instance of this view class.

Add the additional tag, *"ObservableCommandInterpreter"* to the command interpreter if you do this part. Your existing factory method for this class now returns this improved command interpreter.

If you do this step, your command interpreter view and controller must share a JFrame and some component must add components to it. One approach is to make the view create the entire user interface and provide getters that the controller can use to add itself a listener to relevant components of the view. Ideally these getters should be in a separate interface on which the controller depends. The interface of the view can be an extension of this interface. The controller can call the getters in its constructor.

Extra Credit (Toolkit): Action Listeners

Add to your command interpreter controller at least two "action components". One action component should be a JMenuItem and the other a JButton. Each of them should perform some canned (pre-programmed) action on the simulation. For instance the menu

item could move all of the avatars by some distance in the x direction and the button could do so in the y direction. Define readonly properties, “MenuItem” and “Button” to return the instances of JMenuItem and JButton, respectively.

Extra Credit (Toolkit): Animation Progress Bars/Sliders

Connect a progress bar (JProgressBar) or slider (JSlider) for your main method that shows to what extent your animation in the main method has completed. You do not have to use MVC to display the progress bar as it will be too much work for the TAs to check that. However, if you have spare time, you are encouraged to use it. Tag the class (which could be main) that creates the progress bar as “ProgressBarCreator” and define a static readonly property called “ProgressBar” or “Slider” in the class that returns the JProgressBar or JSlider;

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 painting scene view and ObjectEditor. Specifically, the main class:

1. Accesses the scene object (using the associated factory method).
2. Displays it using your inheriting or observing view object.
3. Calls the controller factory method to make the display interactive.
4. Displays the scene using ObjectEditor.
5. Accesses the command interpreter object.
6. Displays the custom user interface for the command interpreter (extra credit).
7. Creates an animation that moves an avatar, sets its text, and moves the avatar.

Your main code should call methods only in the models. This means that it will not cause any changes to the controller widgets such as the text field you created for entering commands. The TAs will test this text field manually. Changes made from the application program to a widget do not cause listener events to be fired. For example, calling setText() on a JTextField does not cause the listener event to be fired. Thus, calling widget methods in the main program does not help you demonstrate your widget listeners

Similarly, The TAs will manually interact with your scene windows to test your mouse and key controller.

Constraints

1. You should use the MVC pattern for all user-interfaces except the progress bar.
2. As before, there should be no warnings from ObjectEditor – if there are spurious warnings, let me know.

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 (main.Assignment9).

Good luck!