Comp 401 - Assignment 9: Toolkits and Graphics

Date Assigned: Thu Oct 22, 2015

Completion Date: Fri Oct 30, 2015

Early Submission Date: Wed Oct 28, 2015

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	lectures.mvc.toolkit Package
MVC and Graphics	PowerPoint PDF YouTube Mix	Docx PDF Drive		-	lectures.mvc.toolkit Package

Regular Credit: Inheriting Bridge-Scene Painter

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.)

This class should provide a constructor that takes the BridgeScene as an argument and registers itself as a listener of the atomic shapes in the screne. 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.

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

Instead of using the above approach, illustrated in class, 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 registers instances of this interface. Whenever the Java paint(Graphics) method is called (by repaint()) in this subclass, it calls 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. 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.   A view object will receive property change events from the model object it paints and thus will also register itself as a listener with these models. It will call the repaint() method in the observable painter, which in turn will call the paint methods in all of the paint listeners.

Bridge Scene Controller

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 original positions. 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 should provide a constructor that takes the BridgeScene as an argument and stores a reference to the model. It can provide other constructors and arbitrary properties.

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.

Those who have created composite objects as shapes may be tempted to have instance addPropertyChangeListener(aListener() methods that add observers to themselves and their children. This is conceptually wrong, as adding a listener to a composite object does not imply adding the listener to its descendants. Observers such as ObjectEditor would end up being added multiple times to an observable, which can cause inefficiencies and even errors.

Extra Credit (Toolkit): Command Interpreter 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 like 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 constructor that takes the CommandInterpreter as an argument. It should also provide a readonly property, TextField, for getting the JTextField. It can provide other constructors and properties.

Extra Credit (Toolkit): Command-Interpreter View

If your interpreter has a read-only error 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. Add the additional tag, *"ObservableCommandInterpreter"* to the command interpreter if you do this part.

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 JTextField 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 are 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. Instantiates a scene object.
2. Displays it using your inheriting or observing view object
3. Displays it using ObjectEditor.
4. Instantiates a command interpreter object.

5. Displays the custom user interface for the command interpreter (extra credit).
6. 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 MVC 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.

Good luck!