Comp 401 - Assignment 9: Window-based MVC

Date Assigned: Wed Oct 24, 2012

Completion Date: Fri Nov 2, 2012

Early Submission Date: Wed Oct 31, 2012

In the previous assignment you first created (observable) models, and then created a widget-based controller (and optionally a widget-based view) to interact with one of your models – the command interpreter. In this assignment, you will further understand MVC by using this framework to interact not with high-level widgets but low-level windows. This means that you will (a) update the display not by calling set methods on a widget but by overriding the window paint() method; and (b) receive input by observing not action events but mouse and key events.

You will create your own user interface to interact with and display the OZ scene, reusing the observable models you created in the previous assignment. This task has been divided into two steps: creating the OZ view and implementing a mouse and key based controller. You will also adapt the implementation of ARoundedTextField so you understand its control flow, and use it in the implementation of the command interpreter user interface, so that you better appreciate the use of interfaces.

For extra credit, you can embellish these user interfaces and their programming in many ways.

Like previous assignments, the assignment may be changed in minor ways in response to student questions. So please look for tracked changes before you submit.

The following new material is relevant to this assignment.

401	MVC and Graphics/Window Systems (10/17, 10/22, 10/24)	PowerPoint	PDF	MVC Chapter		MVC-Windows	lectures.mvc.toolkit Package

Like the previous assignment, the key is understanding the relevant material. Once you do so, it should be straightforward.

Regular - Credit OZ View: Inheriting Painter

Create a view that displays the Oz scene and reacts to changes to models in the scene. This means you must implement a view class to display the model objects in the scene that is a (direct or indirect) subclass of Component and implements a paint() method to draw all graphical objects in the logical structure of the OZ scene model. This implies that the view object must register itself as a listener of all the model objects it paints. It also means that the view object will 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.)

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.

Extra Credit OZ View: Observable Painter and Multiple Paint Listeners

Instead~~ance~~ 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, which should include a paint method – a method that takes an argument of type Graphics or Graphics2D. Now create a subclass of Component that provides a method to registers instances of this interface. Whenever the paint(Graphics) method is called (by repaint) in this subclass, it calls the paint method in each of the registered paint listeners. Let us call this subclass an *observable painter.*

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 to 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.

OZ Mouse and Key Controller

Define a controller that listens to the mouse and key events of the window displaying the Oz scene. (This window will be the view object 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 'd', 's', or 'o' in this window, then the Dorothy, Scarecrow, or Oz avatar, respectively, should move to the last click point. If the user types the letter 'r' then all avatars should return to their original positions.

In case your key listener does not receive key events, then call setFocusable(true) in the constructor of the OZ view. If this also does not work, make your OZ scene view a subclass of Panel. My experience is that these two steps are not necessary if a frame has only one input component, but I may be wrong. These two steps were taken in the implementation of ARoundTextField as there can be many such fields in a frame.

Lower-case only Round Text Field

Copy the classes and interfaces presented in class to create a round text field (ActionListenable, TextComponentInterface, and ARoundedTextField) and change ARoundedTextField to create a lower-case only text field, that is, a text field that converts mixed case letters (e.g. MoVe) to lower case ones (e.g. move) before giving them to its action listeners or displaying them to the user. This means you must change both the setText() and keyTyped() methods. Naturally, no conversion is done if the characters are not letters (e.g. " 123").

Extra Credit Round Text Field

- *Character deletion*: allow the user to delete the last character of the text field
- *Enabled/disabled status*: add to the interface ActionListenable the methods boolean getEnabled() and setEnabled (boolean newVal), which define the Enabled boolean property in a widget. Implement this property in ARoundedTextField(). If the widget is disabled, the rounded rectangle should be filled with a special color (which can be the backrground color of the text field obtained by calling the getBackground() method in the text field), and the characters entered by the user should have no effect. To fill the rectangle you will need to use the Graphics fillRoundRect() method instead of the drawRoundRect() method. If you change the color for the round rectangle, you will have to change it back to the original color when you draw the text. Graphics defines a Color property (with the associated getter and setter).
- *Carat:* Display a vertical line that is approximately at the position of the last character in the text field. To determine the exact location at which this carat should be drawn you can call the getFont() method on the Graphics object passed to the paint() method and the getFontMetrics() method on a Font. To determine the approximate position, try to determine the approximate number of pixels in a character on average. If you implemented the enabled/disabled status, then do not display this line if the widget is disabled.

- *Modular text field:* ARoundedRectangle implements the input and output processing for a StringBuffer data object. Make this class more modular by creating a separate class to process input, which we will refer to as the key listener class. Thus a text field will now involve cooperation between three objects: an instance of StringBuffer, the instance of the new ARoundedRectangle , and an instance of the new key listener class, which correspond roughly to a model, view, and controller. The new ARoundedRectangle class will implement the same interface as the old one. It will instantiate both the StringBuffer class and the key listener class. The key listener class will listen to key events announced by ARoundedRectangle instance , and will call write methods in the instance of StringBuffer instance and the repaint() method in the instance of ARoundedRectangle.

Interface-based Controller of Command Interpreter

Copy the ATextField and AJTextField classes presented in class to your project. Change your implementation of the command-interpreter controller to use TextComponentInterface ~~instance~~ instead of the text field class you used in the earlier assignment (e.g. JTextField). If you displayed the error property, then change also your view of this property to use the interface rather than the class.

Animating Demoing Main Class

Write a main class that (a) creates and visualizes an instance of the Oz scene and the command interpreter; and (b) creates an animation that moves the avatar, sets its text, and rotates each of its rotatable parts.

Your main program should visualize the Oz scene using both ObjectEditor and the OZ view and controller you created.

It should visualize the command interpreter using both ObjectEditor and three manually created user interfaces, displaying instances of AJTextField, ATextField, and ARoundedTextField, respectively. This means that your main program should create three instances of the interface-based controller (and view if you show the error property), and pass to each controller (and view) instance a different implementation of TextComponentInterface. All three controllers and the ObjectEditor UI will share a single model object. Each controller will be associated with a separate Frame, and thus a separate user interface.

As in the last assignment, the TAs will interact with the controllers manually – your main method should call only model methods.

As in the last assignment, do not call the OEFrame refresh method.