MVC in a Java/Swing Application

May 21, 2007

Spend enough time writing code, and you will eventually hear of the Model-View-Controller (MVC) paradigm: your data, business logic, and presentation should all be separate entities. Consider how J2EE applications use combinations of servlets and filters for the controller layer and JSP pages for the view layer of an MVC-based web application. With that being said, I have not seen as much effort directed at desktop application. Examples for Swing applications tend to lump the entirety of the control layer (and indeed even business logic) directly into the view layer—that is, into the various frame and panel classes.

In this article, I will present one way you might put some separation between the view and control components in a desktop application. The example I present will be, by necessity, rather simplified. It will be possible to code the entire demo application in one class. It may even be faster that way. However, larger applications with more complex requirements tend to not fit into the one-class solution category, and it is for these larger, more complex applications that you might appreciate having an extra design strategy in your arsenal.

The Demo App, Defined

Consider that you want a simple Swing-based application that allows the user to enter text and then have certain special characters be escaped. Specifically, you want to escape a few of the characters that would cause trouble in an XML or an HTML document: the greater-than, less-than, and ampersand characters. (How many online forum posts ran into trouble when the poster simply copied-and-pasted a big chunk of code without accounting for the special characters?)

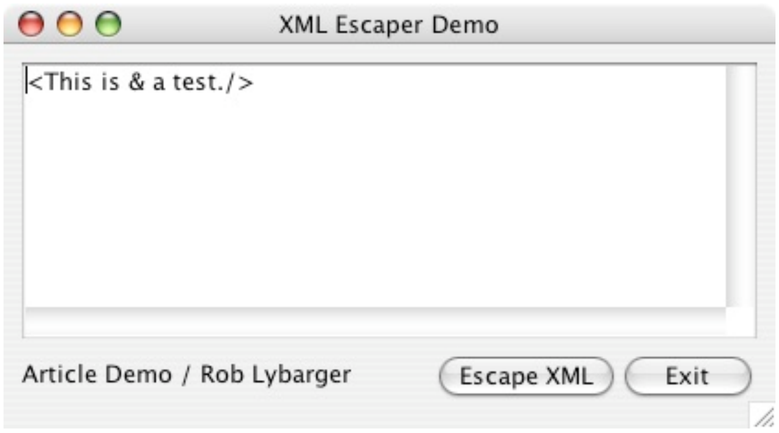

Figure 1: Screenshot of demo application (Mac OS X platform)

As I mentioned above, this can be quite simply done in a single class that extends JFrame and implements ActionListener to pick up the button press. The actionPerformed() method could directly yank out the text from the text component, perform whatever operations needed to transform it, and then replace the text in the text component with the modified form.

What happens in the more elaborate applications where the button, text component, and frame class all need to be in separate classes? Certainly, it is still easy to make the program function, but where do you put the workhorse code? The simplest solution tends to put the workhorse code in the top-level frame class because it tends to be able to see all the UI components underneath it. This also becomes non-optimal as more and more workhorse code builds up in the top-level frame class.

What happens if you completely overhaul the layout of the user interface? Maybe a few panels appear to better organize existing components, a menu comes into the mix, or you change widget sets (such as from Swing to SWT) entirely. Maybe you want to add command-line batch capability where the UI is not even shown at all. The more you find the business logic that affects what information the user sees getting intertwined with the display logic that allows the user to see and manipulate that information, the more strongly (and needlessly) coupled they become. The event handling code for a button has to account for the new panels, for example. Designs such as these will hit a wall eventually.

Enter the Controller

An alternative solution I am presenting here involves the creation of a "controller" class whose job it is to contain the interactions of the user interface components and to interact with other "business logic" classes. This class is not a descendant of any Swing class—it is a direct subclass of the humble Object class. The controller will need to basically perform two functions: keep track of references to user interface components that will be updated or modified in some way (such as the text field in our demo), and provide a set of methods that other components can directly call in their event handler (such as the button in the demo).

The primary use case of the demo application is this:

1. The user enters some text and then presses the button.
2. The button's actionPerformed() method simply calls a method in the controller class.
3. The controller class, storing a reference to the text component, grabs the current text value.
4. The controller class uses a utility class to transform the text.
5. The controller class inserts the modified text back into the text component.

Note: The controller does not, for the purposes of the demo, need a reference to the button itself because it does not take any actions that affect the button. It only affects the text component.

Designing the controller

The next consideration is how the controller class itself should be designed. My opinion is currently that, given the nature of what it is doing, only one instance of the controller class is ever needed. The user is, after all, single threaded with respect to their usage of the application. Given the way that various user interface classes might need to all interact with this single controller instance, a completely static-method class design might suffice. Instead, I prefer to go with a singleton design pattern for the controller, providing a static getInstance() method for all the user interface classes to utilize, but otherwise providing non-static methods for the event handlers to talk to. (The singleton class has a benefit over the fully-static class in that static methods do not inherit into a subclass, should you ever be faced with the need to so do.) Next, some setter methods need to be present to initialize the reference(s) to the interface element(s) that the controller needs to interact with. Finally, some methods need to be present that trigger something interesting to happen.

Tip: Where possible, I recommend that the controller class use the most abstract or most generic forms of user interface components possible. For example, you can use JTextComponent instead of JTextField or JTextArea, or JToggleButton instead of JCheckbox. This allows you to absorb a few user interface design changes with no impact to the controller class itself.

For the demo, the basic skeleton of the controller is:

```
public class AppController {
   //singleton class details
   private static AppController instance = null;
   protected AppController() { ... }
   public static AppController getInstance() { ... }

   //controller details
   private JTextComponent textComponent = null;
   public void setTextComponent(TextComponent tc) { ... }
   public void handleEscapeAction() { ... }
   public void handleExitAction() { ... }
}
```

Refer to the downloadable files for this article for the complete source code.

Using the controller

When working with the class that the text component resides in—most appropriately during initialization and layout—make a call to pass a reference to the text component to the controller:

```
...
JTextArea textArea = new JTextArea(rows,cols);
AppController.getInstance().setTextComponent(textArea);
...
```

When setting up the event listener for the button, feel free to make an anonymous class implementation:

```
...
JButton escapeButton = new JButton("Escape XML");
escapeButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      AppController.getInstance().handleEscapeAction();
   }
});
...
```

Aside from the actual code for the handleEscapeAction method, that is pretty much it. Note how adding a menu item to a menu bar can be just as simple. More importantly, note how the controller neither knows nor cares how its handleEscapeAction method is called, nor exactly where the text component itself lives in the overall design of the application (nor does the button really know what happens when its event fires, if anything.) Also note how much simpler it would be to redesign the user interface components: the interactions between components are stored in a separate controller class that need not change. (Just be sure to properly initialize the controller's component references in the redesigned code user interface code. See the 'Careful Initialization' section below.)

Business Logic

So where is the business logic in all this? After all, you might have taken inter-component control code out of the UI, but the controller class still needs to track reference to a few UI component references (plus know how to interact with them). In the case of the demo app, one last class is used to contain the details of the "business logic," which for you is the actual escaping of characters in some String object. If you download the source code file attachment for this article, note that the Escaper class contains one method that takes one String argument and returns another String argument. This represents the actual workhorse code for the application. In your real-world application, this would do something far more interesting: load or save files, talk to a database, perform some interesting calculation, or whatever.

The net effect for the demo app is that the controller class shown here represents a sort of "bridge" between the Swing-specific code to talk to the JTextComponent instance and the purely business-oriented code in the Escaper class. One immediate implication of this is that you could provide a separate controller class entirely that works completely off of a command line (by prompting for a string as input via System.in and displaying the result via System.out) but still uses the exact same business logic in the Escaper class.

Other Thoughts

Other issues you should not forget are threaded tasks, IDEs, careful initialization, and multiple controllers.

Threaded Tasks

One area of development that always tends to cause grief is threading your long-running business tasks. On one hand, if you don't, the user interface goes into an unresponsive limbo state until the task finishes. On the other hand, if you aren't extremely careful, bad things happen. One major sticking point in a Swing app is this: The user interface elements should really only be updated from the "event dispatching thread," not from your own worker threads. The Swing library provides the static method SwingUtilities.invokeLater(...) to give your worker thread the ability to affect the user interface in a safe manner. However, even knowing you need to do this, it still can be troublesome to put this code in the right place. After all, when a lot of code and a lot of classes are flying around in your editor, your brain easily tends to associate methods in your JFrame class as all running in the event dispatch thread—this isn't true. Offloading the overhead into a controller class helps emphasize the separation, in my opinion (as does carefully documenting that some method is called from a worker thread.)

IDEs

I admit I would still rather hand-code a user interface (after sketching it on paper) than use any of the GUI builders I have tried to date, but those I have tried show common traits that makes a controller class design rather viable: Interface components have post-initialization hooks you can use to pass their reference to the controller class, and event handlers let you either directly connect with arbitrary methods or let you write a quick line of code to do the same. This being the case, once you create a minimal skeleton controller class (with empty methods), the IDE can help you even more rapidly connect user interface elements with those methods. Your UI and business logic can safely coexist with the autogenerated code.

Careful initialization

Take some care in your controller's action methods to check whether the user interface component references are null before you try to access them. If they are null, you either aren't setting them when the components were initialized, or the action methods are being called earlier than you think. This is, of course, a risk regardless of how your application is designed, but it bears noting here.

Multiple Controllers

Larger applications might benefit from the presence of more than one controller. You might have the main frame components managed by a separate controller from a dialog's controller, for example. In situations like these, any common behavior you might have between controller classes can be in a parent class. (Here is where use of factory and singleton patterns provide value over the more simple all-static patterns.)

Closing Remarks

I hope to have demonstrated another tool you can put in your application design toolbox while simultaneously touching on why the MVC approach is worth embracing. Feel free to download the files for this article to see the source code for yourself. Remember: This was a simple demonstration—a "hello world," if you will. However, if you have followed all the way to the end, surely you can already see how applicable it might be in the real world. Although I feel this technique provides a certain amount of architectural flexibility, I also know there is also no such thing as "one solution fits all." I leave it to the interested reader to judge the merits of this approach in their own projects.