Creating Interactive GUIs with Swing's MVC Architecture

April 7, 2004

Part 1: Model-View-Controller and Swing

Many GUI-based applications, client interfaces, and widget toolkits use the Model-View-Controller (MVC) architecture as a primary design pattern to present, manipulate, and store data for the end users. Java's Swing toolkit is no exception. Every visual component in Swing follows the MVC pattern to achieve its task. Model-View-Controller was build into Swing from the very beginning. In this article, I'll show how model-view-controller architecture is intertwined with Swing and how its robustness and strengths can be used to create extremely flexible and powerful graphical user interfaces that require minimal effort to modify once all components are in place.

This material is split in to two articles: This part describes the underlying implementation of Model-View-Controller in Swing and the second part demonstrates a concrete example of the MVC pattern's strength by showing how use three fundamental mechanisms in Java to alter data presentation at run time without changing the correlating model. But, so you don't think that I left all the good stuff for last, the first part of this article will come with fully working code that you can apply to an enterprise level UI project, with minimal changes.

Swing and MVC Design

In a commercial application, there are often requirements to have a table of information populated from a database, or some other real-time data source. Constant revising or requesting data on every change to its presentation (such as a sort or filter) would be extremely expansive and prone to performance bottlenecks. A great number of different caching techniques have been devised to improve performance in these cases, but there is a much simpler way to populate table once, show it, and then modify the presentation only, without touching the underling data. This technique can be applied to any visual components, including lists, drop-downs, and trees. And those UI components that don't have a lot of data, such as buttons, can still benefit from the ability to change their 'look and feel' without actually changing their functionality.

Well, as it turns out, Model-View-Controller in Swing does just that! It decouples presentation from data and from operation on that data. The controller is actually collapsed into a delegate object and is not truly separated from view or is not a separate class; this is why MVC in Swing is sometimes called "separable model architecture." The designers of Swing did this because it was "difficult to write a generic controller that didn't know specifics about the [particular] view."

All graphical components in Swing, and I do mean *all*, follow Model-View-Controller architecture. Therefore, according to the Swing specification and package structure, every component has a presentation class, model class, and a public interface. There is a separation, hoverer, between a GUI-based state model and an application-specific data model. The application-specific model is usually a model dealing with or containing real user data, such as a table, list or text area. A GUI state model is usually a model that does not relate directly to the user data, such as model of a button or a 'progress slider' component. In this article, I'll concentrate on application-specific data models.

If you've superficially used Swing to create basic Java GUIs or used any IDE to do it for you, you probably did not even notice that there is a separation between visual components and their data models. That is because Swing will install a default model for every component that you use. For the developer, there is no sense in creating a new model for every Button on the screen and then assign it to the button. The same goes for the complex components, such as Tables. Just creating one and adding text to it works fine and does not require explicit model object creation. However, the existence of a separable

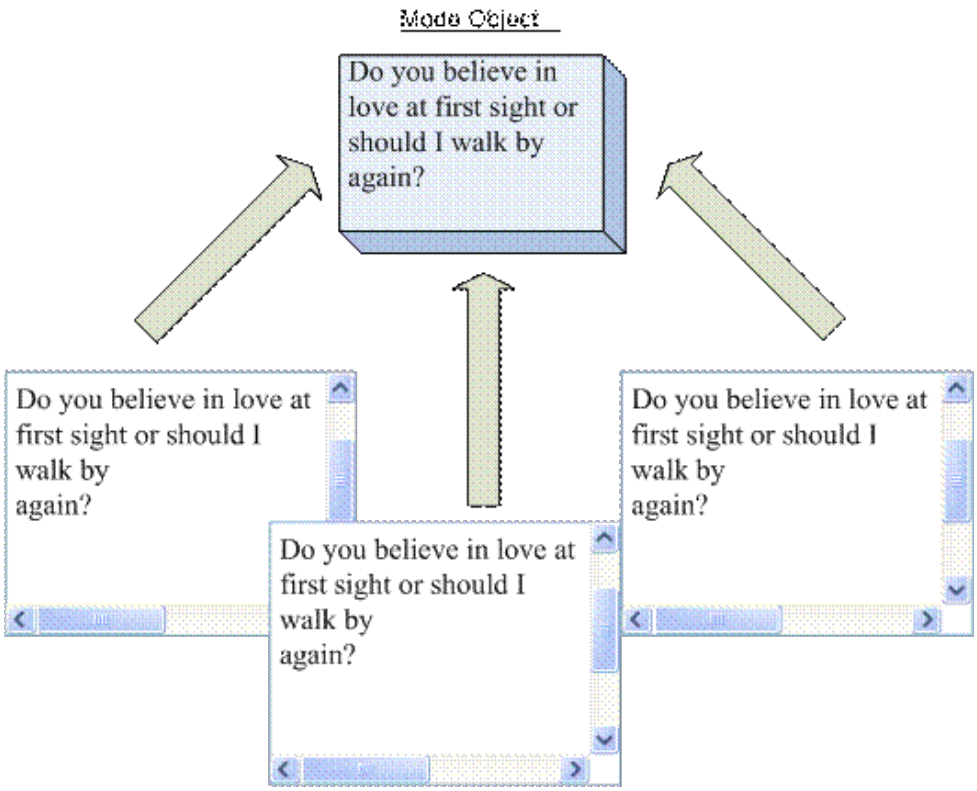
model in Swing allows programmers to do much more advanced GUI development, and was added from the initial design of Swing for a very important reason!

The controller, as I've mentioned, is not clearly apparent and is not used as in traditional MVC design pattern, but it does exist. The View layer of the MVC in Swing is objects used to render (or show) components on the screen; all visual components from Buttons to Sliders to Text Areas are part of the view layer and therefore have presentation classes. The way to truly utilize MVC in Swing is to explicitly install a new model object or chains of them. Several presentation classes may use the *same* Model class to contain the data they are presenting. They don't even have to be of the same type! What I mean is that you can create multiple components, such as text area boxes, and place them in different parts of your application. All of these text areas can have one model object associated with them. Any time one text area is updated—by user typing or deleting text, for example—all other text areas will reflect that change and show same text! Here is sample code that creates a model Object and two View objects, and installs a model into them. By comparison, if you create a View object with a default constructor, a default model would be used.

```
Document doc  = new DefaultStyledDocument();     // MODEL object
JTextArea ta1 = new JTextArea(doc);              // VIEW object 1
JTextArea ta2 = new JTextArea(doc);              // VIEW object 1
```

But wait, that is not all. You can have one model object assigned to view objects of different types, such as JTextArea and JTextField and they will both use it. For example, this code is legal:

```
Document doc  = new DefaultStyledDocument();       // MODEL object
JTextArea ta  = new JTextArea(doc);                // VIEW object 1
JTextField tf = new JTextField(doc, "Name", 1);    // VIEW object 2
```



Model Layer

The ability to separate presentation from data also lets developers modify the presentation of the *same* data. Therefore, issues such as sorting of filtering of pre-fetched data become very easy to solve and do not require re-fetching of the data on every user requested change in the view. You create one model object and install it into one (or many) view object(s), but you can also subclass the Model object and create your own custom models. Swing provides an abstract model class and a default concrete model for every visual component. See **Listing 1** for a table of presentation classes and their corresponding models. Models usually have methods to get and set data and some specific methods correlating to the functionality of the view class, such as **getColumnCount** in the JTable model class. Therefore, the abstract model class only defines these

required methods and the concrete one actually has their implementations (it extends the abstract class, as you probably have guessed).

For instance, the JTable view class has *AbstractTableModel* and *DefaultTableModel*. If you want to create a custom model for your JTable, with custom table header columns and their names, you can subclass DefaultTableModel and overwrite only the methods relating to the header info. **See Listing 2**.

You also can extend AbstractTableModel and implement all methods yourself. DefaultTableModel keeps data in a Vector of Vectors data structure, so you'll need to create a compatible memory storage. The same is true for every default Swing model; they all keep data in some kind of generic collection class. All models also have methods to get data and in some cases its index position, if it's applicable.

JList and JTable both have methods to insert and access data based on the index within the internal memory storage. This is a very important feature because it can be used to manipulate the presentation *without* changing the underling data.

The trick is to use three fundamental features of the Java language: **inheritance**, **passing objects by reference**, and **event/listener mechanism**.

Models in Swing have *no* information of any views; therefore, if you use a custom model, any changes in its data are conveyed to the view by an event/listener mechanism. Because Java passes all complex objects by reference, you can create a chain of custom models, all containing a reference to one main model with data and their own indexes of that data. The custom models would inherit all required methods and overwrite index methods. In the event that a user requests a change in the view, an index would be recalculated and an action event would be propagated to all interested models. This would result in a change in the view but not the model! (Please see the second part of this article for a full implementation of this technique.)

Presentation and View Layer

The view layer is what the end user sees on the screen; because we already know that its data (what is represents) is separated, the Swing development team was kind enough to make it interchangeable and even extendable to custom looks-and-feels. This is completely different from the method I've just described to change data presentation. It is an application-specific GUI modification and not data specific. What that means is that you can switch the way your GUI looks, make a Java application look like a Solaris Motif program on Windows XP, for instance. Code like this (before any Swing components' initialization) would do the trick:

```
try {
UIManager.setLookAndFeel("
          com.sun.java.swing.plaf.motif.MotifLookAndFeel");
} catch...
```

And, even though your JTextArea and JTree would look different, it would still contain the same data and work in the same way; you can still use the same data-specific filtering and sorting techniques. The mechanism by which it works is called a Pluggable Look and Feel (PLAF) and is beyond the scope of this article, but you can find a great tutorial on Sun Microsystems' site about PLAF (see "A Swing Architecture Overview" in the references section).

Conclusion

In this part, I've described the fundamentals of Swing separable model architecture. I've gone over the structure of Model objects and their correlation to the View objects, and the way you can install custom models into your UI components. I also briefly described how the power of separation can let developers change the presentation without touching the corresponding information, both by means of event mechanism and by the Pluggable Look and Feel feature of Swing. By using custom models, you can achieve great flexibility and superior user functionality, at the same time avoiding unnecessary requests of data, caching, or other methods of altering presentation.

The included source code contains the full JTable example from Part 2 of this article with several custom models to filter and sort data.

Source Code

Download the source code here.

Listing 1

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/data
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JMenu	ButtonModel	GUI
JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/data
JRadioButtonMenuItem	ButtonModel	GUI/data
JComboBox	ComboBoxModel	data
JProgressBar	BoundedRangeModel	GUI/data
JScrollBar	BoundedRangeModel	GUI/data
JSlider	BoundedRangeModel	GUI/data
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	data
JList	ListSelectionModel	GUI
JTable	TableModel	data
JTable	TableColumnModel	GUI
JTree	TreeModel	data
JTree	TreeSelectionModel	GUI
JEditorPane	Document	data
JTextPane	Document	data
JTextArea	Document	data
JTextField	Document	data
JPasswordField	Document	data

Listing 2

```
class PlanetTableModel extends DefaultTableModel {
   private static String[] planetModelNames
      = new String[] { "Moon Name", "Planet", "Atmosphere",
                       "Population" };

   private String filter = "All";

   public int getColumnCount()
   {
      return planetModelNames.length;
   }

   public String getColumnName(int c) {
   return planetModelNames[c];
   }

...
```

References

Core Swing Advanced Programming
By Kim Topley
ISBN: 0130832929
Publisher: Prentice Hall; December, 2000

Core Java 2, Volume II: Advanced Features (5th Edition)
By Cay Horstmann and Gary Cornell
Publisher: Prentice Hall; 5th edition (December 10, 2001)

A Swing Architecture Overview
http://java.sun.com/products/jfc/tsc/articles/architecture

Advanced MVC
http://www.ifi.unizh.ch/richter/Classes/oose2/03_mvc/02_mvc_java/02_mvc_java.html#4%20Models%20as%20Proxies

Swing Model Filtering
By Mitch Goldstein
http://www-106.ibm.com/developerworks/java/library/j-filters/

About the Author

Vlad Kofman is a System Architect working on projects under government defense contracts. He has also been involved with enterprise-level projects for major Wall Street firms and the U.S. government. His main interests are object-oriented programming methodologies and design patterns.

Solutions

Whitepapers and eBooks

MSDN Spotlight for Developers
Internet.com Cloud Computing Showcase
Microsoft TechNet Spotlight
Helpful Cloud Computing Resources
MORE WHITEPAPERS, EBOOKS, AND ARTICLES

Webcasts

MORE WEBCASTS, PODCASTS, AND VIDEOS

Downloads and eKits

MORE DOWNLOADS, EKITS, AND FREE TRIALS

Tutorials and Demos

Demo: Google Site Search
Virtual Event: Master Essential Techniques for Leveraging the Cloud
Article: Explore Application Lifecycle Management Tools in Visual Studio 2010
Internet.com Hot List: Get the Inside Scoop on IT and Developer Products
New Security Solutions Using Intel(R) vPro(TM) Technology
All About Botnets
MORE TUTORIALS, DEMOS AND STEP-BY-STEP GUIDES