Course of Programming in Java: Model-View-Controller

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Head First Design Patterns
Chapter 12. Compound Patterns

A Swing Architecture Overview

The Inside Story on JFC Component Design By Amy Fowler
MVC from *Head First Design Patterns*
VIEW
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.

CONTROLLER
Takes user input and figures out what it means to the model.

Here's the creamy controller; it lives in the middle.

MODEL
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

Controller

1. The user did something
2. Change your state
3. Change your display
4. I've changed!
5. I need your state information

View

This is the user interface.

Model

Here's the model; it handles all application data and logic.
Observer

Observable

Model

class Foo {
    void bar()
    .dollar();
}

Observers

My state has changed!

All these observers will be notified whenever state changes in the model.

View

Controller

View

I'd like to register as an observer

Any object that's interested in state changes in the model registers with the model as an observer.

The model has no dependencies on viewers or controllers!
The view only worries about presentation; the controller worries about translating user input to actions on the model. The controller is the strategy for the view—it’s the object that knows how to handle the user actions.

We can swap in another behavior for the view by changing the controller.
The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.
A pulsing bar shows the beat in real time.

A display shows the current BPMs and is automatically set whenever the BPM changes.

The view has two parts, the part for viewing the state of the model and the part for controlling things.

You can enter a specific BPM and click the Set button to set a specific beats per minute, or you can use the increase and decrease buttons for fine tuning.

Decreases the BPM by one beat per minute.

Increases the BPM by one beat per minute.
Here’s a few more ways to control the DJ View...

You can start the beat kicking by choosing the Start menu item in the “DJ Control” menu.

You use the Stop button to shut down the beat generation.

Notice Stop is disabled until you start the beat.

Notice Start is disabled after the beat has started.

The controller is in the middle...

The controller sits between the view and model. It takes your input, like selecting “Start” from the DJ Control menu, and turns it into an action on the model to start the beat generation.

The controller takes input from the user and figures out how to translate that into requests on the model.

All user actions are sent to the controller.
The BeatModel is the heart of the application. It implements the logic to start and stop the beat, set the beats per minute (BPM), and generate the sound.

The model also allows us to obtain its current state through the `getBPM()` method.
The beat is set at 119 BPM and you would like to increase it to 120.

View

You see the beatbar pulse every 1/2 second.

The view is updated to 120 BPM.

View

Because the BPM is 120, the view gets a beat notification every 1/2 second.

View is notified that the BPM changed. It calls getBPM() on the model state.

Controller

...which results in the controller being invoked.

Click on the increase beat button...

The controller asks the model to update its BPM by one.

BeatModel

on()
setBPM()
off()
getBPM()
The DJ view displays two aspects of the BeatModel...

...the current beats per minute, from the BPMObserver notifications...

...and a pulsing "beat bar" pulses in synch with the beat, driven by the BeatObserver notifications.

We've separated the view of the model from the view with the controls.

This is the part of the view that you use to change the beat. This view passes everything you do on to the controller.
DJ example: source code

- BeatModelInterface.java
- BeatModel.java
- DJView.java
- ControllerInterface.java
- BeatController.java
- DJTestDrive.java
- HeartAdapter.java
- HeartController.java
- HeartTestDrive.java
Architecture of Swing

Design Goals

1. *Be implemented entirely in Java* to promote cross-platform consistency and easier maintenance.

2. *Provide a single API capable of supporting multiple look-and-feels* so that developers and end-users would not be locked into a single look-and-feel.

3. *Enable the power of model-driven programming* without requiring it in the highest-level API.

4. *Adhere to JavaBeans design principles* to ensure that components behave well in IDEs and builder tools.

5. *Provide compatibility with AWT APIs where there is overlapping,* to leverage the AWT knowledge base and ease porting.
Swing architecture is rooted in the *model-view-controller* (*MVC*) design that dates back to SmallTalk. MVC architecture calls for a visual application to be broken up into three separate parts:

- A *model* that represents the data for the application.
- The *view* that is the visual representation of that data.
- A *controller* that takes user input on the view and translates that to changes in the model.
The delegate

We quickly discovered that MVC split didn’t work well in practical terms because the view and controller parts of a component required a tight coupling (for example, it was very difficult to write a generic controller that didn’t know specifics about the view). So we collapsed these two entities into a single UI (user-interface) object, as shown in this diagram:

The UI object shown in this picture is sometimes called *UI delegate*. This new quasi-MVC design is sometimes referred to a *separable model architecture*. 
To MVC or not to MVC?

- Component’s view/controller responsibilities as being handled by the generic component class (such as JButton, JTree, and so on).
  - E.g. the code that implements double-buffered painting is in Swing’s JComponent class (the "mother" of most Swing component classes).

- Look-and-feel-specific aspects of those responsibilities to the UI object that is provided by the currently installed look-and-feel.
  - E.g. the code that renders a JButton’s label is in the button’s UI delegate class.
Separable model architecture

Swing defines a separate model interface for each component that has a logical data or value abstraction. This separation provides programs with the option of plugging in their own model implementations for Swing components.
<table>
<thead>
<tr>
<th>Component</th>
<th>Model Interface</th>
<th>Model Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JButton</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JToggleButton</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JCheckBox</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JRadioButton</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JMenu</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JMenuItem</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JCheckBoxMenuItem</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JRadioButtonMenuItem</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JComboBox</td>
<td>ComboBoxModel</td>
<td>data</td>
</tr>
<tr>
<td>JProgressBar</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JScrollBar</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JSlider</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JTabbedPane</td>
<td>SingleSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JList</td>
<td>ListModel</td>
<td>data</td>
</tr>
<tr>
<td>JList</td>
<td>ListSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JTable</td>
<td>TableModel</td>
<td>data</td>
</tr>
<tr>
<td>JTable</td>
<td>TableColumnModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JTree</td>
<td>TreeModel</td>
<td>data</td>
</tr>
<tr>
<td>JTree</td>
<td>TreeSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JEditorPane</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JTextPane</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JTextArea</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JTextField</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JPasswordField</td>
<td>Document</td>
<td>data</td>
</tr>
</tbody>
</table>
GUI-state vs. application-data models

- **GUI-state models** are interfaces that define the visual status of a GUI control, such as whether a button is pressed or armed, or which items are selected in a list.

- It is possible to manipulate the state of a GUI control through top-level methods on the component, without any direct interaction with the model at all.

- **Application-data models** are interfaces that represent some quantifiable data that has meaning primarily in the context of the application, such as the value of a cell in a table or the items displayed in a list.

- Provide clean separation between their application data/logic and their GUI. For truly data-centric Swing components, such as JTree and JTable, interaction with the data model is strongly recommended.

- For some components the model categorization falls in between GUI state models and application-data models, depending on the context in which the model is used. E.g. BoundedRangeModel on JSlider or JProgressBar.
Shared model definitions

Common models enable automatic connectability between component types. For example, because both JSlider and JScrollbar use the BoundedRangeModel interface, a single BoundedRangeModel instance could be plugged into both a JScrollbar and a JSlider and their visual state would always remain in sync.
The separable-model API

If you don't set your own model, a default is created and installed internally in the component. The naming convention for these default model classes is to prepend the interface name with "Default." For JSlider, a Default-BoundedRangeModel object is instantiated in its constructor:

```java
public JSlider(int orientation, int min, int max,
               int value) {
    checkOrientation(orientation);
    this.orientation = orientation;
    this.model =
        new DefaultBoundedRangeModel(value, 0, min, max);
    this.model.addChangeListener(changeListener);
    updateUI();
}
```
If a program subsequently calls `setModel()`, this default model is replaced, as in the following example:

```java
JSlider slider = new JSlider();
BoundedRangeModel myModel =
    new DefaultBoundedRangeModel() {
        public void setValue(int n) {
            System.out.println("SetValue: "+ n);
            super.setValue(n);
        }
    });
slider.setModel(myModel);
```

For more complex models (such as those for `JTable` and `JList`), an abstract model implementation is also provided to enable developers to create their own models without starting from scratch. These classes are prepended with "Abstract".

For example, `JList`’s model interface is `ListModel`, which provides both `DefaultListModel` and `AbstractListModel` classes.
Model change notification

Models must be able to notify any interested parties (such as views) when their data or value changes. Swing models use the JavaBeans Event model. There are two approaches for this notification used in Swing:

- Send a lightweight notification that the state has "changed" and require the listener to respond by sending a query back to the model to find out what has changed. The advantage of this approach is that a single event instance can be used for all notifications from a particular model – which is highly desirable when the notifications tend to be high in frequency (such as when a JScrollBar is dragged).

- Send a stateful notification that describes more precisely how the model has changed. This alternative requires a new event instance for each notification. It is desirable when a generic notification doesn’t provide the listener with enough information to determine efficiently what has changed by querying the model (such as when a column of cells change value in a JTable).
Lightweight notification

The following models in Swing use the lightweight notification, which is based on the ChangeListener/ChangeEvent API:

<table>
<thead>
<tr>
<th>Model</th>
<th>Listener</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedRangeModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
<tr>
<td>ButtonModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
<tr>
<td>SingleSelectionModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
</tbody>
</table>

The ChangeListener interface has a single generic method:

```java
public void stateChanged(ChangeEvent e)
```

The only state in a ChangeEvent is the event "source."
Models that use this mechanism support the following methods to add and remove ChangeListeners:

```java
public void addChangeListener(ChangeListener l)
public void removeChangeListener(ChangeListener l)
```

To be notified when the value of a JSlider has changed, e.g.:

```java
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // need to query the model to get updated value...
        BoundedRangeModel m =
            (BoundedRangeModel)e.getSource();
        System.out.println("model changed: " + m.getValue());
    }
});
```
To provide convenience for programs that don’t wish to deal with separate model objects, some Swing component classes also provide the ability to register ChangeListeners directly on the component (so the component can listen for changes on the model internally and then propagates those events to any listeners registered directly on the component).

So we could simplify the preceding example to:

```java
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // the source will be the slider this time...
        JSlider s = (JSlider)e.getSource();
        System.out.println("value changed: " + s.getValue());
    }
});
```
Stateful notification

Models that support *stateful notification* provide event *Listener* interfaces and event objects specific to their purpose. The following table shows the breakdown for those models:

<table>
<thead>
<tr>
<th>Model</th>
<th>Listener</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListModel</td>
<td>ListDataListener</td>
<td>ListDataEvent</td>
</tr>
<tr>
<td>ListSelectionModel</td>
<td>ListSelectionListener</td>
<td>ListSelectionEvent</td>
</tr>
<tr>
<td>ComboBoxModel</td>
<td>ListDataListener</td>
<td>ListDataEvent</td>
</tr>
<tr>
<td>TreeModel</td>
<td>TreeModelListener</td>
<td>TreeModelEvent</td>
</tr>
<tr>
<td>TreeSelectionModel</td>
<td>TreeSelectionListener</td>
<td>TreeSelectionEvent</td>
</tr>
<tr>
<td>TableModel</td>
<td>TableModelListener</td>
<td>TableModelEvent</td>
</tr>
<tr>
<td>TableColumnModel</td>
<td>TableColumnModelListener</td>
<td>TableColumnModelEvent</td>
</tr>
<tr>
<td>Document</td>
<td>DocumentListener</td>
<td>DocumentEvent</td>
</tr>
<tr>
<td>Document</td>
<td>UndoableEditListener</td>
<td>UndoableEditEvent</td>
</tr>
</tbody>
</table>
The following code dynamically tracks the selected item in a JList:

```java
String items[] = {"One", "Two", "Three"};
JList list = new JList(items);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener
    (new ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent e) {
            // get change information directly
            // from the event instance...
            if (!e.getValueIsAdjusting()) {
                System.out.println("selection changed: " +
                    e.getFirstIndex());
            }
        }
    });
```
Ignoring models completely

As mentioned previously, most components provide the model-defined API directly in the component class so that the component can be manipulated without interacting with the model at all. This is considered perfectly acceptable programming practice (especially for the GUI-state models). For example, following is JSlider’s implementation of getValue(), which internally delegates the method call to its model:

```java
public int getValue() {
    return getModel().getValue();
}
```

And so programs can simply do the following:

```java
JSlider slider = new JSlider();
int value = slider.getValue();
//what’s a "model," anyway?
```
Pluggable look-and-feel architecture

A new L&F is a very powerful feature for a subset of applications that want to create a unique identity. PL&F is also ideally suited for use in building GUIs that are accessible to users with disabilities, such as visually impaired users or users who cannot operate a mouse.

In a nutshell, pluggable look-and-feel design simply means that the portion of a component’s implementation that deals with the presentation (the look) and event-handling (the feel) is delegated to a separate UI object supplied by the currently installed look-and-feel, which can be changed at runtime.
Look-and-feel management

The UIManager is the API through which components and programs access look-and-feel information (they should rarely, if ever, talk directly to a LookAndFeel instance). UIManager is responsible for keeping track of which LookAndFeel classes are available, which are installed, and which is currently the default. The UIManager also manages access to the Defaults Table for the current look-and-feel.

```java
public static LookAndFeel getLookAndFeel()
public static void setLookAndFeel(LookAndFeel newLookAndFeel)
public static void setLookAndFeel(String className)
```

As a default look-and-feel, Swing initializes the cross-platform Java look and feel (formerly known as "Metal"). The following code sample will set the default Look-and-Feel to be CDE/Motif:

```java
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```
The UIManager static methods to programmatically obtain the appropriate LookAndFeel class names:

```java
public static String getSystemLookAndFeelClassName()
public static String getCrossPlatformLookAndFeelClassName()
```

So, to ensure that a program always runs in the platform’s system look-and-feel, the code might look like this:

```java
UIManager.setLookAndFeel(
    UIManager.getSystemLookAndFeelClassName());
```
Dynamically Changing the Default Look-and-Feel

When a Swing application programmatically sets the look-and-feel (as described above), the ideal place to do so is before any Swing components are instantiated. This is because the UIManager.setLookAndFeel() method makes a particularLookAndFeel the current default by loading and initializing thatLookAndFeel instance, but it does not automatically cause any existing components to change their look-and-feel.
And so if a program needs to change the look-and-feel of a GUI hierarchy after it was instantiated, the code might look like the following:

```java
// GUI already instantiated, where myframe
// is top-level frame
try {
    UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel"));
    myframe.setCursor(
        Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    SwingUtilities.updateComponentTreeUI(myframe);
    myframe.validate();
} catch (UnsupportedLookAndFeelException e) {
} finally {
    myframe.setCursor
        (Cursor.getPredefinedCursor
        (Cursor.DEFAULT_CURSOR));
}
```
public static LookAndFeelInfo[] getInstalledLookAndFeel() method can be used to programmatically determine which look-and-feel implementations are available, which is useful when building user interfaces which allow the end-user to dynamically select a look-and-feel.