Simple Java Application Using MVC

1. Example Application

The purpose of this example is to show some Java code and a very useful way of decomposing an application into classes which separate the core logic of the main objects from the details of the human computer interface. First we begin with a simple description of what the application should do using the idea of user stories to express the requirements (from Extreme Programming). The initial set of stories is as follows.

a. The program should have 4 internal states. Start, Finish and 2 intermediate states.
b. The user may navigate between states by using 2 buttons, “Next” and “Back”.
c. The interface will visually provide feedback about the current system state.

Note that this example is simple and rather abstract. A more realistic application would have states that might represent things such as whether or not various information has been supplied, if a computational step has been completed, if an error has occurred, if a game is over, etc. Often state information is held in variables explicitly for that purpose. Clearly tracking the state an object is in can clarify design issues and make the software easier to code and understand.

Notice that stories are small and lack many details. The idea is that a customer is readily available for the development team to meet with to work out the details. In this example we will assume all the stories should be implemented for this iteration. The first code that should be written (aside from a “spike” if necessary) are the tests which help to define what your objects should do. A brainstorming session with the client helps to fill in some details. Techniques like paper prototyping (http://www.uie.com/articles/prototyping_risk/) can help the team communicate and to understand requirements through a hypothetical interface. Figure 1 shows an early design idea for the application’s Graphical User Interface (GUI).

![Figure 1: Sketch of application’s proposed user interface (GUI).](image)

2. Establishing the core logic - focus on the Model

Sometimes it is hard to think about the basic objects when all you can imagine is the sketch of the GUI. However, it is commonly recognized, that the GUI and the underlying functionality are most often better implemented as separate but cooperating objects. The underlying functionality (often referred to as the model) is independent of the particular GUI used to represent and manipulate it. This follows the Model-View-Controller paradigm (see Figure 2). We can develop the model separate from the interface. This better allows porting to different window systems, development of GUI’s by trained GUI designers, clearer coding of business logic, etc. There are 2 immediate
advantages for us: 1. Development can proceed without needing to know GUI programming; 2. Unit tests are much more easily automated if you don’t need to worry about the GUI.

Thus we focus first on the model component. This object should be able to represent the states required. It should have methods that allow the view and controller to communicate user manipulation requests and to respond to state queries. The model class for our example is `StateModel`.

Development with Extreme Programming would begin by writing tests which will help drive the design of the model. These tests would also act as documentation on how the model is expected to work and confirm that behavior of the model code is correct. The test and code-to-pass-test cycle is very tight, sometimes occurring in just minutes. An example test is given in Figure 3.

```java
public void testBasicNavigation() {
    StateModel model = new StateModel();
    model.nextState();
    assertEquals("State1", model.getState());
    model.nextState();
    assertEquals("State2", model.getState());
    model.backState();
    assertEquals("State1", model.getState());
    model.nextState(); // move to state 2
    model.nextState(); // move to finish
    assertEquals("Finish", model.getState());
    model.backState();
    assertEquals("State2", model.getState());
}
```

**Figure 2**: The MVC separates the model from the way it is presented and manipulated by the user. Left, the class which represents the model for the example application.

**Figure 3**: A test for typical navigation behavior using `backState` and `nextState` methods.
To be clear, testing and coding goes back and forth very quickly and is done by the same programming pair. Here we have skipped several steps of Test Driven Development to show you some code examples. The code shown does not represent final code but is a snapshot of the code at a given time. Code can be continually refactored to improve its design.

Without showing feedback of what state the model is in the interface would be incomplete. In the Java Swing GUI shown below, the Start state is shown in red rather than with an arrow.

The Back and Next buttons also need to be hooked up to the model.

The source on the next page is an example of how the GUI might be coded in Java. It also shows how the model code is hooked into the GUI. This GUI highlights the current state in red.
package cs435ex1;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class StateApp {
    JLabel startLabel;
    JLabel state1Label;
    JLabel state2Label;
    JLabel finishLabel;
    JButton nextBut;
    JButton backBut;
    public StateApp() {
    }
    public void updateLabels(StateModel m) {
        // set all to unselected - black
        startLabel.setForeground(Color.black);
        state1Label.setForeground(Color.black);
        state2Label.setForeground(Color.black);
        finishLabel.setForeground(Color.black);
        // query the model to find and show the selected state
        int theState = m.getState();
        switch (theState) {
            case StateModel.START:
                startLabel.setForeground(Color.red); break;
            case StateModel.STATE1:
                state1Label.setForeground(Color.red); break;
            case StateModel.STATE2:
                state2Label.setForeground(Color.red); break;
            case StateModel.FINISH:
                finishLabel.setForeground(Color.red); break;
        }
    }
    public JLabel initLabel(String text) {
        JLabel lab = new JLabel(text);
        lab.setHorizontalAlignment(JLabel.CENTER);
        return lab;
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame("Example State Application GUI");
        final StateApp app = new StateApp();
        final StateModel model = new StateModel();
        // Create Buttons
        app.nextBut = new JButton("Next");
        app.nextBut.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // make the button click do something
                model.nextState();
                app.updateLabels(model);
            }
        });
        app.backBut = new JButton("Back");
        app.backBut.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // make the button click do something
                model.backState();
                app.updateLabels(model);
            }
        });
        // group the buttons in a JPanel
        JPanel buttonPane = new JPanel();
        // uses default flow layout
        buttonPane.add(app.nextBut);
        buttonPane.add(app.backBut);
        JPanel labelPane = new JPanel(); // group the in a JPanel
        // Use a GridLayout to align the labels in a single column
        labelPane.setLayout(new GridLayout(4,1));
        labelPane.add(app.startLabel = app.initLabel("Start"));
        labelPane.add(app.state1Label = app.initLabel("State 1"));
        labelPane.add(app.state2Label = app.initLabel("State 2"));
        labelPane.add(app.finishLabel = app.initLabel("Finish"));
        // Place the buttons and labels into the frame
        frame.getContentPane().add(buttonPane, BorderLayout.SOUTH);
        frame.getContentPane().add(labelPane, BorderLayout.CENTER);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        //Make sure the true initial state is shown
        app.updateLabels(model);
        //Finish setting up the frame, and show it.
        frame.pack();
        frame.setSize(340,300);
        frame.setVisible(true);
    }
}