Core XP Practices with Java and Eclipse: Part 2

In order to get something useful to the customer and allow him to provide us with feedback, we start on the graphical interface for the application. Graphical user interfaces (GUI) are hard to test, so we keep as much logic as we can in the model. A GUI that defers most of the logic to other objects is referred to as a thin interface (its implementation is thin on logic). Figure 1 shows how the GUI of the application will call the model using the same methods that have already been created and tested.

**UNIT TESTS**

<table>
<thead>
<tr>
<th>TestPlaceMarkers()</th>
</tr>
</thead>
<tbody>
<tr>
<td>game.placeMarker();</td>
</tr>
<tr>
<td>game.getBoard();</td>
</tr>
<tr>
<td>assertEquals();</td>
</tr>
</tbody>
</table>

A. board positions with markers

Q. What is on the board now?

```java
game.getBoard();
game.placeMarker();
```

**Model**
Underlying, application object, business logic, data, etc.

**View**
Presents the model info to the user.

**Controller**
Allows user to manipulate/query the model

User wants to place another marker on board.

**GUI**

**Figure 1:** The Unit tests have driven the model using the same function calls that the application will use once the GUI has been implemented. Place the logic in the model so it can be well tested and the GUI left “thin”.

1. **A first pass at the graphical interface**

   An obvious feature of this interface should be the 3x3 grid which represents the Tic-Tac-Toe board. Since we want the user to be able to interact graphically, we’ll assume that clicking on a board location should indicate to the system that the user wants to place a marker in the corresponding location. A reasonable approach might be to represent the board locations with an array of buttons. Each button should display text representing what is on the board at its corresponding position.

**Figure 2:** Creating a basic interface with Swing.
Swing is the modern graphical interface library created by Sun for Java applications. Objects and interfaces are defined for most of the standard interface items. The main window of an application is created using a `JFrame`. Components of this window are placed into its content pane. `JPanels` are often used to group items together within a window (or other container such as another `JPanel`). Components (such as buttons) are added to containers (like the content pane or a `JPanel`) and arranged by the policies of layout managers. Several layout managers are available, each with its own rules about sizing and arranging components. Some code to create a Swing version of the application as sketched in Figure 2 is given in Figure 3.

```java
package tttEx;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TTTGameApp {
    private JFrame frame;
    private JButton[] boardPosition;
    private TTTGame model; // the game logic
    static final int PREFER_BUTTON_SIZE = 90;
    public TTTGameApp() {
        model = new TTTGame();
        frame = new JFrame(); //Create the main window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel theBoard = new JPanel(new GridLayout(3, 3));
        boardPosition = new JButton[9]; // make the board using a button for each position
        for (int i = 0; i < 9; i++) {
            boardPosition[i] = new JButton();
            boardPosition[i].setText("");
            boardPosition[i].setPreferredSize(new Dimension(PREFER_BUTTON_SIZE, PREFER_BUTTON_SIZE));
            boardPosition[i].setName(""); // location num. as name String
            theBoard.add(boardPosition[i]);
        }
        frame.getContentPane().add(theBoard);
        frame.pack();
        frame.setVisible(true);
    }

    private static void createAndShowGUI() {
        TTTGameApp tttApp = new TTTGameApp();
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Figure 3: `TTTGameApp` class. A start at implementing the GUI (view and controller)
This class has a `main()` method to get the application up and running. Even though it looks like the application abruptly ends, it has created the GUI and started another thread which stays live and runs the interface. The method `createAndShow()` will be called and it will instantiate an instance of the `TTTGameApp` class. The `TTTGameApp` class constructor is called creating the top level window (the `JFrame`, `frame`). A `JPanel` is created to hold the board consisting of a grid of `JButtons`. Initially the `JButtons` display text representing how they are numbered so the position numbers are clearly shown by the interface. The `JPanel` is added to the frame’s content window then the window is made visible. The `TTTGameApp` object also holds a reference to a `TTTGame`. It is not used yet but we know we need to make calls to the model to support the visualization and control of it.

To run your application, save the `TTTGameApp` source file and select the Ant build target “run”. Your application interface should appear and look like the following (see Figure 4). Note that you can exit the application by clicking on the close box as this option was set by `frame.setDefaultCloseOperation()`.

---

**Figure 4:** First pass of the `TTTGameApp` graphical interface.

---

Improve this first pass by implementing the following modifications.

1. Add the title “Tic-Tac-Toe” to the window.
2. Create a `TTTGame` model object and have the view code ask the model what is actually at each grid position. Display the true contents of the board as the text for the `JButtons`. (Initially the board is empty).

You should also start to get familiar with reference materials so you can look up the characteristics of various Swing components. The Java api (applications programmers interface) is available on line or you can download a copy onto your computer. The on line link is [http://java.sun.com/j2se/1.4.2/docs/api/index.html](http://java.sun.com/j2se/1.4.2/docs/api/index.html). For instance, the `JFrame` section will tell you how to add a title to the window.

### 2. Responding to the User

If you run the interface above and click in the grid locations you will notice that a button click seems to happen but that nothing else appears to be going on inside your program. You need to hook up the user’s button clicks so they can be translated into requests to send to the model; this is the aspect that the controller code does. In terms of the Swing code, buttons generate actions and objects can be registered as listeners for
actions. Listeners have a chance to run some code in response to an action being fired. To make your application do something, the buttons have to have listeners that will be called in response to the user pressing them.

The snippet of code shown in Figure 5 illustrates how this might be done. Note, the `actionPerformed()` method should pass a user request along to the model. The model’s logic will decide what to do and, if necessary, the model will change state. The view cannot know what has happened inside the model without asking for the latest model information. The view asking for information should not modify the game state.

When a `JButton` object is created, register a listener for it.

```java
theButton.addActionListener(listenerObject);
```

The listener implements the `ActionListener` interface so it must have a public void `actionPerformed()` method. The same object may create the buttons and also be its listener. In that case, register this as the listener object. The `actionPerformed()` method of the listener will be called when the button is pressed. More than one button can share the same listener but in that case it is usually necessary to determine which button was actually pressed. Such information can be found through the `event` object passed to the `actionPerformed()` method. In the example below, the process was simplified because the buttons were assigned unique names as they were created (see `setName()` in Figure 3).

```java
public void actionPerformed(ActionEvent event) { // implements ActionListener
    JButton whichBut = (JButton) event.getSource();
    int position = Integer.parseInt(whichBut.getName());

    // position is the button number pressed - pass request to model
    // model may change internally - refresh visual with new info
}
```

**Figure 5:** Hooking buttons up. Using ActionListeners

While trying to hook the model up to the GUI you may notice that even though you tried to imagine how things would work earlier (when you were developing your tests), method calls now seem awkward. It is fine to go back and change the model if necessary (continuous design improvement). If the model is changed, you should also go back and update (refactor) your unit tests. Unit tests should always run at 100%. Hook up the model so the users can place markers (X always goes first) on the board. An attempt to place a marker on top of another marker should be rejected by the model. (For now the model should just not allow it. No pop-up dialogs etc. The same player can keep trying to make a move until a legal move is made).

### 3. More testing

The application should now have covered the basic functionality of the first 3 stories. Your tests should cover all of your model’s functionality. The customer has submitted sample games which should also be tested (Figure 6).

**Figure 6:** The customer has submitted some sample games as tests to see if your game logic does as he expects. Implement tests to confirm that these games are played correctly. The small subscripted letters indicate button clicks for a particular player on a turn number.