Eclipse Visual Editor: A GUI Building Tool

User interface designers can work in parallel with the software developers. Interface designers might use a tool as demonstrated here, to construct the interface. The tool generates code that might be used in the final product, as part of a prototype, or as a way of clarifying requirements to others on the team. The structure of the code generated by these tools can be quite different from that generated by hand. In our example, we use the Eclipse Visual Editor to generate Java Swing code for an example application that lets two players play tic tac toe on the computer.

1. Overview

We assume that we have communicated with the user using sketches or paper prototypes and know that an important feature of this interface is the 3x3 grid which represents the Tic-Tac-Toe board. The user should be able to graphically enter his move by clicking on a board location. A reasonable approach might be to represent the board locations with an array of buttons. An easier approach which gets the prototype up faster simply incorporates an image of one of the earlier sketches to represent the game grid. Figure 1 shows the starting elements.

![Figure 1: Creating a basic interface with Swing.](image)

Swing is the modern graphical interface library created by Sun for Java applications. Java objects and interfaces are defined for most of the standard user interface items. Additionally, components and containers

![Figure 2: 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”.](image)
may be customized by subclassing. The main window of an application is created using a \textit{JFrame} with a content Pane as its main container. Another container type, the \textit{JPanel}, is often used to group items together within a window (or other container such as another \textit{JPanel}). Components (such as buttons) are added to containers and arranged and sized by the policies of layout managers.

Rather than weaving the game logic throughout the GUI code it is best to try to keep these two aspects separate. Conceptually, the object(s) representing the core logic are referred to as the “Model”. Figure 2 shows how the GUI code of the application will call methods in the Model. The Model View Controller (MVC) paradigm, portrays the GUI as being composed of objects that visualize the Model, the View, and objects which manage the control of the View and the Model, the Controller. The goal is to create a “thin” GUI which defers most of the processing to the Model. As shown in Figure 2, a Model can be tested independently from the GUI. In fact, Extreme Programming expects the tests to be written before the Model itself is even coded. Developing test first promotes better design and generates tests which serve as documentation and a safety net to assure that code continues to perform correctly even after modifications. Here we will focus on the GUI and assume that the Model is being developed by another group (this is another benefit of the MVC).

2. Getting Started with Eclipse

1. Decide where you want to place your Java projects. Eclipse uses the idea of a workspace directory to store projects in. You can make Eclipse start up using your preferred workspace. If you are working in the lab and make use of a networked drive your work will be available to you no matter what machine you are on.

2. Create a new Java project (File --> New --> Project --> Java Project). Let Eclipse make a project directory for you in your workspace (See Figure 3). Pick a name that will be easy for you to recognize (tttGUI). Don’t worry about the other options in this dialog since we will be using an Ant build script to control the compile and run procedures.

![New Java Project dialog](figure3.png)

**Figure 3:** Eclipse 3, New Java Project dialog. Define project name and hit “Finish”.

Within your project directory it is good practice to group different types of files into different directories. In particular it is generally a good idea to keep your source code separate from the compiled code. Java also
uses a close correspondence between java packages and directories. It makes life much easier to set up these directories carefully early on so they work well with the build script. Add a new source folder to your project called src (for source code files). Add a package (ex484) to the src folder. See Figure 4.

**Figure 4:** Highlighting the src directory and popping up the right menu is an easy way to create a New Package within the src directory.

### 3. Beginning Construction of the Interface

Add a new visual class to your package by selecting the New --> “Other ...” option with the right mouse button over the package. Make a subclass of JPanel and name it, TTTBoardPanel, as shown in Figure 5.

**Figure 5:** Highlight the ex484 package and pop up the right menu. Select the New --> Other ... and pick the Java --> Swing --> JPanel visual class option. Create a TTTBoardPanel class which subclasses JPanel and does not include a main() method.
Code for the class will be generated and displayed in the source code view while another view will show the JPanel visually as a grey box. Your views may be arranged differently. Find the Palette view which shows the various components that can be used to construct an interface. Click on the JLabel item (under Swing controls) and then click within the grey box. This creates a new JLabel component and installs it inside of the JPanel subclass you just created. Originally the JLabel is shown as small box with the text JLabel in it. Select the JLabel and find the properties view. Change the “text>” field so that it is empty. Instead of text, the label will be used to hold an image. Set the icon property of the label to refer to the image sketch of the tic tac toe grid (tttGridSketch.jpg). Your result should be similar to that shown in Figure 6.

The TTTBoardPanel class created here can be referred to by other interface code. Later it can be replaced by a more thorough implementation. To adjust the size, use the properties view to set the size and preferred size of the panel to be 304,304 (this is based upon the size of the image used). Eclipse can run this code (even though there is no main method) so we can verify that it is constructed correctly. First select the panel then open the Run menu and select the Run as... Java Bean option. The panel should run showing the grid image. Quit the test by clicking on its close box.

4. Creating the Main Application Window

The TTTBoardPanel just created is one component of your interface. Now we will create the main application. This class will implement the GUI and in the final product will, link up to a model. We will create this code inside the same package so, with the mouse over the package icon, pop up the right button menu then select New ... Other... Java Visual Class. Under the Swing options, choose to make a subclass of JFrame and name it TTTAppGui with the option to make the main method checked. See Figure 7.
Visually you will see a grey box with a title bar representing the main application window. If you look at this class with the Java Beans view you will see that the frame contains a contentPane. Pick the Choose Bean option in the palette and a dialog box will appear. Start typing “TTT” in the topmost field and classes with matching names will appear. Select the TTTBoardPanel class created earlier and name the new variable theBoardPanel (see Figure 8). This allows you to incorporate that class visually into the interface. After dismissing the dialog with an “OK”, position the cursor in the center of the main window and click. This positions the board panel in the middle (center) of the contentPane of the frame. To adjust the size of the frame, select the frame (the, this object, in the Java Beans view) switch to its properties view and change the size field to “-1,-1”. This tells the frame to compute its size based upon the size preferences of its components and its layout manager. Edit the title property of the frame so that its title is “Tic Tac Toe”.

To make the application do something, main() has to be modified to create an instance of itself for display. Examine the source code and change main() to the code shown in Figure 9:

```java
public static void main(String[] args) {
    TTTAppGui app = new TTTAppGui();
    app.pack();
    app.setVisible(true);
}
```

Figure 9: Modified main method of TTTAppGui class.
When run, will open a window as shown to the right.
The application can now be run using the Run menu, Run As --> Java Application option. The expected result is shown in Figure 9. Clicking on the close box will cause the test application to quit.

5. Adding Some Basics

An early pass at this game should include an area for feedback status (“X’s turn”, “Tie Game”, etc.) and a button to start a new game. We’ll place these widgets underneath the game board inside of the contentPane. Controls are often grouped using containers (JPanels are commonly used in Swing). Each container, such as the contentPane, has a policy determined by its layout manager to control how the items it contains are sized and positioned within it. The contentPane by default uses a BorderLayout which divides its area into five regions: a center and the four directions of the compass. We already positioned the game board at the center, now we’d like to place the status information and new game button at the bottom (South). Since only one item can be in any direction, we will need to create an intermediate container to hold the status and button, and place that container within the contentPane.

Create new widgets by using the palette. If it is too difficult to accurately place a widget within another one using the graphical view, use the Java Beans view to do it. First create a new JPanel. Name it bottomPanel and place it in the South section of the contentPane. Graphically, it might be difficult (impossible?) to place it in the right section but once it is in the contentPane you can also edit the source code to adjust where it goes. Look for the routine that prepares the contentPane (getJContentPane()) and find where the bottomPanel is being positioned. Change it so that it will be placed in the SOUTH.

    jContentPane.add(getBottomPanel(), java.awt.BorderLayout.SOUTH);

Now create a JLabel named statusMsg and a JButton named newGameBut . Item names can be set using the Java Bean view right menu --> rename option or the Properties view field name property. The text property sets the text displayed by the component. Make the label say “Your turn player X” and the button say “New Game”. Place both of these components in the bottomPanel. Make the label appear to the left of the button. Running at this stage should give the result shown in Figure 10a. This is almost what is desired.

![Figure 10a: Running the application after the bottomPanel with a label and button have been added.](image)

Instead of having these components bunched toward the bottom center we would rather have them along the bottom but with the status information along the left side and the button on the right. This is just the sort of thing that is determined by a layout manager. Select the properties for the bottomPanel. There is a property named layout that determines what sort of layout manager is used. Change it so that it uses a BorderLayout. As you do this, the visual representation will change and the label and button positions will be adjusted. It is

![Figure 10b: The same application with a change to the layout manager used by the bottomPanel.](image)
doubtful that the components will always be moved where you would like them. Check the components’ 
“constraint” properties and adjust so that the label is constrained to the West while the button is constrained to 
the East. After making these changes and running, gives the results shown in Figure 10b. Figure 11 shows the 
visual, source and Java Beans views of the interface so far.

6. Responding to the User

In a real application, user actions such as button presses should direct the behavior of the application. A 
quick way of indicating this in a prototype is to have the prototype display a message stating what should 
actually happen. Running your application now and clicking the “New Game” button, doesn’t even do that. To make the button do something, we have to add at least a minimum of event handling to the program.

Position the mouse over the New Game button and bring up the right menu. Select Events --&gt; actionPer-
formed. This will create an event handler for the button so it can respond when clicked on. If you have a 
source view open when you click on the actionPerformed item in the Java Beans view you can see the Java 
source code involved. It creates an inner class that doesn’t do anything but send a message to System.out.

We can modify that message so it now says, “Start a New Game”. Try that and run the code. Now the but-
tton click does something but not very much - it’s a prototype. Next we’ll look at adding some more function-
ality.
7. Making the Prototype Almost Work

The current version of the board panel cannot determine where the user has clicked within it. It should be developed further to better represent 9 distinct board positions. It might be that during product development, different groups will work on different parts of the application at the same time. One group might be working on general aspects of the interface while another is working on specific components - like the board panel for instance. As mentioned earlier, an underlying model provides the core logic of the application so still another group could be working on the model.

These different groups need to communicate with each other to agree on the responsibilities and communication aspects of their respective components. Here we will expect the model to be able to respond to requests to place a marker at a given position on the board. The model should also have methods which allow the GUI to query it so it can be properly represented. In Java, it is often helpful to develop an interface which specifies the features expected of various objects. Figure 12 shows the interface developed for the tic tac toe model.

```
package ex484;

public interface TTTModel {
    public static final char PLAYER_X = 'x';
    public static final char PLAYER_O = 'o';
    public static final char EMPTY = ' ';
    public static final char NO_WINNER_YET = 'N';
    public static final char TIE = 'T';
    public void newGame();
    public boolean isAutoblockOn(char whichPlayer);
    public void setXautoblock(boolean desiredState);
    public void setOautoblock(boolean desiredState);
    public char[] getBoard();
    public char getWhoseTurn();
    public void placeMarker(int pos);
    public boolean wasPreviousMoveLegal();
    public char isWinner(char[] mark);
}
```

```
// Used to report board location clicked
package ex484;

public interface BoardReporter {
    public void gridClickedAt(int loc);
}
```

Figure 12: TTTModel is the Java interface for tic tac toe model objects

Now the group developing the board panel will know that it can pass the user specified board location (user clicks on a board location) on to a model with the method “placeMarker(pos)”. The view aspect of the GUI can obtain information about what is on the board with the method “getBoard()”. The view can also know whose turn it is and if there is a winner yet with methods “getWhoseTurn()” and “isWinner(mark)”. Once the interface is defined, the prototype can use an incomplete version of the model while development of the actual model is continuing. Other interfaces might also come in handy as shown in Figure 13.

Create the files TTTModel.java and BoardReporter.java within your ex484 source package directory. Replace your version of the TTTBoardPanel.java with the upgraded version (link on the web). Create a new source file, TTTGame.java, in package ex484 representing the model. You can obtain it from the website.
If you open the TTTAppGui, you should still see it displayed visually but now the board looks like a blank 3x3 grid. We need to hookup the model to the rest of the application.

1. Near the beginning of the source code, add a declaration for a model.
   ```java
   private TTTModel model = new TTTGame();
   ```

2. The board panel reports to the rest of the GUI using the BoardReporter interface. This is easily done by making the TTTAppGui class implement the interface. So change the class declaration to make it so.
   ```java
   public class TTTAppGui extends JFrame implements BoardReporter {
   ```

3. Now that you’ve said that the class implements the interface, it needs a “gridClickedAt” method.
   ```java
   public void gridClickedAt(int loc) {
       updateGUI();
   }
   ```
   This method notifies the GUI that the user tried to make a move meaning that the model may have been changed. The call to updateGUI lets the view update itself based upon the model.

4. The code for updateGUI gets the latest information from the model to update the view. For instance, the board has changed, it’s another player’s turn or the game is over. This updateGUI may ripple down through other components that form the view (like the board panel). Add the code for this method.
   ```java
   protected void updateGUI() {
       theBoardPanel.updateGUI();
       int winCode = model.isWinner(new char[9]);
       String msgString = "Unknown";
       switch (winCode) {
           case TTTModel.NO_WINNER_YET:
               if (TTTModel.PLAYER_X == model.getWhoseTurn()) msgString = "X's move";
               else  msgString = "O's move";
               if (!model.wasPreviousMoveLegal()) msgString = "Still " + msgString;
               break;
           case TTTModel.TIE :
               msgString = "Tie Game";
               break;
           case TTTGame.PLAYER_X:
               msgString = "X Wins!";
               break;
           case TTTGame.PLAYER_O:
               msgString = "O Wins!";
               break;
       }
       statusMsg.setText(msgString);
   }
   ```

5. Change the initialize method to hookup the connection with the board panel. The board panel now be able to communicate with the model and the board panel can also signal the rest of the GUI by calling its BoardReporter’s gridClickedAt() method.
private void initialize() {
    this.setContentPane(getJContentPane());
    theBoardPanel.setBoardReporter(this);
    theBoardPanel.modelHookupInitialize(model);
    this.setSize(312, 338);
    this.setTitle("Tic Tac Toe");
}

Now when the code is run, you should start out with an empty board. Clicking on the board will switch locations to the current player’s marker (X or O). The model’s logic is not complete so don’t expect all of the features to work correctly. To run the code, we use a build file. This allows a more portable way of working with images and code in general. The build file has instructions on how to compile and run code and where the compiled code should be placed. To run it, open the build.xml file and look at it with the outline view. Right click the target run, submenu Run --> Ant Buildfile. You can also make a jar file (archive) so that the code can be easily transferred to another machine (images included).

**Figure 14:** The build.xml file is used by Ant to automate common tasks like compiling and running applications

**Figure 15:** The package explorer view of your project should look like this now. The image folder contains all the images used by the application.

You should be able to hookup the New Game button to the model now too. Give it a try.

Another proposed feature is to add check boxes so each user can turn on and off a “Safety Block” feature. Place these components above the game board and consider the connection to the model.