Lab 1 – Tic Tac Toe

Objectives

- Setting up a basic GUI using the MVC design pattern

Reading

You should complete the reading either before or while completing the lab:

TO READ:

This reading was assigned at the end of the last class. It is reiterated again to stress the importance of reading it.

Java Tutorial – Lesson: Getting Started –
http://docs.oracle.com/javase/tutorial/uiswing/start/index.html
Read through the entire trail. It is just a very brief intro to using Swing. It has one example program called HelloWorldSwing. We worked through a better example in class, so don't feel like you need to grab the code and run it.

Java Tutorial – Lesson: Swing Components –
http://docs.oracle.com/javase/tutorial/uiswing/components/index.html
There is a LOT of information here. I want you to go through it enough to understand the different types of information available for your future reference. Try to pay close attention to the simple controls. You will get a sense of how much capability lies in each control. Bookmark the important pages, including the page that describes how to use all of the Swing components: http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html You will frequently jump back to these pages for reference!

Java Tutorial – Lesson: Layout Out Components Within a Container -
http://docs.oracle.com/javase/tutorial/uiswing/layout/index.html
Read through this section, starting with **A Visual Guide to Layout Managers** through **How to Use…** . In particular, pay CLOSE attention to how to use BorderLayout, and GridLayout, as you will use both of those in this lab.

Java Tutorial – Lesson: How to use Borders -
http://docs.oracle.com/javase/tutorial/uiswing/components/border.html This is one of the pages contained above. I'm listing is explicitly to draw your attention to this important page. Read through it to get an understanding of how to set borders.

Sarang's Java Programming on Safari, Chapter 14. Read the first two sections titled, "Layout Managers", and "Using Layout Managers".

Introduction – The Model/View/Controller Design Pattern

We have started learning how to design GUI applications using the Java Swing framework. For this lab, you are going to take the first steps toward getting a working GUI for a basic tic-tac-toe game. If you are

not familiar with the game, jump over to this page to remind yourself how it is played: http://en.wikipedia.org/wiki/Tic-tac-toe

For this exercise, we are going to follow a very important design pattern for GUI development – the **Model View Controller** design pattern. We will (or already have) discussed this extensively in class. To summarize, the MVC pattern separates the code development into three general classes:

- **Model** – The classes representing the model of the program contain all of the data and control logic behind the guts of the program. IMPORTANT – there is NO GUI CODE contained in the Model code. The model depends on only other classes which are involved with maintaining the data and logic behind the program
- **View** – This is the viewable representation of the model. For example, consider a spreadsheet of salary data for a company. You can view the data as a table in a spreadsheet, a bar graph, a pie chart, etc. The important part is that the view can change depending on the needs of the user, but the data is independent of the view. What is viewed depends on the model. As the model changes, the view will need to update.
- **Controller** – This is the connection between the model and the view. The controller handles all user input, and updates the model and the view accordingly. It tends to be closely tied to the view, and in fact, you may see that in Swing development, sometimes it is acceptable to combine the View and the Controller into one group. When this is done, it is generally referred to as the **User Interface** (UI). However, strict MVC design separates a GUI app into these three groups.

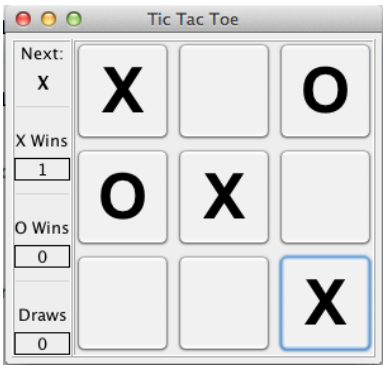
How will this break down in your tic tac toe program? You are going to develop the following classes, over the course of this lab, and the next homework assignment:

Model

- `TTTBoard` – This class represents all of the data and logic behind the tic-tac-toe game. It will have absolutely NO GUI logic in it whatsoever. It contains the data structure to manage the state of the board as the game is played, contains methods for changing board values, and most importantly, it will contain methods for detecting whether or not a player has won, lost, etc. It will eventually contain methods to help the computer play "intelligently." Most importantly, there should not be a single line of GUI code in this class.

View

- `TTTMainView` – This will present the main frame for the program, and is extended from `JFrame`. Your frame will eventually look something like this (with some additional controls needed for the final game):



As a top-level container class, it has a content pane that acts as a container to store the components to be viewed. The content pane in `JFrame` is initialized with a `BorderLayout` layout manager.

You will notice that there are two panels in the GUI:

- `TTTBoardPanel` – This is extended from `JPanel`, and has a `GridLayout` displaying 9 `JButton` instances in a 3x3 fashion, each button representing an individual square in the game. This panel is placed in `BorderLayout.CENTER` position in the `TTTMainView` content pane.
- `TTTStatsPanel` – This is also extended from `JPanel`, but arranged as a `GridLayout` of 1 column. There are six `JLabel` components, three acting as simple labels `"Next:"`, `"Wins"`, and `"Losses"`, and three acting as actual placeholders that the controller will handle updating as the game progresses. Since this represents the primary data to be displayed in this program, it is placed in `BorderLayout.WEST`.

Controller

- `TTTController` – this is a class that takes a `TTTBoard` instance, and the `TTTMainView` instance, and handles the connection to the underlying board data and logic, and the UI interaction when the user presses buttons. This will be done during the next lab.

You will have a main program, contained in an entirely separate class called `TTTMainGame`. This code simply instantiates the model, view, and controller, ties them together appropriately, and displays the frame. At this point, the Swing framework takes over, using your listeners to capture any user interaction with your program.

For this lab, you are going to set up the Model and the View.

Exercise 1 – Thinking about the Model

In the MVC, the model is the data and control logic in your system that is independent of any GUI code. In otherwords, your model should be able to operate independently of the type of GUI that is displayed. There really is no point in working on the view until you understand the data and logic contained in the model.

Let's go through the very basic description of the game:

Simplified description of tic tac toe (adapted From Wikipedia - http://en.wikipedia.org/wiki/Tic-tac-toe

Tic-tac-toe is a game for two players, X and O, who take turns marking squares arranged on a 3x3 grid, which we will call the *board*. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row on the board wins the game.

Before you begin, step away from GUI thinking for a moment. Remember, a model behind a GUI application should represent the data and the logic behind a system independently of the GUI. So, read through that description. Pay attention to the nouns, as these are good candidates for classes. Minimally, you should identify the following classes:

- Game
- Player
- Square
- Board
- Mark

You will adapt these classes for the model of a tic tac toe game. However, before you begin, take a moment to draw up a UML diagram. Show the relationships between each class, and include multiplicities. Your diagram should show a single Game class in the center. A Game has two players. A Game has one Board. A Board has nine Squares. Each Square can have 0 or 1 Marks. Etc. Your single UML diagram should be able to depict all of this clearly! Don't worry about outlining every method and attribute for the class. This first exercise is designed to get you thinking about the game, and more importantly, to get you in the habit of **thinking about your object design before you code**!

Exercise 2 – Implementing the Model

Before you start working through the steps below, take a moment to make some quick notes of what behaviors each of the objects listed above should have. Remind yourself of the coupling vs. cohesion argument discussed in lecture.

ALL CLASSES IN THE MODEL WILL BE STORED IN THE `game` PACKAGE!

Step 1 – Create a `Mark` enumeration

Players create "marks" in squares on the tic tac toe board. So, let's create an abstraction of the different marks that players can make in the game. This saves from error checking for bad data being entered in the board. Set up an enum called `Mark` that has three constants:

- `NONE`
- `X`
- `O`

You can create this as a separate enum in **`Mark.java`**, in the `game` package. (If you did not create a `game` package, do so now!)

Step 2 – Create the `Board` class to represent a tic tac toe board

A board is a 3x3 grid of squares, where each square can store a `Mark`, as we've indicated above. You might want to create a separate Square class, but this is not really necessary at the moment. Create a

new class called `Board` in your `game` package. Start by listing your first private member as a 2D array of `Mark` enum values. We'll call it `board`. This will implicitly represent the 9 squares on a board, and keep track of what mark has been placed in each square:

```
public class Board {
    private Mark[][] board;

    …
}
```

Next, consider the different states that a board can be in:

- `NEW` – New board just created or cleared out – i.e. no marks have been placed
- `PLAYING` – In the middle of a game – i.e. nobody has won, and there isn't a draw
- `WIN` – A player has won
- `DRAW` – Every square is filled with a `Mark.X` or `Mark.O`, and nobody won.

Inside of your `Board` class, create a `State` enum with the above four constants. So far, the start of your class in **`Board.java`** should look as follows:

```
public class Board {

    enum State {
          NEW, PLAYING, WIN, DRAW;
    }

    private Mark[][] board;
    private State state;

    // Methods begin here…
```

Notice that we created the `enum` inside of the `Board` class. This is a completely legitimate design using a technique known as **Inner Classes**.

Now, start working on your methods. Remember, this class represents all of the logic for managing a board. However, it does not handling controlling game play! Minimally, you should have the following members:

- Constructor – create the 2D array for `board`, then initialize the `board` with nine `Mark.NONE` values. Initialize the `state` to be `State.NEW`
- `public void clear()` – clear the board (i.e. set all spaces to `Mark.NONE`)
- `public boolean setMark(int row, int col, Mark m)` – set the space at [row][col] to the specified `Mark m`. Be sure to call `updateBoardState()`. However, only allow it if the board is NEW or PLAYING, and the square is empty. If the play could be made, return true, otherwise, return false. This is going to be the critical method for a player to make a mark on the board.
- `public Mark getMark(int row, int col)` – get the mark at [row][col]
- `public State getState()` – get the current board state
- `public String toString()` – Return a String to represent the object (i.e. the board, with marks, and the board state)

- Some other *helper* methods you might find useful:
  - `isPlayable()` – Boolean method that returns `true` if the board state is not a WIN or DRAW
  - `isEmpty(int row, int col)`
  - `isDraw()`
  - `isWinFor(Mark m)` – Check to see if the current board is a win for the mark specified in `m`.
  - updateBoardState() – A private helper method to update the `state` of the board depending on the current marks on the board.

JUNIT – Create the appropriate JUnit tests for the important methods in this class.

Step 3 – Create the `Player` class to represent a player in a game

A player is an object that actually makes marks in the board. Ultimately, for the third part of the assignment, you are going to have two types of players – Human and Computer. For now, just create a simple player class.

A player has a `Mark` it always plays (either an X or O). Also, it has a single board that it always plays on. It can do one thing – attempt to make a move on a board.

Create a new class called `Player` in the game package. It should have two private members:

- `theBoard` – a reference to the actual board that the player is playing on
- `myMark` – The mark that the player always plays with (will always be either `Mark.X` or `Mark.O`)

Create the following methods:

- Constructor – Takes two parameters, each to initialize the above data members
- `public boolean move( int row, int col)` – this is the function that is called when a player makes a move at a specified row, col position on `theBoard` (i.e. it should use `setMark` of the `Board` class). It should just return what `setMark` returns.
- `toString()` – return a string that represents the current player, using `myMark` as the string output.
- You might ultimately want other `get` methods? Perhaps not, but with all of these classes, keep in mind that you might need a few.

Go back to the `Board` class:

- Add an additional public method called `isWinFor(Player p)` that returns `true` if the current board is a winning board for the current player.

Again, you may (or may not) want other methods in all of your classes as you progress through your code.

JUnit – Create the appropriate JUnit tests for the important methods in this class.

Step 4 – Create the `Game` class to represent an entire game

Go back to your original UML diagram. What does a `Game` consist of? This class should represent everything required for a single game of tic tac toe.

Private members:

- `private Board theBoard`
- `Player players[]` – An array of two players, initialized to an X player, and an O player, both using the same instance of `theBoard`.
- `int iCurPlayer` – An integer representing which player's turn it is.

Methods:

- Constructor – No parameters. It initializes the private data, and sets the current player to 0.
- `playGame()` – This is a test method that is designed to test out the game functionality. In a loop, do the following:
    - print the board
    - print the current player
    - ask for a position on the board to play
    - make the move. If it was invalid, then go back to the previous step.
    - Is the board still playable? Then change the player, and go back to the top. Otherwise, print who won, or whether the game was a draw.
- Create a static `main` method in the class that instantiates one `Game` instance, and calls the `playGame` method.

JUnit – Create the appropriate JUnit tests for the important methods in this class.

Exercise 3 – The View

Follow these steps to get a basic GUI working. You will not have any real functionality yet.

1. In your `Lab15` project, create a new package called `gui`. This package will store all of the view classes.

2. Create a new class called `TTTBoardPanel` in the `gui` package, and make sure it is extended from `JPanel`. This will represent the actual Tic Tac Toe board. In this class, you should have one constructor. In the constructor, do the following:

    a. Call `this.setLayout` to use a `GridLayout`, initialized to be a 3x3 grid, with 5 pixels for spacing on all sides

    b. Call `this.setBorder` to initialize a raised `EtchedBorder` around the board. (Yes, you will need to refer to the API and / or the tutorials to figure out how to do this!) Feel free to experiment with other borders if you want.

    c. Use `this.add` to add 9 `JButton` instances to the panel. You might want to consider using a loop to instantiate them, rather than create 9 different variables. For each button object, use `setPreferredSize` to set each button to have a size of 75x75 pixels or larger. (HINT: Use `new Dimension(75,75)`. I also recommend that you

store the references to each of these instances in an instance variable that is a two-dimensional array of size [3][3]. You might find this handy later.)

3. Create a new class called `TTTMainView` in the `gui` package, extended from `JFrame`. This class will represent your main frame for the app. For now, we will place the above `TTTBoardPanel` in the frame so you can make sure it looks correct. NOTE: Do NOT create any `main` method in this class!

   In the constructor for this class, you should do the following:
   a. Set the default close operation to `JFrame.EXIT_ON_CLOSE`
   b. Set the title of the frame to `"Tic Tac Toe"`
   c. Set the frame to appear in the center of the screen.
   d. Set up a border for the content pane of the frame to be an instance of `EmptyBorder(5, 5, 5, 5)`. We will use the default layout manager of `BorderLayout`, so there is no need to change that.
   e. Create an instance of `TTTBoardPanel`, and add it to `BorderLayout.CENTER`
   f. Call `this.pack()` to compress the size of the frame to the smallest size needed to display all of the components properly. (This will be determined by any calls to `setPreferredSize` on any of the components displayed. Since you called this method on the buttons displayed on the board above, these will ultimately determine the minimum size of the entire frame.)

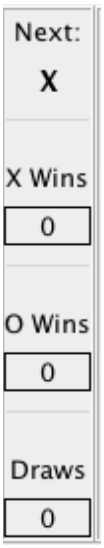
5. Next, we will create the stats panel that will display information about the game being displayed. Create a new class called `TTTStatsPanel` in the `gui` package, and make sure it is

extended from `JPanel`. In the constructor for the class, do the following:

a. Call `this.setLayout` to use a `GridLayout`, initialized to be a grid of 1 column (set the number of rows to 0), with 5 pixels for spacing on all sides

b. Call `this.setBorder` to initialize the board of the panel to use a raised `EtchedBorder`. (Same as above.)

c. You will use eight `JLabel` instances and three `JSeparator` instances to create a panel to allow the user to track game progress. Four `JLabel` instances are just labels, but four are important indicators that will be updated as the game progresses: one to show the next player, and three to show the number of wins, losses, and draws, respectively. Your panel should look like this:

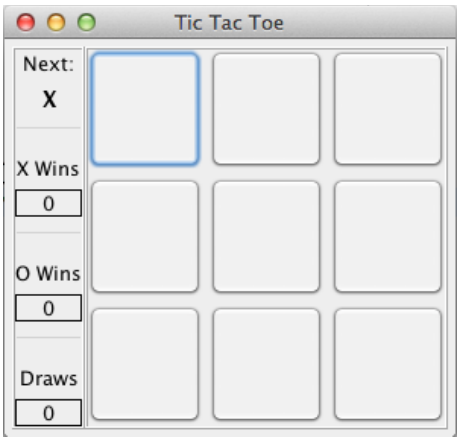


To center the labels in the panel, use the `setHorizontalAlignment` method with a parameter of `SwingConstants.CENTER` on each label.

Place a border around the number of wins, losses and draws, use the `setBorder` method with an instance of `LineBorder`, with a parameter of `Color.BLACK`.

d. Make the label for the next player (the X above) to appear in **bold**. Do some research to figure it out. HINT: You'll need to use a `Font` instance, and the `setFont` method of the label.

6. Go back to your `TTTMainView` constructor, create an instance of `TTTStatsPanel`, and add it to `BorderLayout.WEST`. Rerun your program and make sure everything is displayed correctly. It should look as follows:



Your next lab will get you through connecting the model and the view with a controller.

Exercise 4 – The Controller

Now, you will set up a very basic controller. Create a class called `TTTController` in the `game` package. Set up the class to implement `ActionListener`. Remember that a controller in an MVC design should connect the model logic, and the view. Thus, there should be (at least) two data members:

- `private Game theGame`
- `private TTTMainView theView;`

The object theGame represents the "model" in your program. Remember – this is the core data and logic in your program that is independent of your GUI code. The controller will handle updating the model based on any GUI events that occur.

For your methods:

- Your constructor takes two parameters, to initialize the private members.
- You need to implement the `actionPerformed` method, which is the actual method that will be called when an `ActionEvent` is generated. Each one of the buttons showing the squares can generate this event. Let's set up the event handler to deal with each possible square that might be clicked. Your logic is going to need to do the following:
  - Determine which button was pressed from the view
  - If the game is over, then do not process anything further. Just display an appropriate `JOptionPane` indicating that the game is over, and ask the user if they want to start a new game. If they do, then clear the board in `theGame`, and update the buttons to clear the X and O from the boards, and return.
  - If you get here, then the board is still playable. So, call the `move` method of the current player to simulate the move in `theGame`, and be sure to update the next player id in `theGame`. Notice – if you designed the move method correctly, then it should already be updating the state of the board in `theGame`. (There are several ways you can proceed with managing the state of the actual game.)

- Update the view to reflect the actual model in `theGame`. (This means, updating the buttons, and updating the status panel with the next player turn)
- Finally, check the board state. Is it a win? Then increase the number of wins in the stats panel, and show a `JOptionPane` indicating the winner. Is it a draw? Then show a `JOptionPane` indicating a draw, and do not increase either player win.

Do NOT forget to go back into your constructor, and add this as an action listener for each of your 9 playable buttons!

Exercise 5 – Update your UML diagram

Your UML diagram will likely need some updating to reflect what the actual design looks like. Be sure to show the entire design, including all classes in the `gui` and the `game`.