Lab 2 – Tic Tac Toe, Part II

Objectives

- Setting up a model to represent the game.
- Setting up event handlers for the GUI.

Prepping for the Lab

There are no steps required to prepare for the lab, other than reviewing your understanding of the game of tic tac toe. ☺

Reading

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

TO READ:

Java Tutorial – Nested Classes -
http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
Read through this section to help you understand more of the Java language support for static nested classes vs. inner classes vs. anonymous inner classes. All of these techniques are heavily used in GUI development.

Sarang's Java Programming on Safari, Chapter 13. Read the entire chapter, titled "Event Processing and GUI Building." This has a lot of information on writing event handlers. You will notice that, in these examples, the author fails to place the initial command to make the GUI visible in the Event Dispatch Thread. Though it will work for the vast majority of our examples, it is not a good idea to neglect the EDT in GUI development! Just focus on the material for event handling.

Java Tutorial – Lesson: Writing Event Listeners –
http://docs.oracle.com/javase/tutorial/uiswing/events/index.html As indicated in lecture, this is a very valuable source for understanding the wide range of different events that you can listen for. Read through the following sections:

- Introduction to Event Listeners
- General Information about Writing Event Listeners
- Listeners Supported by Swing Components

In this tutorial, the vast majorty of remaining sections will describe individual examples on how to write listeners for every possible event. You will refer to this frequently throughout the rest of the semester.

Introduction – The Model/View/Controller Design Pattern

For this second part of the lab, you are going to complete two parts:

- The Model – You are going to implement the critical classes to represent the tic tac toe board, and some very basic logic to play through a game
- The Controller – You will set up your first event controller to allow you to place X's and O's on the board, and use the model logic to manage the board state and check for win, loss, or a draw (i.e., every board square is filled.)

Exercise 1 – Thinking about a design

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 – Establishing the model to represent the board and game logic

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.

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.

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`)

- `private void updateBoardState()` – Create a private helper method to update the state of the board (the `state` instance variable) depending on the current marks made on the board. This is a private method, as it will not be used outside of the class.
- `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`.

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.

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 row / column 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.

Exercise 3 – Setting up event handlers

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;`

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.

Now, go back into your constructor, and add this as an action listener for each of your 9 playable buttons!

Exercise 4 – 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.