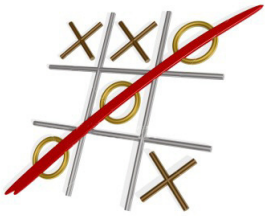
CSE 331, Spring 2011
Homework Assignment #5: Tic-Tac-Toe (50 points)
Due Monday, May 9, 2011, 8:00 AM

This program focuses on creating a graphical user interface in Java with AWT/Swing, event-driven programming, class design, and design patterns. Turn in the Java files listed below from the course's homework web page. This is an **individual** assignment; you should work alone, but you may have a "design buddy" as in past work.

There are no support files provided for this assignment. We also do not specify what classes you should have, other than a class named `TicTacToeMain` that contains a `main` method to launch the program. You do not need to turn in any testing program (such as a JUnit test case) on this assignment, but you are welcome to do so if you like.

FYI: The code you write on this assignment may form the basis for the next homework assignment.

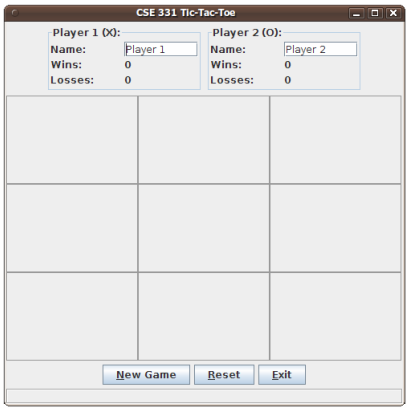
Background Information about Tic-Tac-Toe Game:



Tic-Tac-Toe is a simple game for 2 players played on a 3x3 grid. Each player is given a letter; the first player is "X" and the second player is "O". The players take turns where each places his/her letter in a square of the board until one player wins by placing three of his/her symbol in a line. The line can be horizontal, vertical, or diagonal. The game also ends if the entire board fills with no player completing such a line; which represents a tie.

Tic-Tac-Toe is a simple game that is often used when teaching game theory and combinatorics, because it is possible to exhaustively consider every possible sequence of moves when deciding what move to make. There is an optimal strategy for each player, and if both players follow optimum strategies, the first player to move ("X") has an advantage. Optimal first-player play can force no worse than a tie for every game.

Program Description:



In this assignment you will design and implement a basic graphical user interface for playing a sequence of games of tic-tac-toe. Your program will keep track of game state and information about the two players, including their names and how many games each has won and lost. Both players in the game are controlled by the human users sitting at the computer; you do not need to implement computer player logic.

The GUI window is of size 500x500 px. Its top area consists of a pair of top sections about the two game players, including each player's name in an 8-character-wide text field, wins, and losses. Each player information area appears at its preferred size, just large enough to fit its contents, and is surrounded by a titled border. The text labels for each player at left and right have equal width and are aligned into two columns.

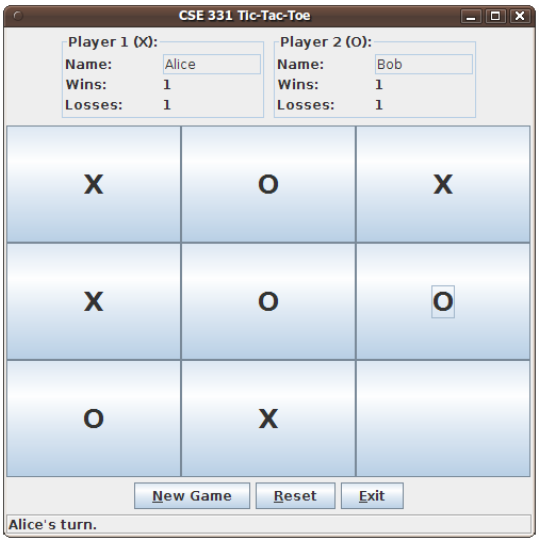
The middle region of the GUI contains a 3x3 grid of nine buttons representing the squares of the game board. The 3x3 grid of buttons is sized to fill all available extra space in the window. The text on each button is shown in a 24-point, bold font, derived from the same font face as the default font for buttons.

Underneath these buttons are three other buttons for starting a New Game round, Resetting the game state, and Exiting the application. These control buttons are centered and sized to their preferred sizes. Under these buttons is a status label stretching horizontally across the bottom of the window that displays various information during and after the game. This label's text is initially "Welcome to Tic-Tac-Toe!". The status label is surrounded by an "etched" style border.

The program can be thought of as being in various states. Initially when the program loads, the game is not yet begun. In this state, the main game buttons are disabled and cannot be clicked. The user is able to edit the players' names. The initial name text is "Player 1" for the first player and "Player 2" for the second player. If the user types an illegal player name, a dialog box (option pane) appears reading, "Illegal player name(s)." and the game does not begin.

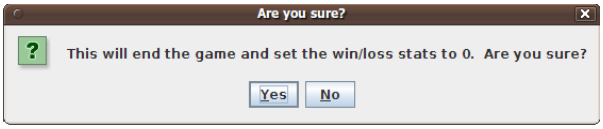
Once the players have typed their names and the user clicks New Game, a game begins. Now the program can be thought of as being in the "game in progress" state. While a game is in progress, the nine game square buttons should enable, allowing the user to click them. A game in progress has a notion of which player's turn it is. The first player, X, always gets to play first. When a player makes a play on an empty square, that player's symbol (X or O) is placed into that square and it becomes the other player's turn. During the game, the status label should always display information about the current player whose turn it is. For example, if the first player's name is Alice and it is her turn, the status label should read, "Alice's turn." While a game is in progress, your GUI should prevent the user from changing the text in the player name fields. If the user clicks on a square that is already occupied, nothing happens.

If a player makes a play that leads that player to have 3 in a row, the game ends and that player is the winner. If all nine squares become occupied without any player achieving 3 of their symbol in a row, the game ends in a tie. Once the game ends, the program can be thought of as being in "between games" state.



In the "between games" state, the game should not allow any further plays to be made; the nine game square buttons should be disabled. The two player name fields should become editable again, and any changes made to them should be reflected for future games. The status label should contain text reflecting the result of the most recent game. If a player won the game, the message should say so; for example, if the winning player's name is Alice, the status label should read, "Alice wins!" If the game ends in a tie, the status label should read, "The game ends in a tie." If the user clicks New Game, another game round begins.

If the Reset button is clicked, a confirmation dialog box (option pane) appears, asking to confirm the action. It shows the text, "This will end the game and set the win/loss stats to 0. Are you sure?" If the user clicks the Yes button, the program's state should return to exactly the way it was when the program first loads. That is, the player name text fields should become editable and reset their text to "Player 1" and "Player 2"; the win/loss records of both players should reset to 0; any game in progress should stop; the nine game buttons should disable; any game in progress should immediately end; and the bottom status label's text should return to its original state of "Welcome to Tic-Tac-Toe!"



If the Exit button is clicked, the program should immediately exit.

A **sample solution** will be posted to the course web site that you can run to verify the behavior your program should have. You can run the sample solution to verify your understanding of the program appearance and behavior described here. You should match this spec and the sample solution both in terms of the GUI's appearance and resizing behavior, and in terms of the game behavior and logic. The sample solution is our best effort to try to meet this spec, but it might contain minor bugs or errors. If the sample solution's behavior differs from this specification, follow this spec, not the sample solution. Please ask if you are unsure.

Classes to Implement:

This document does not specify exactly what classes you should write, nor what behavior, fields, methods, etc. each class should have. Part of this assignment is for you to choose appropriate classes and contents for each class based on the specification of desired functionality and based on our class discussions of proper class design. You may use any classes you like, but please don't use packages on your classes; place all of them into the **default (unnamed) package**.

A major part of your grade will come from how effectively you design the contents of each class to solve this problem elegantly. You should also incorporate **design patterns** taught in class as appropriate to help you solve this problem.

You don't need to use every pattern taught in class; only use the patterns when they improve the program's design. But you should consider using patterns such as Observable, Factory Method, Composite, and so on.

In general, if an entity is important in this system, you should represent that entity with a Java class. It would be inappropriate to try to solve this program using only one or two classes; there are more important entities in this system. If you opt for a design that has too few classes and omits major entities from the system in favor of simple variables or collections, you may not receive full credit.

Your program uses a **graphical user interface (GUI)**. You are to implement this GUI using the Java Swing library as taught in class. Your GUI code should be well designed and decomposed with code that is easy to read and modify.

The core data classes you add to the program should not perform any **user interaction**. That is, there should not be any `println` statements or `Scanner` input from the console performed directly by them. Instead, your user interface classes should perform all graphical user input/output. Another way of saying this is that you should have good model-view separation in your program.

Do not place core system functionality into the GUI classes. The primary purpose of the user interface classes should be to read input from the user and then hand off this input data to your other core objects to let those core objects do the bulk of the actual work and computations. For example, it is not appropriate for the text UI class to contain the data or code to know what player's turn it is, nor to look at the board squares to figure out which player has won the game.

Development Strategy and Hints:

Some programmers find it easier to develop a graphical program in a "top-down" fashion, starting with features that can be seen by the user (the GUI) and working downward to behavior that works behind the scenes (the core data and model classes). Others prefer a "bottom-up" design where they design and test the core data model classes first, then build a GUI around them that uses their functionality. The style to use is up to you, but we suggest choosing one of these strategies.

If you try to write all of the code at once and then compile/run the overall program, you are unlikely to succeed. This is a large system that uses all of your classes in complex ways. A small error in your code will stop it from functioning entirely, giving you poor feedback about what code does and does not work successfully. Therefore it is important to write and test your code incrementally. Remember that you can insert a "stub" version of a particular method or entire class that simply has a pair of empty `{}` braces for the method's body (or simply returns a dummy value like `null` or `-1`). This may allow you to test unfinished classes together.

It might help you to know that in our own sample solution to this program, we interact with the following Swing **components**: `JButton`, `JFrame`, `JLabel`, `JOptionPane`, `JPanel`, and `JTextField`. We also interact with other GUI-related classes such as `Font`, `Border`, `BorderFactory`, various layout managers, and action event listeners.

Recall that you can enable/disable a component by calling its `setEnabled` method, and you can set whether a text component's text can be edited by calling its `setEditable` method. Recall that you can set borders around various components using the `BorderFactory` class.

You can **exit a Java program** by calling the following method: `System.exit(0);`

To match the GUI appearance described, you will need to create a **Composite layout** consisting of several panels within panels. If your appearance does not match the expected appearance, consult the course materials about each layout manager's behavior, particularly with regard to whether it respects components' preferred sizes.

In part of this program you are expected to pop up a **confirmation dialog box** to confirm whether the user wants to Reset the game state. Do this using the static methods of class `JOptionPane`. Note that the default confirmation box has three buttons: Yes, No, and Cancel. You should change this to show only Yes and No buttons by passing additional parameters to your method call. In particular, you should pass `JOptionPane.YES_NO_OPTION` as the type of the dialog. Note that the game should reset only if the user clicks "Yes", which corresponds to a value of `JOptionPane.YES_OPTION` being returned by the static method call. Similarly, when the user tries to start a New Game with invalid (empty or all-whitespace) player names, a message dialog should pop up saying that the names were illegal. This message dialog should be shown as an "error message" dialog by passing additional parameters; in particular, pass `JOptionPane.ERROR_MESSAGE` to give the message box a special error icon and appearance.

Exception Checking and Error Handling:

Your classes should forbid invalid parameters. If any method or constructor is passed a parameter whose value is invalid as per this spec, you should throw an exception. In particular, you must enforce the following:

- No parameter passed to any method (except `equals`) should ever be **`null`**.
- Important strings such as names should not be **empty** or consist entirely of **whitespace**.
- Integer fields that store counts of real-world things, such as a number of votes, should always be **non-negative**.
- **Indexes**, such as row and column numbers, should always be in the legal range of indexes for the game.
- If you parse data from files as strings and turn the strings into objects, you should check that the strings are in a **valid format** and throw the exception if they are not.

In past assignments you were instructed explicitly to always throw an `IllegalArgumentException` on any error. In this assignment we will let you decide what type of exception is appropriate to throw. In many cases it should still be `IllegalArgumentException`, but in some cases it may make more sense to throw other types such as `IndexOutOfBoundsException` when a client mistakenly tries to access an index outside the bounds of the board.

The main program should not throw any **exceptions** to the console. If any errors occur, it is considered a bug in your code. It is okay (encouraged) for you to incorporate exception throwing into the design of your core classes, but you should make sure to catch those exceptions in the GUI client or somewhere else in the active call stack.

Your GUI may perform its own checking for the validity of various user input values, such as making sure that names are non-empty. But this does not substitute for your core classes' own tests and exception checks. Valid arguments are preconditions for your methods, and preconditions are *supposed* to be things that the client could check for and avoid. Just because this particular client might do so does not absolve you from performing checks in your core classes' code.

Style and Design Guidelines:

It is always important to write your code in a general way, rather than tailoring it to a specific client or usage. In this program, though, you may assume that the board's size will always be exactly 3x3 and that there will always be exactly 2 players in the game. In other words, you don't need to make your code easily adaptable for a board of a different size.

You should, however, work diligently to reduce redundancy in your code. If you find yourself repeating code, make the code into a method or pull it out into its own class to reduce the redundancy.

The guidelines described on pages 7-8 of the **Homework 4 spec** also apply here, such as using a good object-oriented design, using design patterns as appropriate, following the Expert pattern, avoiding representation exposure, and ensuring invariants on the valid state of your objects by throwing exceptions. For brevity's sake, we will not repeat all of those guidelines here, though you are still responsible for meeting them as appropriate.

Document all of your files by commenting them descriptively *in your own words* at the top of each class, each method/ constructor, and on complex sections of code. Use **Javadoc comments** for all external documentation (atop class headers or public method/constructor headers). Include descriptive initial summaries along with proper tags such as `@param`. See Homework 4's spec for a more detailed description of our expectations regarding comments.

As a rough sanity check, our solution contains roughly **290 "substantive lines"** according to the class Indenter Tool.

Submitting Your Files:

Since we don't know what classes you will choose, you should **submit a .zip file** named `hw5.zip` containing all .java source files necessary to build and execute your program. We should be able to compile and run your code as follows:

```
javac *.java
java TicTacToeMain
```

The files in your ZIP archive should be in the root level directory of the ZIP archive; in other words, when we unzip your file, your .java files should appear in the current directory. Do not include .class, .txt, or other supporting files, only .java.