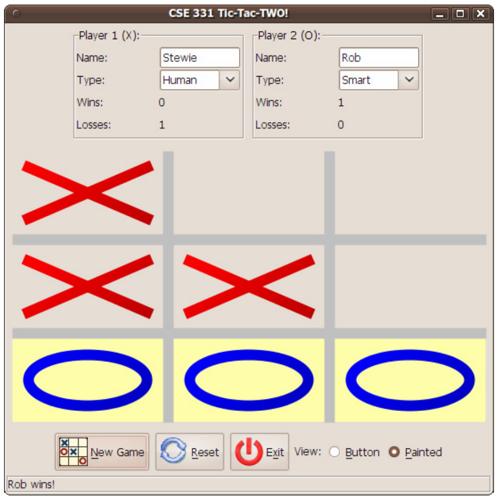
CSE 331, Spring 2011
Homework Assignment #6: Tic-Tac-TWO (50 points)
Due Friday, May 20, 2011, 8:00 AM

This program focuses on design patterns such as Strategy and Observer, Model/View/Controller, 2D graphics, Java features such as packages, and continued experience with graphical user interfaces and OO design. Turn in a ZIP archive of your Java files and other resources from the course's homework web page. This is optionally a **group** assignment; see the Groups section for more details.

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.



Player 1 (X):
Name: Stewie
Type: Human
Wins: 0
Losses: 1

Player 2 (O):
Name: Rob
Type: Smart
Wins: 1
Losses: 0

View: ○ Button ● Painted

Rob wins!

Enhanced Tic-Tac-Toe Game:

In this assignment you will create an improved and revised version of your Tic-Tac-Toe game from the previous assignment. You will add the following major features, as well as other minor changes described in this document:

- Two "views" of the game: a button view and a 2D painted view
- Human vs. computer player support with several strategy algorithms
- OO design improvements, such as design patterns and using packages

A detailed description of each major category of new functionality follows.

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.

Multiple Views:

Your GUI should be modified to support multiple views. A view is implemented as a class that extends `JPanel` and displays the state of the Tic-Tac-Toe game board using graphical components or 2D graphics painting.

The bottom area of the GUI containing the various control buttons such as "New Game" and "Reset" should be modified to contain **two radio buttons** to switch between the two views. A label of "View:" precedes them, and their text reads, "Button" and "Painted". Initially the "Painted" view button is selected and that view is showing. Exactly one button should be selected at all times. When a radio button is selected, the corresponding view immediately appears in the GUI.

The "Button" view is the existing view of the game state that you submitted in the previous assignment, with the game board shown as a 3x3 grid of buttons. You don't need to modify this functionality, though you may need to refactor it to make it modular enough to be used as a removable view.

The **"Painted" view** is a new view that you will add for this assignment. You will draw the state of the game board on a custom canvas component using **2D graphics** as taught in class. The exact appearance of this view is not specified in this document, so you can be creative here. But your view must have the following characteristics:

- It must draw the board using 2D graphics, and not as a set of existing components such as buttons or labels.
- Your 2D graphics should be **non-trivial**, such as using many colors/paints, gradients, shapes, stroke thicknesses, images, anti-aliasing, and/or other features shown in class.

- Your 2D graphics should draw any onscreen shapes by creating shapes objects rather than using the procedural drawing methods such as `drawRect` and `fillOval`.
- It must have some form of **hash lines** (roughly # shape) to divide the squares on the board.
- It must represent the entire 3x3 game board and show the **current state** of the game, including "X" and "O" (or similar markings) in each square that is occupied by each player respectively.
- It must have some indication of whether the **game is in progress**, such as by graying out the board or drawing it in a different color if the game is no longer in progress.
- It must respond to clicks on the various game board squares using **mouse events** to make players' moves.

Your view does *not* need to be able to handle **resizing** gracefully, but you may if you like. It needs to look reasonable only at the default 550x550 window size. Since different systems size their components differently, you should leave a bit of blank space around the bottom/right edges of your canvas to ensure that its contents will fit on all platforms.

Our sample solution shows one example of a painted view that would be considered acceptable. You should *not* attempt to exactly copy this appearance, though yours can be somewhat similar. You can also differ heavily from the expected appearance if you like, so long as your view contains the features described previously.

Hint: Recall that you may need to `validate` the layout of containers if you change their contents while onscreen.

Computer Player Strategies:

In the previous assignment your game allowed two human players to face each other. In this version you will also allow each player to be a computer player rather than a human. A drop-down combo box lists available strategies. "Human" is listed as the first strategy, representing a human player. When "Human" is selected, the user can make moves for that player by clicking the game board while it is that player's turn. When one of the computer strategies is selected, the computer player strategy will automatically and immediately make its move when it is its turn. It is allowable to have two human players, or one human versus one computer, or two computer players.

A computer player makes his moves using one of the following selectable movement strategies. Whenever a strategy algorithm describes moving randomly, the algorithm should choose from available choices with equal probability.

- **Random:** Randomly chooses any unoccupied square with equal probability.
- **Sequential:** Moves in the first unoccupied square, searching from left to right, top to bottom. In other words, this strategy first tries to move in row 0, column 0; if this is occupied, it tries row 0, column 1; then r0 c2, r1 c0, r1 c1, r1 c2, r2 c0, r2 c1, and lastly r2 c2.
- **Blocker:** Examines the game board and, if the human opponent has a "two-in-a-row" sequence that would lead to a win on the next move, blocks that sequence by playing in its third square. If the human player has multiple such sequences, this strategy plays in the square that comes earliest in row-major order that blocks a two-in-a-row sequence. That is, r0 c0 would have the highest priority, then r0 c1, r0 c2, r1 c0, r1 c1, and so on. If the human player has no "two-in-a-row" sequences (such as in the first few moves), this strategy randomly chooses any unoccupied square.
- **Smart:** This strategy is not a completely ideal Tic-Tac-Toe algorithm, but it is smart enough to beat most novice players. Each turn, its algorithm tries each of the following possible moves in increasing order from 1 through 5:
    1. *Win:* If the computer player has a sequence of two-in-a-row where the third adjacent square is unoccupied, play the third square in that sequence to win the game. (If the computer player has two or more two-in-a-row sequences, play the winning move that comes earliest in row-major order.)
    2. *Block:* Otherwise, if the human opponent has a sequence of two-in-a-row, play the third in a row to block the human from winning. (If the human player has two or more two-in-a-row sequences, play the blocking move that comes earliest in row-major order.)
    3. *Corner:* Otherwise, if any corner squares are available, randomly move in one of the corner squares.

4. *Center:* Otherwise, if the center square is available, move in the center square.
5. *Edge:* Otherwise, randomly move in any edge square (a square that is neither a corner nor the center).

Hint: When choosing a random move from a group of possible choices, it can be helpful to store the possible moves in a collection and draw an element from it using a random index (or shuffling the order of the elements in the collection).

If both players are computer players, the game will end immediately when the New Game button is clicked, because the game can make each player's move without any delay for user interaction. So upon clicking New Game, the entire game will be played out and its results shown on the GUI.

To make moves on the game, strategies will have to have some sort of communication with the overall game board in one direction or the other. Though you should not expose the game model's internal representation to strategies or otherwise, you may assume that there is not an evil or "rogue" strategy that would try to do malicious things such as making multiple moves or making poorly chosen moves on behalf of the opponent.

Though our GUI will use the first player as the computer player who uses these strategies, your model code should not otherwise assume that the first player is always a computer or that the second player is always human. In other words, don't hard-code the fact that you are strategizing for Player 1 into your strategy code.

Every strategy may assume a precondition while making its move that the game is still in progress and that there exists at least one empty board square to choose. It is your responsibility to ensure that it is used in such a way by your game.

Your GUI should modify its two upper player information areas to contain an additional pair of components underneath the players' names. The two new components are a label of "Type:", and a drop-down **combo box** containing the names of the strategies as listed above. Initially "Human" (representing a human player) is selected. When the user chooses a strategy from the combo box, the player should begin to use that new strategy for any following moves. The combo box should disable while a game is in progress, so the strategy can be changed only between games.

User Interface Tweaks:

Modify your program's graphical user interface to meet the following new appearance and behavioral constraints:

- The overall **window size** should be changed to 550x550 pixels.
- The **window title** should be changed to `"CSE 331 Tic-Tac-TWO!"`
- The window should appear in the **center** of the screen when it first loads. (See the method `getScreenSize` of the `Toolkit` class.)
- The three control buttons should now have **icons**. The New Game button should use an icon of `newgame.png`. The Reset button should use an icon of `reset.png`. The Exit button should use an icon of `exit.png`.
- Your UI should style itself to match the **native look and feel** of the operating system on which it is running. (See the method `setLookAndFeel` of the `UIManager` class. See p335 of the *Core Java* textbook.)
- The three game control buttons and the two new view radio buttons should be given **mnemonics**, which are underlined letters that can be used as hotkeys to activate a given control by holding Alt (Option on Macs) and pressing the underlined key. The Button view radio button should have B as its mnemonic; the Painted view radio button should use P; the New Game button should use N; the Reset button should use R; and the Exit button should use X. (See the `setMnemonic` method of graphical components.)

Design Improvements:

You should refactor and improve your program's code to meet the following new design constraints. If your code from the previous assignment already met some or all of these constraints, that's great!

- Your overall game model must be **Observable**. Your views must be **Observers** watching this model. As much as possible, all changes that occur in the appearance of the GUI on screen in response to game events should come through observer updates. The game model should notify its observers on any externally visible mutation to its state, whether or not that notification is needed by your GUI for this particular program.

- Your computer movement algorithms must implement the **Strategy** pattern as taught in class. In particular, strategies must fall under a common type hierarchy, and the rest of your model code should be largely ignorant of strategy algorithms. It should simply be supplied and use a strategy without worrying about which one it is.
- Create at least one **Factory** class with at least one factory method for constructing some kind of bulky objects. For example, you can create a factory that constructs graphical component(s) such as buttons. The factory should do the work of setting up the object, such as attaching a listener to a button or adding it to a container, etc.
- You should make your game's overall model into a **Singleton**, since there will be only one game at any time.
- You must use at least **one other pattern** in your solution. For example, you could use the Flyweight pattern for one of your classes if appropriate. Or you could use the Prototype pattern and cloning if necessary. Or you could use the State pattern to keep track of your game's state. (Using the Composite pattern for layout does not count. Using iterators of collections, while it may seem to be a use of the Iterator pattern, also does not count for this requirement. Making one of your own objects have an iterator, though, would.)
- You must organize the classes in your program into **packages**. You should have at least the following packages in your program, though you may add others if you like. If you do not have any classes to put into a given one of the packages below, that's up to you; but we generally expect that you will have at least one class in each.

o `ttt.model`	core game data model classes
o `ttt.view`	graphical user interface, view, and controller classes
o `ttt.strategy`	computer player movement strategies
o `ttt.util`	general utility code

  Your `TicTacToeMain` class to run the program should remain in the default unnamed package.

  See pp. 153-159 of the *Core Java* textbook or the lecture slides for more information about packages. Eclipse can help you move existing classes into packages with its Refactor → Move feature.

Style and Design Guidelines:

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. A major part of your grade will come from how effectively you design the contents of each class to solve this problem elegantly. In general, if an entity is important in this system, you should represent that entity with a Java class.

Part of your grade comes from having good **model/view separation**. In particular, the core data classes of your program should not perform any direct user interaction or store information directly related to a specific kind of user interaction. Instead, your user interface classes should perform all graphical user input/output. Also do not place core system functionality into the GUI classes that would be better placed in the model.

Your model classes are responsible for the same **error handling and exception checking** that was described in the previous assignment. Namely, you should check for `null` parameters, empty/whitespace strings, and negative numbers and throw an appropriate exception if a bad parameter is passed.

You should 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** 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.

Note that our grading process will focus largely on your new code and features and less on older features that already existed in the previous assignment. Though if your new code causes old features to break, you may lose points.

Teams and Buddies:

For this assignment you are allowed to optionally have one partner to work on the program. You aren't required to have a partner; you may complete the assignment by yourself if you prefer. If you choose to have a partner, the two of you may work together fully, sharing all aspects of your code and design with each other. If you have a partner, you and your partner should turn in a single copy of your assignment (you don't both need to turn it in). If you have a partner, make certain that you **document this in the comments** of your files so that the grader will know. You and your partner should decide on which of your two versions of the previous assignment will form the basis of this new assignment.

If you do decide to work alone, you may still choose to have one optional **"design buddy"** for this assignment. Your "design buddy" is one other student in the class with whom you are allowed to speak in great detail about the designs you both have chosen for your classes. In other words, you may share with your buddy the exact classes, methods, fields, etc. that you have chosen to use to implement this specification along with the reasoning behind those decisions, pros/cons you have discovered about your implementation of those decisions, etc. The goal behind this is to allow you a limited amount of greater collaboration, to help make the assignment more manageable, and for you to learn from each other.

If you choose to have a "design buddy," each of you should **document this in your code** by writing your buddy's name in your class header comments. (It is generally good practice to cite sources and collaborators in submitted code.) You should also include the design buddy in the `@author` tag of any Javadoc comment headings atop your classes. If you choose to work with a partner, you may not also have a separate design buddy.

The instructor and TAs will not be providing you with any help or resources to facilitate working as a team, such as shared file storage space, version control repositories, etc. It will be up to you to coordinate this with your partner.

If you do work with a partner, we expect that **both partners should contribute substantially** to the final program. It is not appropriate for one team member to do the vast majority of the work. To help us understand which partners worked on which parts of the program, please indicate in your comments which person was the primary author (the person who did the majority of the work) on each file you submit. If the two of you sat together and co-authored a file, please indicate this; but even in such a case, someone must have been at the keyboard "driving" the input of the code, so indicate this.

Submitting Your Files:

Since we don't know what classes you will choose, you should **submit a .zip file** named `hw6.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 top-level package folders and any .java files in the default package should appear in the current directory. Do not include .class files, but do include any supporting resources that you use in your program or view, such as images.