Make me a Tic-Tac-Toe game!

**You may work on this assignment in groups of 2 or individually.**

It will have 2 modes of operation: Client & Server.

The main() method for starting both versions is in one class called: TicTacToe.

It can be given either 1 or 2 command line arguments:

1. The first command line argument is the port number
2. The second command line argument is the host name of the server, indicating that this process should run as the client.

# Recommended design:

The program follows the Model View Presenter pattern.

## Server

The Server's presenter manages game state. It has one model which stores the game state. The presenter has 2 views. The first view is a GUI component. The gui displays a graphical representation of the game. The 2nd view is an adapter that interfaces via socket to the client version of the game (ServerSocketView).

When the game starts it displays its GUI and awaits an incoming client connection.

Upon connection, the game commences. By convention, the client is X, and X moves first. After X moves, O may move, etc. For each move, the presenter checks that the move is legal, performs the move, then checks to see if the game is over, either by a win or a draw. After each move, the presenter notifies both views (via a property change listener) and the views update. [See: PropertyChangeSupport to help you manage listeners.]

The GUI updates by refreshing its display, and the ServerSocketView updates by sending a message to the client describing the new game state.

When the game is over, a score is updated, and the same sequence occurs to send the game-over updates.

In the next round, the losing player moves first. If the round ended in a draw, then the player who moved 2nd will move first in the new round (they alternate).

This continues until either the Client or Server is closed.

The views also process events to operate the presenter. Some GUI view actions will result in method calls on the presenter. Incoming messages over the socket connection in the

ServerSocketView should be interpreted as actions of the player, and likewise should invoke methods on the presenter.

The socket adapter will need to have a thread to monitor the socket for incoming messages.

## Client

The client is designed similarly, except that its presenter defers to the server's presenter.

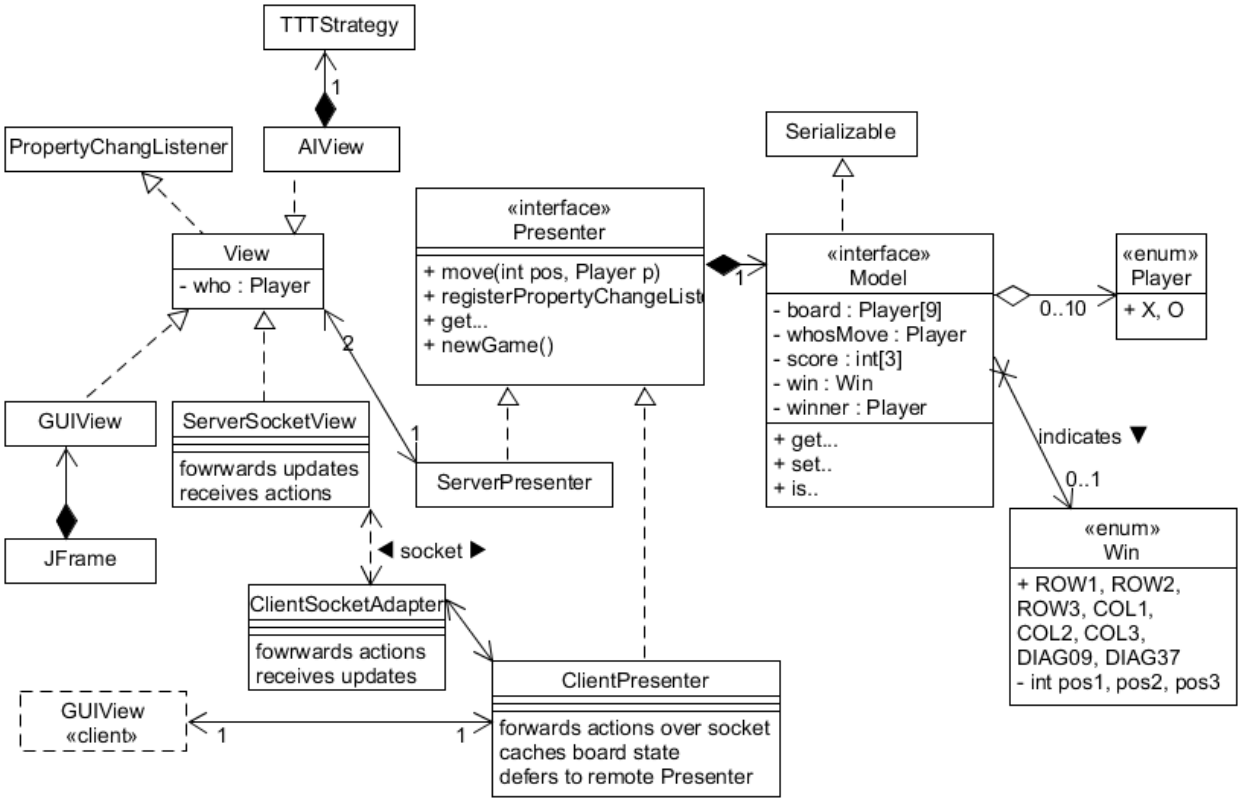
The client presenter may check for the legality of moves.

It's ClientSocketAdapter will craft messages describing user actions to be sent to the server's ServerSocketView, which will interpret the actions and pass them to the ServerPresenter. Likewise, game updates fromt the Server will arrive over the socket to the ClientSocketAdapter. The client will need a thread monitoring its socket for incoming updates from the server.

The client present will keep a cached copy of the game state, but will update it any time that it receives an update from the server. An updated game state results in it's GUI view being updated.

You can use the same GUI view for both client and server. [And you can use two of them on the server side before you implement the socket stuff to accommodate the client.

## UML




Note: AIView is NOT required. This is just indicating where you would insert an AI player.

This UML is neither complete nor completely accurate (eg: interfaces are shown as having variables). Use it as a guide for directing your design and implementation.

## Recommendations:

- Follow TDD for developing the Presenters.
- Use dependency injection to make things more testable.
- Use Java's built-in tools for marshaling.
- The client's presenter should disable input while awaiting a response from the server.