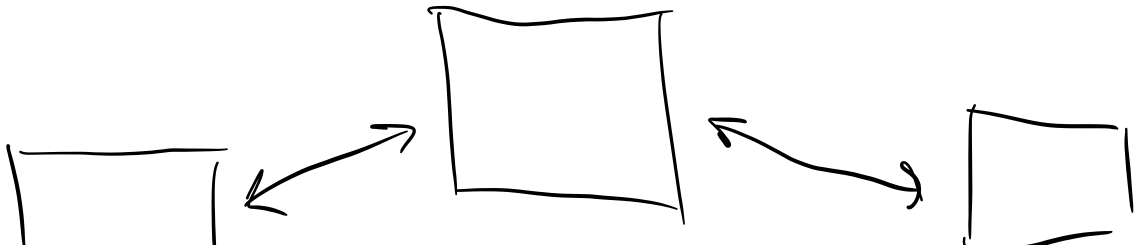
CS 342 **Project#3** **Fall 2022**

Networked Connect Four

Due Date:
Part #1 UML class diagrams and Wireframe:
Wednesday, November 16th 2022, @11:59pm
Part #2 complete programs (one for the server and one for the client):
Wednesday, November 23rd 2022, @11:59pm

Description:

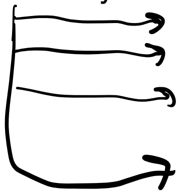
In this project you will implement the classic game of Connect Four. This is a somewhat simple game to understand and play that most of you should be familiar with. Your implementation will be a two player game where each player is a separate client and the game is run by a server. Your server and clients will use the same machine; with the server choosing a port on the local host and clients knowing the local host and port number (just as was demonstrated in class). At the end of each game, each client will be able to play again or quit.



CS 342 **Project#3** **Fall 2022**

All networking must be done utilizing Java Sockets (as shown in class). **The server must run on its own thread and handle each client on a separate thread. Each client must connect and communicate with the server on a separate thread**.

This project will be completed in two parts: Part #1 is the planning stage where you will create a wireframe of your client user interface and well as a class diagrams for both the server and client programs; with all of the classes, interfaces and relations you expect to use in your code.

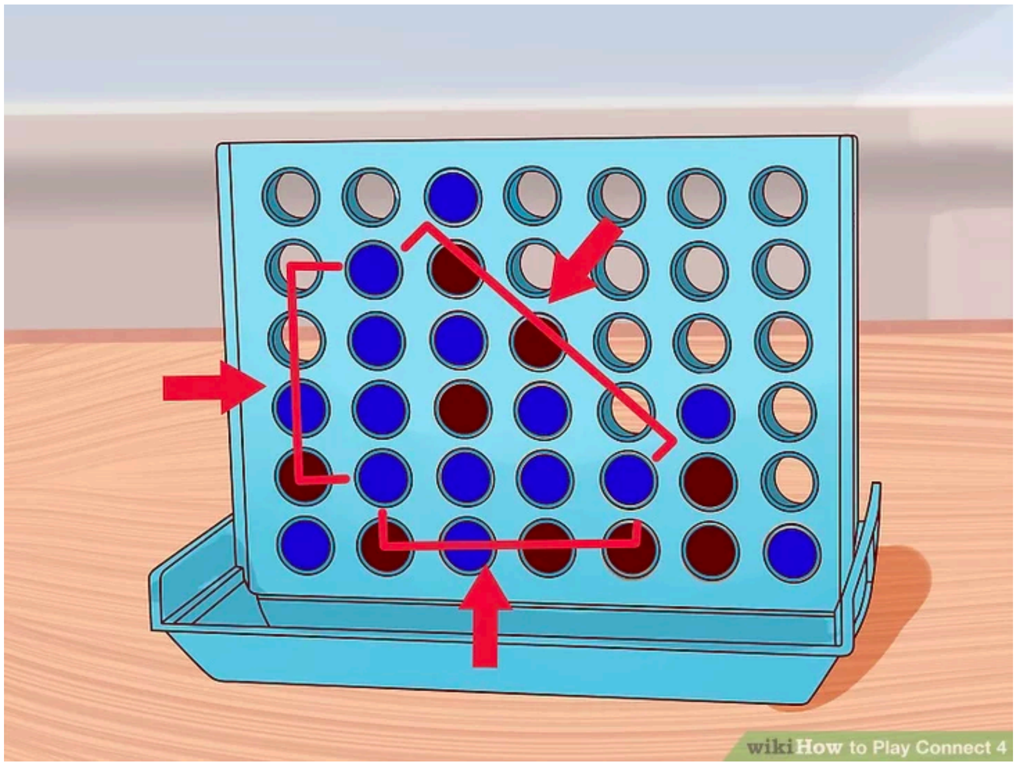


You may work in teams of two but do not have to.

How the game is played:

Connect Four is played on a grid of 7 columns and 6 rows (see image above). It is a two player game where each player takes a turn dropping a checker into a slot (one of the columns) on the game board. That checker will fall down the column and either land on top of another checker or land on the bottom row.

To win the game, a player needs to get 4 of their checkers in a vertical, horizontal or diagonal row before the other player (see image below):



https://www.wikihow.com/Play-Connect-4

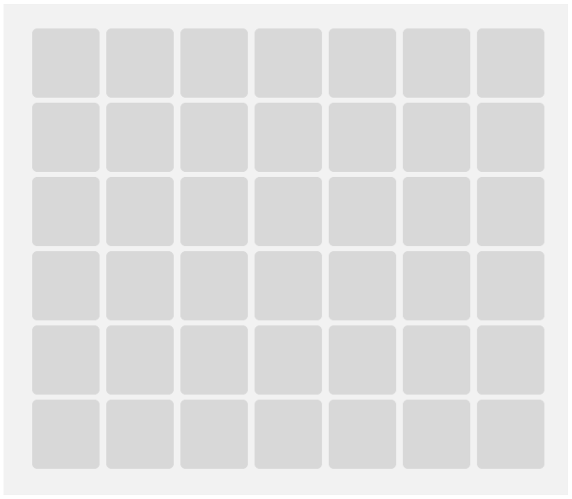
Your implementation of the game:

In your implementation of the game, the game board, in the client program, must be represented by the JavaFX component GridPane.

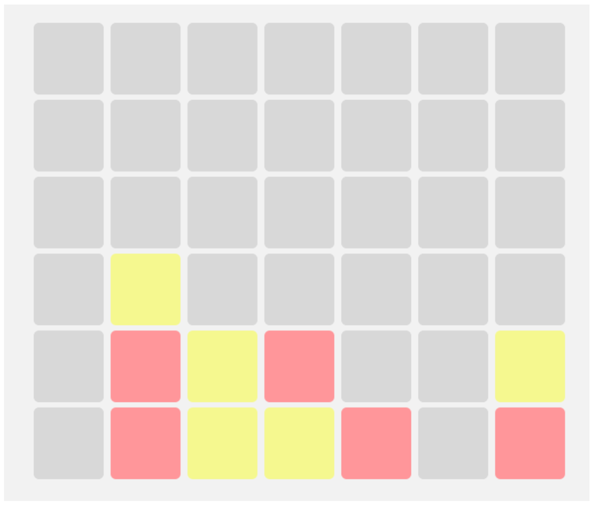
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/GridPane.html

The checkers in your game will be from a class you create called GameButton which will extend the JavaFX class Button.
You will create a game board of GameButton instances that will populate your GridPane.
Moves will be made by clicking on a GameButton in your GridPane (see following images for an example)

This is the only way you are allowed to implement this. Failure to do so will result in a zero for this project.



GridPane with buttons



Some buttons clicked

CS 342 **Project#3** **Fall 2022**

Implementation Details:

You will create ***two separate programs***, each with a GUI created in JavaFX, one for the server and one for the client. This project will be developed as a Maven project using the Maven JavaFX Project posted under “Sample Code” on our course BB site. You will need to download this twice, one for the client program and one for the server program.

You will need to change the <artifactId> in the pom file for each program: for instance, the server program would have:
<artifactId>serverProgramProjectThree</artifactId>

And the client program would have:
<artifactId>clientProgramProjectThree</artifactId>

You may use FXML, .CSS files with controller classes if you would like.
You may add as many other classes, data members, interfaces and methods as necessary to implement this program.
You may **only use JavaFX components** for your GUI.

You may **NOT use Java Swing of Java AWT.**

For the server program GUI:

- A way to chose the port number to listen to ✓
- Have a button to turn on the server. ✓
- Display the state of the game(you can display more info, this is the minimum): ✓
    - how many clients are connected to the server. ✓
    - what each player played. ✓
    - if someone won the game. ✓
    - are they playing again. ✓
- Any other GUI elements you feel are necessary for your implementation.

Notes: Your server GUI must have a minimum of two scenes: an intro screen that allows the user to input the port number and start the server and another that will display the state of the game information. To display the game information, you must incorporate a listView (as seen in class) with any other widgets used. Keep in mind, you can dynamically add items to the listView without using an ArrayList. ✓

<u>For the server program logic:</u>

- It will only allow a game to start if there are two clients connected.
- It will notify a client if they are the only one connected.
- It will keep track of what each player played.
- It will notify clients if there is a win or tie after every move.
- It will update each client with the other clients move.
- It will do all things necessary to run the game.

The Client program GUI:

You are welcome to use/discover any widget, pane, node, layout or other in JavaFX to implement your GUI in addition to the required elements described above.

The following 3 scenes are required:

1) Your program must start with a welcome screen that is it's own JavaFX scene. It will consist of:

Your welcome screen should welcome players to the game and have some sort of design other than the default color and style of JavaFX. In past semesters, this is a good opportunity to use images as background and play with the style of the graphical elements.

It will also allow the user to input the port number ✓ and ip address ✓ to connect to the server. Once connected to the server, the client program will change the GUI to the game play screen.

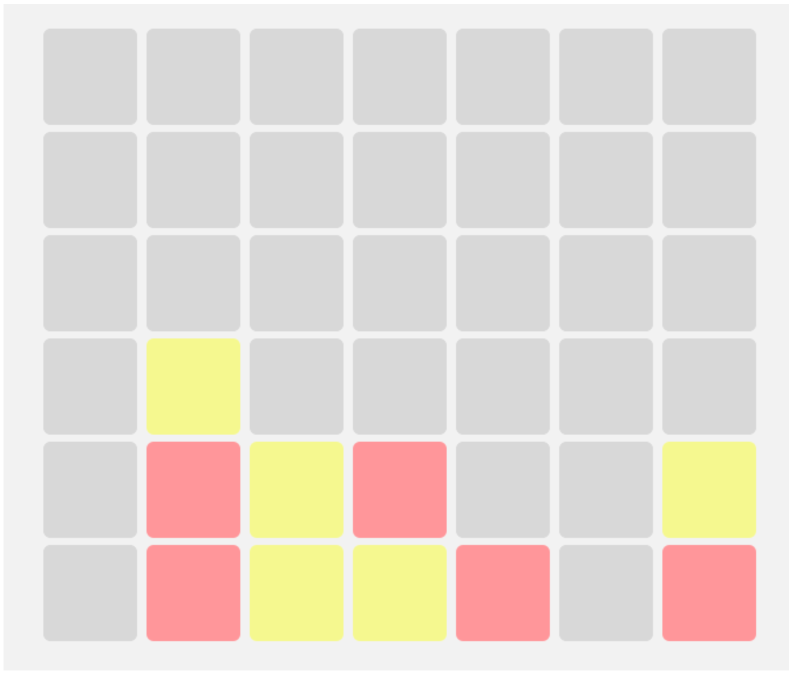
<u>2) For the client game play screen:</u>

- An area to display if it is player one or player two's turn to go ✓
- An area displaying each move made; with the following info for each move
-         which player made the move ✓
-         where the move was made to on the board ✓
-         if it was a valid move (if your program only allows valid moves, omit this)
- The game board (see below for details on this)

For instance, if player one attempted to make the first move of the game in row 3, column 3, you might print out to the GUI:

"Player one moved to 3,3. This is NOT a valid move. Player one pick again."

The game board as a GridPane filled with GameButtons:



- The buttons inside of the GridPane must be larger than the default size so your board is large enough to see and play.
- The buttons must start out as a different color or image than the default color. The image above starts them as "lightgrey".
- Each players move must turn the button a unique color or image. The image above just turns the buttons yellow or red depending on whose turn it is
- The spacing of the buttons in the GridPane must be more than the default spacing. The image above has increased the spacing so one can "see" the grid.
- You may style the GridPane in any other way you think enhances the look and feel of the game.

✓

3) A separate JavaFX Scene that is displayed when a player wins the game or there is a tie. This scene will have three elements:

- A message announcing who won the game or if there was a tie game
- An option to play another game(a button is fine for this)
- An option to exit the program (could also be a button)

✓

Playing the game in your Client Program:

Each client will take a turns choosing a spot on the GridPane. After clicking a GameButton to make a move, that button should change to the color or image that represents that player and become disabled so it can not be pressed again. If it is not a clients turn, they should not be able to make a move. They must wait for the server to send their opponents move, then update their UI with that move, and then allow the client to make their move.

Each players move will be sent to the server through the socket connecting that client. If that move resulted in a win for the client, they will send that move to the server and notify the server that it results in a win. The server will send the updated move to the other client and a message that the game is over. Similarly, if the move results in a tie, the client will send their move and a message that the game ended in a tie and the server will notify the other client of the same.

In the event that there is a “connect four” either horizontally, vertically or diagonally, all remaining enabled GameButtons should be disabled and those GameButtons that make up the four connected spots should be highlighted (you can put text on them, change them a different color or anything that is noticeable so that the players can see it). Your game should pause for about 3-5 seconds and then switch to a third unique scene announcing who won and allowing the client to play again or exit the program.

This should happen in each client program. For example, say client #1 makes a move and it results in a win, they will send the move and message that it is a win to the server and then transition scenes as above. Client #2 will get client #1’s move from the server and the message that it is a win. Client #2 will then transition scenes as above.

If the client decides to play again, the game play scene should come up with everything reset and able to play again. However, if there is not another client to play the game with, they should receive a message from the server that they must wait for another client to join. If the user decides to exit, just terminate the program.

Passing info between clients and server:

You must implement the CFourInfo class. class CFourInfo implements Serializable{}

You will add serializable data members to this class that keep track of the state of the game(i.e. String p1Plays, String p2Plays, Boolean have2players…..). This class will be used to send information back and forth between the server and two clients. This is the only way you are allowed to send and receive information.

Testing Code:

You are required to include JUnit5 test cases for your program. Add these to the src/test/java directory of your Maven Project. While you will not be able to test buttons and user interaction with JUnit5, you can test methods that evaluate game play or implement game logic.

*****Do Not put your whole program in one method. Split things up into appropriate methods and classes.*****

Electronic Submission:

If you worked in a group:

- only one of you needs to submit part #1 and part #2.
- You **must** submit a PDF file called Collaboration.pdf to the submission link on BB. In that document, put both of your names and netIds as well as a description of who worked on what in the project.
- If you worked alone, no need for the Collaboration.pdf.

Part #1: UML diagrams and wireframe

You will create a wireframe of the client GUI as well as class diagrams for the client and the server programs. Put these all in a single PDF file and submit to the assignment link on BB.

Part #2: Two full programs

Zip both Maven projects, The server program and the client program and name it with your netid + Project3: for example, I would have a submission called mhalle5Project3.zip, and submit it to the link on Blackboard course website.

Assignment Details:

Late work is accepted. You may submit your code up to 24 hours late for a 10% penalty. Anything later than 24 hours will not be graded and result in a zero.

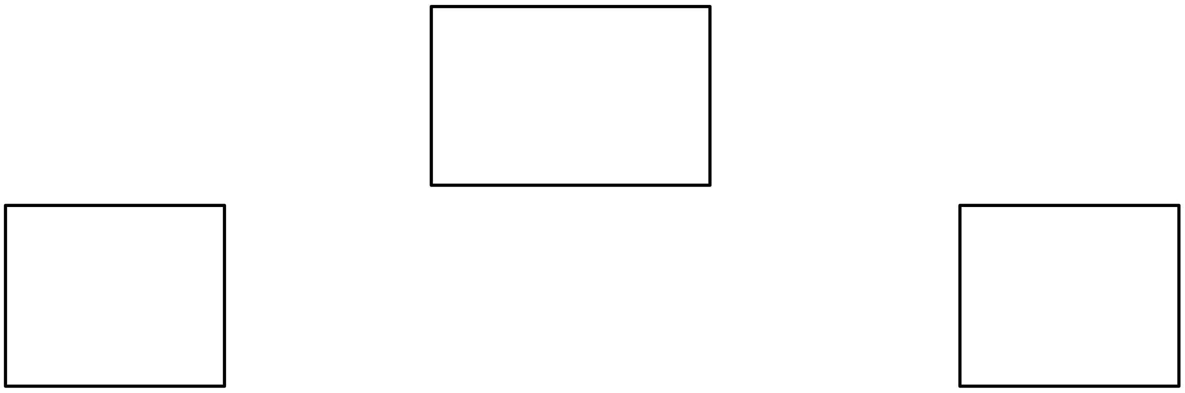
We will test all projects on the command line using Maven 3.6.3. You may develop in any IDE you chose but make sure your project can be run on the command line using Maven commands. Any project that does not run will result in a zero. If you are unsure about using Maven, come see your TA or Professor.

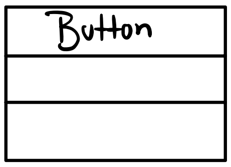
Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

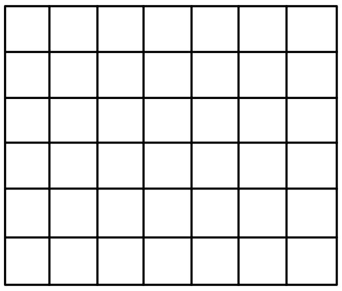
https://dos.uic.edu/conductforstudents.shtml.

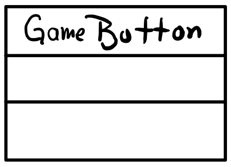
In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing

your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml.







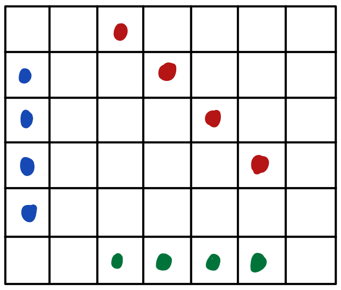


$-1 \longrightarrow$ Empty
$0 \longrightarrow$ PlayerOne
$1 \longrightarrow$ PlayerTwo

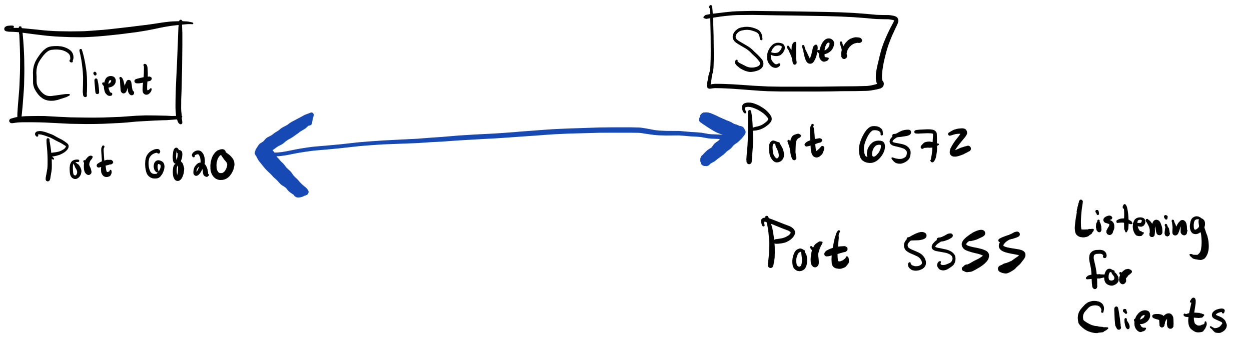
-1	-1	-1	0	-1	-1	-1
-1	1	-1	-1	0	-1	-1
-1	1	-1	-1	-1	0	-1
-1	1	-1	-1	-1	-1	0
-1	1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1

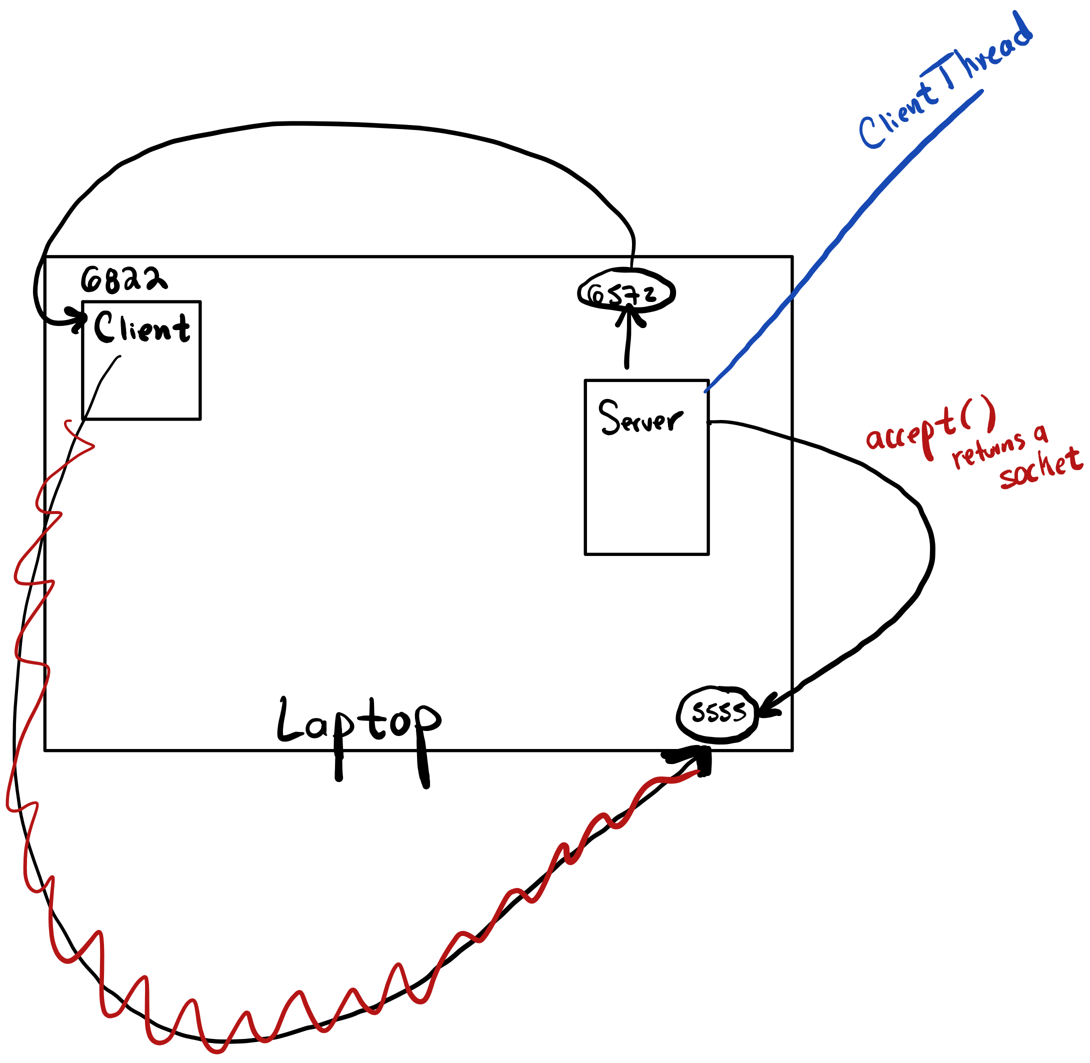
int gbArray[6][7]

GridPane gameBoard

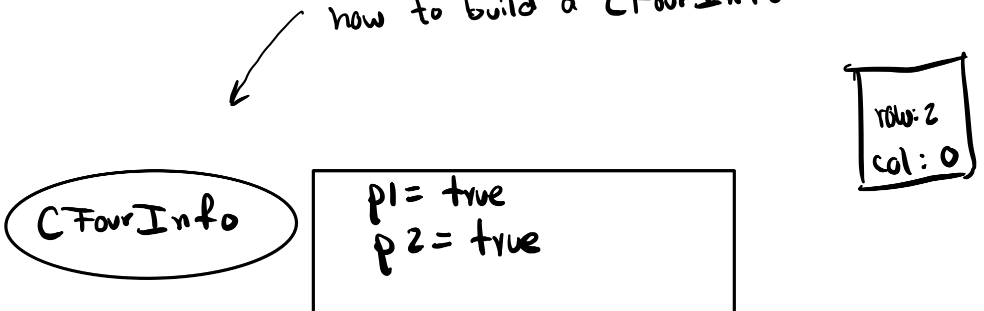


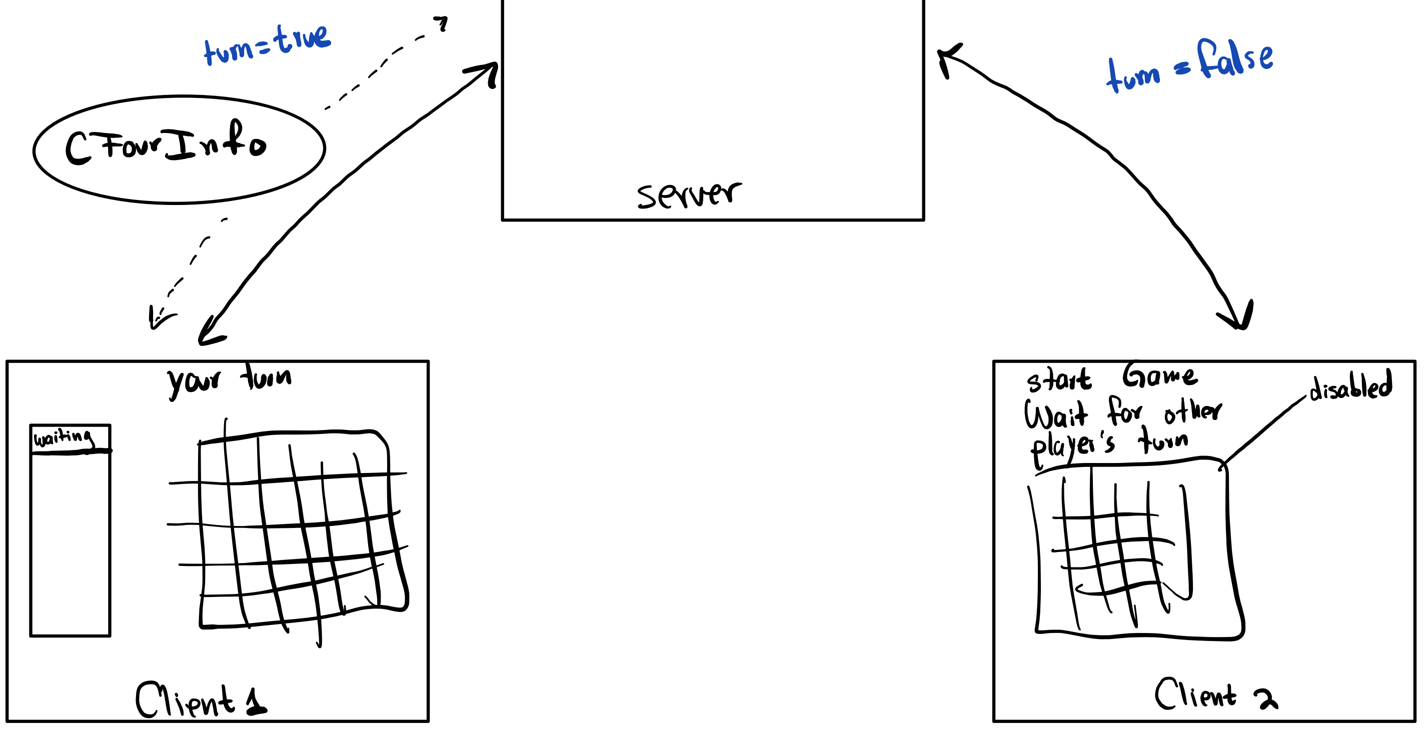
Professor Hallenbeck Project #3 HELP





Server needs to know how to build a CFourInfo





"Have a method that reads what the Object CFourInfo tries to tell me."

If CFourInfo.TwoPlayers = false
listview.add ("waiting")

Simulate playing the actual Game !!!!

Wireframe
↓
Graphical Components
↓
Classes

Player x need not know who he is either 1 or 2 or whatever...

You DON'T send an object.
You send a byte stream with information of HOW TO BUILD that object.

You can keep info from the CFourInfo object in DATA MEMBER or NOT !!!! UP TO YOU.

CAN YOU GET YOUR SOFTWARE TO WORK?

Then you can make it pretty, smart, efficient etc.

GET IT TO WORK!!

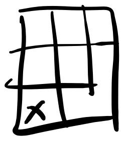
CFourInfo

boolean aPlayers = ~~false~~ true

boolean P1turn

boolean P2turn

How to encapsulate a move?



int row = ^{0} 2

int col = 0

"As long as you know the convention..."

[illegible]

[illegible]