***4 Player Ultimate Tic Tac Toe Final Coding Report***



**CS 440**

**at the**

**University of Illinois Chicago**

**Spring 2022**

**Coders: Brian Kopec, Cole Pearson, Harshal Patel, Nausherwan Tirmizi**

**Developers: Carlos Alves Pereira, Enoc Carranza, Michael Lazzar, Wael Mobeirek**

**Table of Contents**

[I Project Description 4](#_Toc101650476)

[1 Project Overview 4](#_Toc101650477)

[2 Project Domain 4](#_Toc101650478)

[3 Relationship to Other Documents 4](#_Toc101650479)

[4 Naming Conventions and Definitions 4](#_Toc101650480)

[4a Definitions of Key Terms 4](#_Toc101650481)

[4b UML and Other Notation Used in This Document 5](#_Toc101650482)

[4c Data Dictionary for Any Included Models 5](#_Toc101650483)

[II Project Deliverables 6](#_Toc101650484)

[5 First Release 6](#_Toc101650485)

[6 Second Release 6](#_Toc101650486)

[7 Comparison with Original Project Design Document 7](#_Toc101650487)

[III Testing 8](#_Toc101650488)

[8. Items to be Tested 8](#_Toc101650489)

[9. Test Specifications 9](#_Toc101650490)

[10. Test Results 24](#_Toc101650491)

[11. Regression Testing 26](#_Toc101650492)

[III Inspection 26](#_Toc101650493)

[12 Items to be Inspected 26](#_Toc101650494)

[13 Inspection Procedures 26](#_Toc101650495)

[14 Inspection Results 27](#_Toc101650496)

[V Recommendations and Conclusions 27](#_Toc101650497)

[VI Project Issues 27](#_Toc101650498)

[15 Open Issues 27](#_Toc101650499)

[16 Waiting Room 28](#_Toc101650500)

[17 Ideas for Solutions 28](#_Toc101650501)

[18 Project Retrospective 28](#_Toc101650502)

[VII Glossary 29](#_Toc101650503)

[VIII References / Bibliography 29](#_Toc101650504)

[References 29](#_Toc101650505)

[IX Index 30](#_Toc101650506)

# I Project Description

## 1 Project Overview

4-Player tic-tac-toe is based on the original version of ultimate tic-tac-toe but is extended to allow 4 players to play at once. To win the game, you must win 3 of the large grids in a row, and to win a large grid you must win 3 of the squares in that grid in a row. This game allows the use of AI to play with to supplement real players. There is an account and leaderboard system built into the game. There is also a chat system to interact with other players while playing the game. This is to incentivize people to play the game and compete with other players.

## 2 Project Domain

The domain of the project consists of the player portion, the database server portion, and the AI portion. The database server allows a player to create and login to an account, and also keep track of player data and compare it to other players. The player portion consists of logging in to an account. It allows a user to join a lobby with other players and AI bots. It allows a player to check statistics and leaderboards of players. The player will be able to chat with other players in the lobby and during the game. The player will be able to play the game live with other players. The AI portion is there to supplement players. The AIs will have three difficulties and will make moves similarly to how a player can make moves.

## 3 Relationship to Other Documents

This 4-player ultimate Tic-tac-toe project was originally designed by Carlos Alves Pereira, Enoc Carranza, Michael Lazzar and Wael Mobeirek in CS440 in the Fall of 2020. Our group based our project on this and built on additional features on top of it. We have used their project report during the development of this project. This includes the use of their project description, some of their requirements like ‘look and feel’ requirements and test requirements, and also their project designs like their design goals and final system design.

## 4 Naming Conventions and Definitions

### 4a Definitions of Key Terms

Large Grid: The entire Tic-tac-toe board.

Small Grid: A single grid from the large grid.

Player: A person who joins the server and intends on playing the game.

TheServer: instances of servers that host multiple client threads running a client

ClientThread: thread that receives input from its client and sends some ouput based on the input to all other client threads.

Consumer callback: allows communication from the Client class to the GUI.

Client: each individual player that is connected to the server on their own application

Chat: A string shared between players. Format: [TID]: [Message]

TID: The turn identifier assigned to the player when joining the server to determine when they can place a piece

CID: The Current identifier which dictates which turn it is, deciding which player can place a piece

Master board: A 4D array of characters that represent the tic tac toe board. The board that is sent between the players as data. The client’s tic tac toe grids are updated to match the recieved masterboard

### 4b UML and Other Notation Used in This Document

This document follows standard UML format as described by Fowler in *UML Distilled*, 3rd edition. Exceptions noted when made. [1]

### 4c Data Dictionary for Any Included Models

Each individual tic tac toe game is represented by a TicTacToe object which consists of a 3x3 grid of buttons. The overall game board is stored in an array of TicTacToe objects.

A masterboard, which is a 4 dimensional array of characters is passed between the server and the client to update the board.

The login table is a table that holds the ID of a player, their username and their password.

The playerdata table is a table that holds wins, losses, and points of a particular player based on their ID from the login table

The clients array is an ArrayList of ClientThreads, which is used to send information to all other client applications when an event occurs on 1 client.

All data sent back and forth between the Server and Client is done through ObjectInputStream and ObjectOutputStream

Chats are sent back and forth with the chatMsg class. The chat takes a message and prepends the TID of the person sending it. It is received by the server which sends it to all the players and the message is added to the chatlist

# II Project Deliverables

## 5 First Release

Release Date: February 25th,2022

The first release focused on us getting our base game set up. We had functionality for the player to open the program and be able to enter a server with the port number. An infinite number of ports allows for as many servers as needed. For one port number, 4 people can join and play tic tac toe amongst each other. The server had the functionality to intake all 4 players and keep track of their moves. At that point the player can enter the game board and are able to place pieces on the board. The server keeps track of whose turn it is to prevent people from placing pieces out of turn. Once a piece is placed, the board is updated for all the users. At this point we had functioning game logic (to check if a board is won) but it wasn’t integrated with the UI.

## 6 Second Release

Release Date: April 1st,2022

The second release focused on Quality-of-Life improvements. One of the major changes was the ability to register and login. The players were kept track of in a SQL database, this gave them a username and password. Alongside this, game stats could also be tracked, letting players know how many points they’ve accrued and how many games they have won. This promotes player retention since it incentives stat increases. Alongside this, we implement AIs so players can play games even when the 4-player requirement isn’t met. We had an easy AI which was a random number generator and a medium AI that would block the player from winning. Chat functionality was also implemented so players could ~~trash talk~~ interact with each other. The social aspect promotes playing with friends. The UI was also updated to look more appealing and all base game functionality was working.

Figure 1.a:

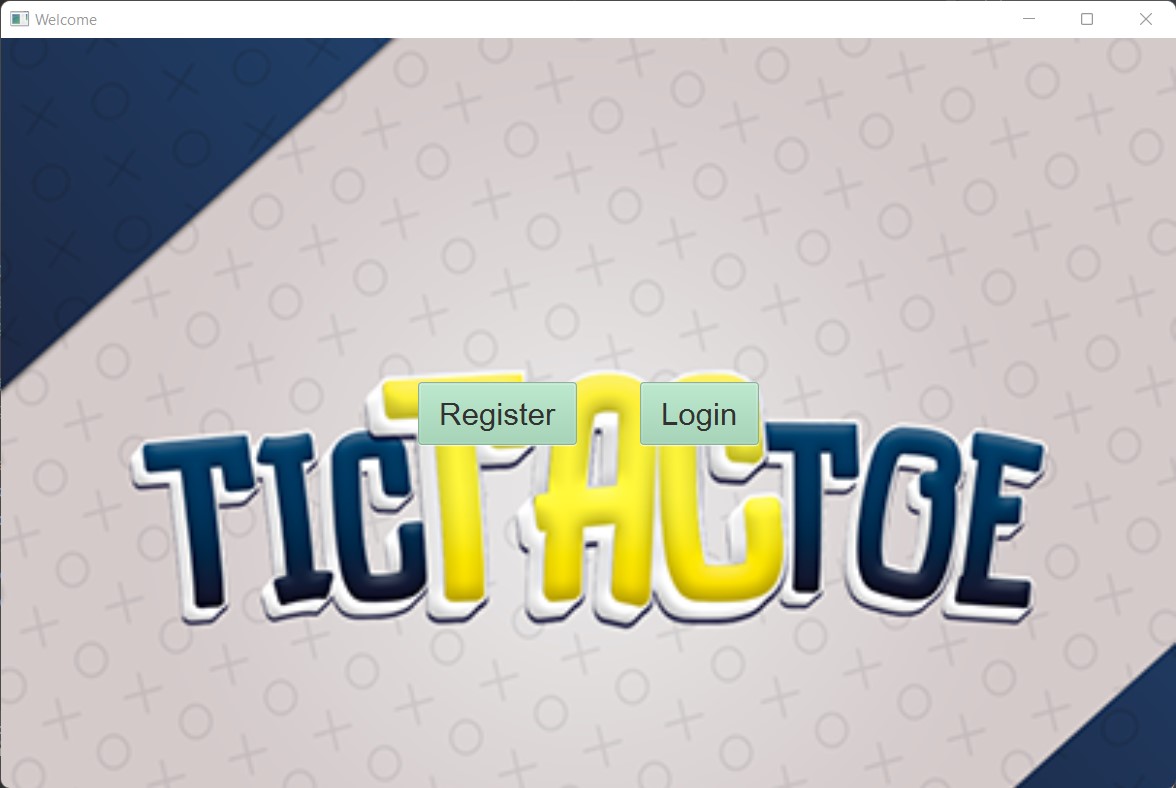
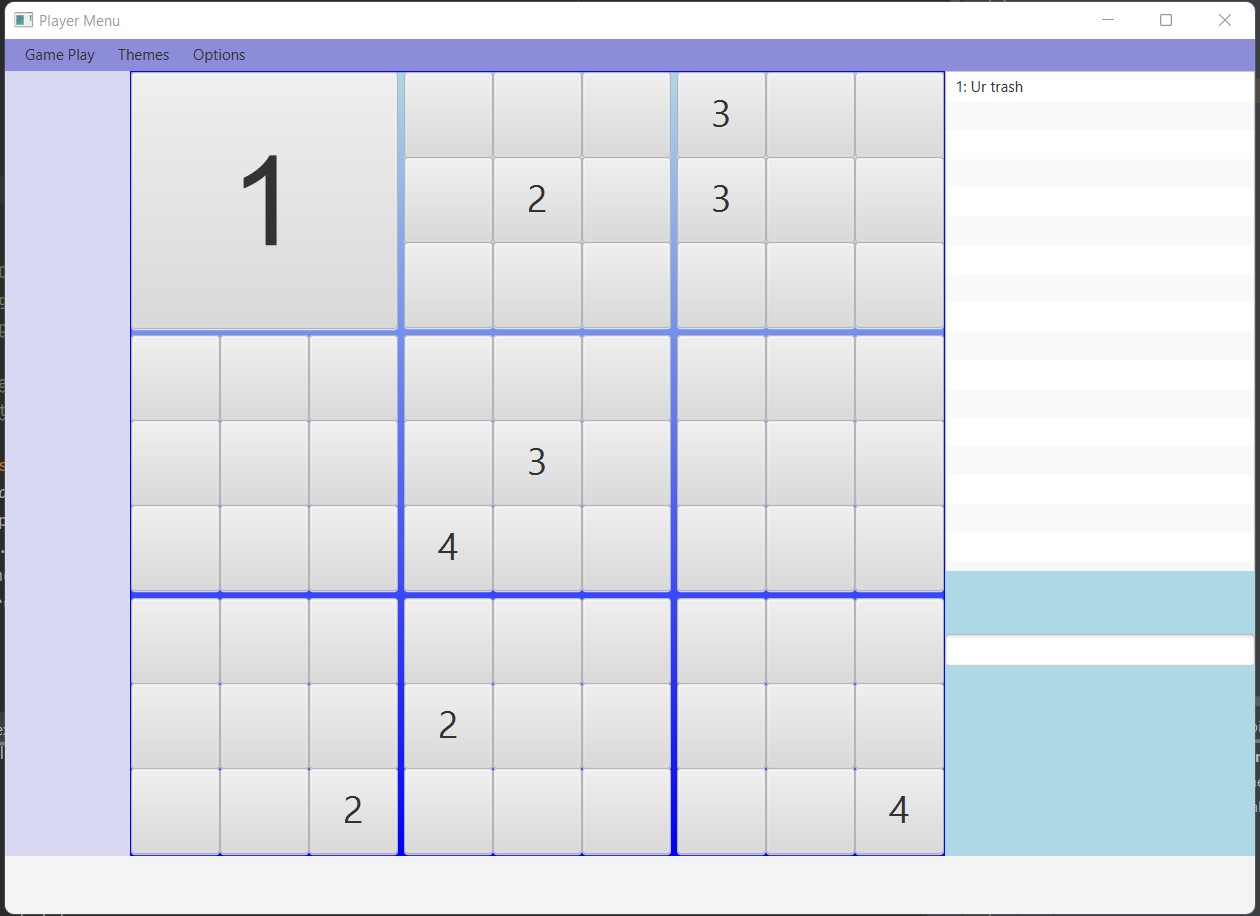


Figure 1.b:



## 7 Comparison with Original Project Design Document

There are many similarities between the original project design and our project. As it says in the project overview of the original project, our project was designed to work with 4 players at once. Another similarity between our projects is the use of AI, we incorporated AI to substitute the use of players. We also went with their approach in the design goals of making the game simple and easy to use. The original project placed an importance on competitive players by including a login account and statistics. We implemented this in our project as well. There are also a few differences between our projects. In the original project, there was to be a game mode with teams. We decided to stick to the standard game mode in our project. We also didn’t add some of the customization aspects from the original project, including the customization of the game board and the use of custom rules. We also decided to add a chatting feature between players to allow them to chat during the game and lobby.

# III Testing

## 8. Items to be Tested

**Item 1:** Client

**Item 2:** Server

**Item 3:** GameLogic

**Item 4:** DBConDao

**Item 5:** GameButton

**Item 6:** TicTacToe

**Item 7:** TurnIdentifier

**Item 8:** ChatMsg

**Item 9:** MasterBoard

**Item 10:** BoardInfo

**Item 11:** Robot

**Item 12:** Medium Robot

**UML of Items Tested:**

Diagram

Description automatically generated

## 9. Test Specifications

All requirements that have been addressed by tests are referred from Fall 2020 Group 22 Final development report [2]

**CSCSTR# - ClientServerConnection**

**Description:** This test sets up a connection between a server and client that is connected to port 5555 and passes a string through the server to client.

**Items covered by this test:** Items 1 & 2

**Requirements addressed by this test:** NA

**Environmental needs:** Must have access to both Server and Client modules and Port 5555 must not be occupied.

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** String with value “test”

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** A string that says “test went through” should reach the Client side of the connection in the Platform.runLater method

**CSCINT# - ClientServerConnection2**

**Description:** This test sets up a connection between a server and client that is connected to port 5555 and passes a int through the server to client.

**Items covered by this test:** Items 1 & 2

**Requirements addressed by this test:** NA

**Environmental needs:** Must have access to both Server and Client modules and Port 5555 must not be occupied.

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** Integer with value 5

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** A int with a value of 5 should reach the Client side of the connection in the Platform.runLater method

**CSCMSG# - ClientServerConnection3**

**Description:** This test sets up a connection between a server and client, passes a string and integer through the server and the server sends back a chatMsg Object with the player number of the client attached to the message to the client.

**Items covered by this test:** Items 1, 2, and 8

**Requirements addressed by this test:** NA

**Environmental needs:** Must have access to both Server and Client modules and Port 5555 must not be occupied.

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** String containing the message and integer denoting the player number that sent the message

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** A chatMsg Object that contains the message sent appended to the player number that sent it should reach the Client side of the connection in the Platform.runLater method

**REG# - RegisterTest**

**Description:** This test registers a user into the SQL database.

**Items covered by this test:** Item 4

**Requirements addressed by this test:** INT - 1

**Environmental needs:** SQL server that contains the database must be running in the background

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** Strings containing the Username and Password

**Output Specifications:** IntelliJ should show a green checkmark on the run tab and the SQL database will have a row added in the login and playerdata.

**Pass/Fail Criteria:** The SQL database should create a new row with the username and password in the login table and a row with wins, loss, and points with a value of 0 in the playerdata table. The ID from the row added in playerdata should correspond to the ID of the row added to login table.

**LOG# - LoginTest**

**Description:** This test retrieves information from the player data table given a username and password.

**Items covered by this test:** Item 4

**Requirements addressed by this test:** INT - 1

**Environmental needs:** SQL server that contains the database must be running in the background

**Intercase Dependencies:** REG test must be completed successfully for this test to be able to pass.

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** Strings containing the Username and Password.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The instance of DBConDao should have the wins, losses, and points stored in its data members for that player whose login information was inputted.

**STATS# - updateStatsTest**

**Description:** This test updates information in the playerdata table given a values to increase the points, wins, and losses by.

**Items covered by this test:** Item 4

**Requirements addressed by this test:** INT – 1, RFT – 1, PAR – 2

**Environmental needs:** SQL server that contains the database must be running in the background

**Intercase Dependencies:** LOG test must be completed successfully for this test to be able to pass.

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** 3 integers representing the wins, losses, and points

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. The database will have updated values for a particular row

**Pass/Fail Criteria:** The SQL database will update the scores for a particular player that has logged in and the values should be increased by the amount that was passed in as input.

**TID# - testTIDInitialization**

**Description:** This test checks if a new TurnIdentifier object has been properly constructed.

**Items covered by this test:** Item 7

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** Integer of 0 representing the 1st player

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** If constructed correctly, nothing will be printed and the Intellij will show a green checkmark. If the test failed, the message “TID WAS NOT PROPERLY INITIALIZED” will be printed.

**CHATINIT# - testChatInitialization**

**Description:** This test creates a chatMsg object and with player 1 sending the message, and checks if correctly updates the message to show the player number and message together.

**Items covered by this test:** Item 8

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** String with the chat to be sent, and integer representing the players turn number

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** If constructed correctly, nothing will be printed and the Intellij will show a green checkmark. If the test failed, the message “MESSAGE IS INCORRECT” will be printed.

**CHATUPD# - testChatUpdate**

**Description:** This test creates a chatMsg object and later updates using the copy constructor

**Items covered by this test:** Item 8

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** String with the chat to be sent, and integer representing the players turn number. Then pass in the object into the constructor.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** If constructed correctly, nothing will be printed and the Intellij will show a green checkmark. If the test failed, the message “TID DID NOT UPDATE CORRECTLY” or “THE CHAT DID NOT UPDATE CORRECTLY” will be printed.

**MBOARD# - testMasterBoard**

**Description:** This test creates a MasterBoard object test wether the Copy constructor successfully does a deep copy of the data.

**Items covered by this test:** Item 9

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file ClientServerTest.java.

**Input Specification:** theBoard array of an instance of MasterBoard will be initialized with some integers

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** If constructed correctly, nothing will be printed and the Intellij will show a green checkmark. If the test failed, the message “Not a masterboard” or “Board not correctly copied and updated” will be printed.

**GRID1# - IndividualGridTest1**

**Description:** This test creates a MasterBoard object tests the isPointScored2 function. It tests a point scored in a column.

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** a MasterBoard filled with values that lead to a point scored, and an integer that represents the player to check the win for

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found player 1 scored a point, and if it failed it would print out "Incorrect IndividualGridTest1"

**GRID2# - IndividualGridTest2**

**Description:** This test creates a MasterBoard object tests the isPointScored2 function. It tests a point scored in a diagonal.

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** a MasterBoard filled with values that lead to a point scored, and an integer that represents the player to check the win for

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found player 2 scored a point, and if it failed it would print out "Incorrect IndividualGridTest2"

**GRID3# - IndividualGridTest3**

**Description:** This test creates a MasterBoard object tests the isPointScored2 function. It tests a point scored in a different grid.

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** a MasterBoard filled with values that lead to a point scored, and an integer that represents the player to check the win for

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found player 3 scored a point, and if it failed it would print out "Incorrect IndividualGridTest3"

**GRID4# - IndividualGridTest4**

**Description:** This test creates a MasterBoard object tests the isPointScored2 function. It tests a point scored in a different grid and in the last column of that grid.

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** a MasterBoard filled with values that lead to a point scored, and an integer that represents the player to check the win for

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found player 4 scored a point, and if it failed it would print out "Incorrect IndividualGridTest4"

**GRID5# - IndividualGridTest5**

**Description:** This test creates a MasterBoard object tests the isPointScored2 function. It tests a point scored in the 3rd grid and in the first column of that grid.

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** a MasterBoard filled with values that lead to a point scored, and an integer that represents the player to check the win for

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found player 4 scored a point, and if it failed it would print out "Incorrect IndividualGridTest5"

**WIN1# - WinnerTest1**

**Description:** This test initializes the isWinner array showing the first row of grids as a winner and checks if the isWinnerFunction correctly identifies the win

**Items covered by this test:** Items 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 3 true statements in a row

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found a winner in the first row of grids, and if it failed it would print out " Incorrect WinnerTest1"

**WIN2# - WinnerTest2**

**Description:** This test initializes the isWinner array showing the second row of grids as a winner and checks if the isWinnerFunction correctly identifies the win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 3 true statements in a row

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found a winner in the second row of grids, and if it failed it would print out " Incorrect WinnerTest2"

**WIN3# - WinnerTest3**

**Description:** This test initializes the isWinner array showing the third row of grids as a winner and checks if the isWinnerFunction correctly identifies the win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 3 true statements in a row

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found a winner in the third row of grids, and if it failed it would print out " Incorrect WinnerTest3"

**WIN4# - WinnerTest4**

**Description:** This test initializes the isWinner array showing the left diagonal of grids as a winner and checks if the isWinnerFunction correctly identifies the win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 3 true statements in a row

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found a winner in the left diagonal of grids, and if it failed it would print out " Incorrect WinnerTest4"

**WIN5# - WinnerTest5**

**Description:** This test initializes the isWinner array showing the left diagonal of grids as a winner and checks if the isWinnerFunction correctly identifies the win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 3 true statements in a row

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return true, since it found a winner in the left diagonal of grids, and if it failed it would print out " Incorrect WinnerTest5"

**WIN6# - WinnerTest6**

**Description:** This test initializes the isWinner array with grids in a row, but not 3 and tests if the isWinnerFunction correctly identifies it as not a win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 2 true statements in a row and 1 misplaced

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return false, since it did not find a winner in the left diagonal of grids, and if it failed and returned true it would print out "Incorrect WinnerTest6"

**WIN7# - WinnerTest7**

**Description:** This test initializes the isWinner array with grids in a row, but not 3 and tests if the isWinnerFunction correctly identifies it as not a win

**Items covered by this test:** 3, 5, 6, and 10

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file GameLogicTest.java.

**Input Specification:** The isWinner array filled with 2 true statements in a row and 1 misplaced

**Output Specifications:** IntelliJ should show a green checkmark on the run tab.

**Pass/Fail Criteria:** The function would return false, since it did not find a winner in the left diagonal of grids, and if it failed and returned true it would print out "Incorrect WinnerTest7"

**AIHOR1# - MediumAIHoirzontalTest1**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring horizontally in the third grid

**Items covered by this test:** Items 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from left to right.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIHOR2# - MediumAIHoirzontalTest2**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring horizontally in the first grid

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from right to left.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIVER1# - MediumAIVerticalTest1**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring vertically from top to bottom

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from top to bottom.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIVER2# - MediumAIVerticalTest2**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring vertically from bottom to top.

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from bottom to top.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIDIA1# - MediumAIDiagonalTest1**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring diagonally from top left to bottom right.

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from top left to bottom right.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIDIA2# - MediumAIDiagonalTest2**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring diagonally from bottom right to top left.

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from bottom right to top left.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIDIA3# - MediumAIDiagonalTest3**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring diagonally from top right to bottom left.

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from top right to bottom left.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

**AIDIA4# - MediumAIDiagonalTest4**

**Description:** This test checks to see if the medium AI correctly blocks a player from scoring diagonally from bottom left to top right.

**Items covered by this test:** 9, 11, and 12

**Requirements addressed by this test:** NA

**Environmental needs:** NA

**Intercase Dependencies:** NA

**Test Procedures:** Running the JUNIT test java file AITest.java.

**Input Specification:** Fills the MasterBoard so that it is 1 move away from winning the entire game from bottom left to top right.

**Output Specifications:** IntelliJ should show a green checkmark on the run tab. Return value will be a Quartet representing the location of a move.

**Pass/Fail Criteria:** The makeMove function from the AI should correctly block the player from winning by correctly returning the coordinates to block the 3 in a row

## 10. Test Results

**ALL TESTS**

**Date(s) of Execution:** 4/23/2022

**Staff conducting tests:** Harshal Patel, Brian Kopec, Cole Pearson, Nausherwan Tirmizi

**Expected Results:** all tests should pass with green check marks

**Actual Results:**

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface, text

Description automatically generated**

**Test Status:** IndividualGridTest4 and IndividualGridTest4 are failing, the rest are passing.

Error Statements:

Text

Description automatically generated

## 11. Regression Testing

NA

# III Inspection

## 12 Items to be Inspected

* + Game Logic
  + AI
  + Game UI

## 13 Inspection Procedures

Checklist:

* + Does not crash
  + Functions as desired
  + Is usable

Small, varying amounts of time from each of our main weekly meetings was devoted to debugging and inspection. Most resolution done outside of meeting

## 14 Inspection Results

* + Game logic was found to be without any flaw, functioning to determine a point winner and a game winner. Inspected by Cole Pearson
  + AI was discovered to not be able to recognize when an individual board was a tie game. This caused the AI to override previously filled in spaces. This was fixed so that the AI would find a new board to place its piece on. Inspected by Brian Kopec and Harshal Patel
  + The user interface of the game board was found to have an issue where the current player was able to override a previously filled in space. This was caused by the improper use of the “==” when comparing strings, and was resolved by using the .equals() function. Inspected by Cole Pearson and Nausherwan Tirmizi.

# V Recommendations and Conclusions

Most of the requirements were filled out and most unit tests passed. The base game is mostly functional with the server client connection succeeding without any issues. The server client connection is fully functional which allows different data to be passed along in the future. So the program can be developed further since it allows modularity with different data types. The two tests that failed were testing the individual grids to make sure the points were being registered correctly. There were some edge cases where the points were not being correctly registered. Further testing and fixing bugs in the point system should be further developed.

# VI Project Issues

## 15 Open Issues

The largest outstanding issue is the lack of a hard AI, which will implement the minimax algorithm.

A cleaner, more elegant user interface is in the works. Focus has been spent on the backend functionality of the program.

Password encryption for our SQL database has yet to be implemented, focus has been instead centered on the setup of the registration/login system.

There is currently no way for the player to know when it is their turn other than seeing that other players’ turns have been made

## 16 Waiting Room

The hard AI is the most important development that needs to implemented on further releases. Currently the Easy and Medium AI allow the player to play the game even if they don’t have any friends but they don’t give enough of a challenge. The minimax algorithm already exists so the actual logic behind the hard AI already exists. Another feature to add would be the usernames to display which player has joined the lobby and which player is chatting. Currently the program uses the Turn Identifier for players when they join but allowing players to have their username displayed will add that personal touch.

## 17 Ideas for Solutions

The hard AI could potentially implement the minimax algorithm. This algorithm can be used to create a perfect Tic Tac Toe AI. All it would need to do is use the algorithm once to figure out which individual board to play on, then use it again to figure out which space on that board it should place its game piece on.

The medium AI does not currently block a potential opponent victory when the opponent has two of their pieces down with an open space in between them. This makes the medium AI considerably easier to beat and fixing this would better bridge the gap in difficulty between the hard and easy AI.

We could potentially display the number of the current player in the top corner of the game screen to tell the player whose turn it is. This could also be switched to the player name in a future implementation.

## 18 Project Retrospective

The project was an overall success. The weekly meeting minutes really helped in organizing different deliverables that everyone needs to work on. This allowed someone to always be working on something and completing a different part of the project (something akin to a multicore application). The meetings also allowed people to align on design standards, because a lot of our parts needed to interact with each other. That method of interaction had to be uniform to work, which was something we were able to achieve. Everyone was able to get their deliverables complete, which made the process less stressful. Individual members could focus on their part and be assured that the other team members would complete their part on time. This also leads to teamwork; team members would actively help others on their project when needed. Harshal Patel really helped everyone with different aspects of the project allowing us to align on one design vision. One thing to change for a real-world environment would be to have two+ persistent weekly meetings. This project would increase meetings as needed but only have one scheduled recurring meeting. In a perfect scenario without exams and class projects, more recurrence would be beneficial/

# VII Glossary

Master board: A 4D array of characters that represent the tic tac toe board. The board that is sent between the players as data. The client’s tic tac toe grids are updated to match the received master board

Large Grid: The entire Tic-tac-toe board. Composed of small grids that are their own tic tac toe games

Small Grid: A single grid from the large grid. A traditional 3x3 tic tac toe board

TheServer: instances of servers that host multiple client threads running a client. One instance of a server is your game with 4 players.

Client: each individual player that is connected to the server on their own application

# VIII References / Bibliography

# References

|  |  |
| --- | --- |
| [1] | M. Fowler, UML Distilled, Third Edition, Boston: Pearson Education, 2004. |
| [2] | J. Bell, "Underwater Archaeological Survey Report Template: A Sample Document for Generating Consistent Professional Reports," Underwater Archaeological Society of Chicago, Chicago, 2012. |
| [3] | A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, Ninth ed., Wiley, 2013. |
| [4] | Robertson and Robertson, Mastering the Requirements Process. |
| [5] | C. A. Pereira, E. Carranza, M. Lazzar and W. Mobeirek, "4-Player Ultimate Tic-Tac-Toe," Chicago, 2020. |

# IX Index

**No index entries found.**