Final Year Project Report

Full Unit – Final Report

CS3821: A Concurrency-Based Game Environment

Maximillian Bartrip

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Zhaohui Luo

Department of Computer Science

Royal Holloway, University of London

March 25, 2022

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

15938

Student Name:

Maximillian Bartrip

Date of Submission:

25/03/2022

Signature:

[illegible]

Contents

Introduction

This document is a final report, detailing my original plan for my project, what I achieved, how I approached the project and an evaluation of the project. The project that I have been working on is a concurrency-based game environment which provides the user an interface allowing them to play multiple different board games at the same time as each other. As was stated in my project plan, I hoped to produce a game which would not only allow the user to play multiple games concurrently, but to also allow the user to play many different board games through the same program. The board games that I aimed to implement were Chess[1], Checkers[2] (or Draughts), Polish Draughts[2], Gomoku[3], and Dama[4]. I have worked to produce many reports and proof of concept prototypes, the knowledge gained from these prototypes culminated towards the end goal of producing the complete board game program. This document shows my plan for the project, the motivation for my project, the design and implementation of the software that I have developed, how to run the software and an evaluation on how my project went, as a whole. Additionally, there is a project diary which I updated throughout my work, to log what I have been working on. There is also a section about professional issues, specifically to do with plagiarism of code.

By completing this project, I believe I will gain lots of valuable skills and experiences that will be transferrable to a future career. I will have to perform independent research, design my own system, develop the software using modern programming tools, manage my own time and schedule and report and present my work to a supervisor who is acting as almost a manager or higher up in this project. This project emulates what it would be like to work on a project in the real world and tests the skills needed to perform as a Computer Scientist in the future. This project will also be a useful learning experience, where I will be able to see and understand the most important areas to focus on, to be successful in a project.

Project Goals:

Below is a list of project goals that I set out to achieve at the start of the project.

- A user interface main menu that allows the user to select what type of board game they would like to play.
- Implementation of a checkers board game with local two player capabilities, with an interactive interface for playing the game
- Using threads to allow for multiple instances of a game to be played on the same computer
- Implementation of additional board games such as Chess, Polish Draughts, Gomoku and Dama, with local two player capabilities, as well as the ability to play these games simultaneously by utilising java threads [4]
- Implementing each game such that they utilise the same functionality and modules without interfering with each other (such as a resizable board that can be used as the location of gameplay)
- Having pop-ups to report to the user any errors that have occurred

- If time allows, implementation of a LAN multiplayer system allowing another user to connect to the game via another device to play against them (this would involve complex transmission of data between the two devices in order to display the same game) would be implemented using Java Socket [6] and ServerSocket [7].

Having spent time working on prototyping parts of the system, I have reviewed these goals and have decided to make some amendments.

- For implementing multiple instances of a game being played, I found an easier way to provide this behaviour without the use of threads which is explained in more detail in the report on prototype implementation
- For error reporting, I found that having pop-ups was quite jarring for the user and lead to a worse user experience, so I would like to change the error reporting to instead be a text-space where the text is printed to the user on the interface when an error occurs
- For the multiplayer implementation, I would like to add a chat feature that allows the two players to communicate with each other, which will run alongside the game, using a thread that watches for messages being sent and received between the users, and displaying the messages in a text-space on an interface

Timeline

Below is analysis of how my actual working timeline differed from the timeline set out in my project plan.

I started work on the Survey report on game environments slightly later than expected, and completed it by the deadline of October 12th, as I started slightly behind schedule, it meant that I only started creating my Concept board game on the 13th of October, when I had planned to start on the 11th.

As I produced my prototype, I found that it was much easier to develop the program in different stages than what I had expected, I had planned to produce the game logic and interface alongside each other, however I found it easier to produce all or most of the game logic first then develop the interface after.

I was able to start working on the interface by October 22nd. I completed work on the user interface by October 27th. And was only able to start work on concurrent execution by November 1st, leaving me far behind schedule. However, I was able to provide concurrent behaviour much quicker than I expected and was able to provide the ability to play multiple games by November 9th.

I was then able to produce my report on my prototype implementation within the scheduled time.

Around January 17th I started work on a second prototype, which was a server-based chat app, which would allow two users to connect to each other and send and receive messages between each other. I initially planned to spend two weeks on this, with the first week being spent creating the server connection and message sending system and the next week creating the interface. However, I struggled a lot on this prototype and ended up completing it on February 10th, leaving me behind schedule.

On February 12th, I began creating my complete system, starting by redesigning and refactoring the code for the checkers prototype.

On February 24th I began to implement the server char system into my game, creating a chat are on the interface and creating the classes necessary to setup the server and client.

I finished implementing redesigning the checkers game on February 27th and began to work on adding the ability to play the game using online multiplayer.

I completed implementation of the multiplayer system into my game on March 6th.

I ended up completing the multiplayer system much later than I had hoped and as such was left with little time to complete the other goals I had hoped to achieve.

Additionally, in my Risk Assessment in my project Plan, I severely underestimated the effect that work from other courses would have on my schedule, which led to me starting work behind schedule on multiple occasions.

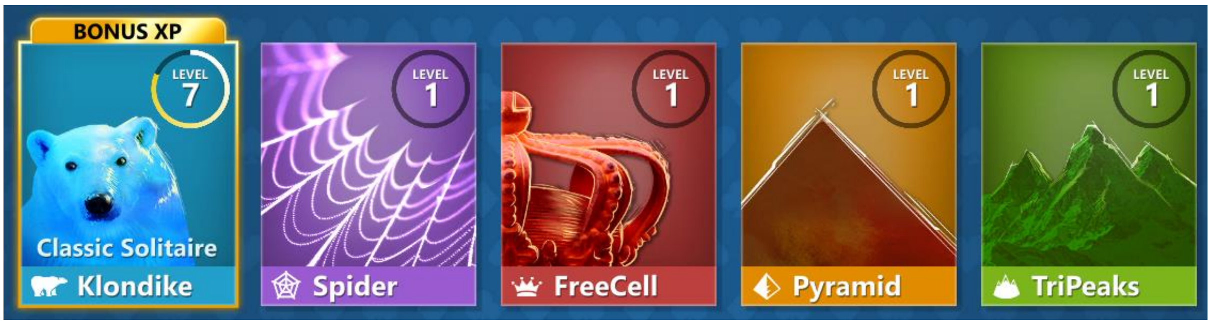
Survey Report: Game Environments

This section looks at multiple game environments that are relevant to my project. For each game environment, I analyse the game's visual design, interesting features and functionalities, detailing how they relate to my project and listing anything that I can take away from looking at the game environments; how I can use these features in my own project as well as looking at how they will improve my project.

Microsoft Solitaire Collection

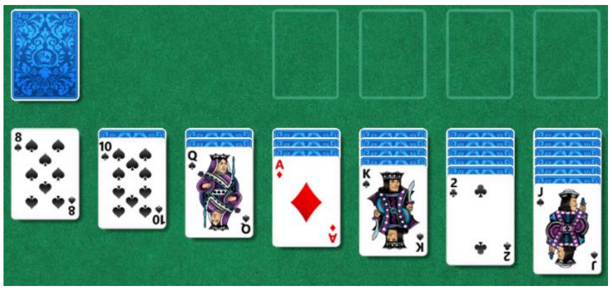
Analysis

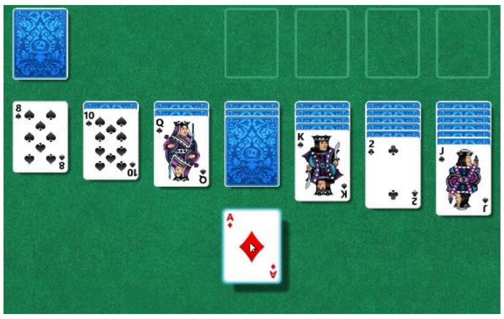
When you enter Microsoft Solitaire Collection, you are greeted by a nice-looking user interface which has five options of different card games that you can play. These options are Klondike, Spider, FreeCell, Pyramid and TriPeaks. Each of these are games that are played with a deck or multiple decks of cards.



For my project this is a useful source of inspiration, as it shows a visually pleasing and easily usable way to present the options to the user; with the name of each game listed and an image and colour representing that game. I would like to use a similar format for the main menu of my game, having five different buttons for each playable game with the name of the game listed with an image of the game and colour coordination to easily identify which game is which.

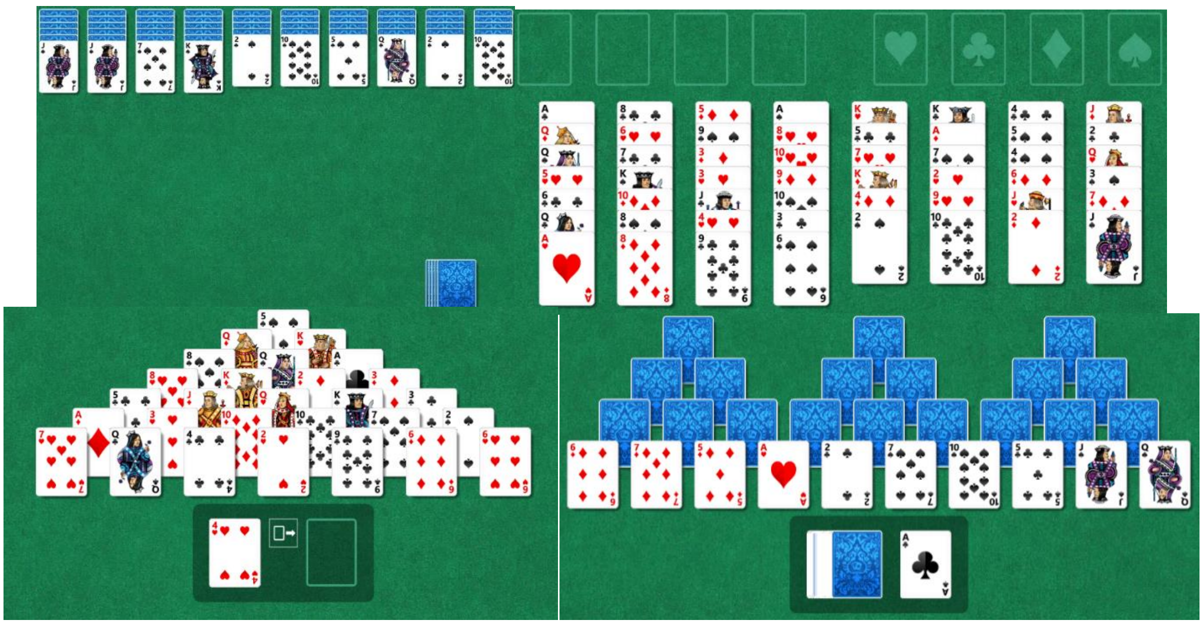
When you select one of these options you are then shown a menu where you can select the difficulty of the game you are playing. You are then put into the game environment where you play the game.





The user can either move cards by holding left click and dragging them to a new position or by clicking a card and then clicking the position they want to move it to. When a card is selected, it shows a blue outline around it. As can be seen in the above image on the right. If the move is illegal, then it either selects the card you clicked if you tried to move to another card or deselects the selected card if you clicked one of the empty spaces, if you dragged the card, it is automatically moved back to its original position. This can be used in my own game to aid the user in playing; when the user selects a piece on the board, that piece should be visibly highlighted so the user knows which piece they are moving and possibly have the ability to either move pieces by simply clicking the piece, then the new position, or by dragging the piece to a new position.

The same game layout and movement functionality is present in all available games, however, each different game works under a different ruleset. This is very similar to what I would like to do with my project, but with board games. Where possible I would like to reuse functionality and resources similarly to how it is done in this game environment. Where each card game has the same background and cards, I would like to reuse the same background and board where possible, and I would also like to reuse movement functionalities with each different board game.



The images above show how resources are reused in each game, with the same card designs and background being used instead of unnecessarily creating a new environment for each game. I would like to reuse piece designs and boards where possible, perhaps using the same board for both chess and checkers and allowing the board to be resized for other games and use the same pieces for games where you only need pieces of two different colours such as checkers, polish draughts, Gomoku and Dama. This will likely improve development time as I am able to reuse code and images instead of having to create new things for each game.

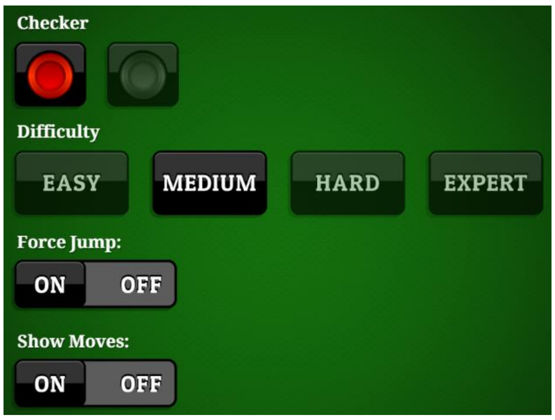
Takeaways

- Create a nice UI menu with suitable images and text so the user can easily choose the game they would like to play
- Have selected piece visibly highlighted to the user
- Reuse movement functionalities and piece design between games
- Design board so that it can be resized and used in multiple games

Checkers Game[8]

Analysis

When you start the game, you are shown a menu where you gave the option to select single player or multiplayer. Once you select this, you are shown a menu where you can choose which colour starts, the difficulty (if playing single player) and options to modify the rules to not force the user to take a piece and whether the game will show the user moves or not.

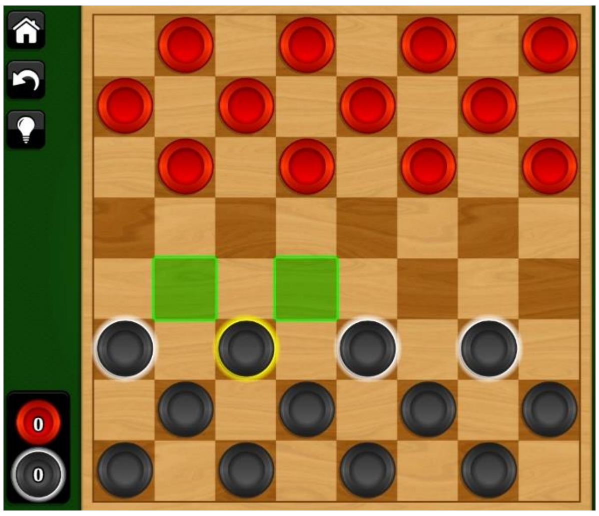

Checker

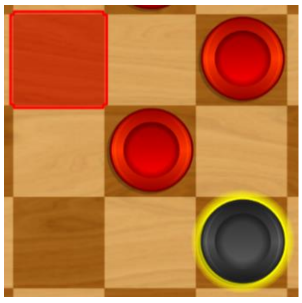
Difficulty
EASY [✓ MEDIUM] HARD EXPERT

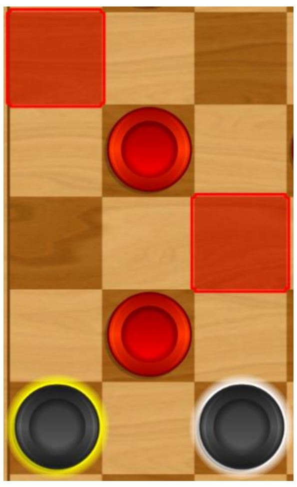
Force Jump:
[✓ ON] OFF

Show Moves:
[✓ ON] OFF

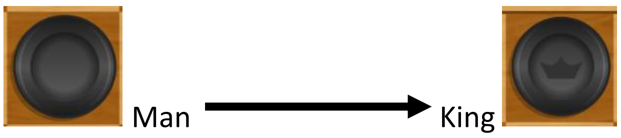
When you enter the match, you are shown a board with the pieces set up, and a score in the bottom left corner which keeps track of how many pieces each side has taken. The pieces that can be moved are highlighted in white, and the selected piece is highlighted in yellow. You can select a piece by clicking on it. When a piece is selected, the legal moves for that piece are displayed to the user by highlighting the possible new positions in green, if the move takes a piece, the position is highlighted in red. Possible moves to take two pieces (double jumps) are also shown to the user if available, and the user can decide between taking just one piece or both.







Once a user has a piece reach the other end of the board, the sprite image used for the piece changes, to signify that it has become a king.



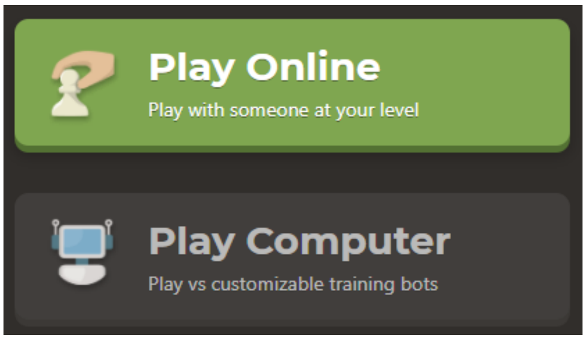
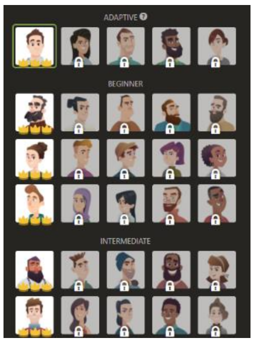
Takeaways

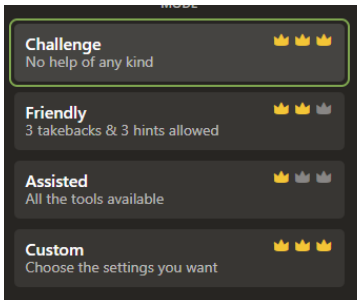
- As well as having the currently selected piece highlighted, have all pieces that can make a legal move highlighted to the user
- Display legal moves to the user by highlighting the possible new positions of the piece
- Ensure that pieces visibly change when they become a king

Chess Game[9]

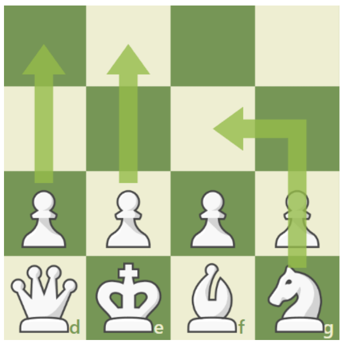
Analysis

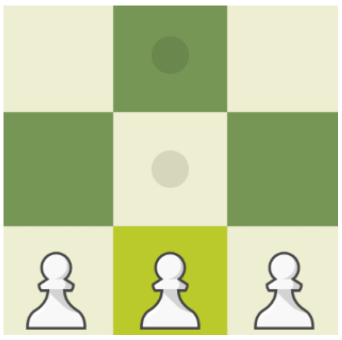
When you enter the website, you are given the choice between playing online or playing against the computer. When you choose to play online, you are asked your skill level and the game searches for an appropriate opponent for you to play against. If you select play against computer, you are given a choice of many different levels or bots to play against, each of varying strength and play style. Additionally, you can enable coaching features which give you insight into the game and there is no time limit. Once you have chosen which AI you would like to play against, you are given the option of choosing a mode to play in, affecting how much help you are given and how many times you can undo moves.

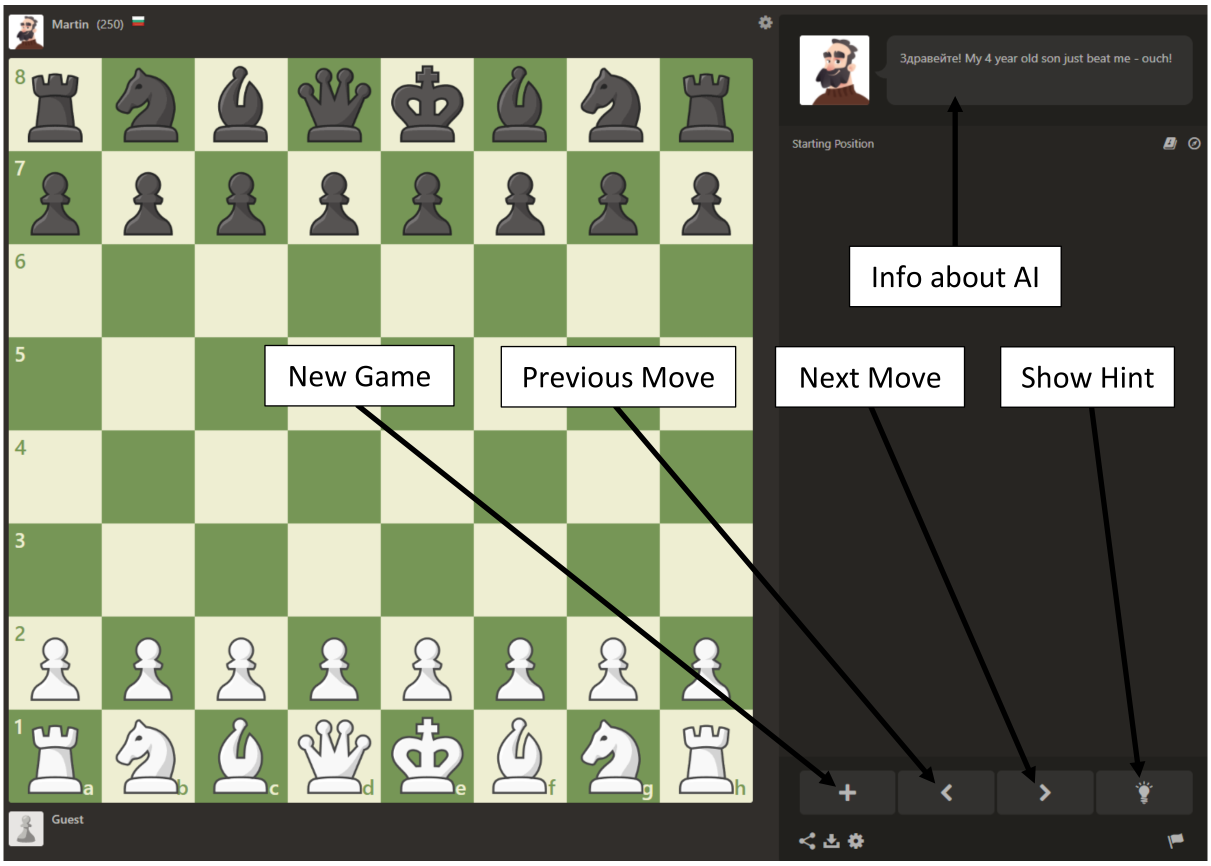

Play Online
Play with someone at your level
Play Computer
Play vs customizable training bots



Challenge
No help of any kind
Friendly
3 takebacks & 3 hints allowed
Assisted
All the tools available
Custom
Choose the settings you want

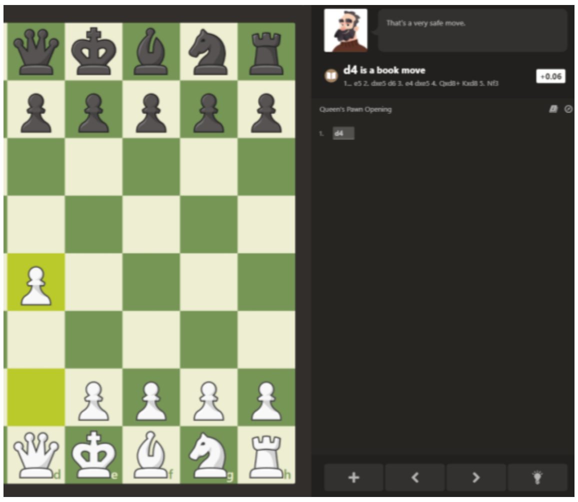
If you select challenge mode it is a normal game of chess with no assistance, if you choose friendly mode, you can undo three moves and can get 3 hints on what moves to make, assisted mode shows you arrows on which moves are best to make in every situation during the game (as shown below). When you enter the game, you are greeted by a board with all pieces laid out on the board, with each piece clearly represented by an understandable design. You can select a piece by left clicking on it, which highlights the tile that the piece is in, to show you what piece you have selected, and shows what legal moves are available for that piece, represented by grey circles on the legal tiles. You can move the piece by then clicking one of the legal tiles or by clicking and dragging the piece to a new position.

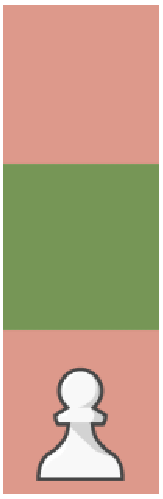




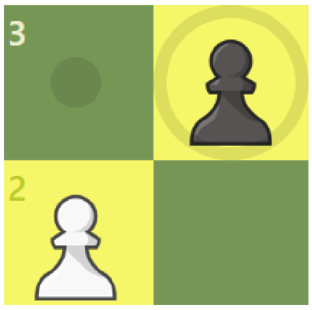


Once you have made a move, your move, it is listed on the right-hand side, and it is rated on how good it is such as saying it is a book move, perfect, a blunder etc. Until your opponent makes a move, the move that you just made is displayed by highlighting the tiles that you moved your piece to and from, once your opponent makes a move, their move is listed on the right and is also rated and highlighted on the board until you make your next move. If you would like to undo your move there is a "Previous Move" button which allows you to step back a move allowing you to change your move, if you then decide that you would like to go back with your original move, there is a "Next Move" button which repeats the reverse move. If you are unsure of what move to make, there is a "Show Hint" button, which, when pressed will suggest a move by highlighting your piece and the suggested new position in red.

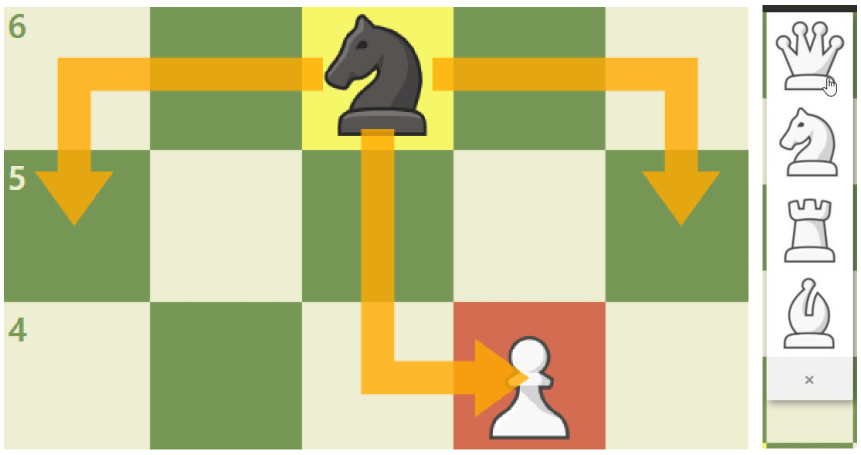




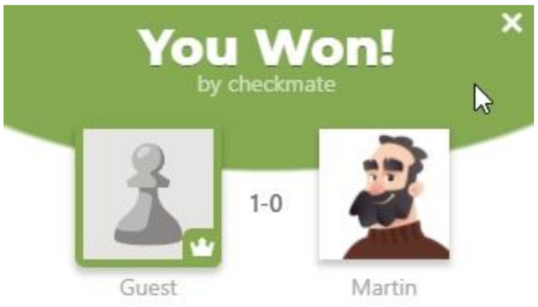
When a move to take a piece is available, instead of the move being represented by a filled in grey circle on the tile, it is represented by a grey circle outline around the piece that you can take. This helps the user distinguish between a move that will take a piece and one that doesn't.



Another feature of this chess game is the ability to right click and draw arrows on the board, this can be used in many ways, such as drawing potential moves the opponent can make on the board. You can also right click a tile to highlight it red, this can be used to signify that a tile is dangerous. These features are likely more suited for high level players that would like to analyse the game as they play it, but can be very useful to see what move you should make next.



Once a pawn reaches the enemies end of the board and can be promoted, the user is shown a small interface with a list of possible promotions that the user can select from (above right), allowing the user to either turn their pawn into a Queen, Knight, Rook or Bishop. This is a nice way to allow the user the option on what to promote to and is laid out in a clear, understandable way. When the game is over, there is a pop-up window which tells you the result of the game, whether it is a win, loss or a draw.



Takeaways

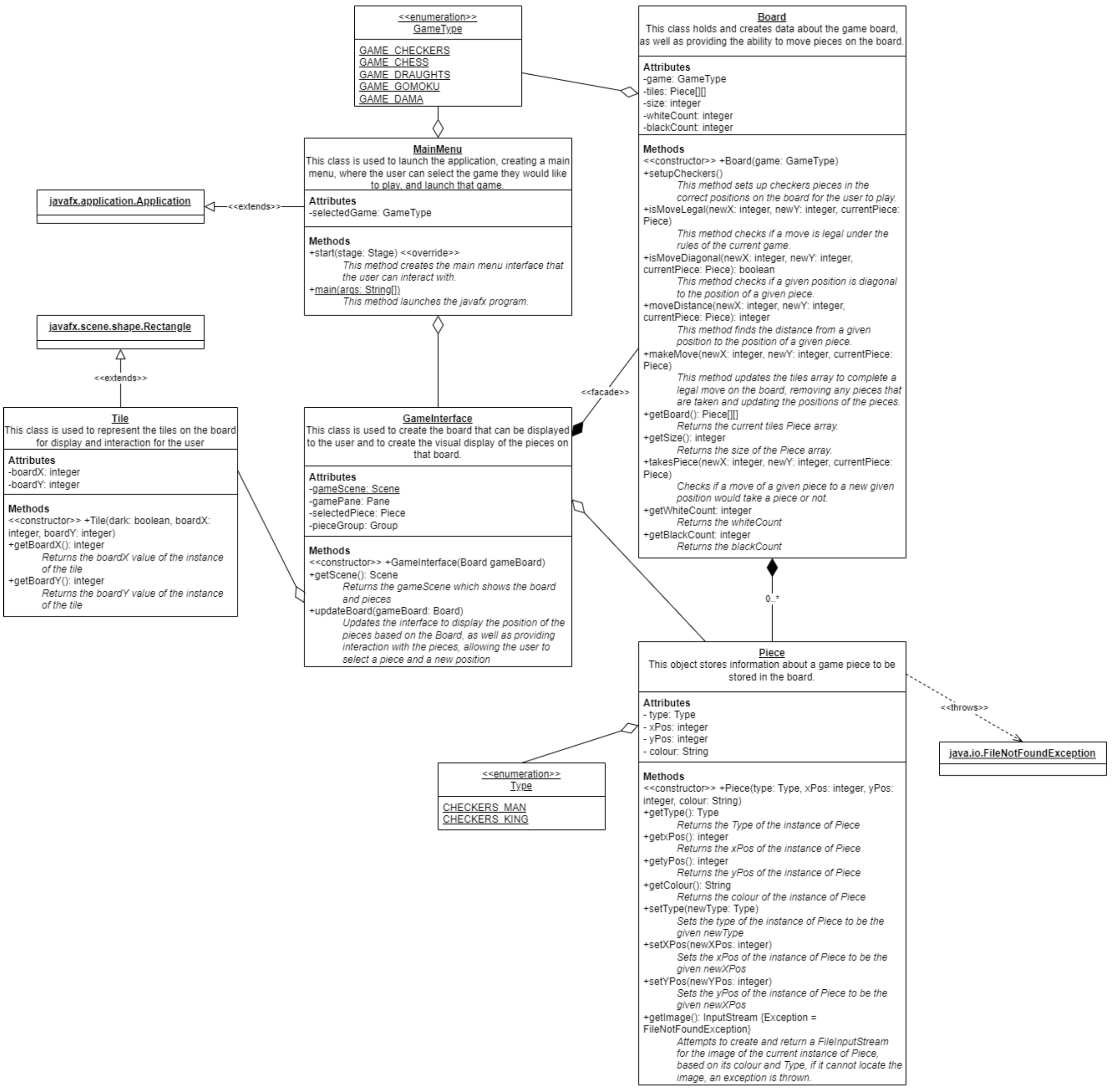
- For each piece in my chess game, have all legal moves listed for that piece when it is selected, to make it easier for the user to play the game (applies to other games too)
- Have the most recent move made by either player shown on the board by highlighting the current position and the previous position (could also be used for other games)
- Have moves that take a piece displayed differently than regular moves, so that the user can easily tell what will happen if they make that move (applies to all games)
- Have the options for promotion appear in a small UI when a pawn reaches the end of the board
- Have a small pop-up window to show the result of games such as who won

Design

This section details the of the programs that I have produced throughout my project, describing the functionalities and the design of each program and explaining in detail how the classes and methods work.

Proof of Concept Checkers Board Game

Below is the UML for the Proof of Concept Checkers Board Game, with more detailed explanations of some more complicated classes and methods below.


<<enumeration>>
GameType
GAME_CHECKERS
GAME_CHESS
GAME_DRAUGHTS
GAME_GOMOKU
GAME_DAMA
Board
This class holds and creates data about the game board, as well as providing the ability to move pieces on the board.
Attributes
-game: GameType
-tiles: Piece[][]
-size: integer
-whiteCount: integer
-blackCount: integer
Methods
<<constructor>> +Board(game: GameType)
+setupCheckers()
This method sets up checkers pieces in the correct positions on the board for the user to play.
+isMoveLegal(newX: integer, newY: integer, currentPiece: Piece)
This method checks if a move is legal under the rules of the current game.
+isMoveDiagonal(newX: integer, newY: integer, currentPiece: Piece): boolean
This method checks if a given position is diagonal to the position of a given piece.
+moveDistance(newX: integer, newY: integer, currentPiece: Piece): integer
This method finds the distance from a given position to the position of a given piece.
+makeMove(newX: integer, newY: integer, currentPiece: Piece)
This method updates the tiles array to complete a legal move on the board, removing any pieces that are taken and updating the positions of the pieces.
+getBoard(): Piece[][]
Returns the current tiles Piece array.
+getSize(): integer
Returns the size of the Piece array.
+takesPiece(newX: integer, newY: integer, currentPiece: Piece)
Checks if a move of a given piece to a new given position would take a piece or not.
+getWhiteCount: integer
Returns the whiteCount
+getBlackCount: integer
Returns the blackCount
javafx.application.Application
<<extends>>
MainMenu
This class is used to launch the application, creating a main menu, where the user can select the game they would like to play, and launch that game.
Attributes
-selectedGame: GameType
Methods
+start(stage: Stage) <<override>>
This method creates the main menu interface that the user can interact with.
+main(args: String[])
This method launches the javafx program.
javafx.scene.shape.Rectangle
<<extends>>
Tile
This class is used to represent the tiles on the board for display and interaction for the user
Attributes
-boardX: integer
-boardY: integer
Methods
<<constructor>> +Tile(dark: boolean, boardX: integer, boardY: integer)
+getBoardX(): integer
Returns the boardX value of the instance of the tile
+getBoardY(): integer
Returns the boardY value of the instance of the tile
<<facade>>
GameInterface
This class is used to create the board that can be displayed to the user and to create the visual display of the pieces on that board.
Attributes
-gameScene: Scene
-gamePane: Pane
-selectedPiece: Piece
-pieceGroup: Group
Methods
<<constructor>> +GameInterface(Board gameBoard)
+getScene(): Scene
Returns the gameScene which shows the board and pieces
+updateBoard(gameBoard: Board)
Updates the interface to display the position of the pieces based on the Board, as well as providing interaction with the pieces, allowing the user to select a piece and a new position
0..*
Piece
This object stores information about a game piece to be stored in the board.
Attributes
- type: Type
- xPos: integer
- yPos: integer
- colour: String
Methods
<<constructor>> +Piece(type: Type, xPos: integer, yPos: integer, colour: String)
+getType(): Type
Returns the Type of the instance of Piece
+getxPos(): integer
Returns the xPos of the instance of Piece
+getyPos(): integer
Returns the yPos of the instance of Piece
+getColour(): String
Returns the colour of the instance of Piece
+setType(newType: Type)
Sets the type of the instance of Piece to be the given newType
+setXPos(newXPos: integer)
Sets the xPos of the instance of Piece to be the given newXPos
+setYPos(newYPos: integer)
Sets the yPos of the instance of Piece to be the given newXPos
+getImage(): InputStream {Exception = FileNotFoundException}
Attempts to create and return a FileInputStream for the image of the current instance of Piece, based on its colour and Type, if it cannot locate the image, an exception is thrown.
<<throws>>
java.io.FileNotFoundException
<<enumeration>>
Type
CHECKERS_MAN
CHECKERS_KING

Piece Class:

The piece class is a class that is used to store information about a game piece, each Piece has a type, which represents what the actual piece is; in terms of checkers, it will either be a man or a king, but for a game such as chess the type may be a queen, pawn, king etc. Each piece also has a xPos and a yPos, which represent the x and y position of a piece along the board, for example, when xPos and yPos are both 0, the Piece is in the top left position of the board; as yPos increase, the Piece moves vertically, up the board; as xPos increases, the Piece moves horizontally, across the board.

The constructor for this class simply sets the variables to the input values.

When called, the method getImage uses the type and colour of the Piece to attempt to return a FileInputStream[10] for the image equivalent to the Piece. If the image cannot be located for any reason, an exception[11] is thrown.

Board Class:

This class is used to create a data structure that acts like a board, storing Pieces in the correct positions along the data structure as they would be on the actual board. The constructor sets the game to be the input GameType, then it sets the size to the size of the board based on which GameType was input; for Checkers, Chess and Dama, the size is 8, as an 8 by 8 board is used; for Draughts, the size is 10, as a 10 by 10 board is used; for Gomoku, the size is 15 as a 15 by 15 board is used. The constructor then creates a new two-dimensional Piece array with “size” amount of rows and “size” amount of columns. Currently only checkers is available to be played, but once other games are made available, the constructor would then setup or call a method to setup the board, creating Pieces in positions in the array to represent the starting positions of the board game.

The setupCheckers method loops through the tiles array and creates Pieces of the correct colour in the correct positions, setting up a checkers game.

The Board class has many methods for checking whether a potential move is legal or not:

> The isMoveDiagonal method is used to check whether a new position is diagonal to the current position or not; if a new position is diagonal, then the absolute value of the new Y position minus the current Y position will be equal to the absolute of the new X position minus the current X position. The isMoveDiagonal method checks whether this is true for the input positions and Piece and returns true if the move is diagonal and false if it isn’t. This method is very useful as all moves in Checkers must be diagonal (it can also be used in chess for certain pieces).
>
> The moveDistance method uses similar code, but instead of returning a true or false, it returns the maximum value of the two absolutes, in other words the greater between the absolute of the new Y minus the current Y and the new X minus the current X. It then returns this value.
>
> The takesPiece method is used to check whether a move should be taking a piece or not, based on the rule of the game being played. For this proof of concept, the method checks whether the move is taking a piece by finding the position between the current position of the Piece and the new position and checking whether that position is diagonal to the current position in a legally movable direction and whether that position contains a Piece of the opposite colour to the Piece being

moved. If the conditions are all true, then the method returns true, if they are not, the method returns false.

For testing purposes, I decided not to implement a turns system into this prototype, as it would make it much more time consuming to test things such as taking pieces, promotion and movement of promoted pieces.

The above three methods are then used in the isMoveLegal method which uses a switch statement for GameType, as each game will have a different ruleset. Currently only checkers rules are setup. For the Checkers GameType, the method first checks if the currentPiece is not null, that the new move is within the bounds of the board and that the new position does not already have a Piece in it. It then checks; that the move is diagonal, using the isMoveDiagonal method; that the Piece is not moving more than 1 tile diagonally, using the moveDistance method; that the Piece is moving only forwards if it is a man and any direction if it is a King. If all of these are true, than the method returns true, but if one is found to be false, the method then checks whether the move is taking a piece by using the takesPiece method. If takesPiece returns true, then isMoveLegal returns true.

The makeMove method checks if isMoveLegal returns true, if it does then it checks if takesPiece is true, if it is, it removes the takenPiece from the Piece array and adjusts the whiteCount or blackCount based on the colour of the Piece that was taken. It then completes the move by updating the Piece array and the xPos and yPos of the Piece. It then checks whether the moved piece is in a promotion position and changes its Type to a king if it is. If the move is not legal, it does not complete the move.

Tile Class:

The Tile class is an extension of the JavaFX Rectangle[12] class, which is used to display the tiles of the board on the user interface and can also be used to provide interaction with the board. The Tile class has two additional attributes, boardX and boardY, which are used to return the position of the tile on the actual board, whereas the regular X and Y of the tile, would return the position on the window of the interface. The constructor sets the Height and Width of the Tile and sets its fill based on the input value of dark; if dark is true, the colour is set to a dark green, if it is false, it is set to beige. It then sets the X and Y values to be the width and height of the Tile multiplied by the input boardX and boardY, so that tiles will appear on the interface in the correct positions. It then sets the boardX and boardY to the inputs.

GameInterface Class:

The GameInterface class is a class that is used to create and update a JavaFX scene[13] which is displayed to the user and can be interacted with allowing them to play a game of checkers with a visual interface. The constructor takes a Board and first the gets the size of the Board, it then initialises gamePane and pieceGroup. Based on the size of the board, it creates Tiles to be displayed on the interface, and adds functionality to them, allowing for them to be clicked on, using a JavaFX EventHandler[14]:

> When the user clicks on a tile it checks if the selectedPiece is null and if it is, a JavaFX Alert popup is shown to the user telling them they must select a piece in order to make a move, if a Piece is selected, then the tile move legality is checked using the boardX and boardY of the clicked tile as the new x and y position, with the selectedPiece as the currentPiece. If the move is illegal, an Alert is shown telling the user that the move they attempted is illegal. If the move was legal then, makeMove is called, and the whiteCount and blackCount are checked. If either the whiteCount or blackCount is 0 after the move is made, an Alert is shown telling the user that the game is over and who won, the game is then closed.

Each tile created is added to the tileGroup. Which is then, as well as the pieceGroup, is added to the gamePane, which will be used to create the gameScene. The updateBoard method is called, and then the gameScene is created using the gamePane.

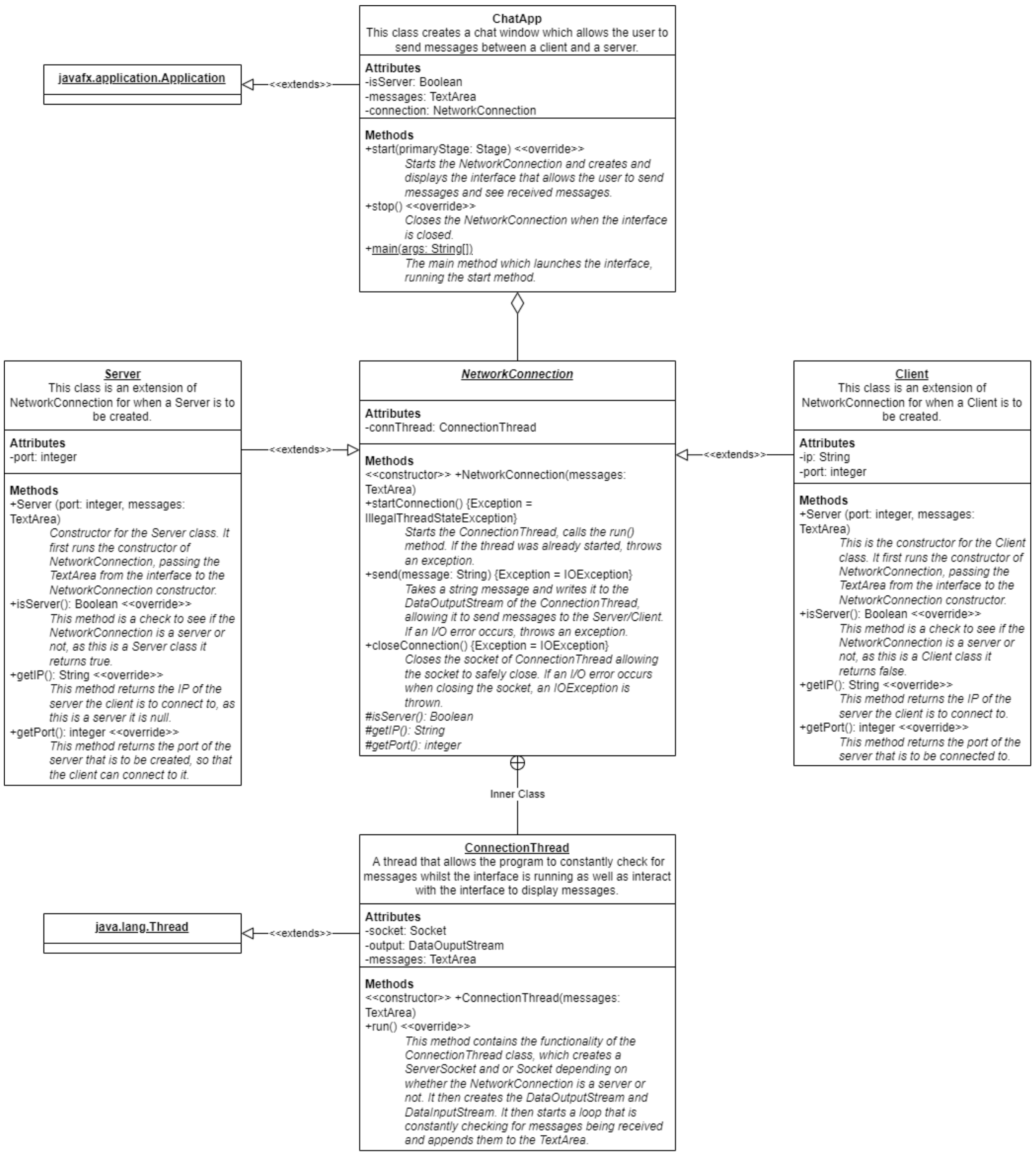
The updateBoard method first gets the size of the Board and the Piece array of the Board. It then clears the pieceGroup, removing all pieces from the interface. It then loops through the pieceArray and for each Piece which isn't null and creates an ImageView of the Piece by using the getImage method of the Piece and sets the position of the Image on the interface. It also adds functionality to the Image using a JavaFX EventHandler, so that when clicked, the selectedPiece is set to the Piece that has been clicked on. The method then adds the ImageView to the pieceGroup.

<u>**MainMenu Class:**</u>

The MainMenu class is an extension of the JavaFX Application[15] class. It has a start method which creates an interface with text asking the user to choose a game type, with 5 buttons, with options of games to play, in this version, only Checkers is available to be selected. There is also a button which says, "Start Game", which initially would change the window to the game window, however it has been adjusted to allow concurrent behaviour, opening a new game window, allowing the user to play multiple games of Checkers at the same time as each other. When the user selects one of the games, the selectedGame is changed to the GameType of the game that has been selected and when the startGame button is pressed a new Board is created with the selectedGame as the GameType and creates a new GameInterface with the Board. It then uses the getScene method of GameInterface and creates a new window with the Scene returned from the getScene method. The main method launches the JavaFX interface.

Proof of Concept Server Chat App

Below is the UML for the Proof of Concept Server Chat App, with more detailed explanations of some more complicated classes and methods below.



ChatApp
This class creates a chat window which allows the user to send messages between a client and a server.

Attributes
-isServer: Boolean
-messages: TextArea
-connection: NetworkConnection

Methods
+start(primaryStage: Stage) <<override>>
Starts the NetworkConnection and creates and displays the interface that allows the user to send messages and see received messages.
+stop() <<override>>
Closes the NetworkConnection when the interface is closed.
+main(args: String[])
The main method which launches the interface, running the start method.

Server
This class is an extension of NetworkConnection for when a Server is to be created.

Attributes
-port: integer

Methods
+Server (port: integer, messages: TextArea)
Constructor for the Server class. It first runs the constructor of NetworkConnection, passing the TextArea from the interface to the NetworkConnection constructor.
+isServer(): Boolean <<override>>
This method is a check to see if the NetworkConnection is a server or not, as this is a Server class it returns true.
+getIP(): String <<override>>
This method returns the IP of the server the client is to connect to, as this is a server it is null.
+getPort(): integer <<override>>
This method returns the port of the server that is to be created, so that the client can connect to it.

NetworkConnection

Attributes
-connThread: ConnectionThread

Methods
<<constructor>> +NetworkConnection(messages: TextArea)
+startConnection() {Exception = IllegalThreadStateException}
Starts the ConnectionThread, calls the run() method. If the thread was already started, throws an exception.
+send(message: String) {Exception = IOException}
Takes a string message and writes it to the DataOutputStream of the ConnectionThread, allowing it to send messages to the Server/Client. If an I/O error occurs, throws an exception.
+closeConnection() {Exception = IOException}
Closes the socket of ConnectionThread allowing the socket to safely close. If an I/O error occurs when closing the socket, an IOException is thrown.
#isServer(): Boolean
#getIP(): String
#getPort(): integer

Client
This class is an extension of NetworkConnection for when a Client is to be created.

Attributes
-ip: String
-port: integer

Methods
+Server (port: integer, messages: TextArea)
This is the constructor for the Client class. It first runs the constructor of NetworkConnection, passing the TextArea from the interface to the NetworkConnection constructor.
+isServer(): Boolean <<override>>
This method is a check to see if the NetworkConnection is a server or not, as this is a Client class it returns false.
+getIP(): String <<override>>
This method returns the IP of the server the client is to connect to.
+getPort(): integer <<override>>
This method returns the port of the server that is to be connected to.

ConnectionThread
A thread that allows the program to constantly check for messages whilst the interface is running as well as interact with the interface to display messages.

Attributes
-socket: Socket
-output: DataOuputStream
-messages: TextArea

Methods
<<constructor>> +ConnectionThread(messages: TextArea)
+run() <<override>>
This method contains the functionality of the ConnectionThread class, which creates a ServerSocket and or Socket depending on whether the NetworkConnection is a server or not. It then creates the DataOutputStream and DataInputStream. It then starts a loop that is constantly checking for messages being received and appends them to the TextArea.

NetworkConnection Class

This class acts as an abstraction layer for allowing a client or server to interact with the JavaFX application. The NetworkConnection constructor is passed a JavaFX TextArea[17] which appears on the JavaFX interface, this is then passed to the ConnectionThread when the ConnectionThread is initialised in the constructor.

ConnectionThread Class

The ConnectionThread class is a private class that is inside the NetworkConnection class. ConnectionThread's run method creates a ServerSocket and or a Socket depending on whether the NetworkConnection is a Server or client respectively. It then creates a DataOutputStream[18] and DataInputStream[19] which allow messages to be sent and received from the Client and Server. It then starts a loop that infinitely checks for messages being received and when a message is received, it output's it to the TextArea on the JavaFX interface.

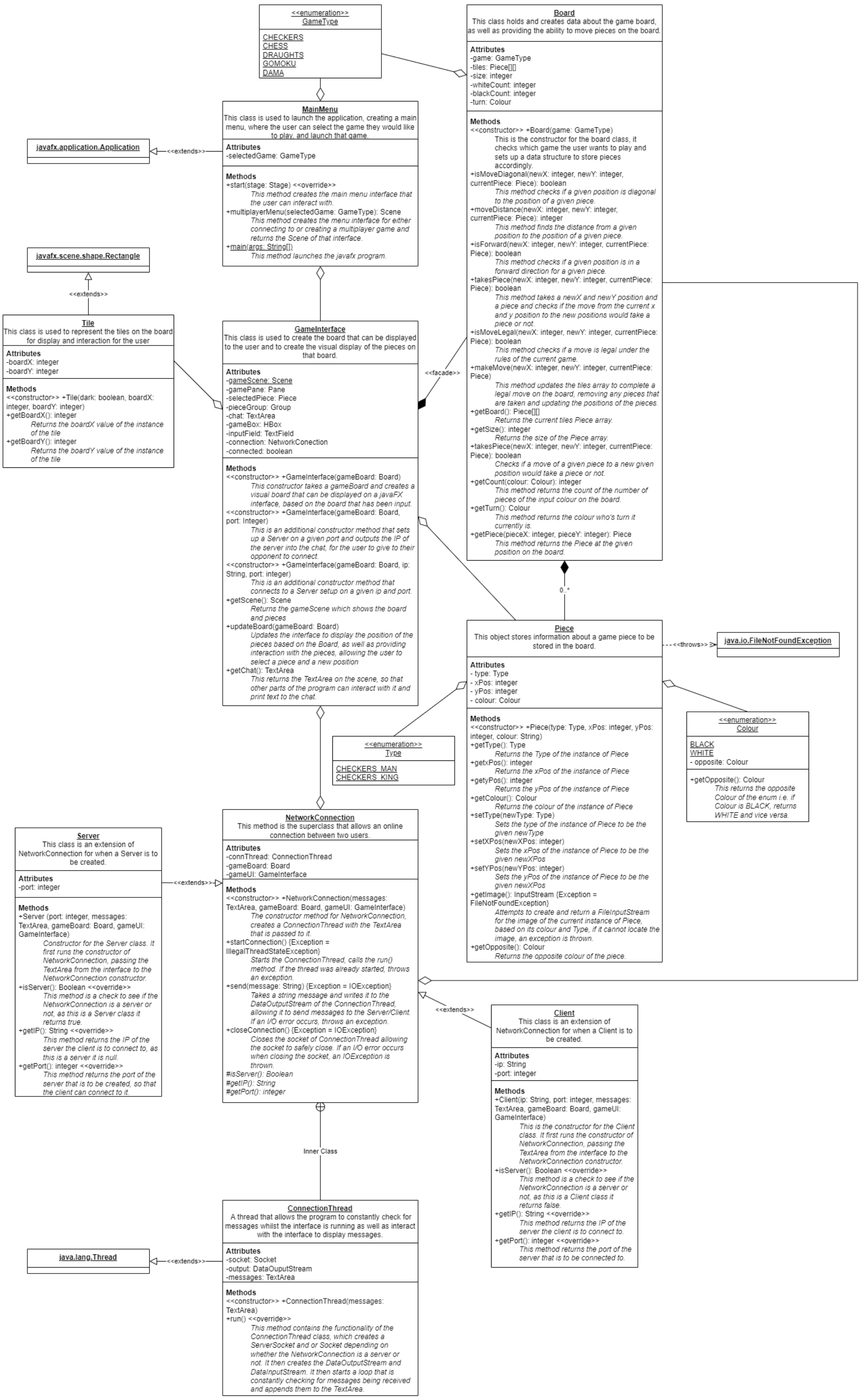
ChatApp Class

The ChatApp class is an extension of the JavaFX Application class. It has an attribute isServer which can be changed to true or false. When it is set to true and the program is ran, the NetworkConnection will be created as a Server and when it is set to false, it will be created as a Client. The class has a start method which creates an interface with a large TextArea, in which messages that are sent and received are displayed, as well as error messages when an issue occurs in the process of sending a message. Underneath this TextArea is a TextField[20] which the user can type into. When the user presses enter, the TextField is cleared and the messages is sent to the Server or Client.

Concurrency-based Game Environment

To make my full software design, I used the knowledge and experience gained in the two prototypes and re-designed my system, combining the ideas from the two prototypes to create the full concurrency-based game environment.

Below is the design is the UML design for my Full Software Implementation of my Concurrency-based game environment, with more detailed explanations of certain classes, methods and how classes are related.



<<enumeration>>
GameType

CHECKERS
CHESS
DRAUGHTS
GOMOKU
DAMA

Board
This class holds and creates data about the game board, as well as providing the ability to move pieces on the board.

Attributes
-game: GameType
-tiles: Piece[][]
-size: integer
-whiteCount: integer
-blackCount: integer
-turn: Colour

Methods
<<constructor>> +Board(game: GameType)
This is the constructor for the board class, it checks which game the user wants to play and sets up a data structure to store pieces accordingly.
+isMoveDiagonal(newX: integer, newY: integer, currentPiece: Piece): boolean
This method checks if a given position is diagonal to the position of a given piece.
+moveDistance(newX: integer, newY: integer, currentPiece: Piece): integer
This method finds the distance from a given position to the position of a given piece.
+isForward(newX: integer, newY: integer, currentPiece: Piece): boolean
This method checks if a given position is in a forward direction for a given piece.
+takesPiece(newX: integer, newY: integer, currentPiece: Piece): boolean
This method takes a newX and newY position and a piece and checks if the move from the current x and y position to the new positions would take a piece or not.
+isMoveLegal(newX: integer, newY: integer, currentPiece: Piece): boolean
This method checks if a move is legal under the rules of the current game.
+makeMove(newX: integer, newY: integer, currentPiece: Piece)
This method updates the tiles array to complete a legal move on the board, removing any pieces that are taken and updating the positions of the pieces.
+getBoard(): Piece[][]
Returns the current tiles Piece array.
+getSize(): integer
Returns the size of the Piece array.
+takesPiece(newX: integer, newY: integer, currentPiece: Piece): boolean
Checks if a move of a given piece to a new given position would take a piece or not.
+getCount(colour: Colour): integer
This method returns the count of the number of pieces of the input colour on the board.
+getTurn(): Colour
This method returns the colour who's turn it currently is.
+getPiece(pieceX: integer, pieceY: integer): Piece
This method returns the Piece at the given position on the board.

MainMenu
This class is used to launch the application, creating a main menu, where the user can select the game they would like to play, and launch that game.

Attributes
-selectedGame: GameType

Methods
+start(stage: Stage) <<override>>
This method creates the main menu interface that the user can interact with.
+multiplayerMenu(selectedGame: GameType): Scene
This method creates the menu interface for either connecting to or creating a multiplayer game and returns the Scene of that interface.
+main(args: String[])
This method launches the javafx program.

Tile
This class is used to represent the tiles on the board for display and interaction for the user

Attributes
-boardX: integer
-boardY: integer

Methods
<<constructor>> +Tile(dark: boolean, boardX: integer, boardY: integer)
+getBoardX(): integer
Returns the boardX value of the instance of the tile
+getBoardY(): integer
Returns the boardY value of the instance of the tile

GameInterface
This class is used to create the board that can be displayed to the user and to create the visual display of the pieces on that board.

Attributes
-gameScene: Scene
-gamePane: Pane
-selectedPiece: Piece
-pieceGroup: Group
-chat: TextArea
-gameBox: HBox
-inputField: TextField
-connection: NetworkConection
-connected: boolean

Methods
<<constructor>> +GameInterface(gameBoard: Board)
This constructor takes a gameBoard and creates a visual board that can be displayed on a javaFX interface, based on the board that has been input.
<<constructor>> +GameInterface(gameBoard: Board, port: Integer)
This is an additional constructor method that sets up a Server on a given port and outputs the IP of the server into the chat, for the user to give to their opponent to connect.
<<constructor>> +GameInterface(gameBoard: Board, ip: String, port: integer)
This is an additional constructor method that connects to a Server setup on a given ip and port.
+getScene(): Scene
Returns the gameScene which shows the board and pieces
+updateBoard(gameBoard: Board)
Updates the interface to display the position of the pieces based on the Board, as well as providing interaction with the pieces, allowing the user to select a piece and a new position
+getChat(): TextArea
This returns the TextArea on the scene, so that other parts of the program can interact with it and print text to the chat.

Piece
This object stores information about a game piece to be stored in the board.

Attributes
- type: Type
- xPos: integer
- yPos: integer
- colour: Colour

Methods
<<constructor>> +Piece(type: Type, xPos: integer, yPos: integer, colour: String)
+getType(): Type
Returns the Type of the instance of Piece
+getxPos(): integer
Returns the xPos of the instance of Piece
+getyPos(): integer
Returns the yPos of the instance of Piece
+getColour(): Colour
Returns the colour of the instance of Piece
+setType(newType: Type)
Sets the type of the instance of Piece to be the given newType
+setXPos(newXPos: integer)
Sets the xPos of the instance of Piece to be the given newXPos
+setYPos(newYPos: integer)
Sets the yPos of the instance of Piece to be the given newXPos
+getImage(): InputStream {Exception = FileNotFoundException}
Attempts to create and return a FileInputStream for the image of the current instance of Piece, based on its colour and Type, if it cannot locate the image, an exception is thrown.
+getOpposite(): Colour
Returns the opposite colour of the piece.

<<enumeration>>
Colour

BLACK
WHITE
- opposite: Colour

+getOpposite(): Colour
This returns the opposite Colour of the enum i.e. if Colour is BLACK, returns WHITE and vice versa.

<<enumeration>>
Type

CHECKERS_MAN
CHECKERS_KING

Server
This class is an extension of NetworkConnection for when a Server is to be created.

Attributes
-port: integer

Methods
+Server (port: integer, messages: TextArea, gameBoard: Board, gameUI: GameInterface)
Constructor for the Server class. It first runs the constructor of NetworkConnection, passing the TextArea from the interface to the NetworkConnection constructor.
+isServer(): Boolean <<override>>
This method is a check to see if the NetworkConnection is a server or not, as this is a Server class it returns true.
+getIP(): String <<override>>
This method returns the IP of the server the client is to connect to, as this is a server it is null.
+getPort(): integer <<override>>
This method returns the port of the server that is to be created, so that the client can connect to it.

NetworkConnection
This method is the superclass that allows an online connection between two users.

Attributes
-connThread: ConnectionThread
-gameBoard: Board
-gameUI: GameInterface

Methods
<<constructor>> +NetworkConnection(messages: TextArea, gameBoard: Board, gameUI: GameInterface)
The constructor method for NetworkConnection, creates a ConnectionThread with the TextArea that is passed to it.
+startConnection() {Exception = IllegalThreadStateException}
Starts the ConnectionThread, calls the run() method. If the thread was already started, throws an exception.
+send(message: String) {Exception = IOException}
Takes a string message and writes it to the DataOutputStream of the ConnectionThread, allowing it to send messages to the Server/Client. If an I/O error occurs, throws an exception.
+closeConnection() {Exception = IOException}
Closes the socket of ConnectionThread allowing the socket to safely close. If an I/O error occurs when closing the socket, an IOException is thrown.
#isServer(): Boolean
#getIP(): String
#getPort(): integer

Client
This class is an extension of NetworkConnection for when a Client is to be created.

Attributes
-ip: String
-port: integer

Methods
+Client(ip: String, port: integer, messages: TextArea, gameBoard: Board, gameUI: GameInterface)
This is the constructor for the Client class. It first runs the constructor of NetworkConnection, passing the TextArea from the interface to the NetworkConnection constructor.
+isServer(): Boolean <<override>>
This method is a check to see if the NetworkConnection is a server or not, as this is a Client class it returns false.
+getIP(): String <<override>>
This method returns the IP of the server the client is to connect to.
+getPort(): integer <<override>>
This method returns the port of the server that is to be connected to.

ConnectionThread
A thread that allows the program to constantly check for messages whilst the interface is running as well as interact with the interface to display messages.

Attributes
-socket: Socket
-output: DataOuputStream
-messages: TextArea

Methods
<<constructor>> +ConnectionThread(messages: TextArea)
+run() <<override>>
This method contains the functionality of the ConnectionThread class, which creates a ServerSocket and or Socket depending on whether the NetworkConnection is a server or not. It then creates the DataOutputStream and DataInputStream. It then starts a loop that is constantly checking for messages being received and appends them to the TextArea.

Piece Class:

The Piece Class is almost identical to how it was in the Checkers proof of concept, with the main difference being that the colour of the Piece is now represented as an Enum called Colour as I felt it was a more appropriate data type to use. Additionally, there is now a method that will return the opposite colour of the piece.

Colour Class:

This is a new Enum class that is used to represent the colour of the piece. It has two values, WHITE and BLACK. It also has an attribute, opposite, which stores the opposite colour for each value, which will be automatically set. The method getOpposite returns the opposite colour for each value. If the Colour is set to WHITE, getOpposite will return BLACK and if Colour is set to BLACK it will return WHITE.

Board Class:

This class is used to create the data structure which stores the pieces and provides the ability to move pieces on the board. The setupCheckers method has been removed and should instead be implemented as part of the constructor. The constructor sets the game to the input GameType and sets the whiteCount and blackCount to 0. It then begins to setup the board based on the game that was selected. It sets the size attribute to the appropriate value for each game and creates the Piece array for this size. It then sets the turn value to be the Colour who's turn should be first, for example, in checkers Black goes first so turn is set to BLACK, in chess White goes first so turn is set to WHITE. It then sets up the board appropriately, looping through the array and creating pieces in the positions where they should be.

A new method, isForward has been added, to more easily check whether or not a piece is moving in a forward direction or not. This method should check the colour of the piece and the new Y position that the move is taking the piece. If the Y is changing correctly based on the Colour of the piece, then it returns true. For example, if White's forward direction is positive Y then it should return true as long as the Y value is increased by the move.

The getWhiteCount and getBlackCount methods have been combined into a getCount method which takes a Colour and returns the count of the given Colour.

The makeMove method now checks whether the colour of the piece whose move is being made is equivalent to the colour who's turn it is, and should update the turn to be the opposite colour when a move is made.

All other methods in the Board class work the same as described in the Checkers proof of concept section.

NetworkConnection Class:

This class acts as an abstraction layer allowing for a client or server to be create and interact with the user interface and board, in order to provide the multiplayer features to the game. It takes a TextArea, where messages can be displayed to the user, the Board which is being represented on the interface, in order to update it when a move is received from the other player, and also takes the GameInterface which it is initialised from, as it needs to update the interface whenever a move received and made.

Server Class:

The server class is an extension of the NetworkConnection class which is used to create a server, allowing a user to host a game on their device for somebody else to connect to.

Client Class:

The client class is an extension of the NetworkConnection class which is used to create a client, allowing a user to connect to a game being hosted on another users device.

ConnectionThread Class:

The ConnectionThread class is a private inner class within the NetworkConnection Class. The run method creates a ServerSocket and or a Socket based on whether the connection is a server or not. It then creates an output and input stream in order to send and receive messages. It then starts a loop which will constantly check for messages while the game is being played. Messages are sent in a way such that the program can tell the difference between a message that should be displayed in the chat, and a message which is telling the program to make a move on the board. If a message should be displayed in chat, then the ConnectionThread appends the message to the TextArea on the interface. If the message is a move that needs to be made, then the program gets the information about what piece is being moved and the new position of that piece, makes the move on the board and updates the interface to show that the move has been made.

The run method also contains an error reporting system that notifies the user if something goes wrong with the connection with the other user, and closes the game if a severe error occurs.

MainMenu Class:

The MainMenu class is adapted to support the ability to set up or connect to a multiplayer game. The start method creates an interface showing the games that are available to play, displayed on buttons, with icons to represent each game. Now, when a game has been selected, instead of the game starting immediately, the user is instead shown a new interface which is created by the multiplayerMenu method.

The multiplayerMenu method returns a scene where the user is asked if they want to play online or locally. If the user selects local, the game will be started as usual, however, should the user select online, the interface will change, asking the user if they would like to join an existing server or to host a game, should the user choose to host a game, a server will be set up and the game interface will be shown, with the ip to the server shown in the chat, so that it can be shared with another user for them to connect. Should the user choose to join a game, they will be asked to input an ip and, if a valid ip address is input, they will be connected to another user and can play a game with them and chat with the via the user interface. The server and client are created via the GameInterface class.

GameInterface Class:

The GameInterface class is used to display and interact with the game as well as providing an interface to report errors to the user and send and receive messages between a opponent, who is connected via multiplayer. The interface should display the game board on the left hand side of the screen with a chat and text input area on the right hand side of the screen.

The GameInterface has three constructors, the first sets up the default GameInterface, the second constructor sets up a Server NetworkConnection and displays the ip of the server in the chat, the third constructor creates a Client NetworkConnection and displays the interface to the user. The constructor also provides the ability to send messages between the Client and Server.

Everything else about the gameInterface is the same, however instead of using pop-ups to display errors or to tell the user that a move is illegal, these messages are shown in the chat, as I found that it was less intrusive to the user and thus improved the user experience.

Implementation

This section details the implementation of the programs that I have produced, first explaining my Software Engineering Process for each, explaining any issues I encountered with each program and how I resolved them, any algorithms that were particularly interesting or of note within my programs, my testing procedures on each program, information on how to run each program and a self-evaluation of my project implementation in comparison to my project goals.

Software Engineering Process

Proof of Concept Checkers Board Game

I wanted to design my prototype in a way that it can easily be adapted to support multiple different types of games to be played. In order to play different types of games, different types of pieces would be needed, with different rules for the moves of each piece in each game. And since the game would be played on an interface and would need concurrent behaviour, it is based on an MVC architecture.

I made sure to store my code on a VCS, committing my work to GitHub whenever I made updates.

I also created Javadoc for each of my classes and methods, which can be seen in the code on my git repository.

I started by first creating a class called Piece that would be used to represent the Pieces that are on the board. In order to distinguish between different types of pieces, it would need to have a variable representing its type. For this, I decided to use an Enumeration, as the Type of the piece would only be one of a specific set of values; currently this only contains CHECKERS_MAN and CHECKERS_KING, but new values can be added for other games, for example, CHESS_PAWN, CHESS_QUEEN, CHESS_ROOK etc. I also gave each piece an xPos and yPos to represent their position on the board. Each piece also must also have a colour to represent which side it is on. I decided to use a string for the colour, however, in hindsight, I believe it would be better to use another Enum which only has two options for colour as the colour will only be one of two values. For the full program I will change this to be an Enum.

I then decided to create an Enumeration for the type of game being played, as this would also be one of a set of values, with the values of: GAME_CHECKERS, GAME_CHESS, GAME_DRAUGHTS, GAME_GOMOKU and GAME_DAMA. I then created a Board class which would be used to store Pieces in a data structure. For the data structure I decided to use a two-dimensional array of Pieces as this has columns and rows like an actual board would, and its size is static once set, this is fine as the board will not change size once it is created. In order to create the correct sized Array for each game Type, I used a switch statement of GameType and set the size based on the GameType that was input. I went on to add methods to the Board class which allows for the position of pieces to be changed.

I used Junit test cases in order to test my methods as I developed them, ensuring that all tests were passed.

For testing purposes, I decided not to implement a turn system, however I designed the legality checking method so that it could easily be modified to allow move checking, by simply adding a variable that stores who’s turn it is and checking whether the move being made is by that colour, then after a legal move is complete, change the turn variable to the other colour.

I then added methods to the Board class to check the legality of moves before making it.

For the interface, the main option was between using JavaFX and Java Swing. I decided to use JavaFX over swing for many reasons. Firstly, I am a lot more familiar with utilising JavaFX over Swing, so by using JavaFX it would reduce development time, as I would not need to spend time learning a new library. Secondly, JavaFX is much more friendly for a MVC architecture pattern that I was trying to achieve. Thirdly, JavaFX is more suited for creating desktop applications than Swing, which is what I was hoping to produce.

I then worked on developing the JavaFX interface for interacting with the data structure in a user-friendly way. A small problem with the interface was that it was difficult to perform Junit testing on, however it was much easier to perform other testing on, as I was able to freely interact with the environment.

After I had completed the functionalities and interface to where I wanted them in this prototype, I adapted my program to allow concurrent behaviour, allowing the user to launch multiple different games of Checkers at the same time, instead of only being able to play one at a time.

Proof of Concept Server Chat App

Due to the nature of this prototype, it was difficult to produce it using a traditional TDD approach, as for the Client to work, the Server must first be created in working condition, and for the interface to fully work, the Client and Server must already be fully working. As such, I created this prototype with a more linear approach, which had testing after development followed by additional development to resolve any issues.

This prototype began with a lot of testing with ServerSockets and Sockets until I found an appropriate solution that met my needs. This prototype was based on a program by Almas Baimagambetov[21]. I mainly used this program to find an appropriate way to set up the Client and Server as I was struggling to set it up in a way such that it can be constantly checking for messages and constantly displaying received messages to an interface.

Concurrency-based Game Environment

After reviewing my proof of concepts, I decided to create my concurrency-based game environment using TDD, redesigning certain features to make them more efficient and fit programming standards better, based on what I had learnt previously. I once again aimed to follow a MVC architecture and stored all of my work on an online Git repository. I had a main design idea of what I wanted to achieve and worked to get there through spiral like TDD cycles. I also made use of JavaDoc to document and comment on my code and explain how it worked.

I used Junit for my TDD on the non-interface parts of my software. For the interface I performed manual TDD, where I performed a series of tests to ensure that the interface was working correctly. The Junit testing files that I used are also stored on my GitHub repository. They are also described in more detail in the Testing Procedures section of this document.

After analysing my proof of concepts and evaluating them, I felt as though there were some areas in which my program could be improved in order to make it more efficient, and fit programming standards better as well as making my program more readable and easier to follow. I found that if code is not readable or is poorly organised, it makes it much harder to work on. I also wanted to improve the user interface so that it looked nicer for the user, I did this by designing new icons for the main menu and pieces on the board.

I started by creating the classes and methods from my checkers proof of concept that did not need to be redesigned. I then, with the help of TDD began to program and test the methods that were being changed, which included the constructor of the Board class, which now contained code which would set up the checkers board within the constructor, rather than using a separate method. I then redesigned the other methods that had caused me trouble in my proof of concept.

In my original proof of concept, I struggled working with Arrays and updated then methods to work better with my newly gained knowledge about Array in Java.

Once I had most of my game logic sufficiently working I began to redesign my user interface. With my new interface I added a TextArea which could be used to display error messages to the user or to send and receive messages when online. This was separated from the main game interface which the user interacts with to make moves, by using a JavaFX HBox to separate the chat space from the game space. This meant that the interface was much easier to use and there was better user experience.

I also utilised GitHub's issue system to leave notes for myself about bugs that I had discovered and could work on at a later date. This was particularly useful as it meant that I would not forget specific bugs and would have details left about what is potentially causing the bug, when it was discovered and how I might be able to fix it.

Issues Encountered

Proof of Concept Checkers Board Game

During development, I encountered a few issues which I had to spend time resolving. The first issue that I encountered was due to my unfamiliarity with two-dimensional Arrays in Java. As I had not used two dimensional arrays before, I accidentally designed my board to be effectively sideways, using Array[y][x] instead of Array[x][y]. Now that I am much more familiar with 2d Arrays I would like to redesign the way the Piece array works for the final system, compared to how it was done in this prototype.

Another issue that I encountered; was with the way I initially designed my legality checking system. When I first designed the legality checking system, it only checked whether the piece had moved forwards by only one place by checking if the new Y position was one greater than the current Y position, however this meant that only one colour of moves was being accepted correctly. I had to change my method to first check the colour of the current Piece and then, based on the colour check if it has move 1 place up or down on the Y axis.

I encountered a few issues when I was adjusting my prototype to support concurrent behaviour. I had initially planned to use threads to provide the ability to play multiple games at once, however I found that it was much easier to simply create a new JavaFX window with each game being an instance of GameInterface. Within this interface, when a game was finished, a command was run to close the JavaFX window, which worked fine for a single game instance, but with multiple games being played at once, winning one game would close all games. I attempted to solve this issue by creating a thread for every new game window that would run that game and close the window once the game was finished, I could not get this solution to work. I instead found that with JavaFX, if the user has a Scene, you can get the window that the Scene is displayed on, so I was able to retrieve the Scene and then get the window and then close only that window, resolving this issue.

Proof of Concept Server Chat App

I encountered quite a few issues when developing this prototype, which led me to spending a much longer time on it than I had hoped. Before I came upon the final solution I used, I attempted to create a program which allowed the user to send messages between two different consoles. However, I overcomplicated the problem and designed a solution that was much more complex and difficult to produce than necessary.

Instead of simply having a Server which is hosted on one device and a Client that connects to the Server from a different device, I attempted to have the host device have both a Server and Client running and an additional Client that connects to the Server from the other device. The Server would receive messages sent out by each Client and relay them to the other Client. This largely and unnecessarily overcomplicated the software as it meant that multiple Clients would have to connect to the Server, and each Client that is connected to the Server must be stored so that the Server can read messages from, and relay messages to each of them.

Additionally, using the console to send and display messages meant that the program had to constantly be checking for a user input on the Console, while also constantly checking for received messages and printing them to the Console. This made the whole program a big mess and led to lots of confusion and wasted time.

After all this, I decide that the best course of action would be to restart on this prototype with a different approach. Instead of using two Clients, only one is required, as the Server can perform all the actions that are required for this program, such as sending and receiving messages, meaning a second Client was unnecessary. Instead, messages from the host would be sent directly from the Server to the connected Client.

Immediately using a JavaFX interface to send and display messages made it much easier, as JavaFX can create an area in which the user can type and wait until they press the Enter key to perform an action. This simplified the sending of messages as I could effectively say "When the enter key is pressed, copy the text from the text field and send it". This made producing the server chat app much more straightforward.

Concurrency-based Game Environment

The main issues that I encountered when working on my main concurrency-based board game software was due to poor scheduling and lack of awareness about how long specific tasks would take and how much time would need to be dedicated to work outside of the project. For example, I severely underestimated how difficult it would be to implement the multiplayer system and once I had completed it, I became very busy with other work and documentation which left me little time to polish, and bug fix my final product. This also meant that I was not able to complete all of the goals that I set out to achieve.

One issue that I particularly struggled with was having the Client-Server system properly interact with the interface. Initially I had planned for the NetworkConnection class to only be an attribute of GameInterface, and to modify the board array and have GameInterface update itself after a move was received. However, I discovered that it was very difficult to update the Interface after a move was made, as the best method would actually be for NetworkConnection to call a method on GameInterface that would update the UI. This meant that I had to pass the GameInterface object to its own attribute, using the "this" keyword in java. This then meant that the NetworkConnection class could call the updateBoard method to update the interface that is shown to the user, thus displaying the move made via a multiplayer connection. This took me a rather long time to figure out and required a lot of time and research.

As my main game was built off the knowledge from the previous two proof of concepts, I was able to avoid a number of issues that I had initially encountered, and was able to plan for future code later, better ensuring that parts of my program would work together when fully developed. Prototyping helped significantly, as it meant that I ran into problems and resolved them much earlier in the project timeline than I would have if I just started without prototyping.

Testing Procedures

The following section describes and explains my testing approach for each part of my program. More information about the testing procedures that I performed for each of my programs can be found in the appendices, showing the test cases that I performed for each program.

Proof of Concept Checkers Board Game

Early on, I used a form of TDD by creating a file that manually ran parts of my code, printing results to the console to ensure that my methods were working correctly, however this file was not committed to my VCS as I planned to delete it and replace it with Junit tests.

In order to test some parts of my system I used Junit test cases. For the Board class I had a BoardTest class which contains the following tests:

> testSetup which tests that the board has been setup correctly by checking three randomly selected positions, one of which should contain a white piece, on that should contain a black piece and one that should contain no piece after the initial board setup.
>
> testLegalMove which tests that isLegalMove returns the correct values by testing a legal move of a white piece, an illegal forward move of a white piece, a diagonal backwards move of a white piece, a diagonal move of more than one place and repeating these test for a black piece.

testMove which tests that makeMove is correctly moving pieces to their new positions, by checking the position where the piece is being moved to, to see if that piece has successfully moved there.

testLegalMoveKing which sets a piece to be a king and tests that moves in any direction for a king are legal as well as testing that double space moves are legal.

testMakeMoveKing which sets a piece to be a king and tests that legal moves are being made by the makeMove method.

Once I had developed the user interface it allowed me to perform unit testing on my system. The test cases that I performed can be found in Appendix A in the Appendices section of this report.

Proof of Concept Server Chat App

For this prototype I performed mainly testing different formats of messages to ensure that messages were sent correctly. As well as testing to see if error messages were displayed properly.

The test cases that I performed can be found in Appendix B in the Appendices section of this report.

Since this was a prototype with limited capabilities, there was not much else that could be tested with it.

Concurrency-based Game Environment

This program was produced using TDD cycles, where I used Junit testing to produce my game logic, this file can be found in the source code, labelled "BoardTest.java". I used GitHub's issues feature to log bugs that I encountered throughout development of this program and my prototypes, leaving comments on the issues with potential information on what was causing the bug and how it may be resolved. This helped significantly to solve issues, as I had a tracked record of each bug and how I resolved it.

BoardTest.java contains a series of tests which check to ensure that the Board class and the classes it uses are working correctly.

testSetup simply starts a checkers Board and checks three locations to see what pieces are located there. The first position should contain a white piece, the second position should contain a null piece and the third position should contain a black piece. This ensures that pieces are being set up int the correct location.

testSize creates a new Board of each game type, and checks the size of the Board for each game type, to ensure that the correct sized boardArray is made for each game.

testDiagonal starts a checkers Board and creates a piece for testing, it then performs three checks to ensure that the isMoveDiagonal method is working correctly, by checking two extremes and a false value.

testDistance starts a checkers Board and creates a piece for testing, it then checks to ensure that testDistance returns the correct value for a given move.

testCount starts a checkers Board and gets the count of each piece colour and ensures that 12 pieces are set up on the Board.

testTakesPiece starts a checkers Board and creates a piece in a take-able position, it then checks whether a move that should take this piece returns true from the takesPiece method and whether another move that should not take a piece returns false.

testIsLegal starts a checkers Board and selects two pieces, one white, one black and checks several potential moves using the isMoveLegal method, checking a legal move, an illegal forward move, and illegal backwards move, an illegal backwards move that lands onto another piece and an illegal forwards diagonal move of more than one place.

testMove tests that the Board is being correctly updated when a move is made, using makeMove, and to ensure that a piece is correctly removed when it is taken. It selects a piece and attempts to move it to a new, legal position, it then checks to see if the Board has been updated to show this move. It then selects another piece and attempts to make an illegal move, it then checks to ensure that the position of the piece on the Board has not been changed. It then creates a new piece in a position where it can take a piece, it performs the move to take the piece, checks that the position of the piece has been updated correctly and then checks to see if the piece that was taken has been removed from the Board. It then checks to ensure that the count for the colour of the piece that was taken has been decreased by one.

testTurns test the turn taking mechanism by attempting to make a white move before black has moved and checking that the position of the piece has not been changed, it then moves a black piece and checks that the black piece has been moved correctly, it then attempts to move another black piece on white’s turn and checks to make sure that the black piece has not been moved, it then moves a white piece and ensures that the position of that piece has been updated correctly.

Once the main game logic was completed, I began to use manual testing in order to develop the interface. The test cases that I performed can be found in the Appendices section of this document, with Appendix C representing the test for the Concurrency-based Game Environment.

Particularly Interesting Code

There were some parts of my program that I found particularly interesting when producing or that required intuitive ideas to work properly.

To create the multiplayer system, I had to modify the way that my NetworkConnection class worked. As the ConnectionThread, which was checking for messages would be running on a separate thread to the interface, it meant that it was difficult for it to directly interact with the interface, this meant that in order to have the NetworkConnection class properly interact with the interface, I needed to pass the GameInterface itself to the NetworkConnection that it created. JavaFX has a nifty method that allows the programmer to run something on the Interface thread from a separate thread, which meant I could run the updateBoard method from the NetworkConnection thread, when a move was made by the other user.

Code on the NetworkConnection class:

```
Platform.runLater(() -> {
      if (message.substring(0, 3).equals("MSG")) {
            messages.appendText("Opponent: "+message.substring(3)+"\n");
      }
      if (message.substring(0, 3).equals("MOV")) {
            String move = message.substring(3);
            String[] moveArray = move.split(":");
      // Moves should be sent in the format "MOVpieceX:pieceY:newX:newY"
            int pieceX = Integer.parseInt(moveArray[0]);
            int pieceY = Integer.parseInt(moveArray[1]);
            int newX = Integer.parseInt(moveArray[2]);
            int newY = Integer.parseInt(moveArray[3]);
            Piece movePiece = gameBoard.getPiece(pieceX, pieceY);
            gameBoard.makeMove(newX, newY, movePiece);
            gameUI.updateBoard(gameBoard);
      }
});
```

I also provided the ability to both send messages and moves between the two online users by differentiating a move and a message using a three-character string which is added before the data is sent. For a message the text “MSG” is added, for a move the text “MOV” is added. The data for a move is then separated by a symbol “:”.

```
int pieceX = selectedPiece.getxPos();
int pieceY = selectedPiece.getyPos();
      gameBoard.makeMove(newTile.getBoardX(), newTile.getBoardY(),
      selectedPiece);
      updateBoard(gameBoard);
      if(connected) {
            try {
                  connection.send("MOV"+pieceX+":"+pieceY+":"+newTile.getBo
                  ardX()+":"+newTile.getBoardY());
            }
```

The code for sending a move is shown above.

```
String message = inputField.getText();
inputField.clear();
chat.appendText("You: "+message+"\n");
try {
	connection.send("MSG"+message);
}
```

The code for sending a message in chat is shown above.

Some other code that I thought was particularly interesting was the system for checking whether a move is legal or not and the methods involved in that system. I originally was using a complex method to check whether a given position is diagonal to another, which involved comparing how much each X and Y value was increased or decreased to see if the position was diagonal, when after some research I discovered that a method used with multi-dimensional matrices to tell if two points are diagonally aligned[16] can be applied to my game.

```
public boolean isMoveDiagonal(int newX, int newY, Piece currentPiece) {
	return Math.abs(newY - currentPiece.getyPos()) == Math.abs(newX -
	currentPiece.getxPos());
}
```

This method compares the absolute value, (which is the total non-negative value of a real number), of the new Y position minus the current Y position and compares it to the absolute value of the new X position minus the current X position. If the two values are equal, then the new position and current position must be diagonal to each other.

I also discovered that another mathematical formulae can be used to get the distance between two positions, this formulae is known as Taxicab or Manhattan distance[26].

```
public int moveDistance(int newX, int newY, Piece currentPiece) {
	return Math.max(Math.abs(newY - currentPiece.getyPos()),
	Math.abs(newX - currentPiece.getxPos()));
}
```

This takes the maximum value between the absolute of the new Y minus the current Y and the new X and current X position. Whichever is the greatest between the two is equal to the distance between the two positions on the board.

Running the Software

Information about how to run the of the programs can be found in the readme.txt file alongside this document.

This is a link to a short demo of the concurrency-based board game working:
https://www.youtube.com/watch?v=S1D_tsSmpTk

Project Evaluation

Overall, I feel that the software I developed was fairly, but not entirely successful. I was able to achieve multiple of the goals that I set out at the start, however I also failed to achieve some of the goals that I had initially hoped to complete. Aside from the standpoint of completing the goals that I set out however, I feel as though I have been very successful in this project. I feel as though I have been able to clearly follow a good project development process: I performed detailed research and produced a detailed plan for my project, organising deadlines for my deliverables and reports; I successfully completed multiple proof of concept prototypes, following good engineering processes and delivered them on time; I learned new skills and theory throughout the project; I developed technical reports detailing the results of my research, as well as evaluating the prototypes that I produced. I feel as though I was able to successfully follow the process of an actual project and understand what it takes to complete a project.

From a software standpoint, I developed a program that allows the user to play multiple games of checkers concurrently, as well as being able to play multiple online sessions with others at the same time. There is also a chat function that allows the user to send messages to their opponent who they are playing online, as well as acting as a space for error reporting. I feel as though I followed a good software engineering process and was able to implement the skills that I learned from previous courses to produce software that fits programming standards well, as well as being well documented. I feel as though I used tools around me such as GitHub and Eclipse to the best of their abilities.

I initially hoped to produce multiple different board games that the user would be able to play, however I was only able to produce a checkers board game. This was mainly due to poor scheduling. Originally, in the second term, I had planned to work on redesigning my system so that it works more efficiently and smoothly, and then develop the multiplayer system. Once the multiplayer system was working, I would begin to add more games to the program. I misestimate the amount of time it would take me to redesign my system and to develop the multiplayer system for my game. Additionally, I failed to correctly account for time that would need to be spent on other work outside of the project such as other coursework. This meant that I did not leave myself enough time to complete these goals. I believe I would have been more successful if I gave myself a stricter schedule with solid deadlines for when features should have been completed, in the second term of the project.

If I was to continue this project, I would like to increase the scope of the program by implementing all the board games that I had initially planned for, and to polish the program further so that it is more intuitive for uses to play. I would do this by performing user experience tests to see what can be improved within the software to make it easier for people to use, based on their feedback. This should be easily achievable, as I specifically designed my software with the intention of more games being easily added later in development.

I have learnt many things about doing a project, but I feel that based on my experience during this year, one of the most important factors when working on a project is solid planning and scheduling, as well as being able to accurately determine how long each task should take and therefore how much time should be set aside for that task. By following a structured plan, with a clear schedule, it makes it easier to know exactly what you should be working on, when and at what time you should aim to be finished by. I believe this where my main failures in my project stemmed from and I feel as though if I was better in this facet my project as a whole would have improved.

Project Diary

The following is my project diary, detailing information about what I had been working on each week during the first term.

October 8, 2021

Worked on producing report on game environments, analysing different game environments, and how they are relevant to my project.

October 17, 2021

This week I worked on creating the data structure for storing information about and the positions of pieces on the board. As well as providing movement for pieces on the board by changing the index in the data structure.

October 24, 2021

This week I worked on only allowing legal moves of checkers pieces, testing functionality, as well as starting to produce the user interface for playing the game. I am currently slightly behind schedule, as I would liked to have already completed most of the user interface, which will then allow me to better test my system by having a visual representation of how the functionality is working.

October 28, 2021

This week I worked on completing the user interface of my game, so that the user can interact with it, as well as the functionality for taking pieces during moves.

November 10, 2021

The past week I worked on adding concurrency to my game, allowing the user to play multiple games at the same time without them interfering with each other. My program is now able to play many different games of checkers at the same time.

November 24, 2021

The past week I have worked on produce the report on my prototype implementation, detailing the design and functionalities, the software engineering process I undertook, any issues I encountered and how I solved them and my testing procedures.

February 14, 2022 (Talking about January 17th to February 12th)

The past few weeks I have been working on using servers in Java and experimenting with how I can implement them into my game effectively.

February 14, 2022

This week I have worked on redesigning and refactoring my game.

February 20, 2022

The past week I have continued to redesign my game as well as updating my report to match with the work that I have done.

February 27, 2022

This week I have completed redesigning my game, and am now in a position where I can start to implement the online multiplayer system into my game. I have also been working on restructuring my final report based on advice from my supervisor.

March 6, 2022

This week I have completed implementation of the multiplayer system into my game. It is now capable of playing the same game across multiple devices, however it still needs some ironing out and bug fixing.

March 14, 2022

This week I have been working on improving my final report, to make it ready for a final review before submission.

Below is my personal programming diary, detailing my process as I created my prototype, I created entries during development each day that I worked on programming.

October 13, 2021

Today I setup my Github repository with my Eclipse project. Made the Type and Piece classes, the Pieces will need to be stored in a data-structure. Likely will use a 2-D array. Will need to get the size of the board for each game to create the array as it is a static data-structure.

October 14, 2021

Created the Board class which has a 2-D array to store Pieces. Created the GameType Enum for storing the type of game that is being played. Modified Piece to add setter methods for the Type and Position as these will change throughout the game. Created a method in Board which sets up positions of the white pieces, based on the starting position of a game of checkers. Using a temporary class to print out the array and manually checking to see if the positions are correct (should setup junit tests). Encountered an issue with the location of white pieces being wrong, caused by the axis of the 2-D array being reversed, instead of X,Y, it's Y,X. Solved the issue by changing the variables around and also discovered an issue with the wrong sign being used on the if statement, should be "<" not ">=". Now that it works, I can simplify it and use the same theory to setup black pieces.

October 15, 2021

Simplified the setup method into a single if statement and added a part that sets up the position of black pieces. I manually tested this method and it appears to work as intended. Created a basic method that can be extended to check if a move is legal or not. Currently this method only checks that the new position is empty. Also added a method that checks if a move is legal and if it is, updates the position of the piece in the array and changes it's x and y attributes using the setter methods. I need to extend the legality checking method to check if a move is diagonal and forwards for men and diagonal in any direction for a king piece.

October 16, 2021

Added switch statement to move legality checking method to allow rules for separate games to be input. Added Javadoc to methods. Flipped x and y values around for the makeMove method as they were the wrong way around. Need to modify legality check to allow only diagonal moves, this could be done by checking if the x value has only decreased or increased by one and the y has only increased by one.

October 17, 2021

Added if statements to check if the move is diagonal or not, if the piece is a king it is able to move two places diagonally. Added a check at the end of the makeMove method that checks if the moved piece is now in a promotion position and changes its type if it is.

October 20, 2021

Added Junit tests for the Board class and discovered an issue with black pieces, not being considered legal, as the legality check method only checks if the y has increased by one, *which would only occur if the piece was white, a black piece's y would decrease by one if it is moving forwards. Modified testLegalMove to check the colour of the piece before checking the y variable. Added tests and found that forwards moves are not being considered illegal moves, I will likely need to redesign my legality checking to ensure that only diagonal moves are considered illegal.*

October 21, 2021

After doing some research[16], I found that it was easier to create a method which checks if two positions are diagonal to each other by checking if the absolute value of the first x position minus the second x position is equal to the absolute value of the first y position minus the second y position. Implemented this into the code. Also added a method to check distance between two positions by returning the greater absolute number between the first y minus the second y and the first x minus the second x. These allowed me to resolve the issue I was facing.

October 22, 2021

Created a main menu interface, with options for different kinds of games. Small quality of life issue with the play checkers button being highlighted in blue as if it has just been clicked when the program is run.

October 24, 2021

Designed icons to represent each type of piece in the checkers game and added them into the Eclipse project.

October 25, 2021

Added a start game button to the main menu and added functionality to the buttons, changing the game type when clicked, when the start button is clicked, it will create a board of that game type. Added a getter method for the size of the board, as it will be needed to iteratively create the pieces on the interface.

October 26, 2021

Added a method in piece to return the image corresponding to the type and colour of the piece. Created a tile class which will be used to create the board on the interface, by iteratively creating tiles in each position.

October 27, 2021

Created UI board using iteration to create tiles. Created a method that updates the positions of pieces on the UI. Discovered an issue, when any piece reaches a promotion zone, it is upgraded to a white king piece, regardless of its colour. Solved this issue by changing URLs for king pieces to the correct URLs (both king piece URLs were for a white piece). Redesigned the king pieces icon to make it more clear that it is a king piece. Completed main game logic.

November 1, 2021

Attempted to add concurrent behaviour, using threads but found that it can easily be done by simply creating a new stage of the game.

November 9, 2021

Discovered an issue with the concurrent behaviour, when any game is completed, all windows are closed. Only the window of the game that has been won should be closed. Attempted to solve this issue by using a thread that would constantly check the amount of white and black pieces left, if there were none, it would close the window of the game that has been completed, however I couldn't get this to work. Attempted to solve this by closing the window inside of the GameInterface class, however I could not close it as it was inside a handle method for a tile. Was able to solve this issue by getting the window, inside of the GameInterface class by getting the scene from a node that was accessible inside the handle method (the tile itself) and closing the window that the tile is on.

November 11, 2021

Made it so that if the user clicks the middle mouse button on a tile, that instance of the game will close, for testing and presentation purposes.

Professional Issues

Source code plagiarism is when somebody copies or reproduces source code without the written permission of the original creator. This is most commonly, in the case of people adapting other people's code minimally, perhaps only changing variable names or including fragments of other's code within their own code, without the original author's consent, or without credit [22]. In software development, it is commonplace to use other people's code in the form of libraries, open-source projects, and frameworks, but it is important, as a programmer to know when using others code is considered plagiarism and the impacts of plagiarising code. Plagiarism is particularly common in classroom work and work of students, especially collusion, which is when multiple students cooperate on work that is supposed to be done individually, sharing their solutions with each other, or working together on the solution [23]. When caught any students involved in collusion may be faced with severe penalties, including having their work completely failed.

Plagiarism is a concern in programming because, as a programmer, it is your responsibility to ensure the stability, reliability, and security of software, but when you simply copy other’s work, you are putting that responsibility onto somebody else, rather than yourself. Additionally, if you copy somebody else’s code to solve a problem, you may not understand the actual solution. This may lead to you become a programmer who cannot solve problems, instead relying on the work of others to fix things. As a programmer it is key that you can resolve issues and break down problems and program a solution.

It is important that you are clear in your program as to what work is your own and what work belongs to others. It is also show clearly how you have modified the code and to state the source of the code and explicit permission that has been given to you to use the code. The best practice to avoid plagiarism [24] is to comment on your code, to not only show what work is yours and what work is from others, but to show a clear understanding of the code that you are using and how it works and to clearly separate your own work from copied work by citing the source of the copied work and stating explicitly in your code what is copied.

The issue of plagiarism has arisen for me personally in my project, as I have used code from open sources in order to complete some parts of my program, particularly the Server and Client aspect of my program uses code from another programmer. To ensure that I was allowed to use the code, I had to check the License of the code to understand what conditions need to be met for me to use the code in my own project. The code that I used had an MIT License [25], which means that the software is “granted to be used free of charge … without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software” under the condition that the copyright notice and permission notice are included in the Software. This meant that I was freely allowed to use the code in my program as long as I cited my source and showed this notice in my software.

To show my use of the software, I clearly labelled the parts of my code that were sourced from elsewhere and explained how they worked within my code comments and JavaDoc. Also, instead of just directly copying code, I modified parts of the code to better suit it’s use in my program.

Bibliography

1. The Rules of Chess on sakkpaalota.hu – This will be the ruleset used to develop the chess game in the full system, it gives a basic overview of each piece and how they move as well as any special circumstances that need to be accounted for, such as pawn promotion, en passant and castling.
2. The Rules of Draughts on masterofgames.com – This provides the ruleset and instructions for playing both Draughts and Polish Draughts that will be used to develop the Checkers and Polish Draughts part of the program.
3. Game Theory of Gomoku on gomokuworld.com – This gives an overview of the rules of Gomoku or Five in a Row, as well as showing reference images for certain rules, this will be used to develop the Gomoku part of the program.
4. Turkish draughts - Turkish draughts game rules – This gives the ruleset for dama or Turkish Checkers.
5. Thread (Java Platform SE 7) on docs.oracle.com – This is documentation about Java threads and how to use them, this is necessary knowledge for providing concurrent capabilities.
6. Socket (Java Platform SE 7) on docs.oracle.com – Documentation about creating a Client to connect to a server and sending and receiving messages to and from the server.
7. ServerSocket (Java Platform SE 7) on docs.oracle.com – Documentation about creating a server for a Client to connect to and sending and receiving messages to and from the client.
8. 247 Checkers – The checkers game that I used for analysis in my report on game environments
9. Chess.com - Play Chess Online - Free Games – The chess game that I used for analysis in my report on game environments
10. FileInputStream (Java Platform SE 7) on docs.oracle.com – Java documentation about FileInputStreams, which I used to learn how to utilise them in my program.
11. FileNotFoundException (Java Platform SE 7) on docs.oracle.com – Java documentation showing the exception thrown when the image file cannot be found.
12. Rectangle (JavaFX 8) on docs.oracle.com – Documentation on the rectangle class that the Tile class is extending
13. Scene (JavaFX 8) on docs.oracle.com – Documentation on the JavaFX scene class
14. EventHandler (JavaFX 8) on docs.oracle.com – Documentation about JavaFX EventHandler
15. Application (JavaFX 8) on docs.oracle.com – Documentation on the JavaFX application class which is used to run the JavaFX application thread and allows for displaying of Windows and interfaces with JavaFX
16. How to know if two points are diagonally aligned? on Mathematics Stack Exchange – This is the theory that my diagonal checking method is based on.
17. TextArea (JavaFX 8) on docs.oracle.com – Documentation about JavaFX TextArea which is used to display messages that are sent by the Client and Server to the user.

18. DataOutputStream (Java Platform SE 7) on docs.oracle.com – Java documentation about DataOutputStream which is used to write UTF messages to the Client or Server.
19. DataInputStream (Java Platform SE 7) on docs.oracle.com – Java documentation about DataInputStream which is used to read UTF messages sent from the Client or Server.
20. TextField (JavaFX 8) on docs.oracle.com – Documentation about JavaFX TextField which allows the user to input messages to be sent to the Client or Server.
21. JavaFX Chat Application by Almas Baimagambetov – This is the chat app that I based my Client – Server system on.
22. Combatting code plagiarism in computer science | Codequiry – Definition of source code plagiarism and additional information about plagiarism.
23. Plagiarism and Programming: How to Code Without Plagiarizing | Turnitin – Information about collusion and solutions to plagiarism in programming.
24. Plagiarism and Programming: How to Code without Plagiarizing | IT Briefcase – Information about how to best avoid plagiarism when coding.
25. License of the JavaFX Chat Application by Almas Baimagambetov – This is the MIT License for the code that I used in my program.
26. Manhattan Distance – The source of the formulae used to determine the distance between two positions on an axis.

Appendices

Appendix A: Test Cases of Checkers Prototype

Test No.	Action	Expected Result	Actual Result
1	Launch program from MainMenu class.	The Main Menu interface should pop up with a window showing text asking the user to select a game, with 5 buttons and a start game button.	As expected
2	Select start game without having chosen a game	Error message should pop up telling the user to select a game	As expected
3	Attempt to select a game other than checkers then select start game	Same as 2	As expected
4	Select checkers from the menu and then select start game	A checkers board should pop up in a new window	As expected
5	Repeat test 4	Another checkers board should pop up in a separate window	As expected
6	Click a tile without having selected a piece	A notification should pop up telling the user that they must select a piece before making a move	As expected

7	Select a piece and attempt to move it directly forwards	A notification should pop up telling the user that the move they attempted is illegal	As expected
8	Select a piece and attempt to move it directly backwards	Same as 7	As expected
9	Select a piece and attempt to move it diagonally forwards more than one place	Same as 7	As expected
10	Select a piece and attempt to move it diagonally forwards one place	The selected piece should move to the selected position without interfering with the other game window	As expected
11	Select a piece and attempt to move it diagonally forwards one place to a position that contains a piece already	Same as 7	As expected
12	Select a piece and attempt to move it diagonally backwards to an empty space	Same as 7	As expected
13	Select a piece and perform a legal move that takes a piece	The selected piece should move to the selected position and the piece that is being taken should be removed from the board	As expected
14	Select a piece and perform a legal move that leaves the piece in a promotion position	The selected piece’s image should be updated to show a king icon on the piece	As expected
15	Select the King piece and repeat test 7	Same as 7	As expected
16	Select the King piece and repeat test 8	Same as 7	As expected
17	Select the King piece and attempt to move it diagonally forwards 2 places	Same as 10	As expected
18	Select the King piece and repeat test 10	Same as 10	As expected
19	Select the King piece and repeat test 12	Sama as 10	As expected
20	Select the King piece and repeat test 13	Same as 13	As expected
21	Perform a move that takes the final piece of the white side	The white piece that is taken is removed from the board and a notification is displayed telling the user that the black side has won, only the game that has been won is closed	As expected
22	Perform a move that takes the final piece of the black side	The black piece that is taken is removed from the board and a	As expected

		notification is displayed telling the user that the white side has won, only the game that has been won is closed	

Appendix B: Test Cases of Server Prototype

Test No.	Action	Expected Result	Actual Result
1	Try to send a message from the Server with no Client connected	An error message should be displayed saying that the program failed to send the message	As expected
2	Try to send a message from the Client with no Server running	An error message should be displayed saying that the program failed to send the message	As expected
3	Attempt to send a message from the Server while a Client is connected	The message should be displayed on both the Server and Client interface	As expected
4	Attempt to send a message from the Client while it is connected to the Server	The message should be displayed on both the Server and Client interface	As expected

Appendix C: Test Cases of Concurrency-based Game Environment

Test No.	Action	Expected Result	Actual Result
1	Launch program from MainMenu class.	The Main Menu interface should pop up with a window showing text asking the user to select a game, with 5 buttons and a start game button.	As expected
2	Select start game without having chosen a game	Error message should pop up telling the user to select a game	As expected
3	Attempt to select a game other that is greyed out and not available (e.g. chess) then select start game	Same as 2	As expected
4	Select checkers from the menu and then select start game	A menu should pop up asking the user if they would like to play locally or online	As expected
5	Select local from the menu	A game interface with a checkers board, and a chat interface should appear	As expected
6	Repeat tests 4 and 5	Another checkers board should pop up in a separate window	As expected

7	Select a white piece by clicking on it	A notification should appear in the chat, telling the user that it is not White's turn, and that Black should make a move	As expected
8	Select a black piece by clicking on it and select a new, legal tile to move it to	The piece should move to the selected tile and the interface should update to show this position change, without interfering with the other game	As expected
9	Attempt to select another black piece by clicking on it	A notification should appear in the chat, telling the user that it is now White's turn, and that Black cannot make a move	As expected
10	Select a white piece and attempt to make an illegal move	A notification should appear in the chat telling the user that the move they attempted was illegal	As expected
11	Make a move that takes a piece	The piece that is being taken should be removed from the interface and should not be selectable, the piece that took the piece should be moved to its new position	As expected
12	Make a move that leaves a piece in a promotion position	The piece that is in the promotion position should be updated to have a crown icon on it, showing that it is now a king piece	As expected
13	Attempt to move the king piece in a legal, backwards move	The king piece should be moved to its new position as a king piece can move backwards	As expected
14	Type into the text field underneath the chat and press enter	The text that was typed into the text field should be removed and displayed into the chat box	As expected
15	Perform a move that wins the game	A notification should pop up telling the user which colour won and that the game will close, when they press ok, only the game that has been won will close, any other games that are incomplete will still be playable	As expected
16	Repeat test 4 and select online from the menu	A menu should pop up, asking the user if they would like to host or join a game, with an option to go back.	As expected
17	Select back from the menu	The user should be returned to the interface to select between online and local	As expected

18	Repeat test 16 Select to host a game	The checkers interface should pop up with a message displaying the IP address of the game to the user in the chat	As expected
19	Repeat test 1 and 16 on a separate device, which is connected to the same network as the device hosting the game, select join a game	The user should be taken to a menu, asking them to input a server address with a button to join or cancel	As expected
20	Select cancel	The user should be taken back to the previous menu	As expected
21	Once again select join a game, copy the IP Address from the chat into the text field and select join server	A checkers interface should pop up on the screen and the user should be connected to the other user	As expected
22	Attempt to send a message between users by typing into the text field and pressing enter	The text should be displayed on the chat for both users	As expected
23	Attempt to make a legal move to start the game	Both interfaces should be updated to show the move has been made	As expected
24	Make a move that wins the game	A notification should pop up to both users saying who has won the game and that the game will close, both games should successfully close	As expected