CS 246 Final Project Design Document

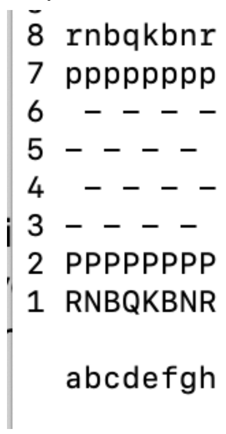
Team Members: Diana Chung, Hugh Chung, Daniel Ephrat
d42chung@uwaterloo.ca, hy5chung@uwaterloo.ca , dephrat@uwaterloo.ca

Overview:

Our group completed the Chess project using a combination of the MVC and the Strategy design pattern. Our program lets the users play against each other, either on the same computer or over a LAN. Alternatively, they can play against computer players, with a range of difficulties (1-4). Level 1 computer will favour any random legal moves, Level 2 computer will favour capturing moves and checks, Level 3 computer will prefer avoiding being captured, capturing moves, and checks, which have all been the standard instructions required for the project. Level 4 computer is the combination of Level 2 and 3 (avoids captures that result in a negative trade) while also preferring moves towards the center of the grid and is capable of checkmate.
Fun fact: You can also make the computers play against each other!
The MVC (Model-View-Controller) design pattern has enabled the group members to complete the implementation of the game Chess with minimal changes that affect the entire program. The TextDisplay is responsible for displaying the game in the command line/terminal interface and the GraphicsDisplay gives a more colorful and easier display of the game in a new window. Shared pointers for TextDisplay and GraphicsDisplay, which both inherits from the Observers class, has been initialized in the init function in chessmodule.cc so that the chess grid and each individual chess pieces can be shown and updated every time the user or a computer makes a new move by notifying the changes to the internal game state.
With the added bonus features, both the TextDisplay and GraphicsDisplay will automatically show up to all respective players (unless specified otherwise), and also supports terminals on different devices within the school server (for access to Xlib).

TextDisplay:

- TextDisplay is inheriting from the Observers Class. Therefore, whenever there is a change of movement from the pieces, the chess graphics will be updated to the most current positions for all chess pieces

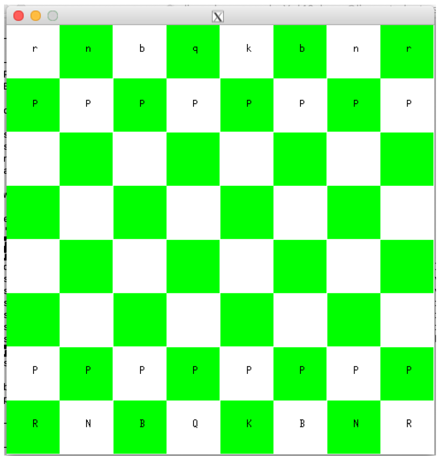


```
8 rnbqkbnr
7 pppppppp
6  - - - -
5 - - - -
4  - - - -
3 - - - -
2 PPPPPPPP
1 RNBQKBNR

  abcdefgh
```

GraphicsDisplay:

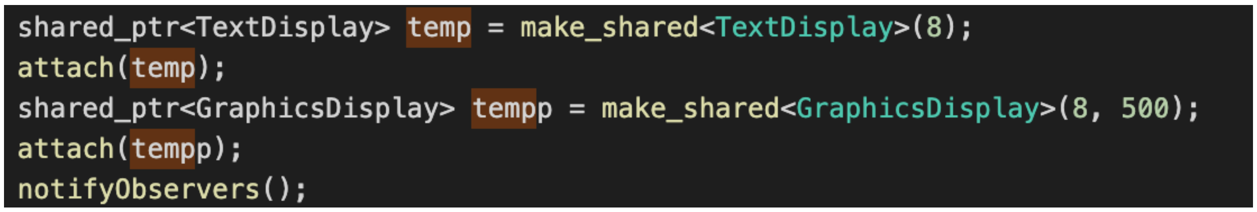
- Similar to how the TextDisplay has been implemented, GraphicsDisplay is inheriting from the Observers Class. Therefore, whenever there is a change of movement from the pieces, the chess graphics will be updated to the most current positions for all chess pieces
- The FillRectangle() and drawString() methods from the Xlib library were used (coupled)
- The windows.h/cc files that handle the mechanics of getting the graphics to display has been imported from the A4Q4 graphics-demo (default windows size has been set to 500 x 500)



How the Display works:
In the init function from chessmodule.cc, there is a shared pointer ‘temp’ which points to a TextDisplay and initializes int n = 8, and another shared pointer ‘tempp’ which points to a GraphicsDisplay and initializes int n = 8 and int size = 500.

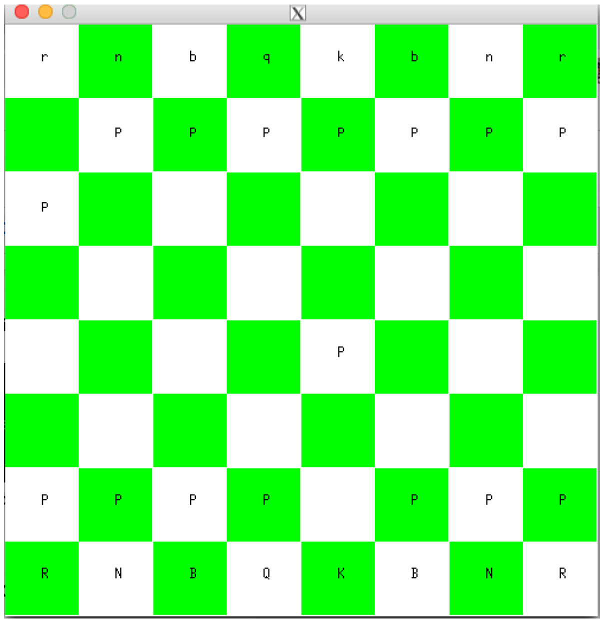
- n represents the number of rows and columns each
- size represents the default windows size (row and column)

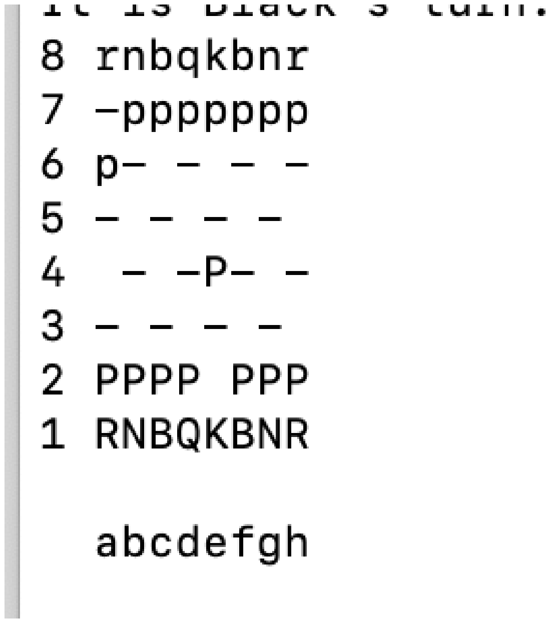
- notifyObservers() is then called to reset the grid with the changed location of the chess pieces



```
shared_ptr<TextDisplay> temp = make_shared<TextDisplay>(8);
attach(temp);
shared_ptr<GraphicsDisplay> tempp = make_shared<GraphicsDisplay>(8, 500);
attach(tempp);
notifyObservers();
```

- This is an example when the white piece has moved from e2 to e4, and black piece has been moved from a7 to a6 :





```
It is Black's turn.
8 rnbqkbnr
7 -ppppppp
6 p- - - -
5 - - - -
4  - -P- -
3 - - - -
2 PPPP PPP
1 RNBQKBNR

  abcdefgh
```

Design:

How did we handle the rules of chess?
We created a logic module called “ChessModule”. This logic module stores all the rules for the pieces, and decides whether a certain move is legal or illegal, and uses these rules to decide how/whether to move a piece. This logic module also stores the locations of the pieces, as well as each piece’s information (among other features)

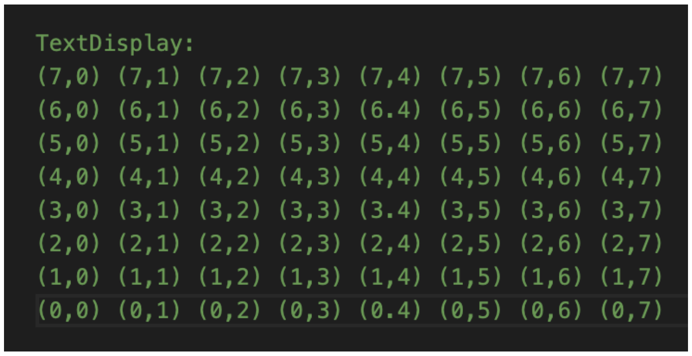
How did we handle user input?
We created a command harness using main.cc which captured the user’s input and communicated it to the ChessModule as appropriate. Main.cc was also responsible for ensuring the input was provided correctly.

What challenges did we face implementing the TextDisplay class?

- Initially, the TextDisplay class has been called Display.
- However, when GraphicsDisplay (which is coupled with XLib) was implemented, an error specifying the obscurity of “Display” would occur when the program ran, even though the only Display that was used was for the text display
- We hypothesized there was another class called Display within the XLib we coupled with the text display module, so proceeded to change the name of the class to TextDisplay for clarity, which solved the problem

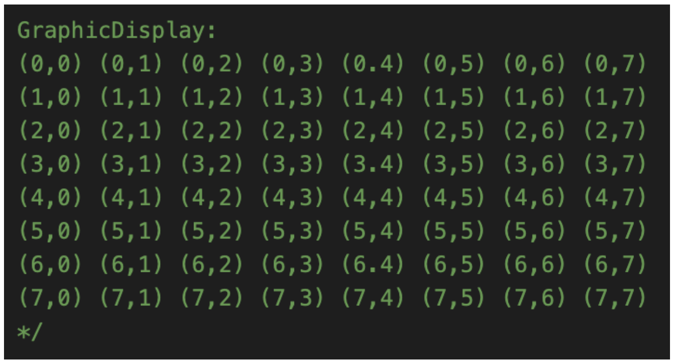
What about for the GraphicsDisplay class?

- In TextDisplay::notify, we have defined the rows and columns such that the bottom left corner would be initialized as (0,0) like the following:



```
TextDisplay:
(7,0) (7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7)
(6,0) (6,1) (6,2) (6,3) (6.4) (6,5) (6,6) (6,7)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7)
(3,0) (3,1) (3,2) (3,3) (3.4) (3,5) (3,6) (3,7)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)
(0,0) (0,1) (0,2) (0,3) (0.4) (0,5) (0,6) (0,7)
```

- When the GraphicsDisplay::notify was implemented, due to how Xlib defined the (x,y) co-ordinates, even with the same nested for loop structure, the (0,0) was automatically defined in the top left corner like the following:



```
GraphicDisplay:
(0,0) (0,1) (0,2) (0,3) (0.4) (0,5) (0,6) (0,7)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7)
(3,0) (3,1) (3,2) (3,3) (3.4) (3,5) (3,6) (3,7)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7)
(6,0) (6,1) (6,2) (6,3) (6.4) (6,5) (6,6) (6,7)
(7,0) (7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7)
*/
```

- Due to this discrepancy, even though most of the code for GraphicsDisplay was a direct copy of TextDisplay, the black and white grid and the order of the rows were inverted.
- The variable i had to be changed to (7-i) for correct implementation

<u>How it differed from your design on Due Date 1 (if it did)?</u>
Nothing significant has been revised from our design on due date 1. We added a few more critical functions and attributes to ensure that the programme runs properly and consistently.

Resilience to Change:

In order to accommodate change to existing features, our group implemented this project using the MVC (Model-View-Controller) design pattern, with different sections of our code controlling different parts of the program. The benefits of using the MVC design pattern the way did are primarily that we can make changes to individual components of the code without impacting the others. The separation of code into different modules reduces coupling, and also reduces the likelihood that new code will introduce bugs into our program.

The ChessModule serves as the ‘Model’, and controls the game state. This includes the current state of the chess board, the locations of pieces and their information, and the rules behind the moves each piece on the board was allowed to make (among other features). The ChessModule was also a subject for the Text and GraphicsDisplays, which were notified whenever the internal game state was modified. Within ChessModule, the individual functions of maintaining and modifying the internal game state (moving pieces, creating a new game, declaring checkmate, etc) were separated and abstracted into different functions themselves, which allowed for more flexibility and use in a wide variety of situations.

The TextDisplay and GraphicsDisplay serve as the ‘View’, and is responsible for displaying the internal game state to the user in the form of a chess board. This was divided into a textual component (displayed in the command line/terminal interface) and a graphical display (opened in a new window). These displays used the Observer design pattern to observe the ChessModule, and as such were notified upon any changes to the internal game state.

The main.cc function serves as the ‘Controller’, and handles the communication of information from the user to the internal game state (i.e. ChessModule). Main.cc is in charge of all user input, and is responsible for enabling command-line input and input flags. Examples of the inputs flags include -seed xxx and -text, and examples for the command-line input include ‘game human human’, ‘move e2 e4’, ‘resign’, etc. (See Demo document for more details)

The Observer design pattern built-in to the displays and ChessModule allows the Displays to display the necessary information to the user(s) without needing to access ChessModule’s code. Similarly, ChessModule doesn’t rely on the displays in any meaningful way. Thus, ChessModule and the displays don’t rely on each other, so if one breaks, the other will remain functional, making our program more resilient to change. Separating main.cc from the ChessModule and the displays similarly reduces the coupling between different parts of our code.

The ChessStrategy class implements the Strategy design pattern, and is inherited by the different difficulties of computer players (Level1 - Level4). Each computer overrides the ‘findBestMove(...)’ method and implements it in a different way, depending on its level. The Strategy design pattern allows for the creation of an unlimited number of unique computer players, each with their own strategies and methods of picking their next move. This allows us to make as many individual changes as we want to any given computer player without impacting the others, while the inheritance structure allows us to add global features to all the computer players without needing to modify each one.

One of the main benefits of using our design pattern is the low coupling built into the program. Each module is largely disconnected from all the others, which provides the following benefits:

- Reduced likelihood that bugs introduced by adding/changing code to any one module will impact the others
- Identifying the portion of code that requires change to support new features is much easier
- New bugs are more easily identified based on the effect they have on the end-user experience, as opposed to manually looking through all the code to find a bug.

The separation of our program into meaningfully abstracted modules is an example of Object-Oriented Programming principles, and greatly increases the cohesion of our modules. Each module has a specific function, which makes adding new features significantly simpler. To add a new input command, modify main.cc. If the player is allowed to make illegal moves, fix ChessModule. If the board isn't being displayed properly, examine the appropriate display, based on which one is being buggy. If a computer player is playing at a level too low (or too high!), check that level's subclass and consider modifying its functions. Whatever the problem may be, our low coupling and high cohesion allows us to quickly and easily identify the source of the problem, as well as the area of code in which changes would need to be made to implement new features.
For an example of how our design patterns make new features easier to implement, see the answer to the third question from the project specification (particularly the last paragraph).

Answers to Questions:

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Answer:
We shall save all of the opening moves in a derived class of our main Chessmodule. Each move will be a vector of strings containing the ordered move; we would then construct another command, call it *Opening*. The user will then enter the command if they wished to use opening moves when they first began the game, for example: After *game human human* then *Opening RuyLopez.*
Then it will move in accordance with the opening move vector from the first to the last move contained in the vector.

Difference between the first and second approaches:

- Instead of making a file called "opening_book.txt" in which each line includes the name of an opening. We decide to keep the initial moves in a derived class. As a result, accessing the opening moves will take less time and less space.

Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Answer:
(This has already been implemented) We made another vector and saved it as a private attribute in our Chessmodule. Its aim is to keep track of all the moves made during the game. When a player needs to reverse his or her previous move, we will launch a new chess game and immediately give move commands using the vector until the very last moves.
We can repeat this process unlimited times to reverse unlimited moves. (Also implemented)

Difference between the first and second approaches:
Instead of having different variables to remember all the moves that have been made scattered throughout the code, they have been stored inside one standard vector for efficiency.

Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

Answer:
Fortunately, the design pattern of our program allows us to support such changes easily.
Let us consider the changes to be made in terms of the changes we would make to each component of our program.

- ChessModule: We would need to change the shape and initial construction of the board, as well as some rule changes as to where certain pieces can go. Beyond that, most of the rules would stay the same, with some minor modifications to the number of players and the way turns are handled. Checkmate would also be different, since the game doesn’t actually end until either a multi-way stalemate, or all players but one are checkmated. (The actual rules of 4-player chess are more complex, but this sufficiently illustrates the scope of the necessary changes)

- Main.cc: Commands would largely be the same, with the exception of how games are started, the number of players to consider, and the effects of resigning. (One player resigning wouldn’t necessarily end the game immediately anymore)
- Displays: The shape of the boards would have to be modified to match those of 4-player chess. Otherwise, no changes.
- ChessStrategy: The superclass would stay the same; the individual computer levels would stay the same as well, since they primarily look for checks, captures, and checkmates. No rule changes would need to be made here, since those rules are handled by the ChessModule.
- The details of the answer we gave to this question in the plan of attack are similar to the details we’ve given here. However, there is now a significantly more clear idea of how those changes should be structured/organized, and which code module should be modified to accommodate these changes. (ChessModule, displays, main.cc, ChessStrategy/Level[1-4])

Extra Credit Features:

- Savelog
  - The savelog lets the user save a local human vs. human game once the game ends, and then lets them load the game back in.
  - The main challenge associated with this feature was how to handle the special cases concerning resigning, checkmating, and various other ways to end the game, since ending the game generated the savelog. Fortunately, the command interpreter was very sophisticated, so we checked all local human vs. human cases where the game could end and created the savelog for each instance. Loading the savelog was implemented by treating each line as an input command, which would then bring the user back to where they had left off.
- Reverse:
  - We allowed the user to undo their moves in a local human vs. human game.
  - The key problem associated with this function was determining how to catch the move position when playing the game. We allow this by inserting move strings into a vector along with movePiece functions. When you call the reverse function, the game can restart and use the caught move to track back to the previous move.
  - As mentioned before in the *Answers to Questions* section:
    - We made another vector and saved it as a private attribute in our Chessmodule. Its aim is to keep track of all the moves made during the game. When a player needs to reverse his or her previous move, we will launch a new chess game and immediately give move commands using the vector until the very last moves.
      We can repeat this process unlimited times to reverse unlimited moves. (Also implemented).
- Level 4 advanced computer:
  - Created a computer player that is capable of checkmate, prefers captures to checks, avoids trades that would result in a decrease in material relative to the opponent and is biased towards moving pieces to the center of the board for a positional advantage. This was challenging because when playing real chess, the conditions for checkmate are often intuitive, which makes communicating the requirements to the computer challenging. An additional challenge came from the sheer volume of possible moves that can be made at any given point in the game, which vastly increased the complexity of the problem
  - There were two barriers to implementation: Recognizing the opportunity for checkmate, and calculating the outcome of a “trade”. (A trade is when you capture a piece, and then the opponent recaptures your piece)
  - To recognize the opportunity for checkmate, we first generated a list of all legal moves for the player in the given situation. Then we temporarily played each move, and if that move would checkmate the other player, then play the move. Otherwise, perform other calculations before moving on to the next legal move.
  - To calculate the outcome of a “trade”, we first generated a list of all legal moves. Then for each move, we generated a list of all legal moves for the opponent once

our move had been made. If any of those moves allowed the opponent to recapture our piece, then our program would evaluate whether the trade was worthwhile (This was done by introducing “points” to pieces, based on tournament piece values). If the trade was strongly negative, then the computer would avoid making that move.

- LAN (Local Area Network) Mode:
    - Implemented with TCP socket programming in a client-server environment. (Figure 1)

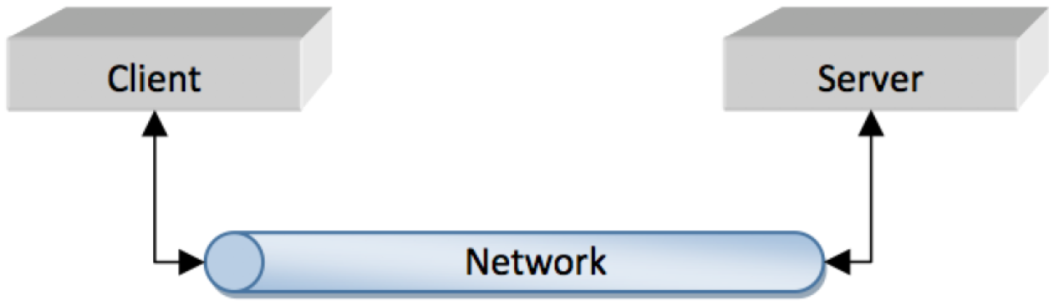


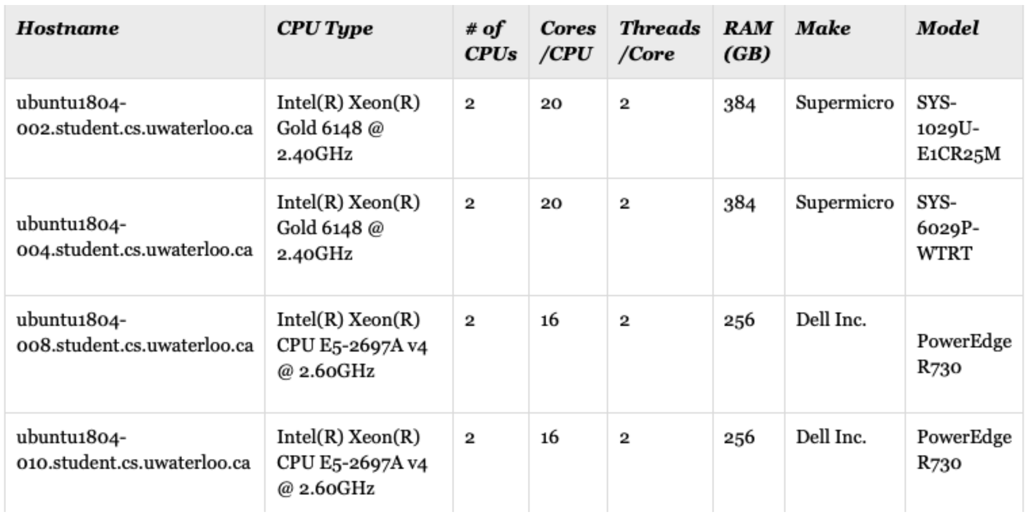
FIGURE 1

- The demo document demonstrated how it works. Here, we will talk about its design.
- OOP design structure:
    - Both server and client are aggregated to our core chess module, but composite to the main.cc module (i.e. our UI/test harness).
- Communication:
    - Stage 1: (Setup) (Main Connection, persistent TCP connection)
        - Both players agree to play a game of chess. One of the players (i.e. host/server) would then collect his IPv4 address and inform the other player.
        - In this stage, one player (i.e., the host/server) will open a server socket on the local open TCP port, which will be referred to as the main communication port. The client (the other player) then creates a TCP socket with the host IPv4 address and port number (where the server is listening). The client will automatically send a handshake message to the host/server after connecting to it. A small welcome message (i.e. handshake message) will then be displayed on both players' interfaces (client and host/server). To inform the user that they are connected with the correct person.

- Stage 2: (Game starts)
  - Both server and client will start their own chess game. They are pitted against a bot. The bot is unable to move a piece on its own. It only moves when it receives the move command from the main connection. Then move the pieces accordingly.
  - In this state, the server and the client communicate solely via the main connection. In the main connection, only two commands would be allowed to communicate:
    - move xx xx:
      - From server to client:
        - The server sends the move command to the client's bot, who is currently playing with the client. The client's bot will move the piece as required.
      - From client to server:
        - The client sends the move command to the server's bot, who is currently playing with the server. The server's bot will move the piece as required.
    - resign:
      - From server to client:
        - The server sends the resign command to the client's bot, who is currently playing with the client. The client's bot then resigns and the program will close all the socket and port and return from the main function.
      - From client to server:
        - The client sends the resign command to the server's bot, who is currently playing with the server. The server's bot then resigns and the program will close all the socket and port and return from the main function.

- End of Communication:
    - Both players (the client or the host/server) can end the process by either resign or finish the game. (i.e. checkmate/stalemate)

- Testing:
    - We already tested the LAN features of the game. As shown below, in the school servers:



Hostname	***CPU Type***	***# of CPUs***	***Cores /CPU***	***Threads /Core***	***RAM (GB)***	***Make***	***Model***
ubuntu1804-002.student.cs.uwaterloo.ca	Intel(R) Xeon(R) Gold 6148 @ 2.40GHz	2	20	2	384	Supermicro	SYS-1029U-E1CR25M
ubuntu1804-004.student.cs.uwaterloo.ca	Intel(R) Xeon(R) Gold 6148 @ 2.40GHz	2	20	2	384	Supermicro	SYS-6029P-WTRT
ubuntu1804-008.student.cs.uwaterloo.ca	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	2	16	2	256	Dell Inc.	PowerEdge R730
ubuntu1804-010.student.cs.uwaterloo.ca	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	2	16	2	256	Dell Inc.	PowerEdge R730

- All four servers have different IPv4 addresses. The game is playable among all above servers.

Final Questions:

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?
    - As a team, this project has taught us that delegating tasks and working together for integration is much more effective than trying to have each person fully understand and implement each component. For example, when one group member was implementing the setup() function, it was much faster for them to write out the high-level details and leave the low-level coding to another group member, who had already written the code necessary to make setup() functional. Another lesson we learned is to not underestimate the time to learn a new technology. None of the group members have done A4Q4, so no one had a clear idea on how to start on the graphics. We expected the graphics display to only take a marginally greater amount of time than the text display, as the logic and implementation would be similar, but learning how to start the graphics from the scratch took up significantly more time than we thought because the XLib and the windows.h/cc had to be read and understood before the implementation.

2. What would you have done differently if you had the chance to start over
    - We would have spent more time on planning and designing. Currently, the chess pieces are in a grid of Green and White with the chess pieces showing up as text. Since graphics was implemented in the end, there was very limited time to study the Xlib library to see if we could’ve enhanced some of the graphics features. For example, instead of having text to represent the chess pieces, we could’ve implemented actual images with more sophisticated colours as the background chess grids.