Programming Report

Timothy Runhaar (s2509148)
Simon Wong Kee Choong (s2422557)

Requirements

Crucial Requirements

- A standard game can be played on both client and server in conjunction with the reference server and client, respectively.
  - Our team's CollectoClient allows a user to play on the reference server.
  - When launching CollectoClient or CollectoComputerClient, user gets asked if he wants to play on the reference server by simply getting boolean from input.
  - When we run CollectoServer we are asked for our name and then the port in which we want to create the server.
  - We put that same IP and port in 2 reference client instances and they can play against each other indefinitely.

- The client can play as a human player, controlled by the user.
  - When user runs CollectoClient the user has to indicate on what server he wants to play
  - After successfully joining the server and logging in he can queue up
  - When he joins a game all commands are displayed to him
  - When it's his turn to make a move, a message "Your turn, make a move will appear"

- The client can play as a computer player, controlled by AI.
  - We have two Client classes, one to play yourself and one to let the AI make the moves for you.
  - When you run CollectoComputerClient you get to choose 1 of the 3 strategies.
  - CollectoComputerClient extends CollectoClient and overrides the determineMove() method called each time it's client's turn.
  - When its client's turn to play, move is sent automatically thanks to AI.

- At any point in time, multiple games can be running independently and simultaneously on the server.
    - Games are started as Threads when thread QueueChecker detects 2 clients(clienthandlers) in the server's queue, and launches a game with these 2 clients as parameters.
    - We have tried this by connecting 4 reference clients to the server and successfully having 2 games running at the same time.

Important Requirements

- When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.
    - When we run CollectoServer, port is asked.
    - If we could not create serverSocket at that port then the user gets asked if he wants to try again. We then ask for the port number again.

- When the client is started, it should ask the user for the IP-address and port number of the server to connect to.
    - When we run CollectoClient, the user gets asked in the main method if he wants to join the UT reference server. If boolean is false, then we ask for server ip:port that client wants to connect to.
    - We then start the CollectoClient with ip and port as parameters
    - Client will try to connect to that server, if unsuccessful, user gets asked if he wants to try again

- When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.
    - When a client is in a game he can enter “hint” in the console. This will display all valid single moves or valid double moves.

- The AI difficulty can be adjusted by the user via the TUI.
    - A user can choose from one of the 3 strategies when they launch the CollectoComputerClient.

- All of the game rules are handled perfectly on both client and server in conjunction with the reference server and client, respectively.
  - After extensive trials we have found the client and server to always display the same board after each move.

- Whenever a game has finished (except when the server is disconnected), a new game can be played without needing to establish a new connection in between.
  - When a game is over, a client can type in “queue” to join the server's queue and play again.

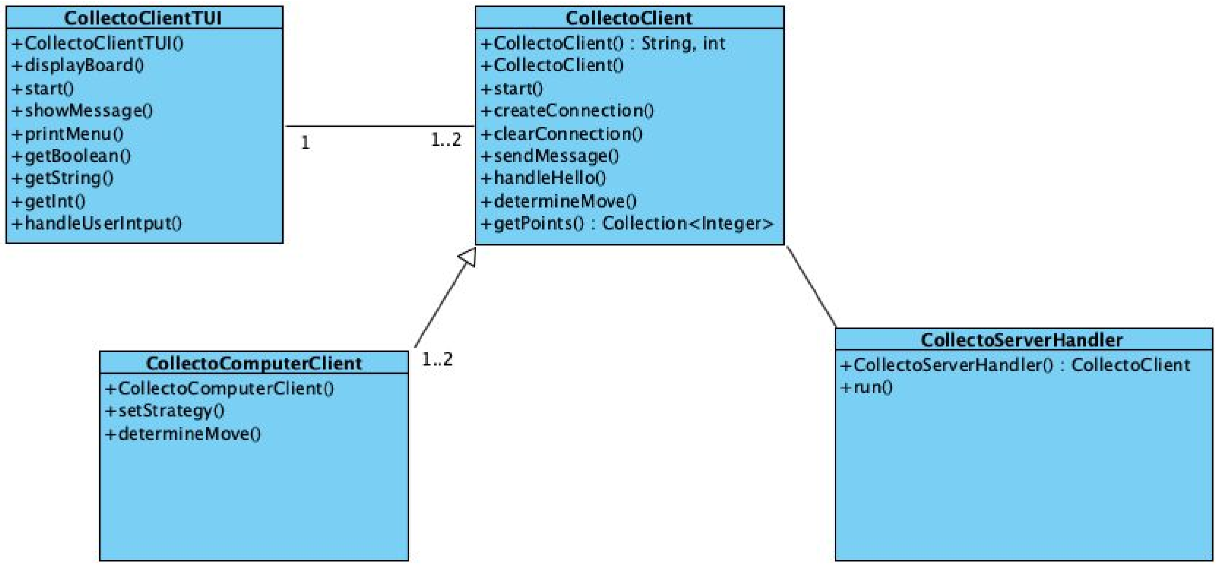
- All communication outside of playing a game, such as handshakes and feature negotiation, works on both client and server in conjunction with the reference server and client, respectively.
  - On our server, when a client connects he first has to send us a Hello, and he has to successfully login in order to be able to play. This is done automatically.
  - On our client, the HELLO message is sent automatically. When the handshake is successful, the user is asked to login with whatever username he wants.
  - When username not taken, server sends “LOGIN” to client, else “ALREADYLOGGEDIN”
  - If the client is already logged in, an error message is sent by the server with the username he is logged in with.

- Whenever a client loses connection to a server, the client should gracefully terminate.
  - When a client loses connection with the server, “Disconnected from server” and “Closing the connection…” displayed and client is terminated.

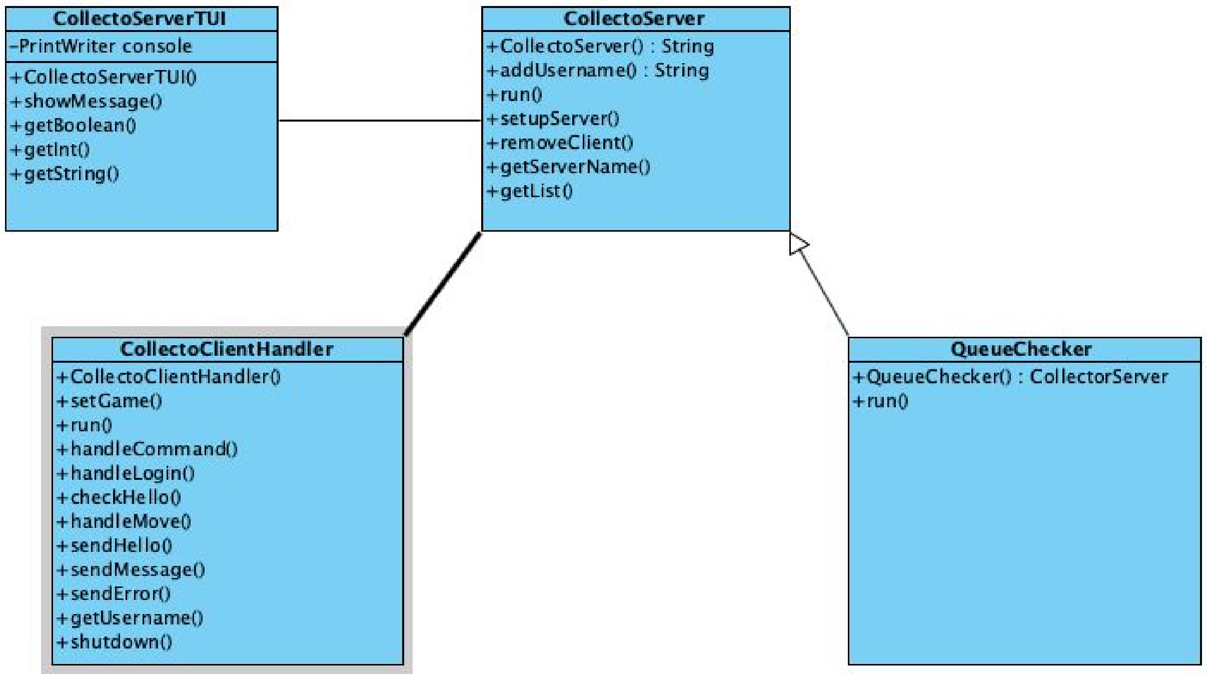
- Whenever a client disconnects during a game, the server should inform the other clients and end the game, allowing the other player to start a new game.
  - When a client is in a game and he disconnects, the disconnects variable in that game is set to that client(the one who disconnected). This will end the game and send GAMEOVER~DISCONNECT~”OTHERCLIENT” to the other client, telling him he won.

Realised Design

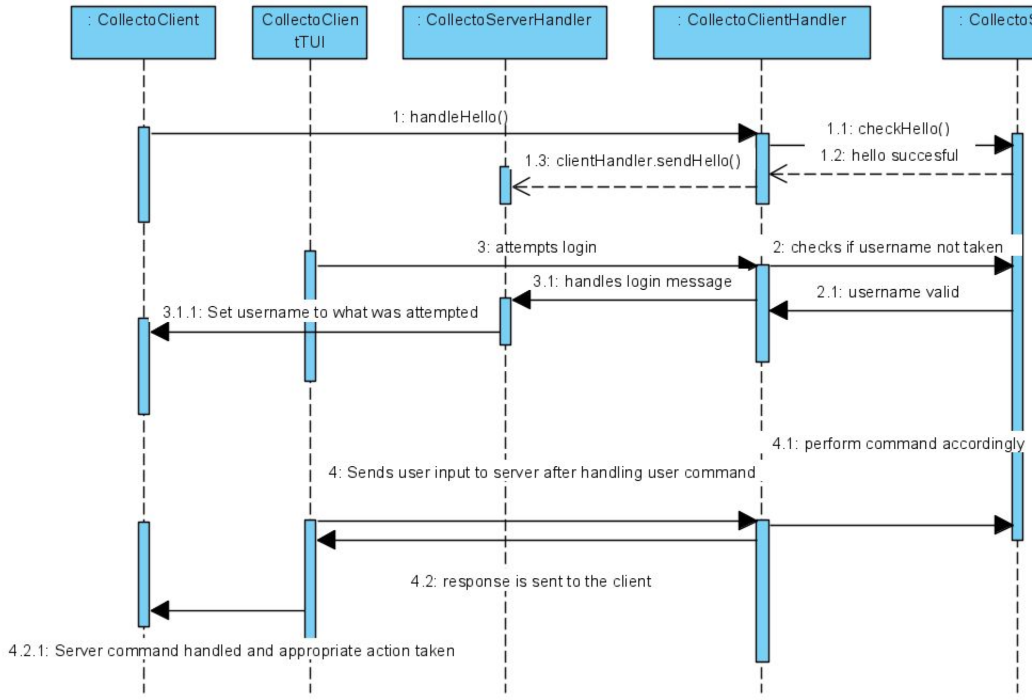
In order to segregate the classes for ease of viewing and editing, we have 5 main packages: Model, Controller, View, Testing and Utilities. The Model package contains our Board.java and Game.java, these two classes will work together with the client and server when it is called the appropriate time. The Controller package handles all the user inputs and interactions with the game, it consists of: AI, client, server, exceptions, and player. This package is essential as it deals with the connection between the client and server, the display of each TUI as well as the different strategies built. Our Collecto testing package contains tests for both the client and server (integration testing), and for the board itself(unit testing). Here we test the various features of our client and server class and if they interact with each other correctly. The Utilities package simply contains the TextIO class, used in both the client and server to get user input from the Console.



Class Diagram Representation of the CollectoClient



Class Diagram representing the CollectoServer


: CollectoClient
CollectoClientTUI
: CollectoServerHandler
: CollectoClientHandler
: CollectoServer
1: handleHello()
1.1: checkHello()
1.2: hello succesful
1.3: clientHandler.sendHello()
3: attempts login
2: checks if username not taken
3.1: handles login message
2.1: username valid
3.1.1: Set username to what was attempted
4.1: perform command accordingly
4: Sends user input to server after handling user command
4.2: response is sent to the client
4.2.1: Server command handled and appropriate action taken

Sequence Diagram to represent the Client-Server Communication

Our design

In our initial design we listed the various functions that our Collecto game should have. Functions such as how our threads would work with each other, the client-server architecture, difficulty of game play, synchronizations of threads and displaying the game itself. We created a class to illustrate our hypothesis on how the organization of our packages and classes should look, it indicated the basic relationships between each class and the multiplicities between them. At first we had an oversimplified class design and structure. Because we almost didn't have any experience working with servers and clients without peers, we changed the way we designed the project a couple of times at the start.
Our Server has a QueueChecker, a TUI, and creates a clienthandler for each Client socket that connects to the server. The QueueChecker checks if the queue has 2 clients and starts a new Game thread on the server. The game takes 2 clienthandlers as parameters and sets that game as the game variable of the 2 clienthandlers.
This allows us to call move from within the clienthandler on the game that is now not to to null anymore. That call is then handled by the Game thread, giving out appropriate responses to said clienthandler.

Our client takes ip and port as parameters. When start() is called, hello handshake is verified, and login is asked. Then serverhandler thread is started in and tui

loop is started. In order to handle server messages and user input at the same time. We keep track of whose turn it is during the game thanks to the local boolean variable mymove. When a move is done, that move is sent to the client's local board, and determineMove is called.
DetermineMove only prints messages asking to move to TUI. But when we have a ComputerClient, determineMove actually tries to determine a move thanks to the strategy we have set and the local board, which is always in sync with the server's board.

Reflection on Initial Design

Our Initial design was very very unsatisfactory. It was way too simple and was in all truth based on a simple game like the local tic tac toe game we had coded previously in the module. So the most important part of the design, the client and server, were almost ignored. This caused us to be in kind of a stalemate after having coded the Board, because that's when we needed to start implementing the client-server part of things.
If we had to restart again we would definitely put more into making a better and more complete initial design. As this project has clearly demonstrated to us the value a good design brings to the implementation of code.

Concurrency Mechanism

Once a connection has been established between the client and server, a Collecto game is created. When the CollectoServer class is started, it prompts the user to input a username. A thread in the server class is launched and a message (CollectoServer by “username”) is displayed on the TUI. When the thread starts it requests for a port number and will create a socket at that port. If this does not succeed, the process starts again.
In the CollectoServer class, when a server socket is started, it creates a new thread implementing the runnable class queue checker and checks every 2 seconds if there are 2 clients in the server queue. If there are two clients, a game thread is created with the clients as arguments
In the client the tui handles user input and the serverhandler thread handles server messages in concurrency.

System Tests

Testing Board:

- Test if starting board is valid
- Test starting validsinglemoves
- Test valid single moves
- Test valid double moves
- Test creation of game via NEWGAME message

Integration testing:

Testing the client and server:

- Assert that the client class can create a connection with the server
- Assert that a Hello handshake is performed not needed because it is implemented in the client and server to do handshake automatically. If the handshake was not successful, the client or server cannot continue.
- Assert that after a player logs in, and that his username is added to usernames list after successful login.

Overall Testing Strategy

Our Testing Strategy was mainly based on a lot of trials and testing every possible situation. Whether that was the server disconnecting, the client disconnecting, the server receiving invalid messages, or untimely messages, or the same for the client. W We had to test a lot manually because we did not manage to appropriately use JUnit for integration testing. But I can now say that our client and server handle unexpected commands well.
They both take appropriate measures when needed, as we were unable to find a situation where the server or the client got stuck or got out of sync with each other.
As for the actual JUnit tests we have tested the creation of a board, and the reliability of its validmoves methods.
We test that when a board is created with a list of fields made from a NEWGAME message that the fields all correspond to the ones sent.

Then we succeeded in doing a small integration test by launching one server and one client. We test if connections are made and then we test if login successful, thanks to the usernames list on the server.

Reflection on process

Timothy Runhaar
At the start of this project I really did not feel like I had the competence to complete it. I was quite confused and frustrated because I didn't really know where to start and had a lot of questions on how to design it but more importantly how to design it the best way possible. So it took quite a while for us to really start working on it, just out of inexperience is projects of this magnitude. But after hours and hours of coding, a lot of debugging, and immense trial and errors. I managed to completely grasp the concept of a Client-Server architecture. And now looking back on the project I think to myself why was I so scared to take it on. But I am immensely happy that we succeeded in making a functional server and a functional client. Now the only thing that I regret not having to properly make is the tests. Which we simply did not know how to integrate on a client server architecture. But overall I would say we did a good job, and I hope you can agree with me, that in the end we present to you a good project.

Simon Wong Kee Choong

In our initial planning, we had a brief overview of what our Collecto game should look like. Timothy and I created a list of requirements we needed to complete the game along with a class diagram to illustrate how the different files would interact with each other. We began our project by creating the game logic and strategies followed by building the client-server connection. Reflecting back, the game reveal was in week 6 before the holidays and we only had 2 and a half weeks to build the game. Since the networking part was the toughest to understand and to work on, Timothy and I could have prioritized the client-server classes in the beginning before working on the game logic.