## Code review

The code works as intended and has an average readability. What I did notice is that the more I coded the messier it got in terms of readability, mostly because the code is less consists and not as clean and well documented as the beginning. Since, I focussed more on the functionality of the game and code in this limited time, the readability and efficiency of the code took a hit. I preferred adding some functionality and optimizing my AI, over cleaning up my code. I can imagine that reading the code now can be difficult for another person to fully understand. However, if I would have spend a bit more time on the project I could improve this. To be more concrete, in certain parts of the code there are functions with a bit of overlap or weird if statements. In addition, some parts of the code are not very optimized. For example, I implemented the repetition rule (where two boards in a game cannot be the same) simply by storing the board and going through them one by one to see if a board is similar. This is obviously not very optimal and efficient. Different strategies, such as looking at the score and mapping that to certain board or looking at similar move only, would increase the efficient by a lot. However, this is something I decide not to spend to much time on, especially since it will not make a huge difference for this use case. Overall, I am still quite pleased with the result given the time spent. However, next time I might want to spend a bit more time on cleaning up the code and finalizing it. Coding/design wise I do not think I would do thinks that much different. I already did a project just like it, which I felt like gave me a small advantages over the others in terms of design choices. Although, I could spent more time on thinking decisions over and writing out the design, which could help me in the long run. Also, I would probably incorporate more exceptions in my classes which could improve my error handling. Since we did not decide on strict error handling in the protocol (just an error notification), I did not use exception that well. In general, I would catch general exceptions which give back an error that something went wrong. For future projects it would be better to give more relevant feedback by creating/catching better exceptions.

Furthermore, for my AI I used the ChatGPT API to try to get some good moves from a LLM. This decision was based on two main principles. The first is, that I wanted to test the integration of ChatGPT in application and see how it works and performs. Secondly, I do not know the game well enough to say with confidence what a good move is, this also made validating the performance of the AI model difficult. Looking back on this decision I am glad that I implemented it, since I think getting a taste of using LLM and implementing these sort of things are of great importance of my learning goals. However, the performance of the model was not as good as I initially expected it to be. In order for the AI to work, there were two main challenges (besides getting a paid API key). The first one was getting it give back an integer for the index. The responses from ChatGPT could differ quite a lot given the same prompt. So I create a function that looks for an integer in a long string, if it would not give back a single number. And I created a loop with a counter of five cycles to ask ChatGPT again if it would give me an indication of a good move with the index, if it was not able to do so in the first place. If none of these steps worked, I would choose a move out the list of valid moves. Finally, I tested AIs (with different prompts) and let different play against each other to have an indication of the best prompt to use.

## Development process

Another challenge we encounter was with the protocol. This was a group challenge, since we needed to decide on a way to communicate with each other. This took some meetings and trial and error to work towards a final protocol. I think we all learned how difficult it can be to create an airtight protocol for computers to communicate with each other.

For the testing part of the project, I used units tests for the Board and Game class. Which I tested right after implementing the classes. Since I wanted to make sure that when I continue with other classes, the issues can not come from these “core” classes. For the other classes (mostly Server and Client related), I went to test them when working on them and also afterwards when we tested to protocol with each other. In addition I performed two small test runs where I would take certain logical flows (before and during a game), but also with wrong input to make sure that it would handle it well. For the client side I did not write a test, but I did test it myself. It is however less important, since I constructed the client in such a way that it basically can send what ever it wants and it is the server’s job to handle it well.

## Design discussion

In general the process of creating/designing Go went relatively smooth. I did not spend to much time on the design part, since I already had an idea on how to design a game (TicTacToe and Collecto). I started with two important classes, the board class and the game class. I decided that the board class should contain all the logic of the game. It should place/get stones, check if moves are valid, calculate scores and capture the ‘groups’. The game class contains functions to keep track of the state of the game, which player has which stone, whose turn it is, what is the current score, is the game over. These two classes do have some auxiliary classes/enumerators, such as Move, Groups, Stone and Player. Next to the game, the server and client needed to be implemented. Here I used the general structure that was used in the university exercises. A server has multiple client-handlers (one for each client), every client handler has one server connection which listen on/send to a socket. For the client, every client has a client connection which has the same function as the server connection. The server has a list of games and client handlers, the client handler communicates the input from the user to the client and back. For the client side I created a GoTUI to connect to a game, login, queue, play a move, or when the stones are not looking too good… resign.

For the communication protocol the entire group decided that we take the protocol we already used for the university as a starting point. We only needed to fit it to the current game and make some decisions on how much and what information we would share over the connection. The main discussion that occurred during this meeting was with regards to the move-protocol. The discussion was about if it was enough to only send the move statement over the connection, or if there should have been some way of making clear who’s turn it is. I thought it was fine to only send the move and let the client side take care of keeping track who’s move it was, however I was in the minority. Also, since the other would not know how to easily implement it, it would help them a lot to incorporate it, which is another reason why we ended up with the ‘MAKE MOVE’-protocol. This protocol is send from the server to all client handlers from a game to emphasize that it is their turn. Another small detail is how we used two integers to do a move. It would have been easier to only use one integer which represents the index, however we opted to make our protocol compatible with either one or two integers for a one and two dimensional approach.

For every main class I created a test class which tested (almost) all methods, including the methods in the auxiliary classes. The goal is to have a high coverage, in addition to testing many different ways on how a function could possible respond in different scenarios. For the server and client testing I used some run testing to see if the expected results would come back, and if it could handle wrong output without crashing and providing useful messages.

## Run tests

TEST1 (Logging in and queuing):

Client1->Server: Connecting

Server->Client1: HELLO~To log in, please respond with: LOGIN~<username>

Client1->Server: LOGIN~Kasper

Server->Client1: ACCEPTED~Kasper

Client2->Server: Connecting

Server->Client2: HELLO~To log in, please respond with: LOGIN~<username>

Client2->Server: LOGIN~Kasper

Server->Client2: REJECTED~Kasper

Client2->Server: QUEUE

Server->Client2: ERROR~ You need to login before you can queue.

Client2->Server: LOGIN~Opponent

Server->Client2: ACCEPTED~ Opponent

Client2->Server: QUEUE

Server->Client2: QUEUED

Client2->Server: QUEUE

Server->Client2: "You are removed from the queue, to queue again type: QUEUE"

TEST2 (Start Game, do moves, winner):

[After client1 already connected and client 2 successfully logged in]

Client2->Server: QUEUE

Server->Client1: GAME STARTED~username(Client1), username(Client2)~13(boardsize default)

Server->Client2: GAME STARTED~username(Client1), username(Client2)~13(boardsize default)

Server->Client1: MAKE MOVE

Client1->Server: MOVE~1~1

Server->Client1: MOVE~14~BLACK

Server->Client2: MOVE~14~BLACK

Server->Client2: MAKE MOVE

Client2->Server: MOVE~15

Server->Client1: MOVE~15~WHITE

Server->Client2: MOVE~15~ WHITE

Server->Client1: MAKE MOVE

Client1->Server: MOVE~k

Server->Client1: ERROR~ Something went wrong, make sure to follow the protocol.

Client1->Server: MOVE~0

Server->Client1: MOVE~0~BLACK

Server->Client2: MOVE~0~BLACK

Server->Client2: MAKE MOVE

Client1->Server: MOVE~40

Server->Client1: ERROR~Not your turn

Client2->Server: PASS

Server->Client1: PASS~WHITE

Server->Client2: PASS~WHITE

Server->Client1: MAKE MOVE

Client2->Server: QUEUE

Server->Client2: ERROR~ You cant queue when you are in a game.

Client1->Server: PASS

Server->Client1: PASS~BLACK

Server->Client2: PASS~BLACK

Server->Client1: GAME OVER~Winner username(client1) (BLACK) with a score of: 2 versus 1

Server->Client2: GAME OVER~Winner username(client1) (BLACK) with a score of: 2 versus 1