The online Collecto Game

This Document is written by:
Vo Nhat Minh (s2503042)
Jesse Hermjan Snoijer (s2572362)

Date: 22/01/2021

Table of contents

Introduction

For this project, we created an online version of a board game. This game is the game Collecto. This is a German game that works with balls that you can move in directions. When 2 balls are next to each other the balls can be removed from the board and be added to your collection. A move can only be made when there are balls to collect after performing the move. When no single move is possible, a double move may be made. When there is no double move possible, the game has ended. Then for every 3 balls of the same color, you gain a point. The winner is the person with the most points. Is it a draw, then the winner is the one with the most balls collected. If this is also the same, the game ended in a draw.

We started by creating a plan and making a class diagram and 2 sequence diagrams. While coding we saw we forgot a few things in our design so we needed to change some things in our design.

For this project, we first created the game to play on 1 computer. Then we created a handler that is able to connect to a server for the Collecto Game that uses the protocol that the teacher provided. In that case, multiple users are able to play the game online against each other. Then we created our own server to run the game on. Finally, we created a bot with a naive strategy and a smart strategy that is able to play the game for us.

Justification of minimal requirements

The crucial requirements:

1. **A standard game can be played on both client and server in conjunction with the reference server and client, respectively.**
   When the client is started, the user is asked to play the game online or offline. If you type “y” you will have to type the IP address and a port number to connect to the server, otherwise, you can play the game offline on your computer. After type the information of the server, CollectoClient will automatically do Hello to establish the connection with the server.
   To start the server, you only need to type the port number. When there is a new client trying to connect to the server, it will be shown in the server's TUI.
2. **The client can play as a human player, controlled by the user.**
   Every time at the first of a player’s turn. TUI will ask the user “Do you want the bot to play the game? (Type y/n for Bot/Human option)“ => Type “n”/”no” to choose human player
3. **The client can play as a computer player, controlled by AI.**
   Every time at the first of a player’s turn. TUI will ask the user “Do you want bot to play the game? (Type y/n for Bot/Human option)“ => Type “y”/”yes” to choose computer player.
4. **At any point in time, multiple games can be running independently and simultaneously on the server.**
   When a new CollectoClient connects to the CollectoServer, the server starts a CollectoClientHandler on a different thread, because of this, multiple things can be done independently for each client.
   When a client joins the queue with command “QUEUE”, CollectoServer.queue will add its CollectoClientHandler.
   When the number of client is greater than 2, then the queue will remove 2 first CollectoClientHandler in the queue and spawn a CollectoServerGame between them (their starting turn will be decided randomly)
   The reference memory location (shallow copy) of CollectoServerGame and both CollectoClientHandler will be saved by each other (CollectoServerGame will have the memory location of both handler and handler also has one of the game). Therefore, multiple games can exist at the same time.

The important requirements:

1. **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 the server is started, it will enter a while loop that asks for a port-number (“Please enter the server port?”) (to make sure the user types correctly) and tries to create a socket on it. If this is successful it will exit the while loop and wait for clients to connect. Else it will notify the user that the port is already in use and asks if the user wants to try again. If the user does not want to try again the program is terminated. Else the user will repeat the previous process until it exits or finds an available port.
2. **When the client is started, it should ask the user for the IP-address and port number of the server to connect to.**
   We made our client in such a way it will first ask if the user wants to play the game online or offline. In offline mode a game is started without connecting to the server. If the user choses online the program will enter a while loop (to make sure the user type correctly) that asks for an IP address (“Enter the IP or host to connect to ?”) and a port number(“What is the portNumber?”). Then it will try to create a connection. If this is successful the program will exit the while loop and automatically do the hello command. If it is unsuccessful the program will ask the user if it wants to try it again. If not, the program will throw ProtocolException and terminate otherwise it will repeat the previous process for asking an IP address and Port until it succeeds or the user indicates it wants to exit.
3. **When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.**
   Everytime when the turn of human player come, You **first have to** type “MOVE” and enter, then the CollectoClientTUI (in particular inside the HumanPlayer class’s method determineMove) will send a message “Do you want to get a hint (Type y/n) ?”
   If the user type “y”, then the client will check if it is a single move or double move, then try all possible single/double moves and print out which moves are valid.
   When it finish the hint, TUI will ask again for a move if it is a single move or ask for the first move of a double move and then the second move (you have to type them one by one)
4. **The AI difficulty can be adjusted by the user via the TUI.**
   Every time at the first of a player’s turn in the online game. You **first have to** type “MOVE” and enter then the TUI will ask the user to choose between human player or computer player. And then if you choose a computer player it will ask ““Do you want bot to play the game ? (Type y/n for Bot/Human option)“ if you want a smart bot or naive bot to play for you.
5. **All of the game rules are handled perfectly on both client and server in conjunction with the reference server and client, respectively.**
   1) The setup board
   When the game starts, the server will spawn a CollectoServerGame between clients and their starting turn will be decided randomly.

The game will first initialize a 7x7 board (with 48 balls divided equally to 6 (colors blue, yellow, red, orange, purple and green) with no 2 same color balls are adjacent to each other) and the middle position of the board will be empty. Then the game will check if there is a possible move for the game or it will re-initialize the board again.
When the create board procedure has finished, CollectoServerGame will send the correct board to both clients.
In the client side, it will receive the board and make a copy of the board and also check if it is a valid board.

2) Make a move
Whenever the game is updated, the board of both sides(in CollectoServerGame and client's CollectoGame) will check if there is still a possible single move. If yes, the board will force the client who in turn to make a single move. Otherwise, the player can make a double move. If the CollectoServerGame find that there are no possible moves, then it will calculate the score and decide the result of the game and send to client
After each move, the board of both side will collect all adjacent balls and display updated board

3) Collecte balls and calculate score
For both server and client side, the board will collect all adjacent balls and calculate the score of both players.
In the client TUI player can see how many balls, scores and which balls they and their opponent got.
When the game is over, who has the most scores will win, (if scores are equal then it will count the number of balls both players got), otherwise it is a draw game.

6. **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 ended it is printed who won the game and the reason why. After that, all the variables game and playerRole(player's current turn in the game) will be reset to null and -1 (in both CollectoClientHandler) (the game then will be then removed by Java auto garbage collection). Then the client can command "QUEUE" to join the queue again and wait for a new game.
7. **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.**

   1) ERROR and ProtocolException
      In any procedure if the protocol is not followed by the CollectoServer, ProtocolException will be thrown and CollectoClient will remove the connection.
      If CollectoClient send a wrong command, the CollectoServer will send back ERROR + reason
   2) HELLO procedure:

When a new client is trying to connect to the CollectoServer, they will send HELLO protocol to each other, in the procedure if the protocol is not followed then, ProtocolException will be thrown and remove the connection.

3) LOGIN procedure:
   The client will send a LOGIN with his username to the server. The client cannot do LOGIN with a different name while is connecting to the server
   CollectoServer will check if the username is already in the server's client list. If not it will send back LOGIN, otherwise will send ALREADYLOGGEDIN
4) LIST procedure:
   The CollectoServer will keep the list of all online players.
   The client can doLIST every time after finishing the LOGIN procedure, and the server (doLIST) will send the correct list of online players.
5) QUEUE procedure:
   The clients can only doQUEUE when they are not in the game (will be checked by both sides). The CollectoServer will add the client to the queue waiting for at least 1 other player or can start immediately if had one.

8. **Whenever a client loses connection to a server, the client should gracefully terminate.**
   We even implemented a little bit more. When the client tries to write something to the server and this is not possible a ServerUnavailableException will be thrown, then the following happens. First inside the class CollectoServerHandler the BufferdWriter and BufferdReader will be closed and then the socket will be closed. Then inside the class CollectoClient the reference to the CollectoServerHandler will be set to null. Also the while loop that waits for user commands in the class CollectorClientView will shut down. Then the client will ask the user if it wants to connect to a new server, if the user answers “yes” the client will ask again for an IP address and port number as described in point two and the view is started again. If the client types no, The program will end.
9. **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 client’s disconnection appear, it will throw a ClientUnavailableException, if the client is in a game then the CollectoServer will doGameOver with DISCONNECTED reason to the other client and the variables game and playerRole in CollectoClientHandler of other player will be reset and he can command “QUEUE” to wait for a new game.

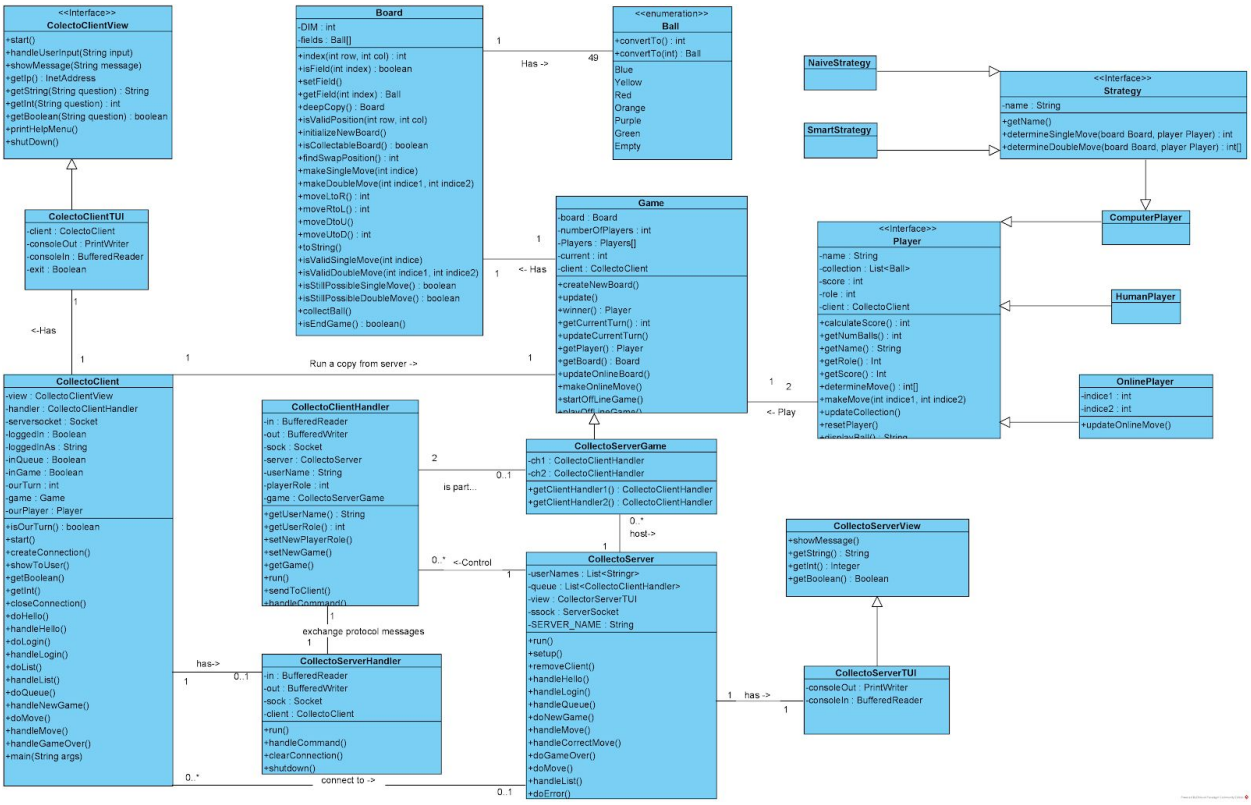
Explanation of realised design

Package :

We separated our programme into 6 packages: collectoclient, collectogame, server, exceptions, protocol and test

+ collectoclient : Contains all the files following the Model-View -Controller for the Client side.
+ collectogame: All the game core logic and data including board, game, player, strategy.
+ server : Contains all the files following the Model-View -Controller for the server side.
+ test : All the test including unit test, integration test and system test for the whole programme
+ protocol : All the protocol format to communicate between the client and server
+ exception: All exception, error can happen during the running. We create ClientUnavailableException(when client disconnects from the server), ExitProgramme (when client exit from TUI), ProtocolException(when server or client does not follow the protocol communication) and ServerUnavailableException when client cannot find the server (to send message, or server was down).

Class Diagram:



- We designed our Class Diagram with Model-View -Controller.
- The View are CollectoClientTUI class which displays the reply from server and current game's situation and CollectoServerTUI class which displays the command from client and server reply. Both of them are connected to CollectoClient and CollectoServer respectively. These classes contain methods to get a boolean, string or an int from the user's input so they return the correct data type(int, boolean or string). The class CollectoClientTUI also contains a method showMessage to display the communication between Server, Client and User and also the game's situation.
- The Model is the Game (with the Board inside), Player , CollectoClient and CollectoServer class:
    + Game class: (and the CollectoServerGame implementation
      Each of the Client and Server has its own Game version to check all the conditions of the game. However the Game on the server side is the main game(CollectoServerGame) with the additional data of two ClientHandler are in the game , the game on the client side is the copy version from the server.
      The main role of the Game is create new board(createNewBoard), update game's situation(update, updateCurrentTurn), decide the result of the game(winner), createNewBoard method to make sure the intializedBoard is correct has at least one possible move, makeOnlineMove-to update the move

from online player, and updateOnlineBoard to copy the board from server to client's side.
The game will have 1 board and the information of 2 players who are playing the game with the starting turn was decided randomly.
The game can run offline game (in only client side) or online game when connecting to the server

- The Board is with all main core logic of the Collecto board game make a new board(initialzeNewBoard and fix it with findSwapPosition), how to make a move(makeSingle/DoubleMove), initialize a new board, check valid moves, board(isValidPosition, isValidSingle/DoubleMove, isCollectableBoard), check if there is still possible move or end(isStillPossibleSingle/DoubleMove, isEndGame), make change after a move(collectBall). The Board also has a converting method(moveLtoR, moveUtoD, ..) to change move's indice to correct row/column and direction.
- There is also a enumeration Ball in which we use to convert the color to corresponding integer and vice versa. (Empty, Blue, Yellow, Red, Orange, Purple, Green) -> (0,1,2,3,4,5,6). We use Ball to make it more understandable and easy to check than digits.

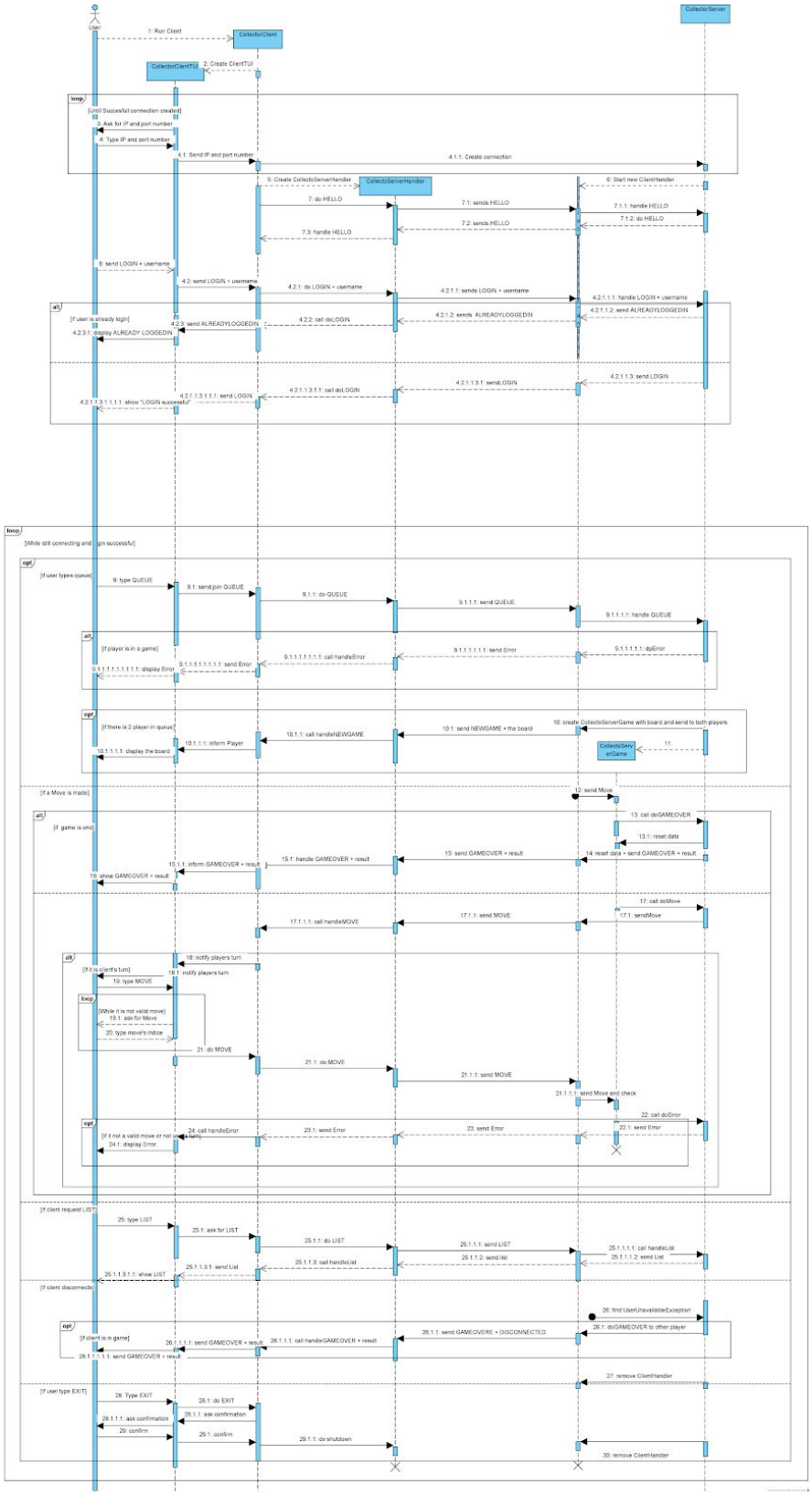
+ Player abstract class:
  Player class represents the user in the game. The player class will store all the starting turn of player (role), scores, balls collection, calculate score and decide the next move depending on the board.
  Player has 3 implementation classes which are HumanPlayer, OnlinePlayer, ComputerPlayer. The Human Player is the player who decides his move on his own (he can get the hint). Computer Player is the BOT player whose move is decided by the computer's strategy. We have 2 levels of strategy: the Naive and Smart. Naive will only make a random valid move. Smart will aim for the score and the best move each turn. OnlinePlayer is the player on the other side of the server playing with you. Its move will be updated(updateOnlineMove method) when the server sends the move and applies it to the game.
+ CollectoClient and CollectoServer:
  They have the main method which is the starting point for the client and server. Their roles are establishing and clearing the connections(createConnection, closeConnection method) with each other, ensuring the connection follows all the protocols properly (Example: doHello, handleList, doQueue, doError ...) and running a game.
  CollectoServer is the one who will host and run the new CollectoServerGame when 2 players are in the queue. Then the game data will be stored by both CollectoClientHandler and vice versa.
  CollectoClient will also get the copy version of the game and also run it on the client side. It has also all data of the user's players (ourPlayer) and will inform the user when it is their turn (ourTurn) (by update the turn every time receive MOVE from server and check it with ourTurn)

The CollectoClient also contains the methods getInt, getBoolean and showToUser. These methods only function is to forward the input to a different method in the view and receive and return the data. We made it this way because most of the classes that need user input had already saved the CollectoClient object. It was also possible to also save the View object in all these classes, but this seemed easier and more clear to use, because otherwise we needed to add another variable to constructors of methods which then become quite large or add a setView method everywhere which takes more lines of code than this solution.

- The Controller is the CollectoClientHandler and CollectoServerHandler classes.
    + CollectoClientHandler:
      CollectoServer is designed to handle multiple clients at the same time, so every time a new client is trying to connect to the server, it will spawn a new CollectoClientHandler to specifically deal with this client's command and call the correct method in CollectoServer. The handler will be removed when the connection is ended.
      The handler also represents the player on the server side. Every time there are two CollectoClientHandler in queue, the game will be started, and the handler will be assigned with the data of new game, the player's data.

    + CollectoServerHandler:
      Its main role is formatting the command from the user flowing the protocol to send to the server. And it also read the command from server, format it and call the corresponding method from CollectoClient.

Sequence Diagram 1 (Client - Server communication)



This is the sequence diagram with the lifelines user, CollectoClientTUI, CollectoClient, CollectoServerHandler, CollectoClientHandler, CollectoServer.

The start of the diagram:
This diagram starts with a running server, so the setup of the server is not included. The user runs the CollectoClient and because this diagram is about the connection between the server and the client, some things are left out. For example the first question if the user wants to start an online or offline game. This is an option if users want to play offline on its own client or play online on the server. If a user chooses online, the CollectoClient will immediately starts with creating a CollectoClientTUI and enters the loop to create a connection.

Creating a connection:
This happens in the first loop in the diagram with the condition: “Until Successful connection created”. This will happen in the method “setup()” in the CollectoClient class. The TUI will ask the client for an IP address and a port number. It will also check if port number is an integer, if not it will ask again. Then it gives them to the CollectoClient that will try to create a connection with a server on the given address. If this fails the program will ask if you want to try again, if yes the loop repeats, if no the program terminates.

The hello:
After a connection is successfully created the client will check if this is a CollectoServer by sending it a hello according to the protocol and expects to receive a hello back according to the protocol. If this is not the case an error is thrown and the connection is closed. Then the client is asked if it wants to connect to a different server, if yes the program will start above the loop for creating a new connection again. If not, the program will terminate by throwing ProtocolException. The methods used for the client are “doHello" and “handleHello", these methods are located in the CollectoClient class. For the server it is the method “handleHello" located in the CollectoServer class.

The login:
After the hello, the program will inform the user it needs to first login before it can do other commands because this is required by the server. All other commands will result in an error message sent by the server. So the only two commands available are: “login” & “exit”. In the diagram it is only shown that only the login is available because otherwise the diagram would become quite messy so that is why we chose to show it this way. The client needs to type “LOGIN + username” it wants. This is then sent to the server, if this username already is in use, it will reply with ALREADYLOGGEDIN if the username is available it will reply with LOGIN. The user will be informed if the choice was available, if it was not the user needs to repeat the process with a different username. The methods used for the client are: “doLogin" and “handleLogin", these methods are located in the CollectoClient class. For the server the method “handleLogin” is used which is located in the CollectoServer class.

The big loop:

After a successful login the program will enter a big loop where it receives all user's commands with an exception, not shown in the diagram because this diagram is about the protocol, but I will explain them in here.

- **QUEUE -** This command can be used to enter or exit the queue for a new game.
    - In the CollectoClient is saved if the user is already in a queue, so the second time the command is used, the user will be asked if it meant to leave the queue. This command is also unavailable during a game. The method used for the client is "doQueue" which is located in the CollectoClient class.
    - For the server it is the method "handleQueue" which will add a player's CollectoClienHandler to the queue. If the queue has more than 2 players, it will remove the first 2 players and start a CollectoServerGame between them. The server will send Error messages if any player is sending "Queue" when in a game. The server also doesn't send back any reply for the queue if it's successful.
- **LIST -** This command can be used to get a list of all the players that are online and logged in on this server. This command is always available.
    - The methods used for the client are "doList" and "handleList", these methods are located in the CollectoClient class. "doList " will send requests to the list to the server, "handleList" will receive the list of online players from the server and present in a more readable way for the user.
    - For the server the method "handleList" is used to send the list of online players to the client, which is located in the CollectoServer class.
- **NEWGAME**

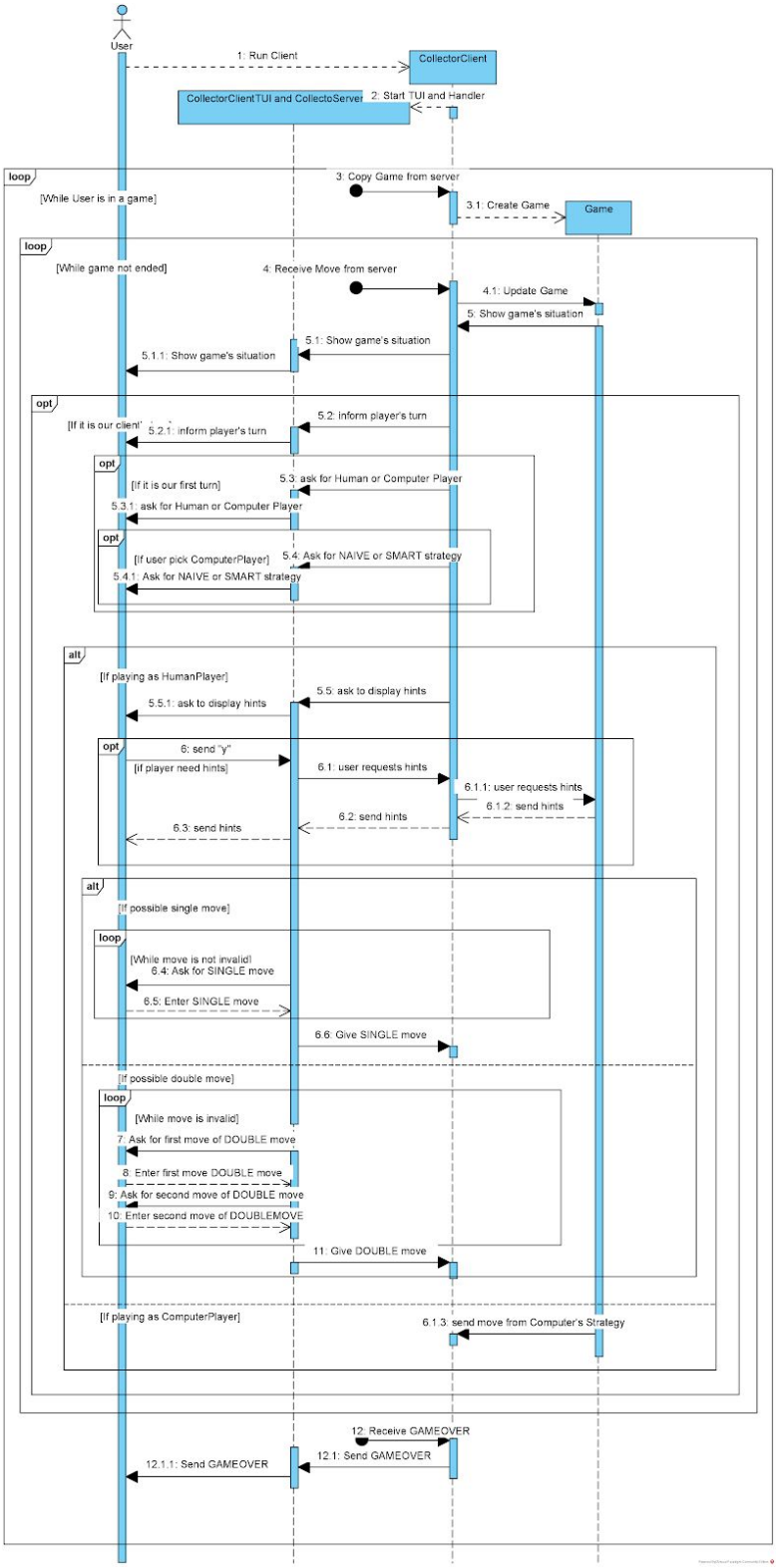
  - For the server it is the method "doNewGame" that is located in the CollectoServer class. It will be called by the handleQueue, when the queue detects that there are more than 2 people in the queue. The doNewGame method will create a CollectoServerGame and generate a correct board, decide starting player randomly and link the CollectoClientHandler (which represents the client's player too) and game together (by assigning a shallow copy for both sides), then it will send the generated board to both clients.

  - For the client side, the CollectoClient will call handleNewGame when it will receive a board from the server when a new game is started and the two players' usernames. This method will create a copy version of the game and 2 players, and get the turn of its client to inform the user when his turn comes (by checking his name in the message from the server).
- **MOVE:** This command is used to send your move to the server or to receive the opponent's move from the server. It cannot be send by user if he is not in a game or not his turn (server will send and Error message and client will also prevent that)
    - For the server, when any player makes a move it will be received by handleMove, it will first check if it is his turn or it will return an Error message. Then it checks if the move is a valid move or not. If it is a invalid move it will return an Error message. If it is a valid move it then will be applied and updated to the CollectoServerGame and check the winning condition. If the game is still

going then it will call doMove to inform both players, otherwise it will call gameOver.
- For the CollectoClient, it keeps track who's turn it is, so the player can only call "doMove" when it is his turn. The methods will send the move that is determined by the user to the server. The "handleMove ", however, will work in whatever turn, it will get the move from the server, check it if it is correct then it will apply it and calculate the score and collection of both players and display them through TUI to the users .

- **GAMEOVER -** This command is send by the server in only 2 situation
  - There is no other possible move and the server will calculate the result and send to both clients, this can be: you/the opponent won the game (VICTORY), the game ended in a draw (DRAW) . This method is doGameOver which will be called by doMove when it check there is no moves
  - The other situation is one player disconnected(DISCONNECTED). It will then stop the game and send the result to the other player. The method doGameOver in this time will be called when the server find ClientUnavailleException and check if the player is in game so it will send to the still online player the GAMEOVER DISCONNECTED reason.
  - On the client side, when it receives this message, it will display the result to the user.
- **ERROR**
  - This command will be used by the server sending to the client via doError method with reason that is explained above.
  - For the user there is not a method that handles this, inside of "handleError" located in the CollectoServerHandler will display the Error message from the server.
- **EXIT -** This command can be used by the client to exit the server. First a confirmation is asked to be sure the user really wants to exit. The connection with the server will be closed gracefully and the user is asked if it wants to connect to a new server. If yes, the program starts again at the top of the diagram. If no, the program will be terminated. This command can be used all the time. When a client disconnects the server will delete all the references to the client, remove the clientHandler and if the client was in a game, let the other client win by sending a GAMEOVER command. For the client the method "closeConnection" is used to close the connection which is located in the CollectoClient class. For the server the Client handler will notice the connection is closed and then the method "shutdown" will be called located in The CollectoClientHandler class.

SequenceDiagram 2 (User - Client in game communication)



This diagram shows the interaction between client and user while it is a game. To make it more clean and convenient I merge CollectoClientTUI and CollectServerHandler into 1 object. The reason is because this diagram is only about the communication in the game. And the TUI and Handler only work as the view to show information and an input reader for the user.

- **Start game**
  The game will start when it receives a NEWGAME command with the board from the server. It then create an copy version of the game on its own side with 2 Online player data (because the game will only be update if it receive a MOVE from server, so it need to have 2 Online player)
- **User's first turn:**
  The CollectoClientTUI will ask user to choose between Human Player or Computer Player (if Computer Player then need to choose between Naive or Smart Strategy).
- **Every turn:**
  The CollectoClient will wait for the doMOVE from the server. When the server sends the move, it first will update the game's situation, then update the current turn and display it to the user.
  If the current turn is the user's turn then it will inform the player "Alright this is OUR TURN", otherwise it will display "Waiting for other player's move …"
- **User's turn**
  User can only make a move when it come to the user's turn or client will prevent and inform that "It's not our turn"
  - Playing as HumanPlayer:
    + If the user chooses to play on his own, first the CollectoClient will ask if the user wants to see "hints". If user type "yes" then it will display all possible moves in the game.
    + Then the game will check if there is still a possible single move, then the TUI and Handler will inform the user and force them to type a valid single move or it will wait until a single move is type.
    + If there is no possible single move, the game will check if there is still a possible double move. For the double move, the TUI will first as for the 1st double move, and then 2sn double and users need to type them separately (if it is not a valid move then user need to type again)
  - Playing as ComputerPlayer:
    + After the first move, if the user decides Bot to make a move for them, then the game will calculate the move and send it directly to the CollectoClient.
    + For the Naive strategy, it will only make a random but valid move
    + For the Smart strategy, it will make the best move based on its algorithm.

Concurrency mechanism

The client side

The client side of the program is fairly simple. If the user choses while starting up the client to play a game offline (so not connect to a server) there are no new threads started. All happens on a single thread. If the user however choses to play a game online (so connect to a server) the server handler is started on a new thread and the original thread will enter a while loop in the view constantly waiting for user input. These two threads do use the same data, but never change the same variables. So both threads read most of the variables, but only one of the threads changes a variable. For example the variable “inGame” is read by the view-thread but can only be changed in the serverHandler-thread. This way there will never be an error and that's why we don’t make methods synchronized. Making methods synchronized is also not the solution to this problem if there was an error, because the two threads never use the same methods. When the user indicates to exit, the serverHandler-thread ends and the while loop in the view stops. Then the user is asked if he/she wants to connect to a different server or shut down entirely. The same happens when the server disconnects.

The server side

In the server side we create a new thread for every new connection that is created with the server. Here it is possible for concurrency errors to occur because a lot of threads are able to access and change the data at the same time. For example the “handleQUEUE” for a new game where we save the clienthandler. It is possible for two threads to change that data at the same time, that's why this method must be synchronized to prevent concurrency errors. The same holds for the methods “handleLogin” and“removeClient”. However this still does not solve the problem, because the variable queue is edited in “handleQueue” and in “removeClient”. The same holds for the variable usernames in the methods “handleLogin” and “removeClient”. That is why we created a synchronized block that is synchronized on the variable. This way only one method and one thread can change it at the same time. The remaining methods do not change the same data so making those methods synchronized is not necessary. When the server disconnects the while loop in the clienthandler-thread will stop and the thread will thus close automatically.

Reflection on design

Class Diagram

- **Parts are still working well:**
    - The Model-View of the collecto client and the relations of CollectoClientView, CollectoClientTUI.
    - The relation between Game - Player - Strategy and Board - Game - Ball
    - Most methods from all classes are kept in actual design.
    - The algorithm for core logic inside Board are working properly as in the initial design (especially the initialzeBoard method)

- **Parts are changed:**

- CollectoClient class:

1) In the initial design, with all the messages from the server, we plan to format in CollectoClient with corresponding methods. However, later we realized that it's better if we create a Controller: CollectoClientHandler which will receive all messages from server, handle messages(LIST~Minh~Tom~Adam) into actual data (Minh, Tom, Adam,..) and call the corresponding methods from CollectoClient. And we complete the Model-View-Controller for client side
2) In our initial design, CollectoClient will only has a Board version instead of a Game, we latter realized that, with only the Board users cannot keep update with some important information of the game like (player and his opponent's collection and score)

- Board class and Ball enumeration:

1) In the initial design, we have "numbering" and "indices" attribute which we want to use to present toString() method for board, but latter we can do without them
2) We need to have some additional method(moveLtoR, moveUtoD, ..) to support converting the move's indice to correct row and direction for the makeSingle/DoubleMove. We also add methods CollectableBoard() and collectBall() to check if the move is valid and collect the ball for each player (which we forget in our first design). These methods help us to separate our concerns and balance our methods on the Board.
3) Ball also needs 2 methods convertTo to convert it to corresponding number and vice versa.

- Game class and CollectoServerGame:

1) At first our Game class was quite simple and it just can run offline. We later changed to add more setter and getter methods, to update the data from the server's game, because the client side is just a copy version, it does not run on its own.
2) We need to add 1 more subclass of CollectoGame, it is CollectoServerGame so that it can keep track of the Player's client to send them the updated situations. And CollectoServerGame is connected with CollectoClientHandler and the CollectoServer instead of normal Game

- Player:

1) In our initial design, we did not plan for another player which data will not be input from System.in, so we have to create 1 more subclass for Player - OnlinePlayer, which its data, decided moves will be updated by server.
2) The current Player class has more getter and setter methods than initial design because with the data from the server we have to create the corresponding data for Client's opponent OnlinePlayer.
3) One improvement is that we merge two makeSingle/DoubleMove into 1 method makeMove with the integer array will be returned. With that we now only have to call 1 method to get the move from Player.

- CollectoServer

1) For the CollectoServer we change it to add View so that it now becomes the Model - View - Controller for the server side too. The View will show the updated situation on the server and messages exchange between server and client.
2) We change also our CollectoClientHandler, so now it is not only handle command from client side but it also represent the data of Player for client in a game (playerRole(turn) and game's reference memory address location), which make it is easier to host the game in the server (keep track of player's turn, don't have to make a list of running game because it was saved in handler, and client also can keep updated and make move during the game (because the CollectoClientHandler will only exchange message with its linked client)
3) For the server class we also need to add doError(send ultimate Error when something is going wrong) method which was not included in initial design. List of userName and queue are used instead of list of clients, so we can keep track and differentiate between players on the server and players who are actually in the queue waiting for a game.

- **Parts will be changed in the future:**
  The game class should be divided into OfflineGame, ServerGame and ClientGame with corresponding methods instead of putting them together in Game class, so some methods are only utilized in Server or Client. In our initial design, it is only the Server running the game and the client only has to show the data from it, but later to improve ourTUI and separate our concerns better we will have to improve them.

Another thing is that we put CollectoClient inside of the constructor of abstract class Player, now every child of the class player also needs a CollectoClient inside of its constructor. This comes in handy for the class HumanPlayer, but it is not the case for all the other classes. So next time we would put it only inside of the constructor of Humanplayer.

Sequence Diagram 1 (Client - Server communication) :

- **Parts are still working well:**
  The basic exchange messages of the diagram are still the same, first create a connection, then do the hello, then login and finally all the other commands inside of a loop that can be used independently. Also when threads are spawned stayed the same, except of course for the new class we needed to implement.

- **Parts are changed:**
  There is an extra lifeline for the game because when there are two players the server will create a new ServerOnlineGame.
  We also added a loop around the part where the client needs to login. This was something we already knew what was going to happen but we somehow forgot to put it inside of the initial diagram.
  While writing the List method we discovered that the list can be asked at any moment during the program, and thus also received. That is when we decided we needed a serverHandler which runs on a different thread. So now the messages do not go directly from the collector Client to the clientHandler but now everything goes through a serverhandler.
  Another change we made is adding some unexpected situation (client send an error message or command is not correct or not in right time) and how it will be handled (Ex: client disconnection will lead to the doGameOver immediately) The doError now is also added as a returned message when some unexpected situation happens.
  And finally, because now the “queue” is the starting of the game so we put other exchanges messages for the game (doMove, handleMove, doGameOver) inside a handleQueue and add additional conditions when the game happens and ends.

- **Parts will be changed in the future:**
  Take a longer time and forget some part in our design. Before actually coding we should first think about all the possible situations and cover all requirements in the design. Now we discovered mistakes while coding and it took some time to first think of a solution and then reverse what we already did and make it the right way. That is something we both really learned from this project.

SequenceDiagram 2 (User - Client in game communication) :

- **Parts are still working well**
  -The move interaction between the user as a Human Player and the Client is still correct.
  -The start game when Client receives and creates a copy board from the server is still correct (but the board now will be included in the game).
- **Parts are changed:**
  -The CollectoClient now will have a copy of the game version instead of a board to easier keep track and display more data for the user.
  -Because the game is now online so it will be updated by the move from the server instead of running on its own. Therefore, everytime a move comes, the Collecto need to first update its copy game with the move before displaying it to user.
  - Because we differentiate between Human Player and Computer Player, so now the user will have an option to choose between them instead of making a move on his own.
  - Improvement : Users are now able to get the hints from the game which make it more convenient if they cannot see a possible move.
  -The double move now will ask separately for 1st and 2nd move of Double Move which reduce the chance of mistyping from users (Ex: type 2 - 12 or 21 - 2)
  -And because the game is now updated by the server, it only ends when it receives the GAMEOVER command from the server instead of ending on its own which can handle some unexpected situation. (Because there is a situation where another player is disconnected from the game).

- **Parts will be changed in the future:**

- For the interaction between user and client right now, we think it is quite complete. However, if we had a chance to do it again, we will implement the GUI, to give user more user-friendly experience when interacting with CollectoClient rather than having to enter the move by move

System tests

For the systemTest we wrote 2 test cases classes. The CollectoClientTest and the CollectoServerTest. These two tests together almost cover the entire systemTesting. There are a few cases where this is not the case, but at the bottom of the test explanation is provided what should be done to test the entire system. In the system test we use most of the methods but don’t focus on the special cases because that's where the unit tests and the integration tests are for.

- CollectoClientTest: This test tests all the methods connected to the client and has a coverage of 57.15 over all these classes (the packets collectogame & collectoserver)
    + We test the class CollectoClient (63.7% cover)
    + We test the class CollectoServerHandler (91% cover)
    + We test the class SmartStrategy (45.8% cover)
    + We test the class board (76.6% cover)
    + We test the class Game (46.8% cover)
    + We test the class NaiveStrategy (45.5% cover)
    + We test the class Player (94.4% cover)
    + We test the class Ball (100% cover)
    + We test the class ComputerPlayer (100% cover)
    + We test the class OnlinePlayer (100% cover)
    + We test the ProtocolException (100% cover)
    + We test the ServerUnavailableException (100% cover)

  To test the Client side of this project on its fullest, the class CollectoClient needs to be run and a few offline games should be played. Also an online game should be played using the Smart and the naive strategy. The test does test when the connection between the server fails, but to even better test this, this should be tested at any point in the program. Then the entire client side is tested to its fullest.

- CollectoServerTest: Important note, before running this test, the class CollectoServer should be run and the port number should be 8080! NOTE: we can not see the coverage.
    + We test the class CollectoServer
    + We test the class CollectoClientHandler
    + We test the class Board
    + We test the class OninePlayer
    + We test the class Player
    + We test the class CollectoServerGame

  We are not sure what the exact coverage of these tests are, but we are certain that if a game is played between 2 clients, everything is tested. Something that is also quite important is to run the test a lot of times on the same server to test the concurrency of the different threads.

A few thing we could not write easily a test case for are:

1. When server suddenly disconnects
2. When client suddenly disconnects
3. When a user just does random stuff

Your are able to test this by:

1. Start a server and connect the client to it. Then at any point during running the program disconnects the server. When you as a client try to send a command to the server an error should be shown that the server is unavailable. Then you should get the question if you want to connect to a new server. You should be able to connect to a new server as if you connect to a server in the first place (so all the variables are reset to beginning values) Also when you don't connect to a new server, the program should terminate.
2. When a client disconnects all the references to it should be removed. So the list "userNames" and the list "queue" must not contain the client anymore. If the client was playing a game the other player should be notified that he/she won and the reason is that the opponent disconnected.
3. This is testable by doing random stuff yourself, type unknown commands. Or type commands that you should not be able to use at that time. The system should react by giving proper error messages.

Overall testing strategy

Unit and Integration Test:

For the unit and integration test, we have written all the tests which cover 99% of all methods in the core login part (Board, Ball, Player, Strategy) class.

- Ball test: we test how the ball is converted to corresponding integer value and vice versa. (100% cover) (Unit test)
- Board test: (Unit test)
    + We test all the getter, setter methods by check its return value (100% cover)
    + We test the make single move and double move by check the change in fields (99% cover)
    + We test the create new board (initializeNewBoard) by check if it is a valid board (However because it's a random board, so sometime fixing methods are not fully cover) (84% cover)
    + We test the fixing board method (findSwapPosition) by assign an invalid board and check if it return the position that the ball can swap (100% cover)
    + We test all the check field method (isValidPosition, isField) by set predicided board (100% cover)
    + We test the check condition for the single and double(isValidSingle/Double move), by assign a predicided board and make a move and use isValidPosition. (100% cover)
    + We test the check board (isCollectableBoard) by set some balls in adjacent and the correct board (no adjacent balls) (100% cover)
    + We check the toString method assigning a predefined board and compare the output string (100%cover)
    + We check the collectBall method by assigning a board after a move and check after the method is called, there is no invalid position in the board (100%cover)
    + We check the ending's condition (isStillPossibleSingle/DoubleMove, isEndGame) by assign some board's situation and check move methods (there is still 1 single move, there is no single move, there is no any valid move) (100% cover)
    + We check the the update game situation method (update) by assign some game situation and make some moves (100% cover)
- Player and Strategy test: (Unit Test + Integration Test)
    + Class are not fully tested:
      HumanPlayer class will not be tested because its move will be decided by input. However all methods in HumanPlayer when a user makes a move(singleMove, doubleMove, board check) are fully tested in Board class. The hints for the player is also the method isStillPossibleSingle/DoubleMove which were tested in Board class too. Other methods are inherited methods which will be tested in the parent class.

The SmartStrategy is also (56% cover) because it's hard to cover all the situations of all games in the test. However, all the possible moves from SmartStrategy are tested to make sure they are valid moves.

+ Player test: (Test player and its subclass)
    - We test all the getter, setter methods by check its return value (100% cover)
    - We test updateCollection/updateOnlineCollection with a predefined collection/OnlinePlayer with a given collection and adding it to player and check if it is added correctly (100% cover)
    - We test the calulateScore method by assign a predefined collection and check its return (100% cover)
    - We test makeMove (for player) by assign a Strategy and a predefined board to check if it the board and the player's score and collection is updated (100% cover)
    - We test display player's situation (displayBall method) by assigning a predefined player with given collection (100% cover)
+ Player's subclass OnlinePlayer test:
    - We test online player by create a predefined board and call updateOnlineMove to make sure the move is correctly applied (100% cover)
+ ComputerPlayer and Strategy:
    - We test all getter and setter methods in Strategy by check its return value (100% cover)
    - We test Naive Strategy by call determineMove in a predefined board and check if it's a valid move (100% cover)

- Game test (we cover 90% of the game class): (Integration Test)
    + Some methods are not or not fully tested :
      startOfflineGame (all of its called methods are fully tested, we cannot test because this is a method to create a loop for offline player plays game again.
      printRestult (a method for offlineGame) is call function to check winner is fully tested
      createBoard (to create a board with at least 1 available single move, which is sometimes not needed because the initialzeNewBoard in Board class maybe did create a board with 1 available move) the board is only need to test for always have at least 1 valid move
    + We test all the getter, setter methods by check its return value (100% cover)
    + We check the calculation of result (winner method) by assign all possible situation for both player (1 player have higher scores, same scores but higher number of balls and same scores and same number of balls) (100% cover)
    + We test method for the online game(updateOnlineBoard and makeOnlineMove) by assign an predefined board and valid and invalid move(which will be check in Board class) (100% cover)

    + We test the offline game methods (playOfflineGame, reset) by let 2 bot player playing with each other and check the final board, final result of both player (100% cover)
- CollectoClientTest (covers 63.3%)
    + This test covers all the methods that don't require user input. However, these methods are for starting up the client and to create a connection with the server, so if you can successfully connect to a server, all the methods are tested.
    + This test is also a system test, it does not check every method completely, but it does check the access to all the methods. In the unit tests are the methods checked completely. So if the connection to the methods works well, the entire method does.
    + For other methods we need to test them manually.
- CollectoServerTest
    + We test all the methods for the initialize parts between the server and client (doHello, doLogin, doQueue, doList)
    + For a complete test we need to run a game played between two users and the CollectoServerTest(which we need to test it manually).

Reflection on process

Vo Nhat Minh

For our group, we have done quite well and I'm really appreciating my partner because we have worked very hard. Working with Jesse is a very nice and conscientious partner. Sometimes we have discussions but these discussions are always going in the right direction which did improve our project so much. Jesse is also a productive partner which makes our project rarely behind the schedule and keep up with the plan. He has also taught me a lot about using Git. Through this project, I have realized that Git is a good way for programmers to cooperate in such a big project and save us a lot of time when we are writing code together.

If I have a chance to do our project again, I will do more carefully with the design in all aspects especially with the client and server. Actually, we find out that the initial design helps us so much, it defines the way we build our programme and our system. In our initial design, we planned very well for the Game and Board (the core logic part of Collecto Game) with all their functions, how they work and the connection between them, so when we code we don't need to change them much and most of our design is kept for these parts. However, with the Server and Client, we don't have that detailed design (it's also partly because we learned about it in the very last week, so we were not fully understanding about it at the time of planning). Therefore, when developing them, sometimes we need to add more methods or even other classes which make our final design not clean (for example the object CollectoServerGame was added when we realize that we haven't planned a good way to keep track of the running game and send game's situation correctly to its clients). The lesson that I've learned is to always think carefully ahead before starting to code and if somethings not in the plan appear and need to add I will carefully think about its function and relation with other methods and classes.

Another thing that I will improve in the future is following the Test-Driven Method more strictly. In the project, we had a plan that every time when we write the code, we will write the test cases beforehand or at least right after a method. However, sometimes I do not follow while coding and this leads to a critical consequence that it takes me much more time to find an error when it appears (I cannot differentiate which lines of code are wrong and I did have to debug for 1 hour to find the bug).

Jesse Snoijer

I think we should have spent more time on the design of our code. When you look at our initial design and the design we now have, you notice a big difference. We needed to create a lot more methods than expected and even more classes then expected. Most of these classes and methods we could have known beforehand if we took more time to think about the design in the beginning. So next project I will definitely do this.
We should also have spent more time thinking about the threads. While writing this document I discovered a possible concurrency error in the code. luckily it was a small one, but it could also have been a big one. So next time I work with different threads I will take more time to think what exactly is happening with them and where errors can occur.
Finally I think we should have written the test cases earlier on. We actually put in the planning to do this while writing the code, but eventually we did most of the test cases later on. I think it would be better if we stuck with our initial planning. So next project I will try harder to stick with the planning we created in the beginning.

In my opinion the collaboration between Minh and I went quite well. We had sometimes a different opinion about something but we then discussed it and chose the better one. I am certain this did improve our project and probably saved time because we did it right the first time and thus did not need to change it back later on. Also this made me learn more about writing code so Minh is a nice project partner to work with.