REPORT PROGRAMMING PROJECT (COLLECTO)

Kelvin Jaramillo, s2074206
Kai Ferdelman, s1812432

Explanation of realised design

In this section it will be explained the design of the final structure of the game.

IMPORTANT NOTE: The diagrams to which this section will be referring can be found in the image diagrams folder of the project zip folder, this is because the size of some diagrams do not fit in this document pages.

The diagrams that represent the architecture of the game are:

- Initial Design
- Game Overall (Class Diagram).
- Server Functionality (Sequence Diagram).
- Client Functionality (Sequence Diagram).
- Two Players During Game (Sequence Diagram).
- Board creation (activity diagram)

Below an explanation of each of the diagrams is realized.

Game Overall.

The architecture of the packages, classes and interfaces is:

- communications(folder)
    - client(folder)
        - Client(class)
        - ClientTUI(class)
    - server(folder)
        - ClientHandler(class)
        - Server(class)
        - ServerTUI(class)
    - exceptions(folder)
        - AlreadyLoggedIn(class)
        - ExitProgram(class)
        - ServerUnavailable(class)
        - ProtocolException(class)
        - InvalidMove(class)
        - MoveOutOfRange(class)
        - NotYourTurn(class)
    - ProtocolMessages(class)
    - testing(folder)
        - ServerTest(class)

        - ClientTest(class)
- players
    - Player(interface)
    - Human(class)
    - AI(folder)
        - AI(class)
        - Strategy(interface)
        - MinMax(class)
        - SmartMinMax(class)
        - ThreadedMinMax(class)
        - Naive(class)

- game
    - board(folder)
        - Board(class)
        - Ball(class)
    - Collecto(class)
    - BallsKeeper(class)

There are three main packages.
The **communications** is where the server-client architecture is implemented. The client and the server have their own package where it is defined their TUI, furthermore the server package contains the client handlers that are created for each of the clients.
The **players** package contains the two types of players that the game has, a human player and an AI player, the human player makes plays with the TUI of the client, while the AI player plays automatically. Also the AI player has a strategy interface that is implemented by a naive and min max strategy. The explanation on **how the strategy works** is found later on this document.
The **game** package contains the logic of the game "Collecto": the legal moves, the valid board, the counting of points, etc.

Server Functionality.
This diagram shows the server class implementation. It starts by first asking the user through the TUI a name for the server as well as the port to link the server to, if the port is in use already then the program asks for a new port, once the connection is completed a server socket is created. Then, the server accepts sockets from the clients that want to connect to it, assigns a new client handler to each of them as well as a new thread to handle the commands, this is all running in the main thread of the server. In parallel to the main thread another thread handles the creation of new games once two clients want to queue for a new game. For instance, a more in detail threading explanation is described in the section of **Concurrency mechanism** further in this document.
Moreover, once to clients have been put into a game, the client handler sends a **NEWGAME** command. After that, the client and server can exchange these commands: **MOVE, LIST** or **ERROR**. Depending on the command that the client sends, the client handler can ask the server for the following objects: the game of the client, the clients that are in a specific game or the list

of clients.In the same way, when the command **MOVE** is sent to a client it is also sent to the other client that is playing the same game, this is also the case for the command **GAMEOVER**.

Client Functionality.

The diagram shows the client class implementation. It starts by asking the user through the TUI For: **<IP-address> <port number> <name>**, this then is passed to the client class to initialize the connection with the server by carrying out the HELLO command. Once the client is successfully connected to the server the client can try to log in, by the LOGIN command, if the name of the user already exists in the server, then the client's TUI asks for a new name until login is successful. Once the client is logged in then he can either QUEUE to play a game or ask to see the LIST of players on the server. So far this is running in the main thread of the client, but once the client decides to play and sends the QUEUE request, a **new Thread** is created to take care of the communication with the server during a game, the benefits of this will be further explained in the the **Concurrency Mechanism section** of this document. For instance during a game the commands that the user can use are: **LIST** to see the players connected to the server, **MOVE n/ MOVE n n** to make moves in the game, "hint" to ask what are the possible moves, **HELP** to see the menu of available commands and exit to shutdown the game. Furthermore, commands MOVE and "hint" require access to the player class to make moves in his board as well as to get the available moves during the game. Explicitly for the MOVE command the client first sends the request to the server, then if the response from the server is MOVE the client updates his board, this to make sure the client's and the server's board are the same during the game.

Two Players During Game.

This diagram is meant to show the order of the commands that happen between two clients and their client handlers in the server side since they connect to the server, followed by playing a game and finally after a game is finished.
This was a diagram that was made at the beginning to have a more clear understanding of the commands that needed to be implemented.

Board Creation

The diagram lays out how the board given by the server is created.
First an empty board is created and then filled with invisible balls. The size of the array is based on the side dimension of the board, which by default is 7. Next the program goes through the board ball by ball and as long as it is not the central ball, assigns a random color. With the color set the program checks if now there are no more than 8 balls of that color on the board. If there are, the failure counter 'f' is increased by one and a new random color is assigned. Next the program checks that any already existing neighbors don't have the same color. Should one of them have the same color the failure counter is again increased by one and a new color is assigned. Should the failure counter reach 50 attempts it is statistically unlikely that any of the six colors is possible, the board is scrambled and the process begins again. If at no ball more than 50 attempts are made, the board is completed and can be used by the game.

AI (Minimax strategy)

The AI diagram shows the basic functionality of the implemented main AI strategy.
The Program first checks if there are any possible moves it can make. If so it generates a list of all possible moves and starts with the first move. It creates a copy of the actual board and does the move on it. Next it checks whether the board has a winner or has further moves. If it has more moves it repeats the previous steps until either no more moves are possible or more recursions have been made than allowed by the level. Once this happens the branch is evaluated. For this the BallKeeper returns the points and number of balls for the player. The points are squared and the number of balls are added to create the score. This puts a focus for the algorithm to gather points but if not possible go for the highest number of balls or ideally a combination of both.
Once every possible move on depth 0 is evaluated, the program picks the highest score and returns the associated move.

Concurrency mechanism

The threads mentioned below are visualized in the diagrams: Server Functionality and Client Functionality.
Different threads are used throughout the game to obtain the desired behavior of the program.
Server threads:

- Main thread:
- Thread 1: This thread is in charge of creating a new game when there are at least two clients in the queue list.
- Thread N: These threads are created for each of the clients that connect to the server.

Thread 1 was necessary to implement because it is needed to have a separate loop always checking if the size of the queueing clients is above 1, while still accepting clients that want to connect to the server in a separate loop. Also, thread Ns are necessary to give the client handlers the ability to get games and clients from the HashMap and list in the server class.

Client threads:

- Main thread: This thread is meant to be used for initialization of the client, carry out the connection to the server, read stream for the socket when the client is not in game and send commands to the server when the client is not in game but still connected to the server.
- Thread 1: This thread is created once the client queues for a game and takes care of commands sent by the server. In this thread is where the AI player is called automatically to play a game.

The Thread 1 was necessary to have two separate loops, for reading commands from the server(Thread 1) and sending commands to the server(main Thread), specifically once the client gets in line to play a game, because during game the server only sends the commands MOVE, ERROR or LIST, then it is better to have a separate loop that reacts to those commands, like make moves on the player's board or show messages to the TUI. For instance after a game is finished the client then breaks the loop and a new game is only initialized if the user gets in QUEUE again which is read by the Scanner in the main thread..

MinMax threads:
In the threaded minmax class the minmax algorithm is improved by using multithreading. This is done by keeping the setup and picking of the move in the main minmax thread class, while moving the minmax algorithm and score evaluation into a separate threadable class.
When the strategy needs to evaluate a board it first gathers a list of all possible moves and then starts a new thread for each one of them to evaluate itself. After all threads have executed the scores can be gathered and a move can be decided upon.

Reflection on design

Below it can be found three different lists that analyze the design process of the program.
Originally planned but not implemented:

- Make gameview interface and GUI implementation: a game view interface was not made because when working on the client and server it was needed to have TUI for testing and building, then a quick TUI was made which at the end was taken as the final TUI after some improvements. Moreover, a GUI was not implemented because time did not allow.
- Neural Network strategy for AI players: The team was looking to implement an unbeatable AI player. For this the idea of using Neural Network came to mind, however time and complexity left this idea out of execution.
- Use of only one **thread in clien**t to send and receive, meaning every time a request is sent to the server the client then expects a response, **instead of two threads**, one for sending the users requests and the other to read commands from the server. This was a feature that was found to be necessary to fulfill some requirements, explained in the concurrency section.
- Originally all moves were going to be communicated using the basic type integer. Since double moves needed to return two values though, it was decided to create a basic toupel class that could store two values at once.
- Use of two classes to represent objects in a board: **Spaces and Balls**. At the beginning of the project it was believed to be a good idea, but for simplicity the Space class was removed and now to represent by a ball with the color 0.
- Use of ArrayList to store a board. For simplicity this idea was left out and an Array of integers was used instead. The balls were going to be stored in a list and keep track of their location themselves, but instead the decision was made to remove the location attribute from the balls and get their location by their position in the array.

Originally planned and implemented:

- Using different threads for different client handlers.
- Store the games on the server instead of giving them to the client handler, which it is thought to be not the most ideal, but it works as expected.
- Use of list to store clientHandlers in the server.
- Use of HashMap to store the client handlers as keys and games as values.

Possible improvements:

- At the moment the program stores the games in the server class, meaning that when playing the client handler has to constantly ask for the game to the server, this implies that the server has to loop through the map of clients to get the right game. What could have been done instead is to pass the game to the client directly when the game is created, then the client handler does not have to ask the server for the game of the client handler. Moreover, the reason why this feature was not implemented is because, by the time we noticed it is possible to make it, there was already a working server that behaved as expected and there were more than 5 methods depending on that architecture.
- The code could be restructured partially by moving some methods to other classes. This could lead to a reduction in the lines of code. This especially is true for the board class.
- At the moment the server sends an empty line to one of the players in a game. This was found to be necessary because there is a bug in the code which consists of sending two messages in the same line rather than in a new line, different things were tried to fix it, but the only working solution was to send a precautionary empty line. So, this could be something that can be improved, maybe make the program more synchronized or change the architecture until the bug no longer exists. As a side note, the bug only happens when the player constantly sends moves even if it is not their turn, if the clients respect their turn then this bug would not happen, but for the sake of precaution the solution of the empty line was implemented.

Suggestions:

- To read the requirements carefully, at first it was missed that the command LIST was supposed to be sent at any time during the execution of the program, this led to the need of having two different threads, one for reading the stream and the other for sending streams.

System tests

To make sure the system was working correctly and as expected, testing was carried out in this way:

1. The client was tested with the given server (IP: 130.89.253.65 and port: 4114).
2. Once the client worked as the given server expected, our server was tested with the client.

So, the way it was made sure that the system was working properly, it was necessary to use the given server to improve the client up to the point it did not stop working unexpectedly and all the commands given by the server performed the expected actions on the client's side. This meant to take care of: the order of commands, the storage of the board when playing, the correct actions when ERRORs are sent and finally the correct handle when the server disconnects. Furthermore, once the client was tested and improved, the server could then be tested with the improved client, meaning the server was improved up to the point it behaves similar to the given server.

The reason why this testing-improving procedure was possible to make is due to the fact that a working server was available, however if it was not the case, the way to test the system will be:

- Have an initial server and make a Junit test for it, where the methods that take care of the communication are tested.
- Once the server is tested then the client is tested with the server since it needs the responses from the server to function.

In conclusion in order to test the correct functionality of the client server architecture, it was necessary to have first a working server, it could have been the given server or the implemented server, then once it was working the client can be tested with the working server. However, once the communication was working, it does not mean that the logic of the game was working fine, then proper testing for that was implemented, this can be found in the section next section Overall testing strategy.

Overall testing strategy

The testing strategy consists of making individual tests for the components of the program that do not require to be passed a socket or a stream in order to function; these components are the board and the players. This Player needs to be pass a board in other to be able to execute the methods, the used board in is then: 5~3~4~2~5~3~6~4~6~3~4~3~1~2~5~3~2~1~2~6~5~4~1~4~0~4~1~4~5~6~2~1~5 ~6~2~3~1~5~4~6~5~3~6~3~6~2~1~2~1.

It is very important that the board of the players correspond with the board of the server, that is why the test consists in making moves in the board of the players and expect a specific number of balls to be adjacent. Another important feature for the players is that they must be able to make a board from the board given as above, so that is tested as well. Apart from that, the AI player also requires their strategy to be tested, then given a board the MinMax and Smart Minmax are given a known board where it is known what is the best move according to their specifications and the move for the Ai must be as expected. Below there is an overview of the realized tests.

- Test for the board
    - Make a board object out of a string of numbers.
    - Make moves in their board and withdraw the adjacent balls.
    - Make an initial valid board, that fulfills the requirements of the game.
- Test for Human Player
    - Make a board object out of a string of numbers.
    - Get a valid move as a hint.
- Test for AI player
    - Naive, MinMax & Smart MinMax Strategies
        - Test if their moves correspond to the best move corresponding to their own logic. In this case it is necessary to find a specific place where MinMax and Smart MinMax Strategy find the move that they are supposed to do.
- Test for the game
    - **Needs an adjustment in ClientHandler to work**
    - Makes some allowed and not allowed moves.
    - Checks if all possible moves are returned

Reflection on process

These are the individual reflections after the program has been finalized.

Kelvin:

- Instead of showing messages to the user when something wrong happens, I found out it is better to make a specific exception to handle that.
- I builded the server, client handlers, client and TUIs in parallel, which I think was a good idea, because they have to communicate with each other then it made more sense to me.
- At the moment there is a bug in the server side: sometimes the server sends two messages in the same line and when the client reads that line then it can not execute properly, this happens when the server is hosting a game the client handler has to send the valid moves to both client handlers, but the server can also send another individual messages to a client and that message can get mixed and cause failure in the client side. For now some time has been spent in trying to debug but the only solution to this was found to send an empty line to the other player, eg: A client sends a MOVE request then if the move is valid then it is sent to both the client and the other client, in that case the other client will receive an extra line at the beginning of the message. If time would allow, new ways to synchronize the BufferedWriter would be considered.

Kai:

- I developed the game classes based on the class diagram we designed in the beginning. This at first made it easy to begin but when diving deeper into the classes showed flaws in the way the classes were structured. After this I had to go back and rewrite some parts of code.
- I wrote the AI strategies last. Because of this I didn't have as much time to optimize them as I wish I could have had.
- From the very beginning Kelvin and I used a shared GitLab project. This helped out a lot with planning, sharing and maintaining code.
- Despite mostly working separately, Kelvin and I tried to have a meeting every day to briefly discuss problems, plan ahead and show off what we had achieved. This in my opinion worked very well and kept the project organized.