Collecto, Module 2 Programming Project
Report

- Project Overview:
  - The project focuses on implementing the board game Collecto using Java, with a networked approach. It is expected to utilize the programming and design concepts which were taught in module 2, Software Systems.

- Project Requirements:
  - Crucial requirements:
    - A game can be played in conjunction with the client and the server, and with the client and the reference server.
    - The client can play as a human player.
    - The client can play as an AI player.
    - At any point in time, multiple games can be running independently and simultaneously on the server.
  - 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 the client is started, it should ask the user for the IP-address and port number of the server to connect to.
    - When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.
    - The AI difficulty can be adjusted by the user via the TUI.
    - All of the game rules are handled perfectly on both client and server in conjunction with the reference server and client, respectively.
    - 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.
    - 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.

        - Whenever a client loses connection to a server, the client should gracefully terminate.
        - 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.

- Justification of the minimal requirements:
    - Justification of the crucial requirements:
        - The ***first crucial requirement*** is justified by simply running the server and a couple of clients and playing games. The server and the client abide by the protocol and have their own instances of the game in sync. The moves are checked for legality and can be executed. The gameover situation is handled according to the game rules and the winner is also announced correctly. The client can also operate as intended when it is connected to the reference server and it plays games with several other connected clients.
        - The ***second crucial requirement*** is justified when a user logs in. He just has to enter his user name and the system will define him as a human player. First the command is issued in the TUI as "LOGIN~<username>". After a new game is issued by the server, the client handles the response in the method handleNewGame. From there the actual creation of the players for the local copy of the game is done in the method playerCreation. If the username given by the "LOGIN" command from the client does not include "AI" the Player field in the Client class is initialized as a HumanPlayer.
        - The ***third crucial requirement*** is justified in a similar way as the second one. The difference is that "AI" or "AI+" should be included in the username when the user issues the command "LOGIN~<username>". In this way in the playerCreation method a check is made whether "AI" or "AI+" is included in the username and if it is, the Player field in the class ThreadedCollectoClient is initialized as a ComputerPlayer with the intended Strategy.
        - The ***fourth crucial requirement*** is justified by the server creating a new game for every two clients which join and queue for a game. The games are then handled in the client handlers. This makes it possible to have multiple games in different clients, as the players

will have their own game and do not have to wait for other players to finish. These games can be truly run in parallel, as the moves and the game over situations are synchronized over the game and not on the server. To look further into these dynamics the reader can look up the doQueue and doGame methods of the CollectoServer and can also inspect the methods relating the game dynamics inside the CollectoClientHandler.

- Justification of the important requirements:
  - The ***first important requirement*** is justified in the creation of the server socket. The textual user interface will ask the user to input a port number, this input must be in a number format otherwise the user will be asked again. If the number is given, a server socket will try to start listening to it, if the port is not available the correct exception will be caught and the user will be asked again. The reader can further inspect the instructions by looking at the setup method of the CollectoServer, and the getPort method of the CollectoServerTUI.
  - The ***second important requirement*** is handled in the TUI of the client. The first methods that are called in the createConnection method of the Client class are the getIp and getPort methods of the view. They are responsible for asking the user for input and handling it correctly as there cannot be an error if an invalid port or ip address are given. The two methods are used to assign the local variables “addr” and “port” of the createConnection method in the Client class with their respective values.
  - The ***third important requirement*** is justified with the client-exclusive `HINT` command. When typed when it is the client’s turn to play, a legal move will be chosen by the computer, and it will be shown to the client. The client can execute this move or discard it. The instructions which guarantee this requirement can be seen in the handleUserInput method’s HINT case inside ThreadedCollectoClientTUI and the doHint method inside the ThreadedCollectoClient.
  - The way we handle the ***fourth important requirement*** can be understood as a continuation to the third crucial requirement. After being properly initialized, the user is presented with a message explaining that he can now log with either a simple AI or with a

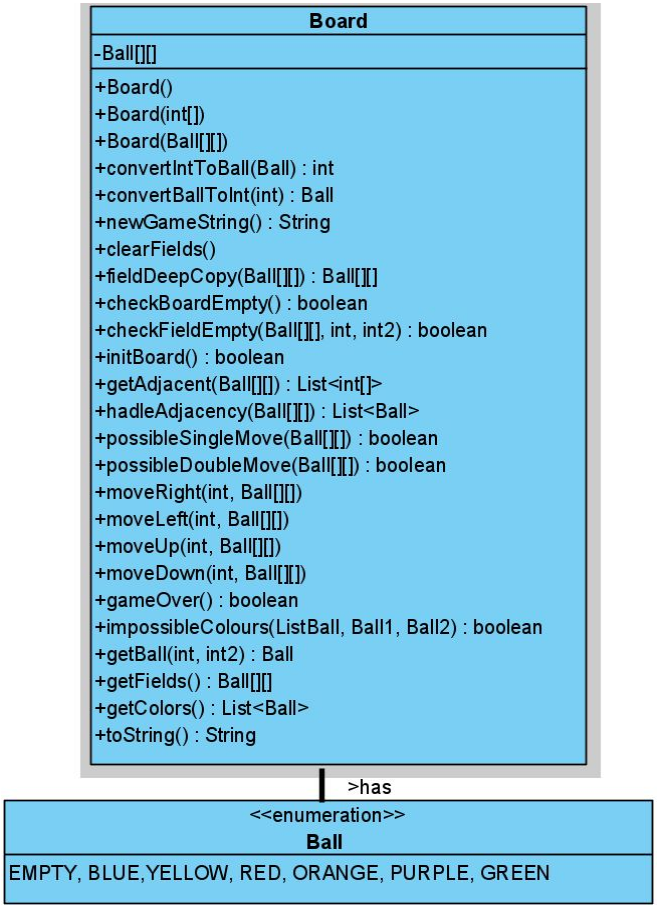
more sophisticated AI to play his/her moves. The respectful additions to the username are “AI” and “AI+”. Depending on the addition the Player field is initialized either with a ComputerPlayer with NaiveStrategy or a ComputerPlayer with SmartStrategy.

- The ***fifth important requirement*** is justified with legality and certain flag checks made by both the client and the server. As both of these components have a game they can check the legality of a move and can handle the necessary actions that should be dealt with after a move is played. The game over situation is checked by both sides as well. Basically the client and the server components do not “trust” other clients and servers, and thus handle the legality and timeliness of every command.
- The ***sixth important requirement*** can be seen in the handleGameOver method of the Client class. After a “GAMEOVER” command is issued by the server and the result is sent to the TUI, only the flag fields “inGame”, “turnToPlay” and “queued” for the respective client are changed to “false” without breaking the connection to the server. Neither the socket or the streams are closed leaving the client the option to continue with a next game.
- The ***seventh important requirement*** is justified by handling the non-game commands as they are intended to be handled. The hello handshake is forced to be done before anything else, and after that the user has to log in. Any tries to skip the initialization will be caught and the user will be warned. Protocol appropriate messages are sent and received by both the client and the server for these commands. The queue command can be typed once and the client will be placed in a queue on the server, it can also be typed a second time to exit the queue. The list command can be typed and the server will send the names of the connected clients.
- The ***eight important requirement*** is justified in the design of the run() method of the Client class. While listening for input from the server if the connection is lost, an IOException will occur. This exception is then caught and handled appropriately. A message is sent to the client that the connection is closed and the closeConnection method of the Client class is called. This method

closes the socket and the streams and also resets the flag fields of the client via the reset method of the Client class.

- The ***ninth important requirement*** is justified by the design of the client handler. When a client has quit in the middle of the game, the shutdown sequence will start for that client's client handler. This client handler then also checks whether the client was in a game and if this is the case it's opponent will receive the appropriate game over message. The methods to look up are shutdown and doGameOverDisconnect inside the CollectoClientHandler.

Explanation of the realised design



- **Board design:**
  - The board class implements the major part of the gamelogic. It plays an important role, if not the most important. The class can be inspected under four major functionalities:

1. Initializing a random board:

- The board initialization follows an interesting process. Before anything we get a list of balls which have eight of each six colors. This is crucial as we have to guarantee that we have to have eight of each six colors. Then we go through each field of the board, get a random ball from the list, check if we can safely put that ball to that field (there must not be adjacent fields with that color), and remove the ball which was placed from the list. If the ball which was chosen cannot

be placed on the field because of adjacency, we simply take another one.

This gives rise to an interesting question of whether there would be any case where it is impossible to place a ball without risking adjacency. Unfortunately such a case is possible if we place the balls randomly, and for this we have a method called impossible colors.

We would reach an impossibility if the remaining balls in the list are no different then the balls placed in the neighbouring fields. But there is a limit to where we can reach this impossibility. If we try to exhaust as many colors as fast as we can, we start risking the impossibility in row 5, column 5 and onwards.

So for those fields, before we even pick a ball from the list, we check whether we have come across an impossibility, if we did, we clear the board and start all over again.

Another check for a valid starting board is made by looking whether there is a possible single move, as if there is not we also start all over again.

The initialization is over if the impossible colors method was not invoked, and there is a possible single move.

2. Handling adjacencies:
   - To handle adjacencies after a move we use two methods. The first method, which is named get adjacency returns a list of integer arrays of size two whose first element relates to the row number, and the second element relates to the column number.

     The idea behind it is very simple. The method goes through each field and checks whether or not the ball is the same with the neighbouring balls. If it is, it creates a temporary integer

array, puts the row and the column to the array, and adds the array to the list which will be returned.

That list is then handled by the second method which is called handle adjacency. The method simply goes through the list which was returned by the first method, gets the balls which were adjacent and adds them to a list, which will be returned for the player who acquires them. The balls that were added to the list are then also changed with an empty ball.

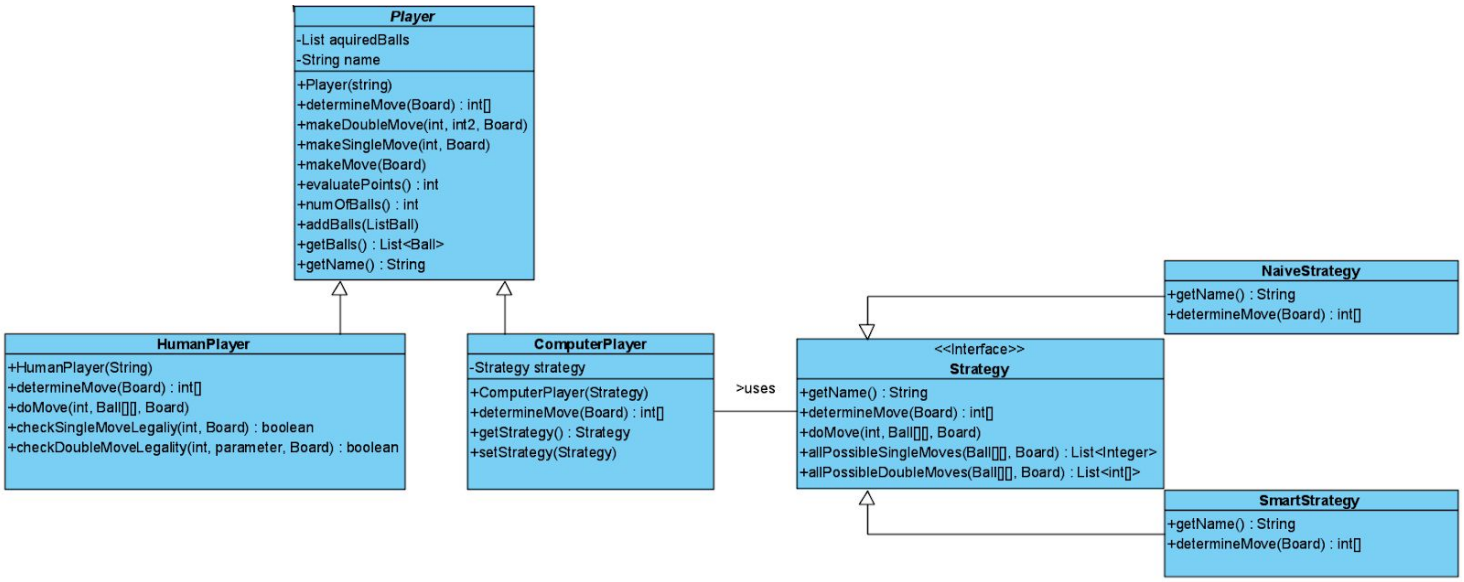
3. Making moves:
   - The moves are made by picking a row or a column, and also the direction. In case we want to move left or right we will shift the balls in the given row to the respective direction, without changing their order. The same idea goes for moving up or down.

     The four methods which are responsible for moving up, down, left and right simply go through the balls of the respective row or column and switch the place of a non-empty ball with an empty ball until there are no empty balls left in the way of the aforementioned non-empty ball.
4. Figuring out possible moves:
   - Before figuring out if there are any possible moves, we should find movable rows and columns. For this we go through the fields and store the indexes of empty balls, those rows and columns then by definition become our movable rows and columns.

     To figure out the possible single moves we create a deep copy before each move we make with the movable rows and columns we have just acquired, if after the move we get any adjacent balls, then it means that we found a possible move. We add all of the possible single moves to a list and then return it.

     To figure out possible double moves we do a very similar thing. We deep copy the board and move the movable rows

and columns and then call possible single moves. Because after our move if we come across new single moves, then we have just made a double move. We add their combination to a list and then return it.



Class diagram for the player and strategy packages

- **Player design:**
  - The players package of the project contains the classes Player, ComputerPlayer and HumanPlayer. They represent the actual players of the Collecto game. The Player class is an abstract one as we want to share functionalities between the ComputerPlayer and HumanPlayer and achieve reusability but there should never be an object of a dynamic type Player. We need the specified functionality of either a human or a computer one. The HumanPlayer class extends the Player class respectively and its purpose is to do moves on the Board class with an input from a user. The ComputerPlayer class extends the Player class and is used when the client/user indicates that he/she wants their moves to be played by an AI.

The Player abstract class has a name and a list of acquired balls. It uses the method makeMove to play moves on a given Board. The helper method determineMove gives the moves that can actually be played and depending on the return, the makeSingleMove or makeDoubleMove methods are called for a single and double move respectively. The interesting thing about the determineMove method is that it is abstract and its actual functionality is chosen by either a HumanPlayer or a ComputerPlayer. The design decision to evaluate the points of the player inside its class was also made. The method evaluatePoints takes care of that. It is also important to mention that the makeMove was only used to develop the early versions of the console game that did not have any networking features. In the client and server the functionality of the makeMove is implemented accordingly. The points are then accessible to the server for each player so that a decision can be made on the conclusion of the game.

The HumanPlayer class has the methods checkSingleMoveLegality and checkDoubleMove legality which check if a move/moves, given by the user, is actually a valid one. This design decision is important because the methods are independent of a Board and can be used in the client with its associated Board. They make use of a helper function doMove and also a copy of the Board that was given. The doMove initiates the move/moves and if there was no adjacency present the 'check' methods return false for the given moves. If there was adjacency present and the move/moves was indicated in the correct range defined in the protocol, it is ok to be played on the actual board reference in the client. In the HumanPlayer class the determineMove method was only used for testing purposes. It handled input from a user playing a console-only version of the game, without any network communication. Its purpose is strictly for development like the makeMove in the abstract class Player.

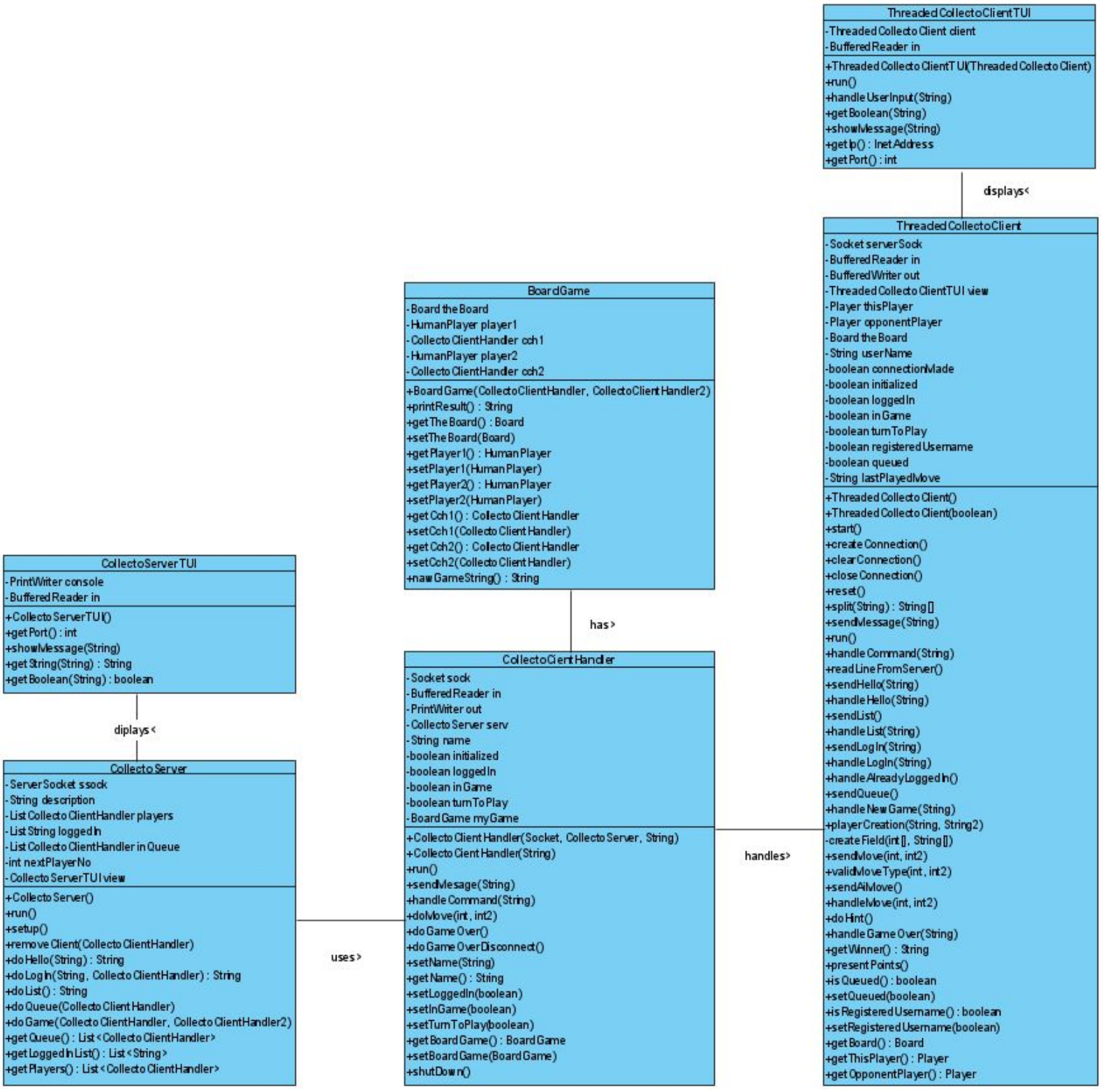
The ComputerPlayer class design consists of a strategy and a detemineMove method that uses the determineMove method of the specified strategy. Depending on the choice of the user, the ComputerPlayer can have either a Naive or a Smart strategy which will directly influence the return values for the determineMove method in the ComputerPlayer class. It is a simple design as all checks for the moves decided by the computer are made in the respective strategy.

- **Strategy design:**
    - The strategy package of the project contains the Strategy interface and the SmartStrategy and NaiveStrategy classes. The strategy is generally the brain behind the computer player and is always defined in the constructor of a ComputerPlayer.

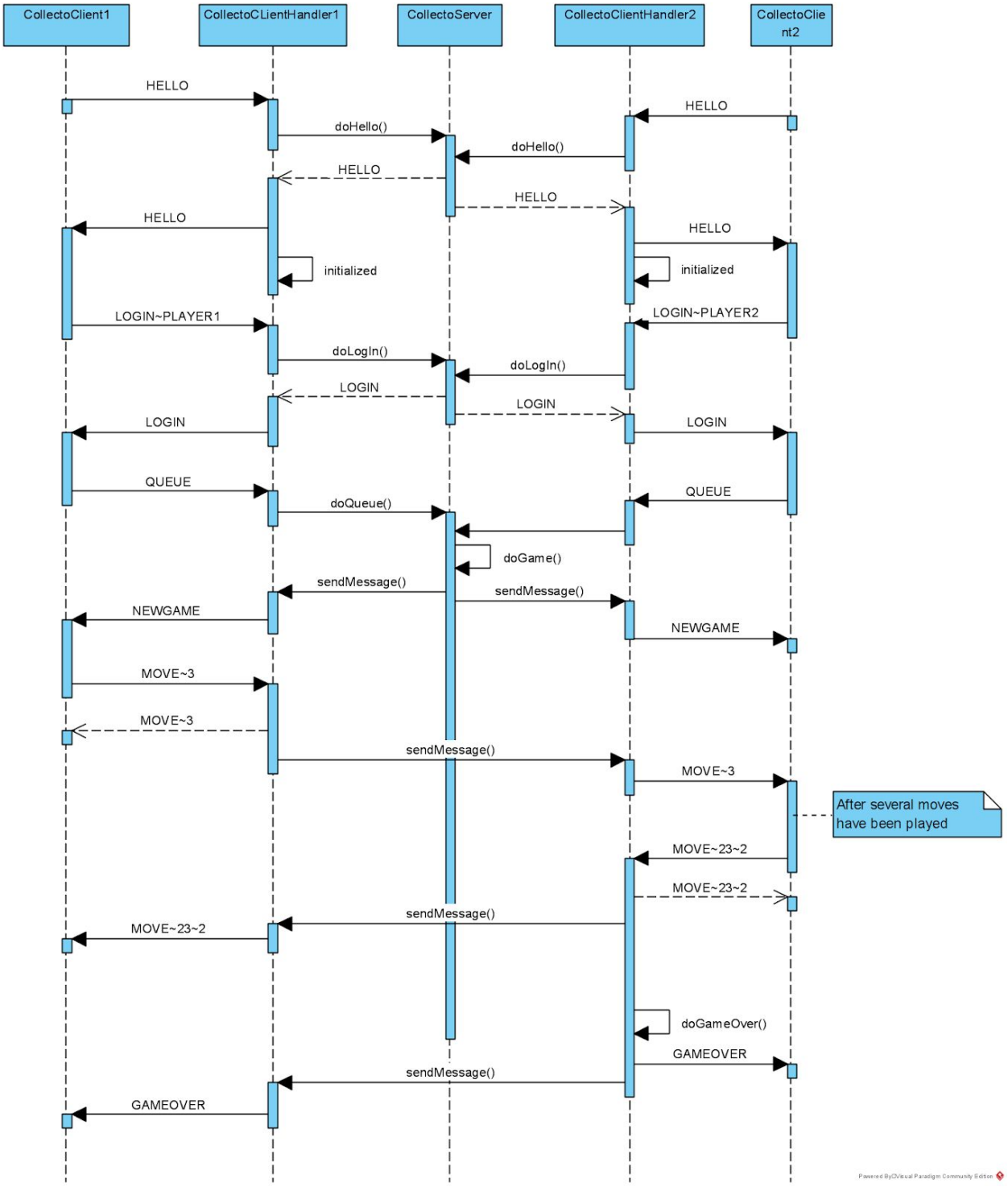
      The Strategy interface has default methods with strict implementations that are shared among the classes that implement it. Namely the methods doMove, allPossibleSingleMoves and allPossibleDoubleMoves. The checks for the moves still need to be made in a similar fashion. The class also has a determineMove method whose functionality is implemented by the two types of strategies.

      The design of the NaiveStrategy class that implements the Strategy interface is very simple as it only has one method for determining the move/moves. The basic idea is that a random possible single or double move on the given board is returned with the help of the default methods in the implemented interface.

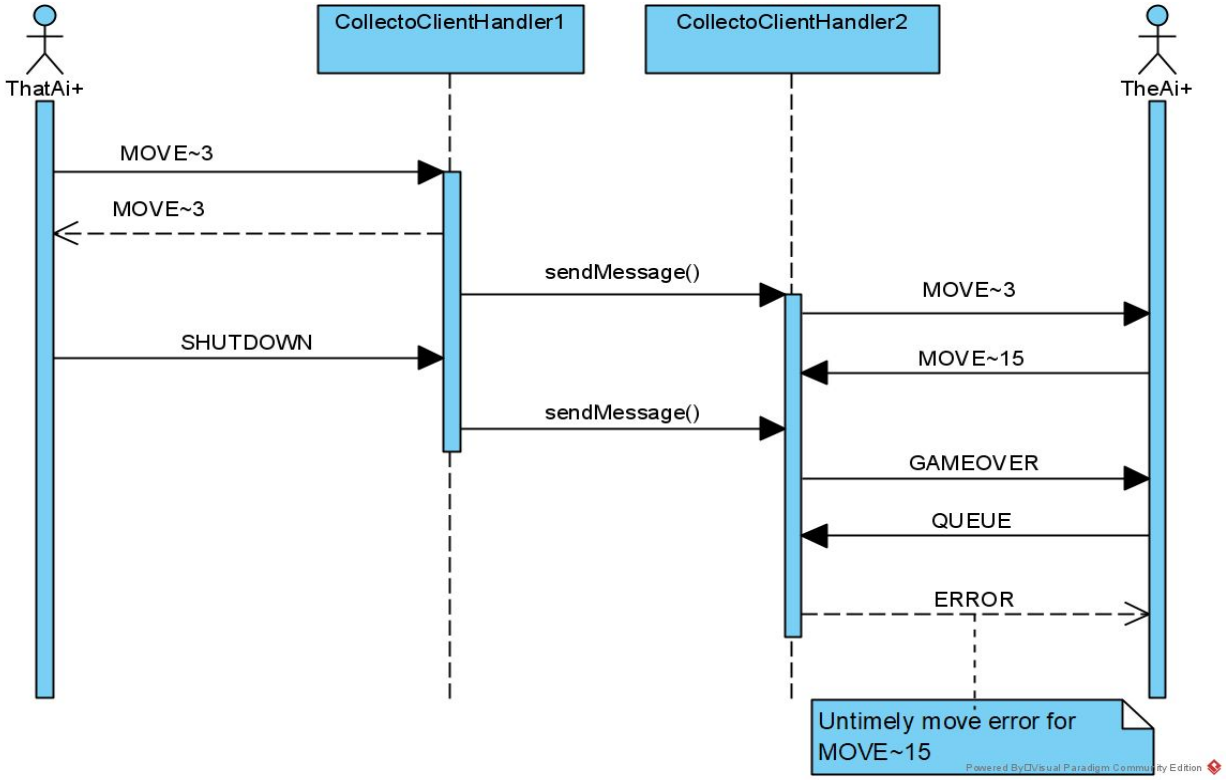
      The design of the SmartStrategy class that implements the Strategy interface even though having a better algorithm to determine the moves, has the same simple design. The copy of the board fields is made for every row and column of the board that have to be moved. Using these methods, the determineMoveMethod can get the possible move that results in most adjacent balls acquired. It stores a local copy of the previous best move and updates it every time the handleAdjacency function of the board gives more balls than the previous best that are also stored in a local variable.



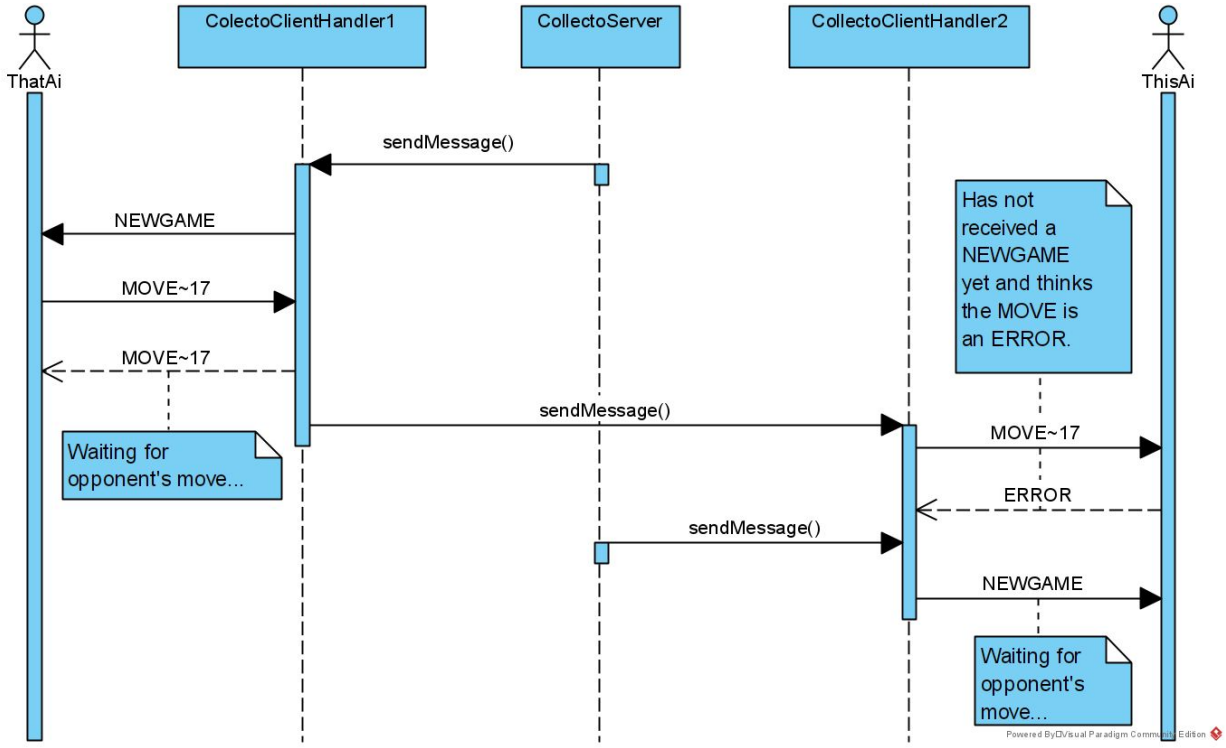
Class diagram of the network communication (relationships with the Board and Player classes are omitted)



First sequence diagram showing intended initialization and game communication



Second sequence diagram showing the rare untimely move situation



Third sequence diagram showing the late NEWGAME message

- **Client design:**
    - The client implements the Runnable interface. The run method simply listens to messages coming from the server and channels them to the appropriate handle methods, because this thread only works with handle methods from here on it will be called the handle thread.

      The main thread also executes a start method which starts the TUI in a different thread, whose run method constantly listens to user input and channels them to the appropriate send methods, because this thread only works with send methods it will be called send thread. A note should be made that this send thread is a daemon thread.

      The design choice behind this was made to handle a server shutdown. We want our client to terminate when the server shuts down and we want to do this gracefully. Thus using System.exit(0) or showing any stack traces is not acceptable.

      When the server shuts down the handle thread's read line will throw an IOException after which we invoke the closeConnection method which closes all of the communication pipes with the server. But this is not enough because the JVM will be kept alive by the send thread's readLine, but now when we change this send thread to be a daemon one the JVM will terminate regardless.

      Another interesting design can be seen in the closeConnection. This is the fact that we close the serverSock first. The reason behind this is that if the user types `EXIT` we come to this closeConnection method, and if in.close() was the first line, we would get stuck. If we look into the implementation of the .close() method we can see that it is synchronized on the same lock as the readLine(). Meaning that when the send thread tries to close it, the handle method, which is constant reading with readLine(), mutually excludes the send thread and blocks it. To workaround this issue we close the serverSock first and by doing so close the in and out pipes as well.

So with these two threads our client takes its general shape. The client also has a bunch of flags to make sure that the user is not trying to send any commands untimely. The client's handle methods also investigate the commands coming from the server and make sure that they abide by the communication protocol. The client's send methods investigate the flags of the user and the appropriateness of the commands. For example for moves, the legality is checked. Let's now dive into these handle and send methods to better grasp their functionality.

sendHello is used to send the first hello handshake of the client with the description. It also makes sure that the hello was not sent before. Here we have no extensions.

handleHello is used to process the server's hello response. It checks whether the initialization was already done, if not it checks the protocol appropriateness of the message and sends error messages back if the protocol was violated. If not, the view notifies the client and the initialization flag is set to true.

sendList is used to send the list command. It checks whether the client finished the initialization and the logging in, if not the user is warned. If however the client passes the check, the list command is sent to the server.

handleList is used to process the server's list response. It's timeliness is also checked in the beginning. And then the message is dissected and the user names are shown to the view. In any protocol violation the ProtocolException is thrown.

sendLogIn is used to send the login request to the server with the username. The initialization and loggedIn flags are checked and appropriate warning messages are shown to the view if the user fails the checks. If the user passes the checks, the login message is sent and the username of the client is changed.

handleLogIn is used to process the server's login response. It's timeliness is also checked in the beginning. If the check is satisfied the user is notified through the view and the appropriate flags are set to true.

handleAlreadyLoggedIn is used to process the server's already logged in response. It's timeliness is also checked in the beginning. If the check is satisfied the user is warned and will have to login again to complete the login process.

sendQueue is used to send the queue request to the server. The appropriate flags are checked and if any one of them fails the corresponding warning message will be shown. If the checks are successful, the queue message will be sent and the type of this queue message (queue request/queue exit request) will be checked and the respective message will be shown to the view.

handleNewGame is used to process the server's new game command. The incoming board's protocol appropriateness and game rule appropriateness is checked and if the checks fail, the correct exception will be thrown or the correct error message will be sent respectively. If the checks are satisfied, the board is created, the player creation method is invoked to create the players, the inGame flag is set to true and the game finally starts!

sendMove is used to send moves to the server. The flags, the move's type (by validMoveType()), and it's legality is checked. If any one of the checks fail, the user will be warned and will have to enter another move. When all the checks are satisfied the move is played by calling the player and the move message is sent to the server. This move that we've sent is also assigned to the flag lastPlayedMove (more about it in the handleMove).

sendAiMove is used to send moves that are determined by the AI. It is called automatically in, either the playerCreation or the handleMove.

handleMove is used to process the server's move confirmation (same with the flag lastPlayedMove) or the opponent's move. The given move is converted to it's protocol appropriate version and checked against the lastPlayedMove, if they are the same this means that we are handling the confirmation of our move, so we can terminate the method. But it may be the case that the opponent will play the same move we just played, so after we handle the confirmation move we reset the lastPlayedMove. The

following is very similar to sendMove, only the opponentPlayer is executing the moves.

handleGameOver is used to process the server's game over message. The first check looks at the local board, if the game over message does not include disconnect and the local board says the game is not over, then the server finished the game earlier. An appropriate error message is sent and the client is made aware of the issue. If the first check is successful, then we check whether the game is draw or it has a winner. In the case of a winner, we compare the local winner, if there is a discrepancy we assume that the local winner is the actual winner. In case of a draw, we check if there actually is a draw, and if there is not we show the actual winner to the view.

These methods form our client. But there is an additional thing our client handles. It is a very rare occurrence that can be realistically triggered by AIs playing games one after another. The interaction can be seen in the second sequence diagram we have drawn. Basically if an AI sends its move really fast and misses the game over message, it will get an ERROR from the server. In our handle command this would trigger the case which handles the messages starting with ERROR. In this case we don't do anything. Basically our AI sent a move really fast right before the game over and it is not a big issue. Now we can't guarantee that this won't happen, as we can't truly synchronize with the server. But we can simply ignore it.

This behaviour was also realised with the reference client. When we were stress testing our server with a bunch of reference clients and closing them randomly, it was sometimes the case that our server sent an ERROR for an untimely move. The reference client simply ignored this message and queued again. Our client is doing the same thing in this very rare occurrence.

- **Server design:**
    - The server was an interesting part to think about. A lot of (a lot) ideas came and went. The crucial part was making the games run truly in parallel. Thus where we synchronize the moves on was an important decision we had to make.

Our workaround is to have a BoardGame, which is a simple encapsulation of the board and players, in the client handlers. The making of the moves are also handled by the client handlers. This way if we have enough cores, the games of different client handler pairs will truly run simultaneously.

But before we investigate the game dynamics further, we should also describe how we handle non-game commands.

doHello is used by the client handlers to send a HELLO handshake response. After calling this method, the client handler sets its initialization flag to true as well.

doLogIn is used to log in the client handler with a unique username. Thus as a parameter it also expects the username of the client and also the client handler which invoked it. The loggedIn field of the server which contains the logged in usernames, is used to check whether the username is unique or not. If it is not, ALREADYLOGGEDIN message is sent to the client. If it is, the username is added to the field, the name of the client handler is updated, and its loggedIn flag is also set to true.

doList is used to create the LIST message to be sent to the client. It simply concatenates the usernames while putting the delimiter in between and returns that string to the client handler.

doQueue also utilizes a list field called inQueue. This method is called by the client handler if the client sent the QUEUE request, and is not in a game, and is initialized, and is logged in. The method checks whether the client handler which invoked it is already in the queue, meaning that this is an exit queue request. If it is already contained the client handler is removed from the queue. But if it is the first time for the client to queue, it is placed in the queue, and the length of the queue is checked. If the length is two the doGame method is invoked and the clients are removed from the queue.

doGame simply creates a game for the two client handlers it gets. It utilizes the BoardGame class and it creates players with the names of the client

handlers. The players are always HumanPlayer as we will always check for the legality of their moves. The first client to play is selected at random. And appropriate flags are set in the client handlers.

Now another noteworthy thing here is that we send the new game message inside a synchronized block which is synchronized over the boardGame. This was a very tricky thing to realize. The reason for this design choice (or its motivation) is shown in the third sequence diagram we have drawn. Basically a possibility is that one of the clients gets the new game message but the other one is late. And the early client sends a move, after which we send the move confirmation to both of the clients. But as far as the late client is concerned the game is still not started, so the client gets confused, which confuses the server and we are in a bad spot.

But one thing we can realize is that the early client sends a move to the method doMove inside the client handler which is synchronized on the board game that was created in the doGame. So if we surround the sendMessages in the doGame with a synchronized block synchronizing on the boardGame we make sure that we have sent the message to both of the clients. So the early client's move won't be able to execute the doMove if we are still in the process of sending the new game message.

After the game creation, moves and the gameover situation is handled inside the client handler. The doMove method checks the client handler's flags and the legality of the sent move. If the checks pass the move is executed on the myGame field which the client handler shares with its opponent.

After each move, the move's confirmation is sent to both of the clients, and the game over situation is checked. If the game is over the result is sent to both of the clients and the appropriate flags are changed.

In case of any disconnection form the client the client handler invokes the shutdown method which checks if the client was in a game. If it was, the client handler sends the appropriate game over message to its opponent.

With these methods and measures for thread-safety we formed our server.

- Concurrency mechanism
  - The project operates with threads in the client and server components. As mentioned above the client has two threads, one which listens to server named handle thread, the other which listens to user input named send thread.

    This separation of handle/send is the first step to thread safety as the threads do not race over any shared methods. The exception to this is the closeConnection, sendMessage, and setQueued methods which are synchronized.

    The run method of the client first gets into a while loop, the reason behind this is that we should wait until the connection is established and then start listening to the server. There are several such flags which are shared between threads. These variables are also volatile so that no thread caches it and disrupts thread-safety.

    Another concurrency mechanism that is utilized is the fact that the send thread is a daemon thread, the motivation behind this was explained in the realised design.

    The while loop inside the client's run is waiting for the connection to be established, which can become a problem if the user wants to exit the program after a failed connection attempt. In the createConnection method it can be seen that we set the connectionMade flag to true after the client refused to make another connection attempt. This termination sequence is an interesting one. As the handle thread is not a daemon one we have to make it escape the while loop and catch the correct exception that will be thrown (null pointer) because of this escape, and terminate the program gracefully with closeConnection().

    On the server side of things we have a thread which accepts and creates the client handlers and client handler threads. The concurrency mechanism is focused on these client handlers.

Server provides some methods for the client handlers. These methods should be accessed in a thread safe way. Meaning no two client handlers should access the same method at the same time. As that can disrupt the reading from the lists, also adding or removing from the lists. Another important concurrency mechanism is found inside the doGame method's synchronization block surrounding the sendMessage method calls, the reasoning behind this was explained in the realised design.

The thread-safety for the client handlers are achieved by synchronizing the doMove, doGameOver, and doGameOverDisconnect on the game instance. With this, these methods won't be executed in parallel. This is important as we wouldn't want to execute a move while we are in the middle of executing the doGameOverDisconnect because of an immediate disconnection after a move. Moves being synchronized over the game also allows for truly parallel games between client handler pairs.

- # Reflection on design

The initial design of the project was the first and probably one of the most stressful parts of the project. We were finally faced with the reality of the task ahead. And we did not know what to do. Nowhere to start from, nowhere to end it. The idea of a class diagram stressed us a lot and the one of a sequence was just not imaginable. One thing that was at least more concrete was the Board class. The idea behind it was almost shaped by the time we had to think of initial design. Finally a decision was made to mix the design of the TicTacToe game and the Hotel application developed in earlier weeks. One for the game logic and the other for the communication which led to the initial class diagram for our project.

Looking back, the overall structure was not far from what we eventually delivered as the final product. That being said, there are also many small and some big design choices that were implemented and differentiate from the initial idea. Let's start with the similarities. All of the classes except two made the final design. Also the communication between them remained the same. There is a TUI for both the server and the client that display to the user updates in the system and also manage the input. The server uses client handlers to manage connected clients. The player package stayed largely the same. The Human and Computer player

define the two types of Player and the Computer player makes use of strategies. The Board class also did not face many changes, it still uses the Ball enumeration class and most of the methods mentioned in the initial design.

Eventually, here were the first design changes. The method integerToRowCol was found unnecessary. The first idea was to store the adjacent balls as integers and then convert these integers to the actual location in a row and column form. The method that eventually is used does not need a conversion and the getAdjacent method of the class stores in a list of int arrays the row and column of the adjacent balls. This is better because an unnecessary step is skipped. The movePossible method was also scrapped as there was no practical need. possibleSingleMove and possibleDoubleMove are mostly used separately so a general method was not needed. Another major difference is the removal of the Game class. Following blindly the design of the TicTacToe, this class was thought to be central in the Collecto application. It did serve its purpose for creating a non-networked console version of the game. It helped the development of the players and the strategies. But eventually was found unnecessary as the Board and the Player classes are used directly in the server-client application. The Game class was initiating moves from the players until the game was over but now the end of the game is decided by the client handlers and also other functions were distributed along other classes. Something of a sequel to the Game class is the BoardGame class that is eventually used in the design of the client handler. It has two players with their respective client handlers and also a board. This allowed for a better and easier distribution of the games among client handlers. It is very easy to see which client handler is responsible for which game and vice versa as there is a BoardGame field in the client handler. The CientHandler itself has also undergone changes. The doMove and doGameover methods were moved from the server as it was needed to synchronize them on the board game instance to ensure the right concurrency mechanism mentioned above. There is also the addition of many flags and their appropriate getters and setters to ensure the correct flow of the commands and that no single command is sent before others or when it is just illegal. Similar flags were added to the client as well. The client was also modified so that there are “send” methods and “handle” methods to properly adhere to the concurrency mechanism instead of just “do” methods for both functionalities. This was a step towards better separation of concerns. An important decision was to make the client and the client TUI runnable objects which were not in the initial plan. This helped to solve problems with blocking and also fixed a hanging thread

issue. The Player class got additional functionalities which could not be thought of in the beginning of the project and one of the strategy types was removed.

If this project was to be implemented again we would spend more time on ensuring the right encapsulation. Many helper functions were added whose only purpose is to aid the functionality of more important methods but their visibility is mostly public. We could also spend a bit more time on the ModerateStrategy class so that the computer player is more diverse. The Game class could also be thought of in a better way so that it is used in the client. The Model-View-Controller pattern could also be implemented in a better way as now there is a mixing between the view and the controller in the TUI. Why we could not manage to achieve these improvements is because of the time pressure imposed on us by the deadline and prioritising the correct functionality of the game, the communication between client and server following the protocol and the thread-safety of the concurrent elements of the system before the aforementioned missing changes.

• System Testing

When we had a working client and a server, we started testing them together and also with references, as these tests are crucial for development. Some of the tests showed us several thread safety issues which were discussed throughout the report.

The system testing also made it possible for us to test connection establishment and disconnecting and handling these disconnections gracefully.

The first test was made by the first client we have written. We connected to the reference server and started executing the commands. What we realised was that we couldn’t queue twice which was in the protocol. This was because our client only had a single thread. We naively thought this issue would become a problem only here, but that was not the case. An unresponsive server would simply demolish our client as we were always reading a line, and that reading was blocking for the client. After these tests we decided on to do the Threaded Collecto Client and it was a success. And we learned another lesson from the old client: the importance of certain flags. To understand the timeliness of a command before

sending it to the server was a responsibility of the new client and it successfully achieved this.

Some of the tests that we have done to test the timeliness of the commands were as follows:

- Client tests:

After proper connection to the server, typing any commands (LOGIN, QUEUE, LIST, MOVE, HINT) before the initialization command (HELLO) would result in a warning that the user has not been initialized.

After the HELLO command it is not possible to initialize again, the expected output would be a warning that the user is already initialized. Only after disconnection the user can initialize with HELLO again.

Typing meaningless commands which would prompt a warning.

Typing any of the commands: QUEUE, LIST, MOVE, HINT before a LOGIN command would result in a warning that the user has not logged in.

Typing LOGIN twice for the same user would result in a warning message that the user has already logged in.

Typing LIST after logging in would give a list of the currently logged in users.

Typing the QUEUE command once will result in a “Queue request sent” message and typing it a second time will result in a “Queue exit request sent” message. This could not be possible without testing the second QUEUE request on the non-threaded client and seeing the blocking result.

Connecting two clients to the server, initializing and logging in should not result in a problem. The two clients are able to join the queue with the QUEUE command which will lead to the start of a game.

During a game, typing any other commands except MOVE, EXIT, HINT and LIST will result in a warning message.

Typing illegal moves with the MOVE command will result in a warning, the illegal move will not be sent to the server to disturb the game flow. A valid move will be expected from the client.

Typing MOVE without numbers but with random things which prompted appropriate warning messages.

Typing HINT when it is this user's turn to play will result in a hinted move message.

Typing any command except LIST when it is not this user's turn will result in a warning message.

After a game the user can rejoin the queue and wait for a new game by typing QUEUE as he is still connected to the server.

Closing the server at any point in time and making sure that the client terminates gracefully.

Trying to connect to a server which is non-existent either because the IP was incorrect or the port, and making sure that we get an error message and have a chance to retry to reconnect again.

Connecting many of our clients to our server and making them play games one after another.

- Server tests:

We connected a bunch of reference clients to test whether multiple games can be played in parallel.

We disconnected clients at random places. To make sure that the server handles such disconnects appropriately. For example disconnecting right before a game ends or disconnecting right after a game starts.

We sent untimely commands to the server and made sure that it sends error messages back.

We tested running multiple servers running on the port and got back a warning to pick another port appropriately.

Stress testing with many reference clients which automatically queue after a game over.

- # Overall Testing Strategy

After calculating the overall test coverage of the JUnit test cases we learned that the result is 79% coverage. This excludes the classes and methods that were used in the early stages of development (Collecto.java, Game.java, TextIO.java, determineMove method of HumanPlayer.java and makeMove methods of the Player.java) and classes that were aborted (ThreadedCollectoClient.java, and ThreadedCollectoClientTUI.java). The JUnit tests utilized special constructors for testing the client and server, and pre-set boards to test players.

ThreadedCollectoClient.java and ThreadedCollectoClientTUI.java were tested 64% percent. What was not tested with JUnit was the methods responsible for connection handling and parts of the methods dependent on the connection. The proper testing of these parts was done during system testing.

The server package falls a bit short with JUnit testing but this is compensated with extensive system testing such as stress testing with multiple reference clients and also ThreadedCollectoClient.java clients. Also many untimely and unreasonable commands were sent to the server to ensure proper handling.

That being said, the focus on unit testing was on the proper command handling on both sides. Special constructors were used to achieve the desired results in the server and client classes.

The player package has 82% test coverage. This is because of the intentional exclusion of the makeMove and determineMove methods which account for roughly 220 instructions.

The game logic package was the second most extensively JUnit tested package with 99.1% coverage. Almost all edge cases of the Board class were covered ensuring the proper handling of the game rules.

The strategy packed was the most extensively JUnit tested package achieving the coverage of 99.6%. All of the AI algorithms and their edge cases were tested to ensure the proper behaviour of the computer players.

All of the unit tests cover their respective part of the system thus no filling in for other tests are required.

What the unit tests did not cover were the connection, game dynamics and fine grained thread safety of the server and client components. These were thoroughly investigated with integration and system testing. Numerous stress testings, feeding bad inputs and intentional disconnections at precisely chosen places in the game dynamics and thread risky areas on both sides were conducted. These risky areas were explained in the realised design and also modelled with the aid of sequence diagrams. This way integration and systems tests complemented the unit tests, giving us confidence with our overall testing strategy.

● Reflection on process

Alex Petrov:
This project was really something. When I first saw how much time was allocated for the development of everything I thought to myself: ‘what is there so much to do for 4,5 weeks?’. I could not really grasp the scope of it. Then after the initial planning, things got clearer for me and I became confident in the process we designed. Now, standing on the finish line, I can say that our initial planning was very realistic. A reason could be the head-start work during the winter break. There were some crossings between the different sections but those were to accommodate the situation that was at hand. One example is the start of the report during the *Implementation and Testing* phase. It was not our intention to start the

report this early but because of different circumstances this was chosen to be the best fit for the time. The ‘documentation along with implementation’ was not strictly followed. I do not think this was a big problem but probably documenting and testing more WHILE implementing the code could be better for the workflow. I have to say I am impressed with the collaboration process. My partner and I managed to solve each problem with a discussion and also respected each other’s opinions on the project work process and the project development itself. There were personal circumstances that could lead to stalling of the project but one of us was always backing up the other. One thing that could be better for future projects is the initial mindset. For the future we can be more composed and concentrated in the beginning to not let the stress of the unknown lead us. Also implementing more industry-recognised version control techniques such as GIT can elevate our workflow.

Kağan Gülsüm:
This project was one of the biggest steps I have taken to become a computer scientist, without a doubt. We had the tools and knew how to use them, but the project was simply a different kind of problem. The answer to it was not in the appendix of a book.

It was our responsibility to think about it and do it. This brought with itself a great amount of stress and literal fear. Starting the project itself, for me, was one of the hardest parts, as its size was blown out of proportion by my imagination.

My partner motivated me greatly and took great steps at the start. And after the gears started rolling we had better decisions and a better workflow overall. We simply should have been in this flow and concentration state in earlier times and this was the biggest lesson for me: Not to get crushed by the size of the problem and simply approach it systematically and calmly.

The initial design was a good systematic approach where we dissected the project into its components, and had a better grasp. Even though we have come a long way from the initial design and changed a lot, it gave us the basic framework to build upon.

In the process of building upon this framework me and my partner had a supportive cooperation. We critiqued each other and worked together to understand and solve the problems we have encountered. After seven weeks of working on different problems we knew each other well and played to our strengths.

The thing which we both failed to capitalize on was the automated testing, in my opinion. We could have done a better job with creating and applying tests. We should have practiced test driven development more. But this was one of the several other lessons we learned. The importance of testing a lot and testing frequently was definitely realised.

27-01-2021
Alex Petrov (s2478412) and Kağan Gülsüm (s2596091)