Software Development (202001064)
Programming (202001066)

Programming pair 38

Programming Report – UNO

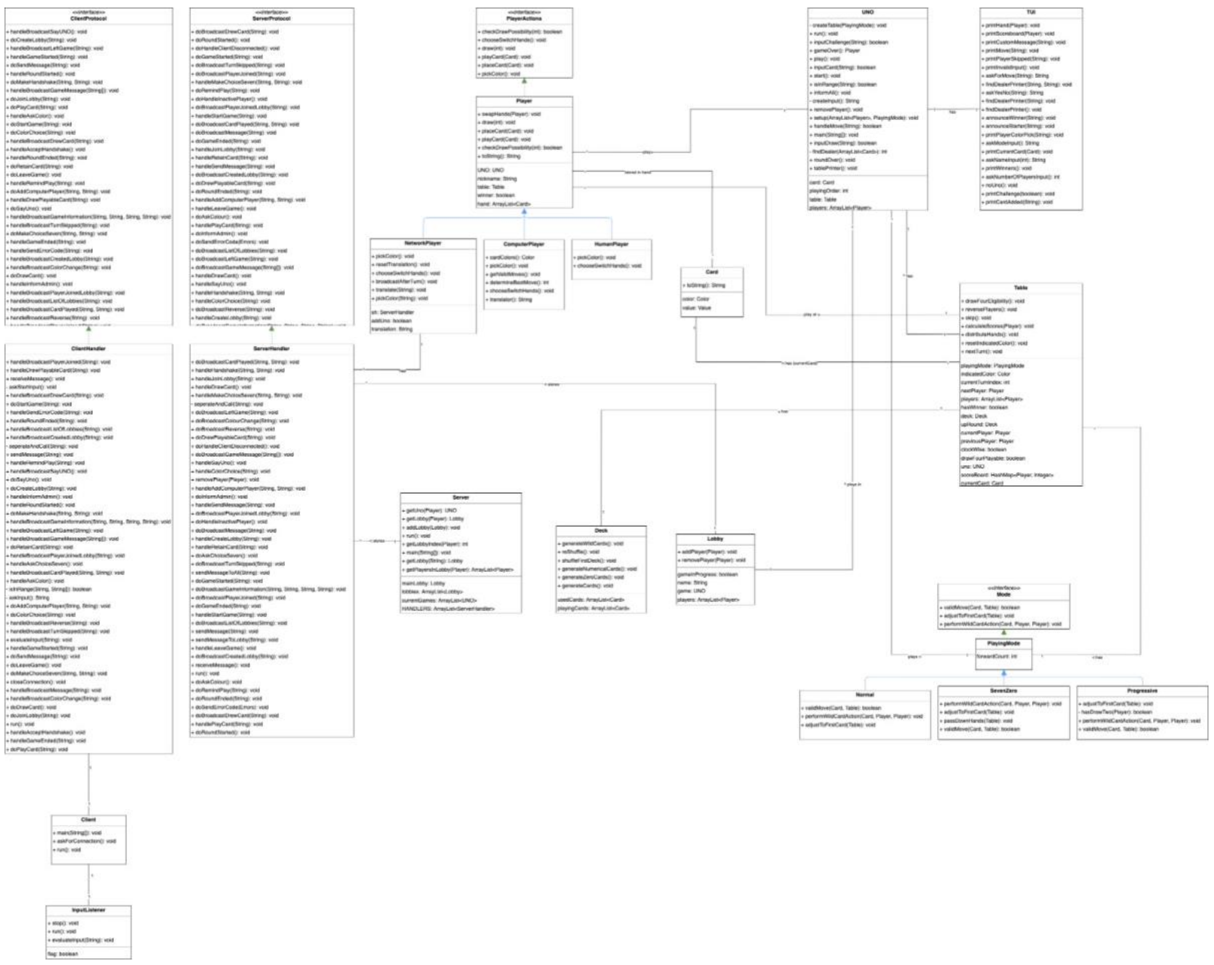
Tom Hansult, s2993074, t.hansult@student.utwente.nl
Gracjan Chmielnicki, s3077489, g.s.chmielnicki@student.utwente.nl

1/February/2023

Table of Contents

Overall Design

Class Diagram



For better view please refer to documents section of our codegrade submission

The class diagram above illustrates the class structure of our project. It is important to note that the ServerHandler and the ClientHandler classes are the implementations of the respective protocol, being the reason, these classes have so many methods. The structure of the project will be discussed in great detail in the following section of the report.

There are two main packages, the server package and the client package, which are both structured according to the MVC pattern. Let's first consider the server package, as it also contains the "local" version of our game.
The model contains multiple packages. Firstly, the Card package and class. It holds two enums Color and Value, that are constructor parameters of the Card class. The Deck class holds Cards in two ArrayLists, the discardPile, here: usedCards and the drawinPile, here: playingCards. The Deck constructor initalizes the two ArrayLists by generating Cards, according to an Uno Deck. The class has the method reShuffle() that reshuffles the usedCards when there is no more cards to draw. The abstract Player class implements the PlayerActions interface, and provides implementation for the useful methods such as isWinner(), playCard() or draw(). The HumanPlayer class extends Player and represents a local player in our Uno. The constructor takes a nickname as a parameter and invokes the

super constructor. It implements pickColor() and choosSwitchHands(), used in the SevenZero gameMode. The ComputerPlayer class also extends Player, takes a nickname parameter and invokes the super constructor. The ComputerPlayer collects all possibleMoves in an ArrayList and determines a move, keeping Wild Cards for the end of the game. It picks the color with the most occurrences in its hand. The third child of Player the Networkplayer builds a bridge between the local game and the server. The NetworkPlayer has the property translation and the method translate() , which translates the message we receive from the protocol, indicating the move of the cleint, to the input form we use in our local game. To be exact, from a String with Color and Value to the corresponding index in the hand of the player, and sets the NetworkPlayer's translation property accordingly, this required a synchronization mechanism which will be elaborated on in the Testing.Bugs section of this report.
Furthermore, the abstract class PlayingMode implements the Mode interface. The Normal class extends PlayingMode and implements the validMove() and performWildCardAction() methods according to the regular Uno rules. The Progressive class, also a child of playingMode implements these methods differently, as in this gameMode you can forward Draw 2 Cards. Likewise, the SevenZero class implements the methods allowing you to pass down hands or swap cards with another player when a 0 or 7 is played, respectively. Lastly, the table, the most complex class of the model, takes the players and the playingMode as a construction parameter. And holds the Deck in addition to several useful properties, such as the currentCard or the currentTurnIndex, the direction of play or the scoreBoard. Its non-default constructor sets up a round by distributing the hands and setting a first card. It provides useful methods such as reversePlayers(), nextTurn() or calculateScores() taking the winner as an argument. The Lobby Class is the implementation of the Lobby extra feature, it has a name, the constructor parameter of Lobby, players and a game and a boolean indicating if the game is in progress. Since each lobby holds a game (here: UNO) property, this also allowed for the implementation of simultaneous games, where each lobby can play their own game.

The controller contains the UNO class, which is basically our Game controller, where the magic happens. Its method start() will, with help of the TUI, ask the user for the starting arguments such as the amount and name of players. The method setUp creates a table object, finds the Dealer by determining who drew the highest card and sets the playingOrder. The play() method contains several while loops, each representing a game, a round, and a move respectively. At the be
The main method calls these functions in the same order as described above. Each move uses the createInput() methods which asks the Player, depending on what child of Player he is, for his input (move). The handleMove() function then evaluates this inputs and returns true or false, meaning whether this move is completed or not. It calls several further functions each written for a different type of input for example "draw", "skip" or "0". Furthermore the Uno class has methods gameOver() and roundOver() which check whether the game or the round is over, in which case it would break out of the respective while loop. The Uno Class implements Runnable, the run method calls the play(), so when this Thread is started Uno has to be set up previously, as when we start it using Thread.start() instead of running the main function, this is done by the server that does not want to ask a local user for the starting input in the console, but a client connected to the server. The controller package also contains the classes Server and ServerHandler. The Server continuously listens for new Clients that want to connect, once they do a new ServerHandler Thread is started and added to the list of ServerHandlers that each correspond to a Client and a ClientHandler. As the Server- and ClientHandler communicate via the protocol the ServerHandler evaluates received messages and calls the appropriate "handling functions", more about each of the commands and the methods handling them can be found in their JavaDoc comments, it provides useful information about the purpose of each method. The ServerHandler has properties like, correspondingPlayer or Lobby, indicating the Player

representing the connected Client in the local game and the lobby this player is in and implements the ServerProtocol and the Runnable interface.
The view package is the third part of the MVC trident in the server, its purpose is communicating to the user, by providing him a User Interface, in our case a Textual User Interface (TUI). And asking the user for input, the controller UNO holds an object of the TUI class and can thus call its functions printing the right output and asking for user input.

As stated above, the client package is also structured according to the MVC pattern, the model here, is empty however, as the client connects to the server which hosts the game and holds the data. The controller package contains both the Client and the ClientHandler classes, the Client tries to connect to the server and if successful starts a new ClientHandler thread. The ClientHandler receives messages from the server translates them in a way it is understandable to the user and then presents it to the user using the ClientTUI. Moreover, using the ClientTUI it will ask for input, when necessary, for example when the client should make a move, or choose a color.
The client's view package contains the class ClientTUI, which is used by the controller to print messages to the user and take input from the user, in the same manner as the server's TUI that is used for local play.

Lastly, the test package contains all Junit test classes, their purpose will be discussed in further detail in the Testing section of this report.

Concludingly, as we were very content with the local version of Uno that we built, offering non required features like saying "uno" or challenging the play of a Draw 4 card, we wanted to find a way for the local version and the networked version of this game to coexist and connect. The NetworkPlayer bridges the two by translating input from the network side to the "language" that we use in the local version of our game.

Requirements Implementation Map

You should be able to play UNO with at least 2 people over a network with a client & server application:
This requirement is implemented by several classes, as it consists of establishing a network connection and using this connection to play a game of UNO according to the defined protocol. Thus, this includes classes neccessary to establish a connection and comminucate via the protocol while playing the game using this connection, therefore first of all the Server, ServerHandler, Client and ClientHandler and furthermore lobby, which will be elaborated on in the bonus feature subsection of this section implement this requirement. Also, the ClientTUI is necessary to provide a user interface for the client connected, so that he can actively participate in playing.

The client should be able to connect to a server, play a game and announce the winner in the end.
This requirement is implemented by the three classes in the client package, the Client the ClientHandler and the ClientTUI. The client establishes a connection to the server and starts a new ClientHandler thread with this connection. The ClientHandler will use the ClientTUI to inform the user about the current state of the game, also announcing the winner in the end and to take input from the user for hi, to be able to participate.

The server should be able to host at least one game with 2 players, following the rules of UNO, and determine the winner in the end.
This requirement is implemented by several classes, as it consists of playing a game of Uno that is hosted by a server and accessed by a client. Thus, this includes classes neccessary to establish a connection and communicate via the protocol while playing the game using this connection, therefore first of all the Server, ServerHandler, Client and ClientHandler and furthermore lobby, which will be elaborated on in the bonus feature subsection of this section implement this requirement. Moreover, for the server to be able to host a game, it will host a local game, thus all classes in the model and the controller package are used to implement this functionality too. The view and test package are not included here, as the Client provides its own User Interface and the tests validate the written code, but do not have any role in playing/ hosting a game.

**It is required to have a UI in the client in order to play the game. This may be a TUI, and
it doesn't matter what it looks like.**
This is implemented by the ClientTUI. The ClientHandler receives and evaluates messages and therefore uses the ClientTUI to inform the client about the current state of the game, important events and notes that he should be aware of, as well as ask for his input.

Your application / codebase should be structured as an MVC application.
This is implemented by all classes, as the MVC pattern gives guidelines for the structuring and factoring of the project. The exceptions here are the test classes inside the test package, as they are not covered by the MVC structure.

Your application should handle common exceptions and errors correctly during running (connection loss, invalid input)
This is implemented predominantly by the controller packages in both the client and the server, as they are responsible for handling and evaluating input and changing data in the model accordingly. Therefore, these packages implement handling common exceptions and errors that occur while running the game. Namely, the classes implementing this requirement are the ServerHandler, UNO, Client and ClientHandler classes.

Computer Player, you should have a computer player to play with. This player should at least only do valid actions.
This is implemented by the ComputerPlayer class, the computerPlayer traverses through his hand and for every card checks if it is a valid move, then out of the possibleMoves he selects one. He picks a color based on the color that occurs most often in his hand.

EXTRA FEATURES

Progressive:
As described in the Global structure of the project section of this report, we have an interface called Mode which is implemented by the abstract playingMode class, providing necessary methods and properties for a game mode in our project. The Progressive class extends gameMode and implements the methods performWildCardAction and validMove according to the rules of the progressive game mode. If a Draw 2 card is played and the next player does have a draw two card in his hand, he is given the opportunity to forward drawing the cards, this will increase the forwardCount in playingMode accordingly and the next player who cannot or does not want to play a draw two card is punished with the number of cards that have been forwarded and his turn is skipped. This also changes the implementation of the valid Move function, as in case the forwardCount is bigger than zero playing a draw 2 card is the only valid move, as otherwise your turn will be skipped, and cards added to your hand according to the forward count. To make this feature fully functioning, also some changes in our UNO controller where necessary, where we sometimes want to check for the current forward count, this however does not interfere with playing in normal or seven zero game mode, as the forwardCount will always be equal to zero in these game modes.

SevenZero:
In equal manner, we implemented the SevenZero class that also extends playingMode and implements validMove and performWildCardAction according to the rules of the seven zero game mode. In comparison to the normal gameMode, validMove does not change. PerformWildCardAction however gains two more cases in which an action should be performed, namely seven and zero. A zero will force every player to pass down their hands in the direction of play, this is implemented by the passDownHands() function of this class. To implement the functionality for playing a seven, we needed to add a method the PlayerAction interface called chooseSwitchHands() which works in similar manner as the pickColor() method, asking the corresponding player (implementation differs for each type of player.) for a choice on who to switch hands with, and after given this choice switches the hand, so performWildCardAction() can simply call this method in case a seven is played. Here, also this does not interfere with playing the game in other modes, as passDownHands() and chooseSwitchColor() will only be called in the performWildCardAction() of this playingMode.

Lobbies:
The lobbies feature had quite significant impact on how we create and join a game over the network. Firstly, we implemented the Lobby class, whose constructor takes a name as a parameter and initializes the players to a new ArrayList. It furthermore holds a UNO object, represeniting the game of this lobby and a boolean gameInProgress, that is toggled once the game has been started. Then, we implemented lobby commands on both the server and client side. For the ClientHandler this completely changed what we do before starting the game as the client now needed to be able to send custom commands to create or join a lobby or request a list of available lobbies. Furthermore, each lobby has an admin that can add computer players to the game and start the game in the gameMode of his choosing. This also required for some server-side validation, as there are multiple scenarios when you

should not be able to join a lobby, for example when the game of the lobby is already in progress or when the lobby is full. The server holds a list of all available lobbies, and the admin will be whoever creates a lobby. There is also always a main lobby by default which a player can join, in which case the first player to join will become the admin.

Simultaneous games:
Having implemented the lobby extra feature, helped tremendously with the development of this extra feature, as each lobby could host their own game and start and play it independently from the games of other lobbies. Therefore, each lobby has their own UNO property. When the server receives a command from the client to start the game, he can check which lobby this client is in and start the game of this lobby, while games in other lobbies remain entirely unaffected. Therefore, this extra feature is implemented in the Lobby class and the Server/ ServerHandler and Client/ ClientHandler classes, or the entire networking package, as well as in UNO as we made it implement Runnable.

Chat:
We did not create a separate class to implement the chat, we implemented it in the ServerHandler and ClientHandler classes. Since our protocol offered support for this extra feature, we could use the commands "SM" for sending and "BM" for broadcasting our message. The Server- and ClientHandler both work in similar manner, when they receive a message from the other side, they call the seperateAndCall function which separates the received string according to the defined delimiter and calls the appropriate "handler" function for this command. This allowed us to broadcast a message to everybody in the lobby, whenever we received a string starting with the "SM" command. On the client side we implemented it in such a way that if you type "SM|" + yourMessage it is not going to be evaluated or sent us game input but recognized as a chat message and can therefore be formatted and handled accordingly.

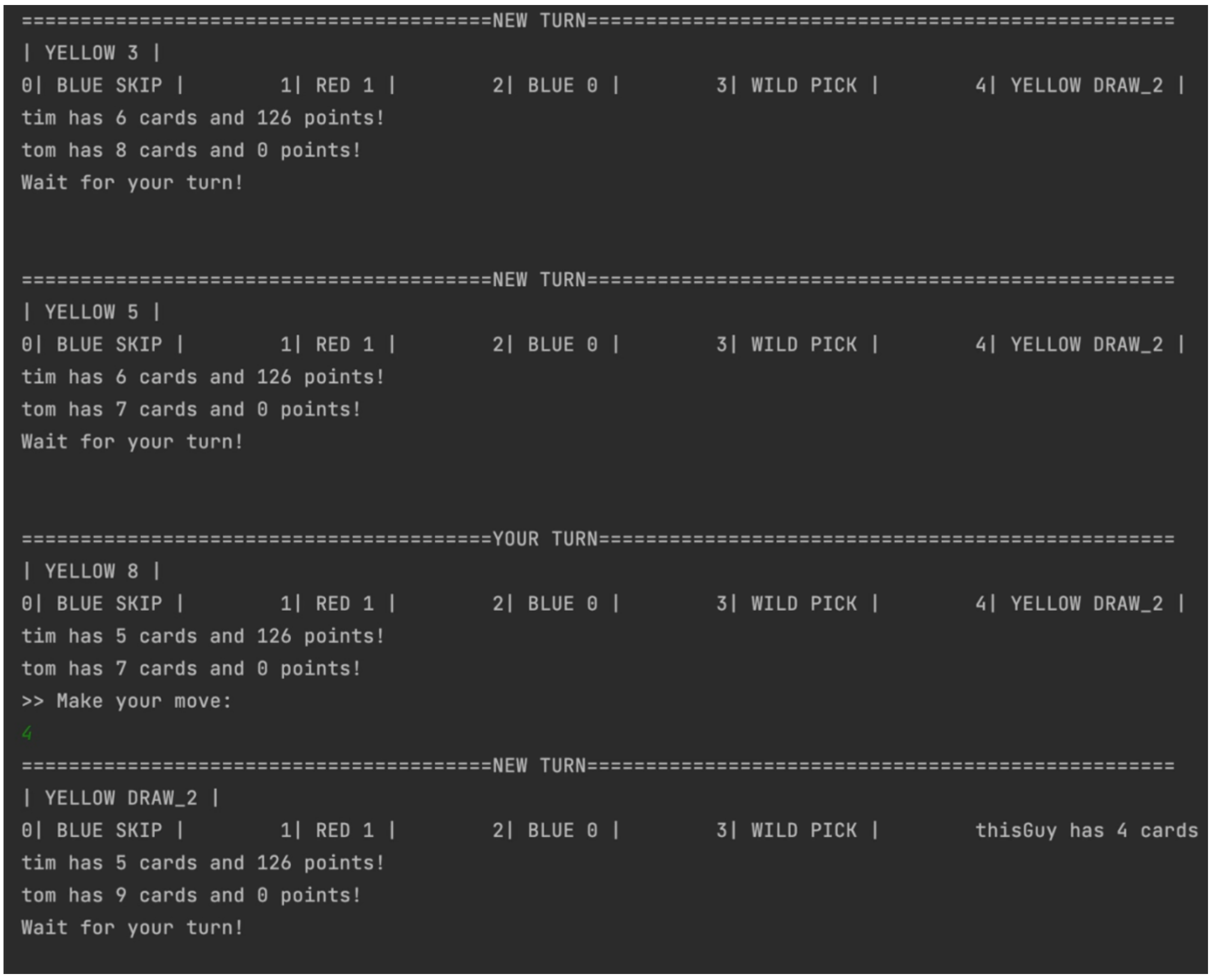
Model View Controller Implementation

Our project adheres to the model/view/controller structure. The basic idea here is that the model holds the data, the controller changes the data in the model and the view is responsible for providing an interface and taking input from the user.
There are two main packages, the server package and the client package, which are both structured according to the MVC pattern. In addition, the test package is not located in client or server, as it is not part of the MVC pattern.
In both server and client, we have packages according to this pattern, everything inside that package belongs to the corresponding part of the MVC pattern.
The structure and design choices have been discussed in the section above, this section will indicate which classes and packages play the role of the model, the view and controller and why that is the case. Let’s first look into the server package and its structure
The model package contains multiple sub packages that all represent the model in the MVC structure. It contains the packages Card, Deck, which contains cards according to a UNO deck. The player package contains the Interface PlayerActions, which is implemented by the abstract player class, the class has three childs: The HumanPlayer, NetworkPlayer and the Networkplayer. Furthermore, the model package contains the table package and class, the most complex class of the package, it holds the players and the deck (thereby also the cards) bringing all classes of the model together. Furthermore, the server's model package contains the Lobby class, which is part of the implementation of the lobbies extra feature. A lobby has a name, players and holds a game property and a boolean indicating whether the game is in progress, why that is the case is stated above in the extra features subsection. As stated in the beginning, the model holds the data that is changed by the controller, and offers useful methods to work with that data. Thus, all classes mentioned here that are

located in the model package of the project adhere to this guideline and thereby the given structure of the MVC pattern.
The controller package contains the UNO class, our game controller. Among others, It holds a table object and provides several methods all related to checking current states of the game and changing them by using the TUI or the network connection to evaluate input received from the user. The TUI is also used to inform the user about current events in the game. Thus, the controller package and specifically the UNO class changes the data in the model according to the flow and sequence of an Uno game and thereby follows the MVC guideline. Furthermore, this package contains the Server class, that allows clients to connect to it and then starts a ServerHandler Thread to deal with the client that connected. The ServerHandler class implements the ServerProtocol and the Runnable interface and is responsible for sending and receiving messages from/ to the client, which is why it is part of the controller packages as it receives inputs from the user, acts accordingly and changes the data of the model accordingly (also with help of the UNO and Server class).
The view package contains the TUI class, which provides the Textual User Interface. It is responsible for taking input from the user and printing information to the console about what is currently happening in the game, like whose turn it is, what cards you have in your hand and what the current card is. The methods of the TUI are called from other parts of the program, whenever applicable, as the TUI itself only provides taking input from the user and printing to the console, but does not account for, when which methods should be called, as this is mostly done by the controller. Thus, also the model adheres to the guidelines of the MVC strucutre and completes the MVC trident.
The Client package is structured in the same manner, having a model, view and controller package. The model of the client, however, is empty as the client only connects to a server, which holds the required data (model) to play a game of uno. The controller package contains both the Client and the ClientHandler in similar manner to the server, the Client tries to connect to a Server and once connected passes this connection to the ClientHandler which implements the ClientProtocol and Runnable. The ClientHandler receives and evaluates messages from the ServerHandler and uses the ClientTUI to communicate with the user. It then transforms the input from the user to the format of the protocol and sends messages back, which is why it is part of the controller package. The view package contains the ClientTUI, as mentioned this class provides the necessary user Interface to print the current state of the game, inform about important events in the game and asking for user input, which is why it is located in the client's view package.

Concludingly, both the server and the client packages are both structured according to the model/view/controller pattern. The test package is the only exception to this structure, as it is not part of the MVC structure and thus located in the src package, together with the client and server package.

TUI



```
========================================NEW TURN==================================================
| YELLOW 3 |
0| BLUE SKIP |        1| RED 1 |        2| BLUE 0 |        3| WILD PICK |        4| YELLOW DRAW_2 |
tim has 6 cards and 126 points!
tom has 8 cards and 0 points!
Wait for your turn!


========================================NEW TURN==================================================
| YELLOW 5 |
0| BLUE SKIP |        1| RED 1 |        2| BLUE 0 |        3| WILD PICK |        4| YELLOW DRAW_2 |
tim has 6 cards and 126 points!
tom has 7 cards and 0 points!
Wait for your turn!


========================================YOUR TURN=================================================
| YELLOW 8 |
0| BLUE SKIP |        1| RED 1 |        2| BLUE 0 |        3| WILD PICK |        4| YELLOW DRAW_2 |
tim has 5 cards and 126 points!
tom has 7 cards and 0 points!
>> Make your move:
4
========================================NEW TURN==================================================
| YELLOW DRAW_2 |
0| BLUE SKIP |        1| RED 1 |        2| BLUE 0 |        3| WILD PICK |        thisGuy has 4 cards
tim has 5 cards and 126 points!
tom has 9 cards and 0 points!
Wait for your turn!
```

Testing

Functional requirements

When a player draws a playable card, he should have the opportunity to play it immediately. We have implemented a test case in the Junit test class UnoTest that briefly tests this functional requirement. The test method is named testInputDraw and asserts that if you draw a playable card the uno inputDraw method returns false, so that you are given the opportunity to play this card. If the drawn card is not playable the function should return true, so the game can proceed to the next turn. Furthermore, we have tested this functional requirement using the TUI, as we wanted to ensure that this does indeed work as intended also in context of the whole game not just the isolated method.

A user should be able to pick a color after playing a wild card. This functional requirement was mainly tested using the TUI, especially given the networking context. There are also Junit test cases that test the proper functioning of setting/ resetting the indicatedColor to ensure that valid move indeed returns true whenever it is supposed to. In a networking context, there was further testing to be done, as we needed to put synchronization mechanisms into place that allow you to only proceed with the game, once the client picked a color. The fastest way to test this via a TUI was by automatically giving players a wild card on their hand, so that you do not have to wait until somebody actually has a wild card on their hand. Then it could be checked what happens if a wild card was placed, without synchronization mechanisms there were multiple clients asked for input, as the game already proceeded without waiting for the color to be picked. Thus, we introduced synchronization via wait() and notify() to our ServerHandler. Then we simply checked if all other clients in the game, are now waiting for the color to be picked.

A user should be asked to enter a valid move, in case he enters an invalid move. We have implemented a Junit test case that checks whether our handleMove function in the controller of our game indeed returns false if the entered input corresponds to an invalid move. The function is called inside a while loop thus, until you enter an input corresponding to a valid move you will be asked for input again. In Junit test class UnoTest the test method testHandleMove tests the return value of handleMove in case of valid/ invalid input. Moreover, we tested this functional requirement using the TUI as it enables you to test it in the entire context of a game not as an isolated method.

A user should be informed about crucial events in the game (turn has been skipped, Color was changed, direction was reversed etc.) This was strictly tested using the TUI, as this functional requirement is mainly about the user experience and what information he receives during a game. Firstly, we tested this locally by letting two computer players play against each other, then seeing what more information we would like to receive than what is currently printed to the console. After that, we compared the output on the server side (representing what would be printed locally) and the output on the client side, to ensure that we are providing the same information and updates about the game.

An inactive user should leave the game and the winner should be announced if there is only one player left. This functional requirement has been tested via the TUI. For testing purposes, we decreased the time a client has to make a move to 10 seconds, and then by connecting multiple clients, one can simply check the consoles to follow what is happening and if the player was correctly removed from the game (another player should now be able to input a move) and when you continue with the round the player should not be asked for a move again as he is removed from the players in the game.

EXTRA FEATURES

Lobbies:
Lobbies have strictly been tested via the TUI, by simply entering the commands to join/ create or request lobbies and then using print statements within the code showing the size of the lobby e.g to validate if a player did indeed join the lobby and games

Chat:
Chat also has been tested using the TUI, we mainly tested if the chat command works as intended, so that a chat message is not accidentally interpreted as a game command.

Simultaneous games:
This extra feature also has been tested using the TUI, we connected several clients to our server, made them join different lobbies and started multiple games simultaneously and then followed what happens over the TUI and validated if everything is going as intended.

Seven Zero:
We have tested this gameMode in different manners, firstly by having a Junit test class with test cases to test passingDownHands and swappingHands, the actions of Card values Zero and Seven respectively. Secondly, by having an entire game run through with computer players in this gameMode in the Junit test class System test and thirdly by simply playing games in this gameMode and follow thoughtfully what is happening.

Progressive:
We have tested this gameMode in pretty much the same manner as the Seven Zero gameMode. The corresponding Junit test class has test cases testForwardCount which tests the update of the ForwardCount (the amount of cards a player has to draw) after playing one or more draw 2 cards, furthermore it tests the updated validMove function of this gameMode, as you would originally be skipped if the previous player played a draw2, but having a draw 2 yourelf it becomes a valid move and you should be allowed to play it and thereby forward drawing cards to the next player. Moreover, just as for Seven Zero, we let two computer players play games and followed their moves, to see what might have gone wrong and on top of that also played games on our own, to better understand when exactly something went wrong and how to prevent that.

On top of this, we have developed Junit tests wherever we thought is necessary, to ensure that classes work as they are intended to. However, the majority of testing and debugging has still taken place with help of the user interface. As it allows to print debugging messages and follow what is happening after a specific move. To make this testing as efficient as possible, whenever we had a specific case, we wanted to test, for example drawing after a wild card to see whether the next player can still make a valid move. We adjusted how we distributed hands to enable us to always be able to play the card necessary for the given debugging and testing purpose.

Bugs we fixed and how?

One of the first bugs we encountered during development was that after a player played a wild card and chose the color, he wanted the game to proceed in, and the next player chose to draw and not play a card, the player after that was not able to play a card. After investigating exactly what happens with indicatedColor(table property that is changed according to pickColor) by printing its state to the console while playing we were able to track that indicatedColor was set to null before it got to the player that wanted to play a card. This was because we reset the indicatedColor to null every time nextTurn() is called, which should not be the case as proven by the example given above. By shifting the reset of indicatedColor to playCard() we were able to fix this bug.

Another problem we have encountered developing the Client(Handler) classe(s) was data race. The NetworkPlayer class translates the input received from the protocol, as Card Strings, to the corresponding card in the index, which we use as input in our local version. We had used wait() and notify() for synchronization in the first place, but while testing with the TUI and following which Thread, accesss what data and at what time, we quickly realised that wait() and notify() do not suffice in this situation and shifted to a synchronization mechanism using locks and conditions . This allowed better synchronization, because it enabled more control over when the data can be set or read. After further testing via the TUI and small adjustments to the locks and conditions we managed to implement a proper way to synchronize the getTranslation() and setTranslation() methods in the NetworkPlayer class.

Testing process of the most complex classes

The following section will elaborate on how we tested our most complex classes, it will discuss both the unit test approach and the system test approach. For more information about our Junit test classes, please refer to the detailed JavaDoc comments we have written for each test case. It explains our approach to each test case, and if applicable also the expected results.

Running the Junit tests with coverage resulted in 100% coverage of the test class its methods and its lines. The only exception here, is the UnoTest, which also achieves 100% coverage for the classes and methods, but for its lines only receives 98% which is due to a test case that tests drawing a card, since the card can either be playable or not it should be distinguished between two possible outcomes, this is done by asserting inside an if statement which results in 2% coverage loss, this is explained in further detail in the JavaDoc comments of testInputDraw function in the UnoTest class.
Furthermore, the test package coverage remains rather low if you run the test classes on their own, as they are mostly segregated from one another and targeted to test a specific class rather than the entire game. However, if you run them one after each other and add the coverage to the active suit instead of replacing it, it will achieve 100% code coverage for the classes and methods and 99% in lines of code, because of the mentioned test case in UnoTest.

UNO:
As briefly mentioned above in the subsection on testing functional requirements, the UNO class, the controller has been tested in three different manners. The first of which is by using a Junit test class, testing the basic functionality of the most complex methods of the UNO class. Secondly, we tested using The SystemTest class which runs through an entire game of Uno with two computer players. We could follow their moves and what exactly happens with the game in an entire run through, all additional information wanted for debugging like current values of variables etc. can simply be printed to the console and evaluated after. Additionally, we also tested just using the TUI and playing a game of Uno with multiple human players or a combination of both human and computer players, this was especially helpful when trying to recreate a specific event that might have caused a wrong outcome, as discussed in the bugs subsection of this section.

Table:
We developed a Junit test class to test the Table class, it provides test cases for the major functions of table, such as testReversePlayers() or testCalculateScores(). On top of testing this class in an isolated manner, Table was also tested with the TUI in combination with UNO. As described above, testing with the TUI and UNO and the SystemTest class, enabled us to follow every move and outcome of the program therefore also allowing us to test the Tables, because problems that occurred with it, that have not been discovered in the Junit tests, could be tracked back to the Table and the code adjusted accordingly.

ServerHandler/ClientHandler:
The networking part of this project was exclusively tested using the TUI. Multithreading imposed quite a challenge on this part of testing. First, simply trying to print values did not really provide an accurate and sufficient testing process to the multithreaded and networked part of this project. It required thoughtful placement and meaningful debugging messages to see what is going on and more importantly, what is going wrong. The classes have not been tested in isolation, but rather in context of the entire game using all its classes. Connecting multiple Client instances to the Server allowed us to track what is going on in the local version of the game (printed to the server console) and also on the Client side of each Client instance running separately. So, we could match the output, see if messages were sent but not received and vice versa or neither. Thereby, giving us great insight into where things were going wrong and where to look in the code but also validate when things were going just as intended.

Conclusion:
To conclude the testing section of this report, we tested our implementation in three different manners. Firstly, using Junit tests, we developed six Junit tests for selected classes and also developed a Junit test class called SystemTest which invokes an entire game run through by two computer players in each playing mode. These Junit test classes and their test case have been developed in thorough discussion about expected behavior and asserting possible wrong behavior, and helped reveal some methods that were not entirely correctly implemented but needed some further adjustments. Secondly, we used the TUI to follow a game run through of two or more computer players while printing debugging messages whenever applicable, allowing us to follow exactly what is happening and at what time. Lastly, we used the TUI to test playing as a humanPlayer with other humanPlayers or in combination with computer Players.
It is important to note that most of our testing was done via the TUI. The networking part for example was tested entirely using the TUI and printing debugging messages, as it provided quick feedback on what might have gone wrong and where and allowed for quick adjustments to be made mostly in Server- and ClientHandler and evaluate if these adjustments did indeed succeed in solving the problem at hand. This has been the major reason why our main testing mechanism was via the TUI and not using Junit tests. However, sometimes Junit tests still came in very handy, when we wanted to test things with a predictable and expected outcome, but this only applied to the local version of our game. As the ability to predict and expect exact values has been curbed by implementing networking and multithreading.

Academic Skills (first student)

Time Management and Procrastination Avoidance

Documenting the first weeks of this module for the Academic Skills assignment, has helped me reflect on my time management, work ethic and procrastination, but also my confidence in the programming topics at hand. One conclusion I draw from this documenting journey, was that simple to do lists work best for me as a time and task management mechanism. If I feel overwhelmed by all the items currently on my to do list, i simply check my calendar and divide the tasks among the next days so that todays to do list gets a lot easier to cope with. Keeping this in mind for managing time and tasks for this project also in regards to collaborating with a fellow student, has helped me tremendously to always keep track of what needs to be done and when. Looking back this has worked out well and if I were to do a similar project again, I would again choose a to do list that is backchecked against my calendar at times as my time and task management mechanism again. Therefore, I did not really need to adapt my approach to time management and stuck to the mechanism that worked for me in the first weeks of the module.
Coming to time management in the project, overall, I am quite content with how me and my partner managed our time during the project. First orientation of when, what should be achieved were marked by the two milestone in December of 2022, the first being the submission of a class diagram and stub implementation which we managed to do before it was due, which was a first positive indicator of proper time management. The second milestone then enhanced this indicator, as the requirement was to have at least one gameLogic function and a unit test of that, by the time we submitted the second milestone we already had implementation of the majority of the gamelogic functions and were already able to start playing first versions of our game. We continued developing to a point, where we had a proper version of our local uno game with only minor bugs. The first academic week of this year was week 7 of the module, in which we were thought about networking, the most crucial part of this project. We did not start implementing network functionality for our project in this week, as it was also the week of the maths exam, to which my preparation was quite time consuming. Looking back, we underestimated the scope of implementing

network functionality a little bit. However, as soon as we were aware of the real scope of it, and the amount of time that needs to be invested into properly debugging and testing this, we quickly adjusted our time plan to be a bit more ambitious, to be able to account for this increase in working effort required for the networking part of this project. This adjustment proved to be very valuable and helped us succeed in finishing the project on time. Constant peer discussions about what needs to be done, allocating clear tasks and continuously tracking them with the help of to do lists has also contributed significantly to the success of my/ our time management.

Strengths and Weaknesses

I would consider my strongest skills in the topics of this course to be in Object Oriented Programming, Testing, control flow and Lists/ Collections. I have had some experience with OOP, before starting this study and following the first two modules validated my knowledge and built on top of that, which is why I am quite confident in this topic now and would consider it as one of my strengths among the topics of this course. Furthermore, I have to say that I really enjoyed using Junit tests, during the assignments of this programming course, as they were able to indicate where your code was wrong and what the expected outcome of it should be. Therefore, I familiarized myself further with the topic, as I also wanted to be able to develop my own unit tests for my code. Spending time with it, researching and following the lectures and lab sessions on this topic has brought me to the point that I can also confidently lists this as one of my strengths amongst the topics of this course. Moreover, I would consider Control flow and List/ Collections as one of my strenghts in this course, as I have already started learning about these topics in Java when I was in high school. Following the first two modules has built my confidence in these topics and expanded my knowledge, which is why I would consider it to be one of my strengths.
I would say that my weaknesses amongst the topics of this course are in networking and multithreading. I can confidently develop code regarding both of these topics and understand the theory behind them, however as they were completely new to me my knowledge and understanding is not nearly as profound as for the other topics mentioned above. The project however has helped me a lot in building more confidence in these topics, as constant discussions, testing and questioning implementation with a fellow student really elevate the understanding of the topics at hand. Still, I feel like I could work more on familiarizing myself with these topics, which is why I consider them to be my weaknessess amongst the topics of this course.
Looking at this in an academic skills context, I would see my strengths in time management, metacognition and self-regulated learning. Time management has always been a strength of mine, when there is a deadline for example and a bigger task is to be done by then I was always able to quickly break this bigger task into smaller, easier to grasp subtasks and then set myself deadlines when these subtasks should have been archieved. Metacognition and self-regulated learning are also strengths of mine in this context, going hand in hand with what I said about time management. For the most part, I know what i can and cannot do so I am aware of my strengths and weaknesses which helps a lot with metacognition and self-regulated learning, as I can estimate pretty confidently, whether I am confident in a topic, need some more practice to be confident enough, or need a lot more time and practice to become confident. Simply being able to reflect on this properly is why I consider metacognition and self-regulated learning as one of my strengths in this context and it also helps tremendously in properly managing my time.
My weakness in this context I would see in the peer feedback, usually I am not the best at expressing my thoughts to other people, mostly because I try to avoid confrontation. But even if I am very content with my peer, as I certainly was during this project, it is still difficult for me to clearly express that. This is a weakness I am aware of, and I will keep working on.

Checkpoint Meetings

The feedback I received during the checkpoint meetings was always very positive and I have to say that I found it quite refreshing being asked to reflect on yourself and your learning journey and doing this together with somebody that has been at the exact situation you are currently in. As my feedback has always been positive it did not have great influence to my learning journey and my way of doing things as they have proved to be quite successful. However, being forced to reflect on how I did things, what time management mechanisms I used and how I have progressed during the Academic Skills part of this module has been really helpful to me in understanding the importance of self-reflection, as this is something I have rarely done before. But even if you feel like things are going very well, after a thorough self-reflection there will always be minor things you realize you can do differently to even increase how efficiently you are studying or how efficiently you are managing your time. One aspect, where the checkpoint meetings definitely influenced me is that I was given the advice to not procrastinate too much as it will only get more and more along the module. Looking back, taking this advice was definitely a very wise choice as the last weeks have been quiet demanding and I would not be sure if everything would have worked out the way it did if I procrastinated more in the beginning of the module.

The peer feedback I have received and given has always been very positive. I truly enjoyed working together with my peer and everything has worked out very well. Thorough peer discussions and collaboration have been very contributing to my learning journey. The peer feedback I received validates that me and my peer share the same thoughts on this and therefore there was no necessary improvements needed on either side.

Academic Skills (second student)

Time Management and Procrastination Avoidance

Throughout the course, I utilized daily planners and to-do lists as my primary tools for time management. I found these methods to be user-friendly and effective in keeping track of my tasks and deadlines. However, I discovered that my initial time estimates for certain tasks were often inaccurate, leading to unexpected time constraints and potential delays. To address this issue, I decided to revise my approach in week 5 by adding a 15% buffer to my original time estimates. This strategy proved to be successful and resulted in a more accurate and comprehensive daily schedule.
As I continued through the course, I found it necessary to adapt my time management approach to accommodate unexpected circumstances and changing priorities. I regularly reviewed my to-do list and calendar, making adjustments as needed to ensure that I was able to prioritize my responsibilities effectively. This allowed me to stay on track and maintain productivity, despite any challenges that may arise.
In regard to the project, I approached time management in a similarly structured manner. I created a comprehensive project plan, breaking down the tasks into smaller, manageable components and allocating adequate time for each. I also accounted for potential roadblocks or setbacks, allowing for additional time if necessary. Through this method, I was able to effectively manage my time and ensure that the project was completed on time and to the best of my ability.

Strengths and Weaknesses

In my analysis of my learning journey, I have identified programming in Java as my strongest skill in this module. Throughout the course, I have developed a great passion for coding and have consistently gone beyond the required exercises to expand my knowledge and understanding of the subject. I have found that I am able to work efficiently and effectively in achieving my designated learning goals.
However, I have also identified a weakness in my ability to focus and stay on task. Despite my efforts to overcome this challenge, I find myself easily distracted and unable to fully control my attention span. Through the use of checkpoint meetings, I was able to reflect on my approach to learning and discovered that listening to music, specifically lo-fi music, helps me to become more relaxed and focused.
In terms of academic skills, I have found time management to be a strength for me. I am able to prioritize my tasks and allocate sufficient time to each one, ensuring that I meet my deadlines and goals. Additionally, I have developed the skill of recognizing my strengths and weaknesses, which has allowed me to address and overcome my challenges more effectively.
The checkpoint meetings have allowed me to re-plan my learning and have made a positive impact on my approach to studying. I believe that I can improve this process by consistently incorporating self-reflection into my learning and seeking feedback from others in order to better understand my strengths and weaknesses.

Checkpoint Meetings

The checkpoint meetings with my mentor were incredibly insightful and productive. They not only allowed me to gauge my progress, but also provided me with a valuable opportunity to reflect on my learning and development. The mentor provided a supportive and engaging environment, encouraging me to engage in the metacognition cycle and regularly assess my skills and understanding of the course material.
Through this process, I was able to gain a deeper understanding of my strengths and areas for improvement. I found that regularly self-assessing my learning every two weeks was essential to staying on track and maximizing my potential. I also used the feedback received from my mentor to supplement my self-assessment, which helped me to identify areas where I needed to focus my efforts and further my understanding.
Overall, the checkpoint meetings with my mentor had a profound impact on my learning and personal growth. By providing me with structured opportunities to reflect and assess my progress, I was able to make informed decisions about how to develop my skills and achieve my learning goals.

Peer feedback was always positive and working together was a true pleasure. Thanks to great effort we both put into this project, we accomplished every designated goal. We always participated lively in the discussion, so that we could avoid any sort of problems.