

Software Development (202001064)
Programming (202001066)

Programming Report – UNO

Wouter Deen, S3032132, s.w.deen@student.utwente.nl
Iulia Costea, S3007715, i.costea-1@student.utwente.nl

January/2023

Contents

1. System Design	4
1.1 Class and package diagrams	4
1.1.1 Class diagram	4
1.1.2 Package diagram	4
1.1.3 Unit tests package	4
1.2 Implementation of functional requirements	4
1.2.1 The network protocol	4
1.2.2 Starting the server	4
1.2.3 Accepting connections on the server	5
1.2.4 Receiving commands on the server	5
1.2.5 Handling input on the client	5
1.2.6 Playing a card	5
1.2.7 Drawing a card	5
1.2.8 Calculating points and announcing winner	6
1.2.9 MVC pattern	6
1.2.10 User interface	6
1.2.11 Handling errors: server-side	7
1.2.12 Handling errors: client-side	8
1.2.13 Computer player	8
1.2.14 Additional features	9
1.3 Model-View-Controller pattern	10
2. Testing	11
2.1. Unit Testing	11
JUnit Testing	11
2.1.1 Game class testing	12
2.1.2 Player class testing	12
2.1.3 Card class testing	13
2.1.4 CardPile class testing	13
2.2. System Testing	14
Academic skills report	19
1. How was your planning influenced by your experiences with the planning and time writing during the Design project?	19

2. To what extent did your planning correspond to the actual progress during the project weeks? What made you deviate from your planning?.....	20
3. Which counter measures did you take to compensate for deviations from your original planning? What was the impact of this on the intended scope or quality of the project?	20
4. What did you learn from this experience for your next (project) planning? Take your answers to the other questions into account and ask yourself how you would want to prevent this or deal with this next time.....	20
5. Suppose that next year you are a teaching assistant for this project. Give at least two do's and don'ts that you would tell your students to help them with their planning	21
Appendix A: Academic Skills planning.....	23

1. System Design

1.1 Class and package diagrams

1.1.1 Class diagram

To convey our design choices regarding classes with their fields and methods, we hereby provide a class diagram. This class diagram is provided as a separate PDF file in the project's [GitLab repository](#). Provided in our class diagram are all classes in our project, with all associations between classes. We also added descriptions to some of the classes, to make their usage or certain design choices clearer. For further explanation about how we implemented multithreading (i.e., what classes are multithreaded and not), see section 1.2.

1.1.2 Package diagram

In addition to this, we made a UML package diagram, which is also provided as a separate PDF file in the project's GitLab repository. We decided on making a package diagram to make it abundantly clear what classes belong to each package. This also highlights our choices regarding the model-view-controller (MVC) pattern. You can read more about this in [section 1.3](#).

1.1.3 Unit tests package

In addition to the *Model*, *View*, and *Controller* packages, our main package (*nl.utwente.cardgame*) also contains a fourth package called *Tests*. We deliberately left out this test package and all its classes from the provided diagrams because tests are not of any importance for running the UNO application.

1.2 Implementation of functional requirements

For this project, we received a list of functional requirements to implement. Below, for each functional requirement, you will find a description of how we implemented it and what classes and packages we used for it.

Our initial offline implementation of the UNO game for deadline 2 (with the game logic as a deliverable), already had a fully functional game of UNO that in theory could host an unlimited number of players. However, for UNO, the player limit is 10. Instead of having separate projects for the client and the server application, we have a single project that contains all the necessary packages and classes to run both as a server and as a client. Starting the game as a server or a client is as easy as specifying “server” or “client” as the first CLI argument for running the JAR file. For more information on how to run the game, see the readme file in the [repository](#).

1.2.1 The network protocol

To play UNO over the network, we made a *Server* and a *Client* package inside the *Controller* package. These classes contain all code to communicate over the network to each other. For communication over the network, we are sending stringified JSON objects. These JSON objects contain key-value pairs, based on the specifications in the protocol of our mentor group. We wrote this protocol and added very detailed descriptions and lots of examples to the readme, which you can view on the separate [protocol repository](#). Here, we wrote in more detail about how sending and receiving commands over the network works, including code snippets.

1.2.2 Starting the server

To start a server, we create a new instance of the *Server* class in the *Main* class (in the *Controller* package), and we execute this server on a separate thread. To start listening for connections on the server, we create

a new `ServerSocket` in the `Server` class's `run()` method, which will be invoked once the server's thread has been started. The server's `run` method will also create two thread pools by using the `java.util.concurrent.Executors` package:

1. a `clientPool` for running all instances of `ClientHandler`;
2. a `gamesPool` for running all instances of `Game`.

1.2.3 Accepting connections on the server

To accept connections, we created an infinite loop, that waits for the `server.accept()` method to return a `Socket`. Since Java functions are synchronous by default, this means that the loop will only start over once it accepted a new socket. With this socket, inside the loop, a new instance of `ClientHandler` will be created and executed in the `clientPool`. A `ClientHandler` is constantly listening for incoming messages from the client, and if it receives a message, it will act accordingly. Because each `ClientHandler` needs to be constantly listening to incoming messages from the input stream from the client's socket, each `ClientHandler` needs to be run in its own thread.

1.2.4 Receiving commands on the server

The `run()` method of the `ClientHandler` will loop for each incoming message and it will try to parse an incoming stringified JSON message back into a `JSONObject`. We read the value of the `command` key from this JSON object and run this through a switch statement. Each case will invoke different methods. For playing a game, there are essentially two important commands: `PLAY_CARD` and `DRAW_CARD`, used for playing and drawing a card respectively. The `PLAY_CARD` command will invoke the `Game` class's `playCard()` method, and the `DRAW_CARD` command will invoke the `Game` class's `drawCard()` method.

1.2.5 Handling input on the client

We have the `InputHandler` class inside the `View` package to handle standard I/O operations of the system. This means reading and printing lines in the terminal by using `System.in` and `System.out`. Most methods that need to read data from the standard I/O will have a designated function for this in the `InputHandler`. This class contains a `print()` method, which receives a JSON object from the `IncomingHandler` and prints it accordingly, depending on the command that it holds. This JSON object is an object that has been received from the server.

1.2.6 Playing a card

When the `ClientHandler` receives a `PLAY_CARD` command, it will call the `playCard()` method in the `Game` class. This method invokes all necessary methods to play a card. First, it will check if it's actually this player's turn, after which it will invoke the `card.isValidMove()` method to determine whether playing this card abides by all the rules of UNO. If not, it will send an error to the client.

If the card is valid, it will call the `playCard()` method on the current player. If the card that the player is trying to play is an action card, this method will call the `performAction()` method on that card. It will then call the `discardCard()` method on the hand, which handles removing the card from the player's hand and placing it on top of the discard pile. After that, in the `Game` class, the `playCard()` method will construct a new JSON object for the `NEXT_TURN` command, following the specification in the protocol's documentation. For playing a card, we made a schematic illustration of the different classes that come into play. Please see Figure 1-4.

1.2.7 Drawing a card

Drawing a card has roughly the same control flow as playing a card. The `drawCard()` method in the `Game` class invokes all necessary methods to draw a card. First, it will check if it's this player's turn, after which

it will call the `drawCard()` method on the current player. This method in the *Player* class contains the actual functionality for drawing a card. It will call the `pickCard()` method of the *DrawPile* class. Drawing a card in our game works differently from the real-world game: we do not have a method for shuffling the cards in the draw pile. Instead, the `pickCard()` method takes care of selecting a random card from the draw pile. The players of our game will never see the difference, but behind the scenes, it results in much cleaner code. After calling `pickCard()`, the `drawCard()` method will add the card to the player's hand by calling the `addCard()` method on the player's *Hand*. This method is inferred by the *Hand* class from extending the *CardPile* class.

In UNO, if you draw a card and it is a valid card to play, you can play it. We implemented this by making the player automatically play this card by executing the `playCard()` method in the *Game* class. This is because our network protocol does not support asking a player whether they want to immediately play this card or not, so we implemented this as an acceptable compromise. For all players with chat functionality enabled, it will send them a *CHAT* command with an accompanying message (the player was able to play the card that they drew). Hereafter, it will construct a JSON object containing the *NEXT_TURN* command and broadcast this to all players.

1.2.8 Calculating points and announcing winner

Eventually, a player will play their last card. Since the *Game* class is a controller class (see the package diagram), we use the `playCard()` method to check whether the player played their last card. This is done by requesting the cards in the player's hand by calling the `getCards()` method of the *Hand* class. This method originates from the abstract *CardPile* superclass of *Hand*. The *CardPile* class is also extended by the *DrawPile* and *DiscardPile* classes, each having its own implementation and additional methods.

If the `getCards()` method returns an empty list, the hand is empty and we call the `calculatePoints()` method on the player that played their last card. This method loops through all the cards of every player in the game in a nested for-loop and adds the points of each card to a variable holding the total amount of points. Reading the points value of each card is done by invoking the `getPoints()` method on a *Card* object. The *NumberCard* and *ActionCard* classes extend the abstract *Card* class and have their own implementation of this method. It returns the points that a certain card is worth according to the official UNO game rules.

After calculating the total points that the cards in the hands of other players are worth, it will add these points to the total points of this player. We broadcast the end of this round to each player's *ClientHandler* by constructing a JSON object with the *ROUND_FINISHED* command. If the total amount of points is 500 or more, we set this player as the winner by calling the `finish()` method on the *Game*. This method will broadcast a JSON object with the *GAME_FINISHED* command according to the protocol requirements.

1.2.9 MVC pattern

We discuss this functional requirement in [section 1.3](#).

1.2.10 User interface

For the user interface of the client application, we built a TUI (Text-based User Interface). This TUI is using the standard input and output of a Java application to read and write messages to and from the console. We use the *InputHandler* class inside the *View* package to communicate with the client using the TUI. We have methods like `print()`, `println()`, and `printError()`, but we also have use-case-specific methods, like `getPort()` and `getWildcardColor()`, which are used for getting the port of the server and getting the color of the wildcard that the user wants to play. We used the provided ASCII interface to give our messages some colors.

1.2.11 Handling errors: server-side

On the server-side, we made sure to catch all possible exceptions. However, a user might try to perform an unauthorized action, like trying to play a card that does not abide the rules of UNO. When something like this happens, we send a JSON object with the ERROR command over the network to the client. This JSON object also contains the error code and a description (please refer to [our protocol specification](#)). The game knows what instance of *ClientHandler* to use for this, since the constructor of *HumanPlayer* receives a *ClientHandler* object. Thus, we can do:

```
if (p instanceof HumanPlayer) {
    ((HumanPlayer) p).getHandler().sendObject(error);
}
```

Figure 1-1: sending an object to the client by first getting the *ClientHandler* of the current player, and then invoking the *sendObject* method from the *ClientHandler* class.

An error will most likely be sent over the network by either the *ClientHandler* class or the *Game* class since these are important controller classes. An example of an exception being caught in the *ClientHandler* is the *JSONException*. This exception is thrown by the *org.json* package that we use for working with JSON objects. When this exception is thrown, it means that it could not parse the received string from the client back into a JSON object. When such an exception is caught, we send a JSON object back to the client, with the ERROR command and an error code and description, as per the network protocol requirements.

The most common network-related exception is an *IOException*. This could mean all kinds of things, depending on where the exception is caught and what the exception's message is. In the *ClientHandler* class, we have a loop in place for reading data from the client. This is done by creating a new *InputStream* from the client's socket input stream and wrapping the *InputStream* in a new *BufferedReader*:

```
@Override
public void run() {
    try {
        out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String req;

        while ((req = in.readLine()) != null) {
            try {
                JSONObject data = new JSONObject(req);
                Command cmd = data.getEnum(Command.class, key: "command");
                System.out.println(displayName + ": " + cmd);

                if (displayName == null) {...} else {
                    switch (cmd) {...}
                }
            } catch (JSONException e) {
                JSONObject jo = new JSONObject()
                    .put("command", Command.ERROR)
                    .put("code", Error.INVALID_COMMAND)
                    .put("description", "Could not parse the client's message to JSON. Is it formatted correctly?");
                out.println(jo);
            }
        }
    } catch (IOException e) {
        if (e.getMessage().equals("Connection reset")) {
            System.out.println(((displayName == null) ? "Unknown client" : displayName) + " disconnected.");
            shutdown();
        }
    }
}
```

Figure 1-2: reading from and writing to the client's socket by using the socket's input and output stream. This method also handles common exceptions.

In the provided code in Figure 1-2, you can see the code for receiving messages from the connected socket (the client) and handling exceptions. If we catch an *IOException* while receiving a new line from the socket (*in.readLine()*), this is likely due to a connection reset. If this is the case, we call the *shutdown()* function on the *ClientHandler*. This will close the socket's input and output stream and remove the client from the list of connected clients in the *Server* class. Also, if the player was in a queue, we remove them from the queue. If the player was in a game, we remove them from the game and also stop the execution of the entire game using the *shutdown()* function in the *Game* class. Finally, to stop the execution of the thread that is running this instance of *ClientHandler*, we call *Thread.currentThread().interrupt()*. Note that the code implies that if we catch an *IOException* here that has a different message than "Connection reset", we ignore it. This is because other *IOExceptions* might be a one-off, and we decided that we do not have to handle these, except to prevent a termination of the current thread.

A class that is even more likely to send errors to the client than the *ClientHandler* is the *Game* class. These errors are for playing an invalid card and trying to play/draw a card while it's not your turn. These are the errors that are most often sent from the server to a client since someone might mistakenly think it's their turn, or accidentally play a card that they are not allowed to play.

Errors are only sent from the server to the client and not vice versa. The *ClientHandler* on the server will ignore incoming ERROR commands if you try to send them.

1.2.12 Handling errors: client-side

On the client side, errors will be received like any other command from the server and will be forwarded to the *InputHandler* to be dealt with. However, we also need to handle certain problems that could occur on the client-side, without communicating with the server. These problems are by design always input problems. For instance, we have a lot of places where only a number input is valid. Catching a *NumberFormatException* is not mandatory, but we always do it in places where we call *Integer.parseInt()* to prevent problems. An example of this is the *getPort()* method, located in the *InputHandler* class in the *View* package. It will read a port from *System.in* and return it to the caller object:

```
public int getPort() {
    printLine(msg: "Please enter the server port:");
    int port = 0;
    while (port == 0) {
        try {
            int input = Integer.parseInt(readLine());
            if (input < 1 || input > 65535) {
                throw new NumberFormatException();
            } else port = input;
        } catch (NumberFormatException e) {
            printError(msg: "Please input a valid port number.");
        }
    }
    printSuccess(msg: "UNO server found.");
    return port;
}
```

Figure 1-3: *getPort()* method inside the *InputHandler* class

1.2.13 Computer player

For implementing the computer player, we chose to run this computer player directly on the server. This means that it is not possible to run a client as a computer player. This not only prevents players from cheating but also provides some useful functionalities. Namely, if a client joins the queue for a game for a

certain number of players, a timer of 30 seconds starts. If not enough players join within 30 seconds to fill up all the player slots, the remaining slots will be filled by computer players. This makes sure that players do not have to wait too long to play a game.

This timer is implemented in the *ClientHandler* class and will be started by the first player to join the queue. We made a separate function to start the timer: *startTimeLeftBroadcaster*. It uses the chat functionality to broadcast the remaining time to each player in the queue. If a client does not support the chat functionality, it will simply not receive the chat and it will just receive the `GAME_STARTED` command after 30 seconds.

In addition to computer players filling up a queue after the time limit has been reached, you can also make computer players play against each other in a computer game (see readme markdown file in the GitLab repository).

We found that for playing a computer game, the fewer computer players that play against each other, the longer a game will take. At some point our computer players were not smart enough to finish a game with only two players, resulting in a range of exceptions. We will talk about this in more detail in the 2nd chapter (Testing), but to sum up, we had to make the computer player smarter and increase the stack size. Please make sure that if you run the game, you run it with the right stack size, otherwise, a *StackOverflowError* might occur.

1.2.14 Additional features

We had to implement at least one additional feature from the list of additional features. We picked the chat, the multi-game server, and the lobby features to implement. We will discuss each functionality and how we implemented it.

Chat functionality

In our network protocol, the *CHAT* command is treated like every other command. When a player enters a message into the standard input that is not a command, the entire input will be treated as a chat message. The *OutgoingHandler* will construct a new JSON object with the *CHAT* command and all other required parameters, as required per the protocol. This will then be sent over the network and will be received by the *ClientHandler* that is dedicated to this client's specific socket. The *ClientHandler* can request all connected clients from the server, and invoke the `broadcast()` method from the *Server* class. This method takes two arguments: a list of *ClientHandlers* (clients) to broadcast to, and the JSON object. The *ClientHandler* will add the *displayName* of this client to the JSON object, after which it will broadcast it to a certain list of clients. If a player is not in any game or any queue, it will broadcast the message to all players in the lobby. If a player is in a queue, it will broadcast the message to all players in the queue. If a player is in a game, it will use the *Game* class's `broadcast()` method to broadcast the JSON object to all clients in the game.

Multi-game server

First off, the multi-game server. This was extremely easy to implement, since we already had the knowledge about thread pools since needed one for the *ClientHandler*, as discussed previously. We created a new thread pool and implemented the `newGame()` method in the *Server* class. This method is used for creating a new game, together with some other things, like clearing the list of *ClientHandlers* in the queue `ArrayList`. This list is saved inside the `queues` hashmap with integer-`ArrayList` key-value pairs, where the integer is the *gameSize* (number of player slots) and the value is the *ClientHandlers* in the queue. Every time a new game is created and constructed, it will be executed in a separate thread using the thread pool's `execute()` method. Running each game on a separate thread prevents a thread block in one game from affecting the other games.

Lobby

The lobby functionality was implemented by combining the chat functionality and by using a list to save all clients that are not in a queue or in a game, which means they will be in the lobby. In the lobby, players can chat with each other, and decide together to join a game with a certain amount of player slots.

1.3 Model-View-Controller pattern

For this project, we decided on using the Model-View-Controller design pattern for implementing the separation of concerns. We implemented this design pattern by separating all packages and classes into three main packages: a *View* package, a *Model* package, and a *Controller* package. Which classes and packages play the role of the model, view, and controller can be seen in the package diagram provided in [section 1.1](#), in addition to the class diagram.

Arguably the most important package of the three is the *Controller* package. This is the glue between the controller and the view: classes in this package execute methods in classes from the *Model* package and send data back to the *View* package for the user to see. Most of the time, a method of a class in the *Model* package is of type non-void, meaning that it does some computations, and based on the specified parameters, it returns a value to the caller method. Some model classes might also void functions, meaning that they do not return any value.

A key characteristic of the model classes, more specifically the classes' methods, is that they are not invoked on their own. To see an example of this behavior, see Figure 1-4. The behavior that model classes often perform computations and return data to a controller class, so that the controller class can invoke the next method on a model class, is illustrated by the double-sided arrows.

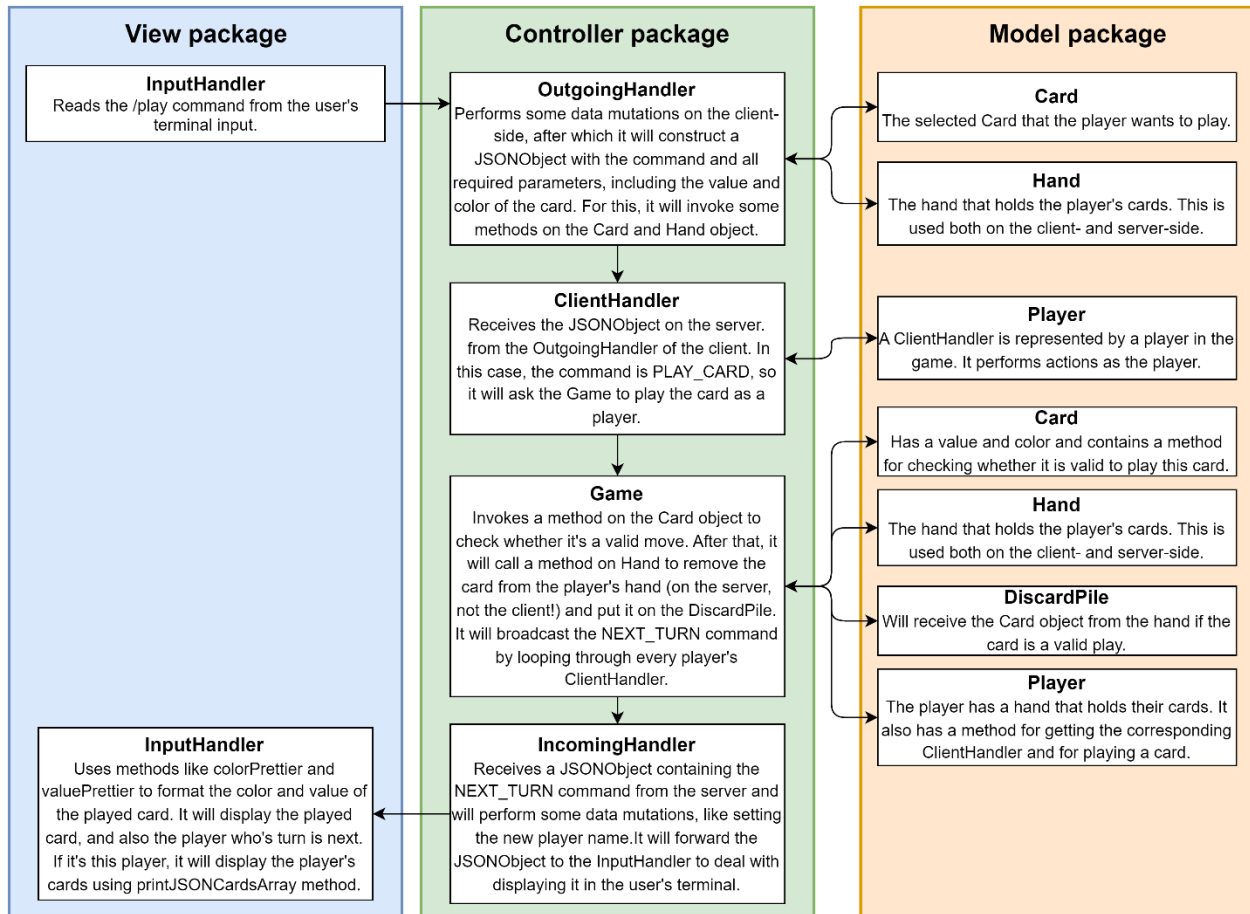


Figure 1-4: schematic overview of the interactions between the Model, View, and Controller packages for playing a card.

As you can see in the above illustration, the *View* package is used solely for handling player interactions with the program. The *InputHandler* class handles reading input from the TUI and the *Controller* package will act on that input using certain model classes in the *Model* package. Of course, Figure 1-4 only illustrates the scenario where a player plays a card. For drawing a card, different methods on different model classes will be invoked at different moments.

2. Testing

2.1. Unit Testing

JUnit Testing

For the JUnit testing, we choose to test the classes `Game`, `Player`, and `Card`. These are 3 of the most complex methods in our project, as most game logic methods are found in these classes and their corresponding subclasses. We used JUnit tests to test the game logic of these classes. It is worth highlighting that some other classes might be regarded as even more complex (in terms of raw lines), like the `ClientHandler` class. However, a lot of this code is working with JSON objects, and also, these types of classes are highly network related and are therefore impractical at best and impossible at worst to test with JUnit test classes.

2.1.1 Game class testing

The *Game* class is the center of our application, where all game-related model classes come together to compose the game. Examples of these classes are *Player*, *Card*, *DrawPile*, and *DiscardPile*. Although some methods of the *Game* class include over-the-network communication, making Junit testing infeasible, most of the methods in the *Game* class, such as `getNextPlayer()`, `nextPlayer()`, `switchDirection()`, and `drawCard()`, are crucially important methods in the game flow, therefore, we created the *GameTest* class, which includes 2 main tests. Moreover, the methods containing network communication, are mostly invoking methods in model classes, which can be individually tested.

Before running each of these test methods, we set up a scenario for the tests to use, using a setup method with a `@BeforeEach` tag. In this method, we create an instance of *Game*, which will initialize all important features of a game: the direction in which the game is played, handing out the first 7 cards to each player, creating a new *DrawPile* and *DiscardPile*, and numerous other things. As the methods we want to test require that the game has players, the setup method also creates players and adds them to the game. Before being able to generate the players (*HumanPlayers* to be exact), we need to create corresponding *ClientHandler* objects for all of them. This is because the *HumanPlayer* constructor takes a *ClientHandler* object as a parameter. As this is a local game, the socket and the port parameters for the constructor of the *ClientHandler* objects can be set to null. After this setup, the new players are added to the game.

The first test will assert whether methods `getNextPlayer()`, `nextPlayer()`, and `switchDirection()` work as intended. To avoid calling methods that involve over-the-network communication, specifically in the `initiateGame()` method, we have extracted the lines that generate a random integer index. This index is used for selecting a random player to kick off the game. As in the setup method we created the array of players, the players should now be in order. We can now call the `getNextPlayer()` method and asserts if the player which has the position in the game of the generated number +1 is the same as the player returned by calling the method. In the test, we are also calling the method `switchDirection()`, and after this, asserting if the direction was changed by again calling the `get next player` method, which should now be the player on the random index we generated.

The second performed test is for the `drawCard()` method. For this test, we simulate a player drawing a card from the draw pile and moving it into the player's hand. As we have not called the `initiateGame()` method in the *Game* class, the hands of all players are still empty. This means that after `drawCard()` is called, there should be one single card in the player's hand. The limited number of tests that call several methods have good coverage of the methods and classes, as pictured below:



Figure 2-1

2.1.2 Player class testing

Another class we tested is the *Player* class. This class contains the part of the game logic which stores the hand of a certain player, contains the method to calculate the points the winner is awarded at the end of a round, and the method `playCard()` which, in the case of an *ActionCard*, performs the action and removes the card from the player's hand.

The setup for this method is very similar to the one performed in the game testing. A new game is created, as well as the *ClientHandlers* needed to connect players to the game.

The first method tested is the `playCard()`. For this, we created a *NumberCard* just as a test object, added it to the hand of the player, and then calling the `playCard()` method, using as parameters the simulated game and

the test card. To assert whether the instructions were performed successfully, we check whether the card was removed from the game by the player.

One of the important methods we tested is from the *ComputerPlayer* class, which extends the *Player*. The method is `calculateBestWildcardColor()`, which picks the best color for the *ComputerPlayer* to choose when playing a *Wild* card. To check this method, we generated a few cards, one of each color and one extra green one, and added them to the hand of the *ComputerPlayer*. After calling this method, we check whether the picked color is green, as this is the most present color in the hand.

One last method checked here is the `calculateTotalPoints()`. Here we create a game of 2 players, one will be the winner, and the other the loser of the round. We added a few cards to the hand of the loser, as the points of these will be summed up and added as the points of the winner. Finally, we assert whether the method calculated the points, which should equal to 78, considering the generated cards.

All these tests resulted in good coverage of the methods, as well as the lines, as pictured below:



Figure 2-2

2.1.3 Card class testing

Card class includes all the methods which handle the actions of a card. For this class, the most important method to test is the `isValidMove()` one. To check this method, we created two separate tests, one for *NumberCards* and the other one for *ActionCards*. These 2 tests start by creating a game, which consequently picks the first card to kick off the game. In the test for numbered cards, we create 2 test cards, one with the same color and a different number, and the other, with different color and the same number. We then call for the `isValidMove()` of the class, and as at least one of the conditions is met, the tests should pass.

The same method is used for testing the `isValidMove()` method for cards that are objects of type *ActionCard*. However, to assure that more possible scenarios are met, we used a `switch()` to create several test cards for each corresponding type.

Other methods we are testing in this class are getters and setters, as well as the `toString()` method. This last one should return the action performed by an *ActionCard* or the number of a *NumberCard* in a string format. This is checked by generating the cards and assessing whether the output matches the expected strings.

Due to the high number of possible cases, this method does not provide full line coverage, but the remaining methods are tested by playing the game, as in section 2.2.



Figure 2-3

2.1.4 CardPile class testing

One more class that we created JUnit tests for is the *CardPile* class. This class is responsible for creating a collection of cards and is extended by three classes. These are the *Hand* class, for holding the cards of a player, the *DiscardPile* class, for holding all discarded cards, and the *DrawPile* class, for drawing cards. For testing, we created a `setUp()` method tagged with the `@BeforeEach` tag, as is usual with JUnit unit tests. We create a new game in this method since *DrawPile*'s constructor requires a *Game* object as an argument. Also in the constructor, the methods `initiateNumberCards()`, `initiateColorActionCards()`, `initiateWildActionCards()` are called, so that the deck should be fully created and ready for testing.

The first test checks the constructor of the *DrawPile*, specifically if all cards are included in the pile. By using the getter for this collection of cards and checking its size, we can assess whether the number of cards is equal to 108 (following the official UNO base game rules). This checks all the mentioned methods that were called in the constructor, albeit in an elementary form, without actually checking each card.

Another method tested is the *pickCard()* method of the *DrawPile* class, by assessing whether after calling it, the number of cards in the draw pile decreased by one. The tested method also calls for the *getRandomCard()* method from the same class. Although this whole test class only calls for 2 methods, these on their own need to call the rest of methods in the class. This results in a method and line coverage of 100%.



Figure 2-4

2.2. System Testing

Some of the most important classes of our project, such as *Client*, *Server*, and their handlers, use over-the-network communication. To test these classes, we considered it would be unfeasible to create Junit tests, therefore, in order to still ensure that they are all working well, we created a main method that would invoke several instances of the class *Client*, and one *Server*, in order to test them.

After we completed the main functionalities of the game, and we had a server and client running, most of the testing was done by playing the game, in order to spot any edge cases, exceptions and errors that we were not handling right. During this process, we spotted and fixed the following issues of game logic and communication between the server and the clients:

2.2.1 After a game is finished, the player can join a new game

This first functional requirement was tested by playing enough rounds so that one of the players in game becomes the winner. Although we had implemented a method for shutting down the game and removing the players from it, this was not enough for them to start a new round. While playing, we noticed that after the first game was finished players were not able to start a new round. This issue was on the client side, as the server was receiving the “JOIN_GAME” command, but a new queue was not set up. This was fixed by adding an *ArrayList* called “lobby” on the server side. This list would receive the players at the beginning of the game, it removes them when they join a queue for a certain game and adds them back in when the game was finished.

2.2.2 Chat availability at all times

The chat of the game should be available to players at all times: while they are in the lobby, while they are in the queue, and during the whole game play. Of course, the chats should only be visible to players in the same queue, same game, or if the player is in the lobby, to all players that are in the lobby. We tested this feature by simulating all of the before mentioned scenarios. In order to do so, we set up a server and connected a few clients to it. We set up clients that would simultaneously be in the same queue or game, as well as clients that would be in different ‘locations’ on the server. As designed, while testing the chat was only sending a player’s messages to other players that were respectively part of the same instance of game, queue, or part of the lobby, and not to all players connected to the server.

2.2.3 Not initializing the game direction at the beginning of each round.

We eventually identified unexpected behavior where the game's playing direction (clockwise/counter-clockwise) on the client side would either be null or incorrect. While this issue did not throw an exception, we noticed while playing that when a reverse card was played, the direction was not changed. We then recognized in our code that we were not initializing at the beginning of each round the direction in which the game was played, therefore its value was null. This issue was in the *IncomingHandler* class, on the client side, and we fixed it by setting the game direction in the command of a new round:

```
case ROUND_STARTED -> {  
    input.print(jo);  
    client.setGameDirection(GameDirection.CLOCKWISE);  
}
```

Figure 2-5

2.2.4 Wildcards not getting removed from the hand on the server side

This issue was noticed while playing the game, in the following scenario: a player was trying to play a wild draw 4 card, but the server was sending an invalid move error, although in the client side the move seemed to be valid (no other cards from the hand were playable). Going through the hand on the server side, we noticed that the Wild cards were not removed from the client's hand. This happened because the client was sending a wildcard with CardValue "Wild", but the CardColor was already set to the color the player was choosing, instead of just CardColor.W. We fixed this problem by adding an "else" branch, where we were checking just for the cards that were of type Wild, and also removing them from the hand. This modification is represented by the else branch in the following code:

```
void removeCard(Card card) {  
    ArrayList<Card> cardsCopy = new ArrayList<>(cards);  
    for (Card loopCard : cardsCopy) {  
        if (loopCard.getValue() == card.getValue() && loopCard.getColor() == card.getColor()) {  
            cards.remove(loopCard);  
            break;  
        } else if (loopCard.getValue() == card.getValue()  
            && (loopCard.getValue() == CardValue.W || loopCard.getValue() == CardValue.F)) {  
            cards.remove(loopCard);  
        }  
    }  
}
```

Figure 2-6

2.2.5 Casting ComputerPlayer as a HumanPlayer exception

In the *WildDrawFour* class that implements the *Action* interface, while running the game, the following exception came up:

```
Exception in thread "pool-2-thread-1" java.lang.ClassCastException: class  
nl.utwente.cardgame.Controller.Player.ComputerPlayer cannot be cast to class  
nl.utwente.cardgame.Controller.Player.HumanPlayer (nl.utwente.cardgame.Controller.Player.ComputerPlayer and  
nl.utwente.cardgame.Controller.Player.HumanPlayer are in unnamed module of loader 'app')
```


Figure 2-7

The highlighted part indicated to us that at some point in the execution, we were casting a *ComputerPlayer* to a *HumanPlayer*. This exception happened during a particular edge case on the client side, when the first card played was a Wildcard and the player was a *ComputerPlayer*. We noticed the mistake was in the last “If” statement of the method. Instead of checking if the `game.getNextPlayer()` was a *HumanPlayer*, we should check for the *Player* `p`, as this is also the one we will be sending the command to. The first picture below shows the `p` variable:

```
Player p = game.getCurrentColor() == null ? game.getCurrentPlayer() :
game.getNextPlayer();
```

Figure 2-8

The current color will only be set after the first card on the discard pile is placed. This means that if the first card is an action card, the `currentColor` will still be null. The action will then be performed on the first player, instead of the next player. Here we can see the modified if statement:

```
if (p instanceof HumanPlayer) {
    ((HumanPlayer) p).getHandler().sendObject(issueCardsCmd);
}
```

Figure 2-9

2.2.6 Creating a copy of the card instead of passing the card

We identified the following issue on a different edge case, the one where the game was picking a *WildDrawAction* as the first card to be played. As the game rules mention, this card can never be the first one to be placed on the discard pile at the start of a new round. While we did implement a check for this, as seen in the method below (Picture 2-10), we should not have created a new card, as we do in line 5, as when adding it to the *DrawPile*, we are adding a copy, and not the actual card from the pile.

```
while(!validActionCard) {
    Action action = ((ActionCard) card).getAction();
    if(action instanceof WildDrawAction) {
        System.out.println("This is an invalid first card. Redrawing...");
        card = drawPile.pickCard();
        addCard(card);
        System.out.println("The first card on the discard pile is: " +
getTopCard());
    } else {
        // ...
    }
}
```

Figure 2-10

To fix this small bug, we removed the line that was creating a new card and just passed to the `addCard` method the card picked from the draw pile, as seen in the snippet below (Picture 2-11).

```
if(action instanceof WildDrawAction) {
    System.out.println("This is an invalid first card. Redrawing...");
    addCard(drawPile.pickCard());
    System.out.println("The first card on the discard pile is: " +
getTopCard());
} else {
```

Figure 2-11

2.2.7 Stack overflow errors

While building the option to play a game between only *ComputerPlayers*, we encountered the `StackOverflowError` in two different spots. A stack overflow can occur when a recursive function calls itself repeatedly to execute a set of instructions. This is because each time a method is called recursively, memory space is taken up in the method's stack. This is where all necessary data to execute a method is located.¹ In a stack overflow error, it is usually the stack frame in the stack that causes the problem. This is where local variables and function arguments are saved. Each thread, and thus, each instance of *Game*, gets its own stack. When a function returns, we remove its stack frame. However, if a function is called recursively, the data of every method call is saved in the stack. Since the first call of the recursive function did not return something yet, the stack will become larger after every recursive call. At some point, if there is not a timely return, the stack will become so large, that there is no space left for another recursion. Another recursive call could then make the stack *overflow*. This is inappropriate behavior since it could pose a risk to the data integrity of other parts of the software (i.e., it could overwrite other data saved in memory). Therefore, to prevent an actual overflow, if the number of calls exceeds the designated memory for the stack, a `StackOverflowError` will occur. This could result from a design flaw, such as a recursive function that ends up in an infinite loop. This is why a stack overflow error is usually a good thing: it is preventing infinite loops. However, it could also be the case that this error occurs during a process that is not a design flaw, but rather a recursion so *deep* that Java's default stack size (256kb for a modern Windows installation) is not enough for the execution.² We found that this was the case while playing a computer game (a game with only *ComputerPlayers* playing against each other).

One `StackOverflowError` we discovered while running a game with only *ComputerPlayers* was an edge case. While playing the game, cards are being moved from the *DrawPile* to the player's *Hands* to the *DiscardPile*. Once the *DrawPile* is empty, all the cards in the *DiscardPile* are added back to the *DrawPile*. To pick a card to place as the first card on the draw pile, we just get a random one from the pile, as discussed in [section 1.2.7](#). However, in this very particular case, the only cards on the discard pile at the time that the *DrawPile* was emptied, were two *wild draw four* cards, which are invalid first cards. Therefore, the method would continuously loop through the cards in the draw pile, trying to pick a new top card, but as there would be no valid card, it would just keep calling the same method, until the stack would be full.

To fix this, we decided to surround the method call that caused the problem with a try-catch block, to catch this `StackOverflowError`. In the catch block, besides sending out a message for the players, the *DrawPile* is reset and the whole game is restarted. This implementation of the try-catch block is seen in figure 2-12. The stack trace which represents the initial exception can be seen in figure 2-13.

¹ Jacob Sorber. (2020, December 23). The Call Stack and Stack Overflows (example in C). YouTube. <https://youtube.com/watch?v=jVzSBkbfdiw>

² Understanding Threads and Locks. (n.d.). https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html

```

Card firstCard;
try {
    firstCard = discardPile.placeFirstCard( game: this);
} catch (StackOverflowError e) {
    System.out.println("The draw pile does not contain a valid card to place as a first card on the discard pile. " +
        "Resetting the game...");
    discardPile.clearPile();
    resetDrawPile();
    initiateGame();
    return;
}

```

Figure 2-12

We would like to emphasize that catching a `StackOverflowError` is rarely the best practice. In this case, we could have also resolved the error altogether by checking whether the draw pile only contained one or two *wild draw four* cards. However, given the time constraints, we opted for this solution.

```

Exception in thread "pool-2-thread-1" java.lang.StackOverflowError
at java.base/java.nio.ByteBuffer.putArray(ByteBuffer.java:1324)
    at java.base/java.nio.ByteBuffer.put(ByteBuffer.java:1181)
    at java.base/sun.nio.ch.NioSocketImpl.tryWrite(NioSocketImpl.java:397)
    at java.base/sun.nio.ch.NioSocketImpl.implWrite(NioSocketImpl.java:413)
    at java.base/sun.nio.ch.NioSocketImpl.write(NioSocketImpl.java:440)
    at java.base/sun.nio.ch.NioSocketImpl$2.write(NioSocketImpl.java:826)
    at java.base/java.net.Socket$SocketOutputStream.write(Socket.java:1035)
    at java.base/sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:234)
    at java.base/sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:313)
    at java.base/sun.nio.cs.StreamEncoder.implFlush(StreamEncoder.java:318)
    at java.base/sun.nio.cs.StreamEncoder.flush(StreamEncoder.java:160)
    at java.base/java.io.OutputStreamWriter.flush(OutputStreamWriter.java:248)
    at java.base/java.io.BufferedWriter.flush(BufferedWriter.java:257)
    at java.base/java.io.PrintWriter.newLine(PrintWriter.java:567)
    at java.base/java.io.PrintWriter.println(PrintWriter.java:710)
    at java.base/java.io.PrintWriter.println(PrintWriter.java:838)
    at nl.utwente.cardgame.Controller.Server.ClientHandler.sendObject(ClientHandler.java:239)
    at nl.utwente.cardgame.Controller.Game.broadcast(Game.java:466)
    at nl.utwente.cardgame.Controller.Game.playCard(Game.java:284)
    at nl.utwente.cardgame.Controller.Game.handleComputerPlayer(Game.java:390)
    at nl.utwente.cardgame.Controller.Game.playCard(Game.java:286)
    at nl.utwente.cardgame.Controller.Game.handleComputerPlayer(Game.java:390)
    at nl.utwente.cardgame.Controller.Game.drawCard(Game.java:364)
    at nl.utwente.cardgame.Controller.Game.handleComputerPlayer(Game.java:387)
    at nl.utwente.cardgame.Controller.Game.playCard(Game.java:286)
    at nl.utwente.cardgame.Controller.Game.handleComputerPlayer(Game.java:390)
    at nl.utwente.cardgame.Controller.Game.playCard(Game.java:286)
    and this goes on and on...

```

Figure 2-13

After solving this `StackOverflowError`, another one appeared during the execution of a computer game. On this occasion, however, the error did not occur due to a design flaw.

In the *Game* class, the `playCard()` method is called first with 2 arguments, the game in which the card is played, as well as the player which is playing the card. To simplify this method which is already quite complex, we created a separate method, `handleComputerPlayer()`. This method checks if the next player is a computer player, and if it is, it first checks whether a card can be played. If this is the case, the `playCard()` method is called again in *Game*. Otherwise, the `drawCard()` method in *Game* is called. So when playing a game with only computer players, the `handleComputerPlayer()` method ends up calling either `playCard()` or `drawCard()`, which calls `handleComputerPlayer()` again after execution is finished. This results in semi-recursive method calls (between two methods rather than one). If the computer players do not manage to finish the game after a certain amount of turns, the call stack will be filled up and a `StackOverflowError` will be thrown. This time, we recognized that there were two ways to fix it.

First off, we made the computer player smarter. The first change made was to choose an `ActionCard` over the `NumberCard`. To do this, we simply first check whether the hand contains an `ActionCard`, and set the chosen card to it, before checking for the `NumberCards`. Another thing that we added was an algorithm to check whether a Wild action is a valid next card, which we were not previously checking for. This last part, paired with the `calculateBestWildcardColor()` method improved our computer player's moves.

Secondly, we found that even though making the computer player smarter helped, we still got a `StackOverflowError` regularly. We recognized that this was not due to a design flaw in our code, but rather, a limitation of the Java Virtual Machine. To fix this, we increased the stack size and did some trial runs to find the optimal stack size.³ Eventually, we settled on 1 megabyte. To run our game with a stack size of 1 megabyte, we provided details in the readme file in the code repository.

Academic skills report

1. How was your planning influenced by your experiences with the planning and time writing during the Design project?

Wouter

For me, it did not make that much of a difference. I already had some planning lessons in high school, a couple of years before the final year, so I knew some of the strategies and I also knew what strategy suited me best. However, it gave me some new insights in that I usually plan a bit too much time for things like math and too little time for things like programming. This is usually because I like to try and one-up myself, because that is how I learn new things. This time around, I tried to plan more time than I thought would be necessary for programming, to make up for this.

Iulia

The planning and time writing was a great reflection opportunity. Before, even if I was planning my tasks and keeping track of them, because I was not putting them down on paper, they seemed more overwhelming than they were in reality. I learned that explicitly listing my tasks and giving an approximate time needed to finish them helps with handling complex assignments. Therefore, while working on the project, I created

³ Understanding Threads and Locks. (n.d.).
https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html

a list of to-do's, which helped me with the clarity needed to complete all the tasks. I think it was especially important in the process of developing the game that we planned our time carefully, and started early on with the design.

2. To what extent did your planning correspond to the actual progress during the project weeks? What made you deviate from your planning?

Our planning corresponded quite well to our actual progress on the project. However, once we were done implementing everything, we needed a couple of days to debug and do a lot of testing, especially by just playing the game a lot of times. Almost every time, we found a new bug. By simply making the server log every command that comes in and every action that is performed, we could find bugs relatively easy. These logs were largely already in place since the code for the initial TUI (when we did not implement the networking yet) would already do this. We left this code there, so it's also possible to see on the server what happens. Overall, debugging took more time than we anticipated, but we were able to plan around this very well, because our planning was quite broad.

Something that took way less time than anticipated, was the computer player. Since we were not sure we would participate in the tournament, as we could not estimate whether our game would be working by then, we decided on making a bare-bones computer player. It turned out that the requirements for making a functioning computer player were quite simple, only choosing a good wildcard color to play took a bit more time, but barely.

We do not think that we had a period where we had a loss of momentum. We knew that we would not spend any time on the project during the two weeks of holiday. The same goes for week 7 since we both had to focus on math. In week 8, although we focused on the programming test, we were able to finish the documentation and enums of the protocol on Monday. This was the deadline that we set with our mentor group. We feel like we spent some more time on the protocol than necessary, but that was mainly because we were very open to answering questions from other groups. We also posted a lot of code snippets and examples to help the other groups communicate using the protocol.

3. Which counter measures did you take to compensate for deviations from your original planning? What was the impact of this on the intended scope or quality of the project?

We did not have major deviations from our original planning, and therefore we did not have to take any counter measures. The only deviation is that we wanted to finish the game by Sunday of week 9, but we ended up also needing the Monday of week 10, meaning that we would have two days for the report instead of three. This was fine, since taking three days for the reporting was a rather broad estimate. We do not think that this made a difference on the quality of the project.

4. What did you learn from this experience for your next (project) planning? Take your answers to the other questions into account and ask yourself how you would want to prevent this or deal with this next time.

Wouter

Usually, I pick a partner that I know can plan very well. This (while sounding contradictory) means that we do not have to plan in too much detail. We both know what is going on and we both know when something must be finished. We do not have to come together solely for the purpose of making a planning. However, I find it useful to make a broad planning for myself, and also for my group in general, like we did for this assignment. To be more specific, I learnt from this project better recognize what to plan and what not to plan. Things like debugging are extremely hard to plan, since you will never know what you can encounter. Only taking a rough estimate for the time necessary to debug is possible. On the other hand, for the project,

we knew what the requirements were, and we knew that we would need 2-3 days to make a report that was up to our standards.

Iulia

For me, planning for this project has been a great lesson in organizing time and resources for a long time, while working on a bigger project. While last module's projects were less complex in my opinion, for this one we had a bigger workload for just 2 people, so it was important that we keep up with the deadlines for the milestones and the final submission. Although 7 weeks seems like plenty of time to develop the game, it was important to keep up with all other subjects and unforeseeable situations. Therefore, we were constantly working on the project, with a break during the holidays, and tried not to leave an unmanageable amount of work for the final weeks. This planning also showed, once again, how important it is to budget enough time for tasks that are not easy to estimate, so that if needed, more time can be spent on them.

5. Suppose that next year you are a teaching assistant for this project. Give at least two do's and don'ts that you would tell your students to help them with their planning.

Wouter

Do's:

1. First off, I would tell my students to try and write down everything that they think that they must do. Then I would ask them to think about whether something is plannable or not, e.g., debugging vs. participating in the tournament. The things that are not plannable, I would advise them to think about a reasonable, a minimum, and a maximum time estimate, and then use the maximum time estimate in their planning. This means that they won't have to compromise on something else if something takes longer than expected.
2. Secondly, I would advise them to make a planning before the Christmas holidays, since that is around the time that they have the deadline for the game logic. I saw some of my friends struggle with that deadline, and some were still struggling with the game logic in week 9 (1 week before the final deadline).

Don'ts:

1. First, I would advise them to not try to make a planning immediately at the start of the project. I would advise them to deliver the class structure at the first deadline first, since the activities that they must do for that deadline are not really plannable, only with a very rough time estimate. The deadline will provide them with valuable feedback.
2. Second off, I would advise my students to try and not shuffle their planning that much after they made it. Of course, they need to leave some room for uncontrolled events, but they should not constantly procrastinate on one thing, while doing the other (also required) things instead.

Iulia

Do's:

1. I believe it is especially important to consider all other subjects while making the planning for the project. Therefore, while making the planning in the first half of the module, it is important to consider all other exams and deadlines they might have, as well as potential resits.
2. I would advise my students to be consistent with following the rough planning they made. It is important to deliver on time the 2 milestones, as this is a first indicator that you are on the right track timewise with the project. If you finish a task before the expected time, start on the next one,

even if it is before the planned time. This is especially important because of how hard it might be to estimate the duration of a task, so it's better to take advantage of any extra time.

Don'ts:

1. Do not schedule most of the work for the project for the final weeks of the module. This is a mistake I've seen some of my friends made, and it would cause great stress and it might be hard to repair.
2. Do not forget to include rest time, as it is important to have enough time to also spend time away from work, especially during the holiday season.
3. Do not procrastinate, regardless of how easy it is to do so. The budgeted time should be enough to cover all tasks, and postponing them might create an overload that can be unmanageable.

Appendix A: Academic Skills planning

Week #	Items	Assignee(s)
4	Class structure: <ul style="list-style-type: none"> ▪ Cards ▪ Card piles ▪ Players ▪ Class diagram 	Iulia & Wouter
5	Game logic: <ul style="list-style-type: none"> ▪ Cards (number & action) ▪ Card piles (draw pile, discard pile and hand) ▪ Exceptions ▪ JUnit test for draw pile 	Iulia & Wouter
6	Game logic: <ul style="list-style-type: none"> ▪ Player abstract class and HumanPlayer ▪ Drawing and playing cards ▪ Playing action cards Other: <ul style="list-style-type: none"> ▪ TUI 	Iulia & Wouter
7	Focusing on math test <ul style="list-style-type: none"> ▪ Documentation of protocol 	Iulia & Wouter
8	Focusing on programming test <ul style="list-style-type: none"> ▪ Protocol enums and examples ▪ Computer player logic ▪ Split codebase up into model, view, and controller 	Iulia & Wouter
9	<ul style="list-style-type: none"> ▪ Convert current app into server ▪ Create client app ▪ Create TUI for client ▪ Implement all protocol commands ▪ Convert own enums to use the protocol's enums ▪ Create chat ▪ Implement multi game server 	Iulia & Wouter
10	<ul style="list-style-type: none"> ▪ Implementation of computer player ▪ Create lobby functionality ▪ Fix bugs ▪ Writing report <ul style="list-style-type: none"> ○ Design (Wouter) ○ Testing (Iulia) ○ Academic skills 	Iulia & Wouter