Ludwig-Maximilians-Universität

SoftwareEntwicklungsPraktikum

SEP

Rummikub-Currygang

Author(s):
Cedrik Harrich
Till Kleinhans
Johannes Messner
Hyunsung Kim
Ella Mayer
Angelos Kafounis

Supervisor:
Nicolas Brauner

28. Januar 2019

Inhaltsverzeichnis

1 Introduction **3**

1.1 Setup . . . 3

1.2 Rules . . . 3

1.2.1 Play . . . 3

1.2.2 Valid Sets . . . 3

1.2.3 Manipulating Existing Sets . . . 4

1.2.4 Winning . . . 5

1.3 Progress . . . 5

2 Tasks **7**

2.1 Minimal Requirements . . . 7

2.2 Further Requirements . . . 8

2.3 Diagrams . . . 8

3 Goals and Milestones **8**

3.1 Main Goal . . . 9

3.2 Definition of Done . . . 9

3.3 Milestone 1: 21.12.2018 . . . 10

3.4 Milestone 2: 04.01.2019 . . . 10

3.5 Milestone 3: 14.01.2019 . . . 10

3.6 Milestone 4: 21.01.2019 . . . 11

3.7 Milestone 5: 28.01.2019 . . . 11

3.8 Milestone 6: 04.02.2019 . . . 11

4 Organisation **11**

4.1 Communication . . . 12

4.2 Workload(NoCode) . . . 12

4.3 Workload (Code) . . . 13

4.4 Programming Specifications . . . 16

5 Functionalities **16**

5.1 UseCase Diagrams . . . 16

5.2 Class Diagram . . . 18

5.3 Sequence Diagram 25

6 GUI **27**

6.1 LoginPhase 27

6.2 GamePhase 29

7 Verfication **29**

7.1 Unit Testing 29

7.2 Manual Testing 30

8 Conclusion **30**

1 Introduction

In this course we will implement a game called Rummikub. It is a tile-based game for up to 4 players, combining elements of the card game rummy and mahjong.

1.1 Setup

There are 104 number tiles in the game (valued 1 to 13 in four different colors, two copies each) and 2 jokers. Initially every player draws 14 randomly from the bag. [2]
The youngest player will start the game. Play begins with the starting player and proceeds clockwise. [1]

1.2 Rules

1.2.1 Play

For a player's first move, they must play a set (or sets in some versions) with a value of at least 30 points. Point values are taken from the face value of each tile played, with the joker (if played) assuming the value of the tile it is being used in place of. A player may not use other players' tiles to make the initial meld. If a player cannot make an initial meld, they must pick up a single tile from the pool and add it to their rack. Play then proceeds to the next player.
Once a player has made their initial meld, they can, on each turn, play one or more tiles from their rack, making or adding to groups and/or runs. If the player cannot (or chooses not to) play any tiles, they must pick a tile randomly from the pool and add it to their rack.
Players may play tiles by amending sets already in play. The only limit to the length of a run is the extremes of the tile values. Groups are limited to four because colors may not repeat within a group. [2]

1.2.2 Valid Sets

All tiles in play must be arranged in sets of at least three tiles. The two valid set types are called runs and groups.
A run is composed of three or more, same-colored tiles, in consecutive number

order.
A group is made from three or four same-value tiles in distinct colors.

1.2.3 Manipulating Existing Sets

During a player's turn, sets of tiles that have already been played may be manipulated to allow more tiles to be played. At the end of the turn, all played tiles must be in valid sets.

- Shifting a run: Players may add the appropriate tile to either end of a run and remove a tile from the other end for use elsewhere. If red 3, 4, and 5 have already been played, a player may add the red 6 to the end and remove the 3 for use elsewhere.
- Splitting a run: Players may split long runs and insert the corresponding tiles in the middle. Thus, if blue 6, 7, 8, 9, and 10 are already a run, the player may insert his own 8 to make two runs: 6, 7, 8 and 8, 9, 10.
- Substituting a group: Players may replace any of the tiles in a three-tile group with a tile of the fourth color and the same value. If blue 6, red 6, and orange 6 are already a group, the player may add the black 6 and remove any one of the other three for use elsewhere.
- Removing tiles: So long as the remaining tiles form a valid run, tiles can be removed from the ends of runs. Any one tile may be removed from a four-tile group.
- Joker Substitution: A joker may not be retrieved before the initial meld. A joker can be retrieved from a set by replacing it with a tile of the same numerical value and colour it represents. The tile used to replace the joker can come from the player's rack or from the table. In the case of a group of 3, the joker can be replaced by the tile of either of the missing colours. A joker that has been replaced must be used in the player's same turn as part of a new set. A set containing a joker can have tiles added to it, be split apart or

have tiles removed from it. The joker has a penalty value of 30 points if it remains on a player's rack at the end of the game

- Harvested Tiles: In the course of a turn, a tile that is harvested from an existing set must be played during the turn; it cannot be kept for later use. Example: if there is a 3, 4, 5 run on the table, the 3 can be harvested by putting down the appropriately colored 6, but the 3 must be used during that turn, not kept in the player's hand for later use.

1.2.4 Winning

Game play continues until a player has used all of the tiles in the rack. That player is declared as the winner. If the pool runs out of tiles, play continues until there is a winner or no player can make a valid play.

1.3 Progress

Update: 2018.12.24
So far we created the LRZ-Gitlab Repo for the project. We made the first designs for the UML-Class Diagram and Sequence Diagram. Moreover we also have a UseCase Diagram and a design Pattern for the view.
Update: 2019.1.4
We adjusted the diagrams split up our workload so we are able to start programming now.
Update: 2019.1.14
We handled our workload with Trello and it worked good so far i.e. were able to stay in schedule. We are actually a little bit ahead. This is the first time we can present a list of features implemented so far:

- The Model/Logic part of the game is as good as finished. It needs to be tested though.
- You can use the main method in Shell so you can simulate a game in the command.
- You can start the main game window. You can draw Stoneboxes and drag Stone Boxes onto the grid.

- Two clints can connect to the server. The game starts and the clients can print/ get the stones on the table and the hand.
- The connection can be tested with the named Shell.

Update: 2019.1.21
We made good progress since last week:

- The model/logic part of the game is tested and finished.
- The network part is now able to interact with the game/model.
- One can start the game now from a .jar File.
- The Workload was documented properly in this document.
- Game Music was implemented.

Update: 2019.1.28
We have a running Game! We still have some minor bugs. And we need to polish the representation. The main tasks we fulfilled this week are the following:

- The model/logic of the game has JUnit Tests.
- The communication part has JUnit tests.
- The network works with the view.
- JavaDocs for most of our Code.
- We implemented all the Minimal Requirements from Section 2.1 for our Program.
- The UML Class Diagrams were updated.
- We have a game Manual.
- It is possible to drag a complete Group/Run by Left Click and STRG.

2 Tasks

In this chapter we will list all our duties that have to be fulfilled until the 4th of February 2019.

2.1 Minimal Requirements

The requirements our program needs to fulfill are [1]:

- The setup from section 1.1 and all the rules from section 1.2 have to be implemented.
- It should be possible to start a game with 2-4 Players.
- The youngest player starts the game.
- The players proceed clockwise.
- If it is your turn you should have 2 options: Draw a card or lay one down.
- The GUI shall be implemented with JavaFXML.
- Model-View-Controller must be used.
- It should be playable across a network with sockets.
- One of the players is the host and also holds the server.
- Information should be broadcasted as serialized Java Objects.
- Clients are sending their moves to the host.
- The host is checking for valid moves.
- If the move is legit the host updates the board and notifies all the other clients.
- If the move is not correct the player gets a notification and is able to try again.
- The host is configuring and starting the Game.

- Every Player is allowed to leave the game. (What about the host?)
- Port: 48410 needs to be used.
- When the game is over, every Player gets a Notification who won the game and the host is asked if he wants to start another game.
- One can close the program with the usual x-Button. Optionally via the menu or a Shortcut.

2.2 Further Requirements

In addition to the minimal Requirements:

- All the public classes have to be commented with a proper Java Doc.
- Unit Testing should be done for the Logic part.
- Final game was played at least 50 times without errors.
- Edge case tested

2.3 Diagrams

We do not only need to implement the program but we also need to have a good documentation that is up-to-date all the time. The central parts of the documentation will be:

- The product requirements document (this document)
- The MVC-class diagram
- The sequence diagram(s)
- The GUI-Design Template

3 Goals and Milestones

In this chapter we will define our main goal and dismember it into smaller Milestones with a workload of approximately a week.

3.1 Main Goal

We are done with our project, if:

1. We have a working and running program.
2. All of the rules from section 1.2 are implemented.
3. All the features from section 3.2 work.
4. Our program fulfills all of the requirements from section 2.1.
5. The documents mentioned in section 2.3 are complete and match our end result.
6. The program was tested for its main functionalities and was played at least 50 times.
7. The executable program and all the needed documents are uploaded to our GitLab account.
8. Our program was reviewed and approved.

3.2 Definition of Done

If the following features are fullfilled by a feature one can commit to master:

- Code is compiling.
- All the (public) methods of the features are documented with JavaDoc.
- All the classes/interfaces of the feature are documented with JavaDoc
- All the methods need to be tested manually.
- JUnit Tests for methods that handle logic or parsing.
- Closed the task on Trello.

3.3 Milestone 1: 21.12.2018

The workload will be:

- LRZ-Gitlab Repo for the project
- invite our Tutors
- First designs of the diagrams.

3.4 Milestone 2: 04.01.2019

The workload will be:

- Adjust diagrams
- Choose Design pattern
- Prototype Implementation

3.5 Milestone 3: 14.01.2019

The workload will be:

- Define our Definition of Done.
- Workplan (Trello)
- First version of our program with a list what is possible right now.
- The Model should be done and the Logic of the game should be implemented.
- The Framework for the server stands.
- We have the first features for our view.
- We implemented a TestShell so we can test our results without the need of the view part.
- Extra: We don't need to connect our threes parts right now but we can already work on the first Interfaces.

3.6 Milestone 4: 21.01.2019

The workload will be:

- Second version of our program with a list what is possible right now.

3.7 Milestone 5: 28.01.2019

The workload will be:

- Second version of our program with a list what is possible right now.
- List of planned/possible features

3.8 Milestone 6: 04.02.2019

The workload will be:

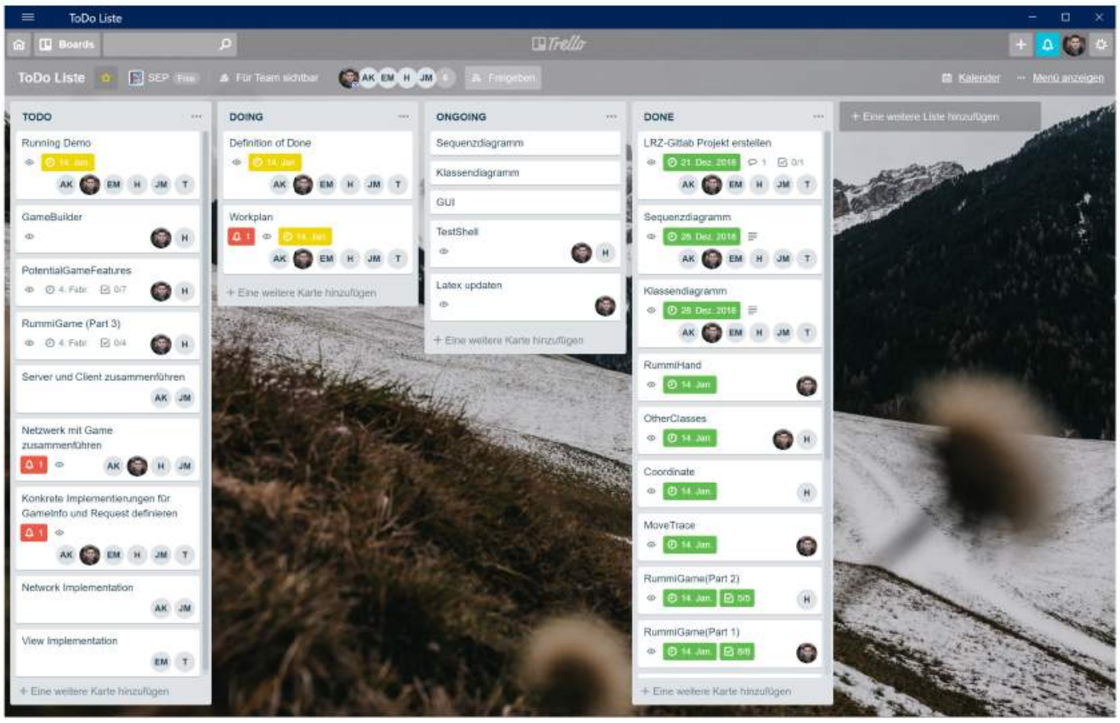
- Check all the Specifications
- Test the program
- Update the diagrams and documents
- JavaDoc and clean up code.
- Implement Features.
- Ship final working product, diagram and tests
- Prepare for Presentation.

4 Organisation

The offical platform for our project will be our gitlab account:
https://gitlab.lrz.de/ru96vik/rummikub—currygang

4.1 Communication

Since we have to use a lot of different platforms (Google Drive, WhattsApp, GitLab ...) we decided to mangage them with a central platform called Trello.If your want to have a look on our workflow click on the following link and you will be given permission to join our Trello Project:
trello.com/sep207
If you just want to have an idea how we handle our work here is a screenshot:



TODO
- Running Demo — 14. Jan — AK, EM, H, JM, T
- GameBuilder — H
- PotentialGameFeatures — 4. Febr — 0/7 — H
- RummiGame (Part 3) — 4. Febr — 0/4 — H
- Server und Client zusammenführen — AK, JM
- Netzwerk mit Game zusammenführen — 1 — AK, H, JM
- Konkrete Implementierungen für GameInfo und Request definieren — 1 — AK, EM, H, JM, T
- Network Implementation — AK, JM
- View Implementation — EM, T
- + Eine weitere Karte hinzufügen

DOING
- Definition of Done — 14. Jan — AK, EM, H, JM, T
- Workplan — 1 — 14. Jan — AK, EM, H, JM, T
- + Eine weitere Karte hinzufügen

ONGOING
- Sequenzdiagramm
- Klassendiagramm
- GUI
- TestShell — H
- Latex updaten
- + Eine weitere Karte hinzufügen

DONE
- LRZ-Gitlab Projekt erstellen — 21. Dez. 2018 — 1 — 0/1 — AK, EM, H, JM, T
- Sequenzdiagramm — 28. Dez. 2018 — AK, EM, H, JM, T
- Klassendiagramm — 28. Dez. 2018 — AK, EM, H, JM, T
- RummiHand — 14. Jan
- OtherClasses — 14. Jan — H
- Coordinate — 14. Jan — H
- MoveTrace — 14. Jan
- RummiGame(Part 2) — 14. Jan — 5/5 — H
- RummiGame(Part 1) — 14. Jan — 8/8
- + Eine weitere Karte hinzufügen

Abbildung 1: This is a small caption how we handle our workload in Trello.

4.2 Workload(NoCode)

In this subsection we register all the non code related tasks, who took care about each task X and who will take care about every specific task (X). See Table 1.

Tasks	Angelos	Hyunsung	Till	Johannes	Cedrik	Ella
LRZ-GitLab Repo						X
Invite everyone					X	X
UML Class Diagram (First Version)	X	X	X	X	X	X
UML Class Diagram (Second Version)	X	X	X	X	X	X
UML Class Diagram (Final Verison)	X	X	X	X	X	X
UML Sequence Diagram (First Version)	X	X				
UML Sequence Diagram (Final Version)	X	X				
UML Use Cases 1 (First Version)					X	
UML Use Cases 1 (Final Version)					(X)	
UML Use Cases 2 (First Version)					(X)	
UML Use Cases 2 (Final Version)					(X)	
GUI (First Version)					X	
GUI (Second Version)			X			
GUI (Final Version)			(X)			
Latex Document (First Version)					X	
Latex Document (Second Version)					X	
Latex Document (Third Version)					X	
Latex Documnet (Fourth Version)					(X)	
Latex Document (Final Version)					(X)	
Prepare Presentation					(X)	
Hold Presentation	(X)	(X)	(X)	(X)	(X)	
Abgabe 1					X	
Abgabe 2						X
Abgabe 3	X					
Abgabe 4		(X)				
Abgabe 5			(X)			
Abgabe Final	(X)	(X)	(X)	(X)	(X)	(X)

Tabelle 1: In this table all our tasked are displayed.

4.3 Workload (Code)

In this subsection we register all the code related tasks, who took care about each task X and who will take care about every specific task (X). See Table 2 and Table 3.

Tasks	Angelos	Hyunsung	Till	Johannes	Cedrik	Ella
Game: Class Coordinate		X				
Game: Class ConcreteGame		X				
Game: Class Game		X			X	
Game: Class Grid					X	
Game: Class MoveTrace					X	
Game: Class Player					X	
Game: Class RummiBag					X	
Game: Class RummiGame		X			X	
Game: Class RummiHand		X				
Game: Class RummiTable		X				
Game: Class Stone					X	
Game: Class TestShell					X	
Network: Class ClientListener	X			X		
Network: Class Controller	X			X		
Network: Class GameInfoHandler	X			X		
Network: Class RummiClient	X			X		
Network: Class Gamebuilder	X			X		
Network: Class RequestHandler	X			X		
Network: Class RummiHandler	X			X		
Network Class Server	X			X		
Network: Class ServerListener	X			X		
Network: Class ServerSender	X			X		
Communication: Class BagInfo	X	X		X	X	
Communication: Class GameInfo	X	X		X	X	
Communication: Class HandInfo	X	X		X	X	
Communication: Class InfoID	X	X		X	X	
Communication: Class StoneInfo	X	X		X		
Communication: Class TableInfo	X	X		X		
Communication: Class WrongMove	X	X		X		
Communication: Class YourTurn	X	X		X	X	
Communication: Class AbstractMove	X	X		X		
Communication: Class ConcreteHandMove	X	X		X		
Communication: Class ConcretePutStone	X	X		X		
Communication: Class ConcreteSetPlayer	X	X		X		
Communication: Class ConcreteTableMove	X	X		X		
Communication: Class ConfirmMoveRequest	X	X		X		
Communication: Class DrawRequest	X	X		X	X	
Communication: Class GetHand	X	X		X		
Communication: Class GetTable	X	X		X	X	
Communication: Class Request	X	X		X		
Communication: Class Start	X	X		X		

Tabelle 2: In this table all our tasked are displayed.

Tasks	Angelos	Hyunsung	Till	Johannes	Cedrik	Ella
View: Class DemoView			X			X
View: Class ClientModel			X			X
View: Class Controller			X			X
View: Class DemoView			X			X
View: Error.fxml			X			X
View: Class GameController			X			X
View: Class GameModel			X			X
View: Class Main			X			X
View: Class NetworkController			X			X
View: Class OpponentHand.fxml			X			X
View: Class OpponentHand.fxml			X			X
Shell: Class Start Controller			X			X
View: Stone.fxml			X			X
View: Class WaitController			X			X
View: Class WaitView			X			X
View: Game.fxml			X			X
View: gameStyle.css			X			X
View: startStyle.css			X			X
View: wait.fxml			X			X
View: startMusic.mp3	X					
View: waitingMusic.mp3	X					X
JUnit Testing of game					(X)	
JUnit Testing of network					(X)	
JUnit Testing of View					(X)	
Combine network with view	(X)	(X)	(X)	(X)		(X)
View: Drag and Drop			(X)			
Wrong User Input(StartView)	(X)	(X)	(X)		(X)	
Layout (WaitView)	(X)	(X)	(X)		(X)	
Layout (GameView)	(X)	(X)	(X)		(X)	
DIfferent Skin (View)	(X)	(X)	(X)		(X)	
Hoverstate for cell(GameView)	(X)	(X)	(X)		(X)	
Drag Stones (GameView)	(X)	(X)	(X)		(X)	
Json				(X)		

Tabelle 3: In this table all our tasked are displayed.

4.4 Programming Specifications

We decided to use Java SDK 8 and IntelliJ as our IDE. For our Unit Tests we use JUnit 4 and for serializing JSon 2.85. Moreover we will try to format our Code according to the Google Java Style Specification.

5 Functionalities

The final product should be a complete Rummikub Game that is playable over a network. To model our game UML Diagrams come in handy.

5.1 UseCase Diagrams

There are basically 2 different phases in this game. First of all you have to start the program and set all the needed information like host or guest, name, age etc. as you can see in figure: 2.

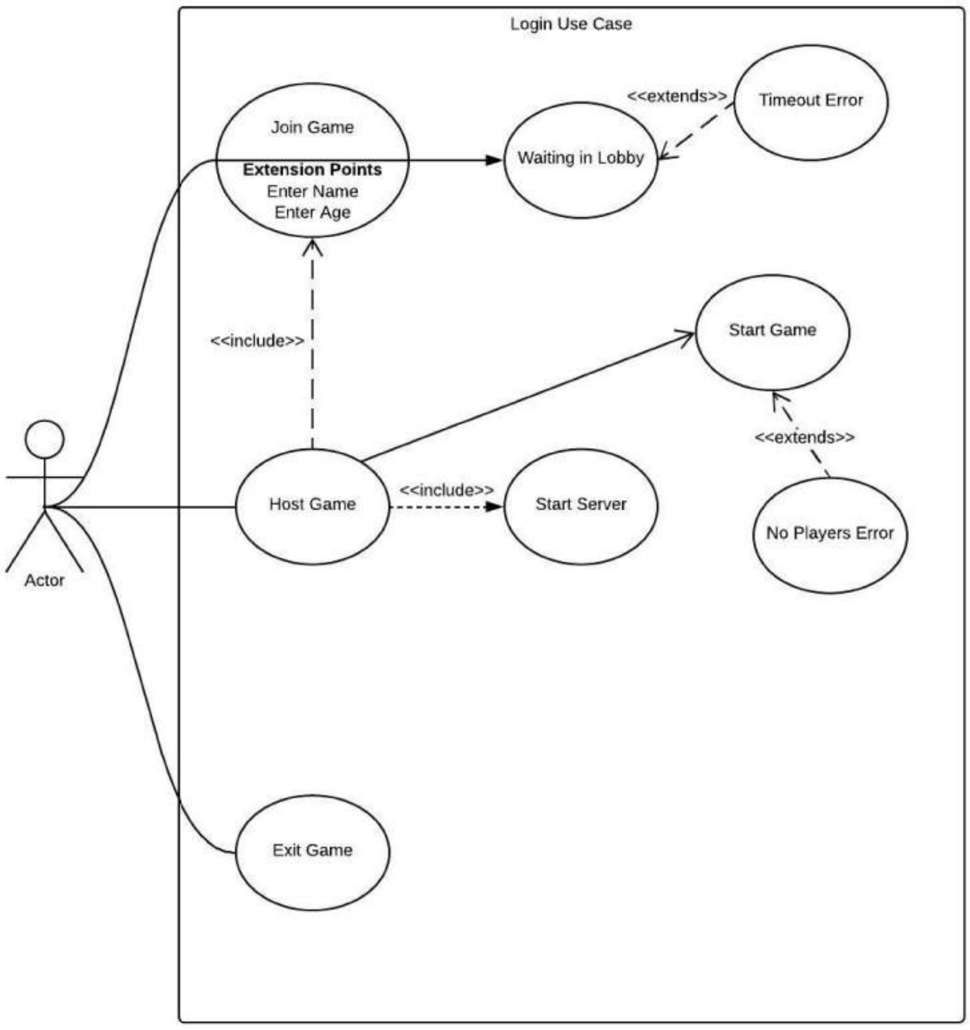


Abbildung 2: Use Case for the login screen interaction

The other standard phase is the acutual game where every player has the same options every turn as seen in figure 3.

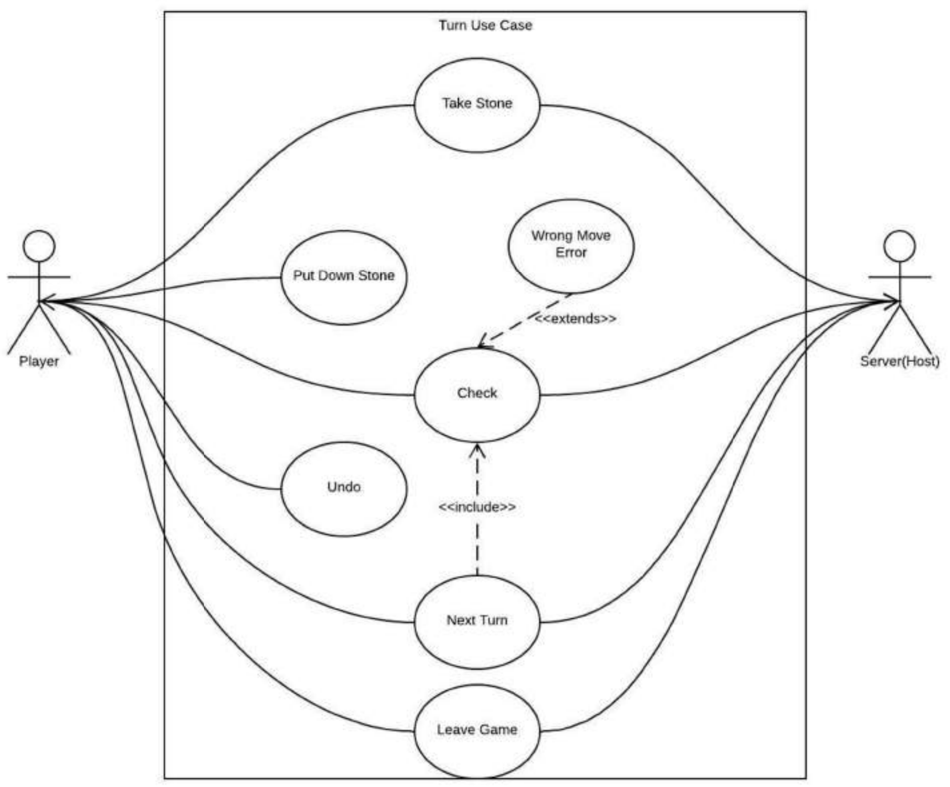


Abbildung 3: Use Case that shows what options every player has in every turn

5.2 Class Diagram

Since the last time our class diagrams have changed a little bit and since it got so big we divided the illustration by their packages as you can see in the following diagrams:

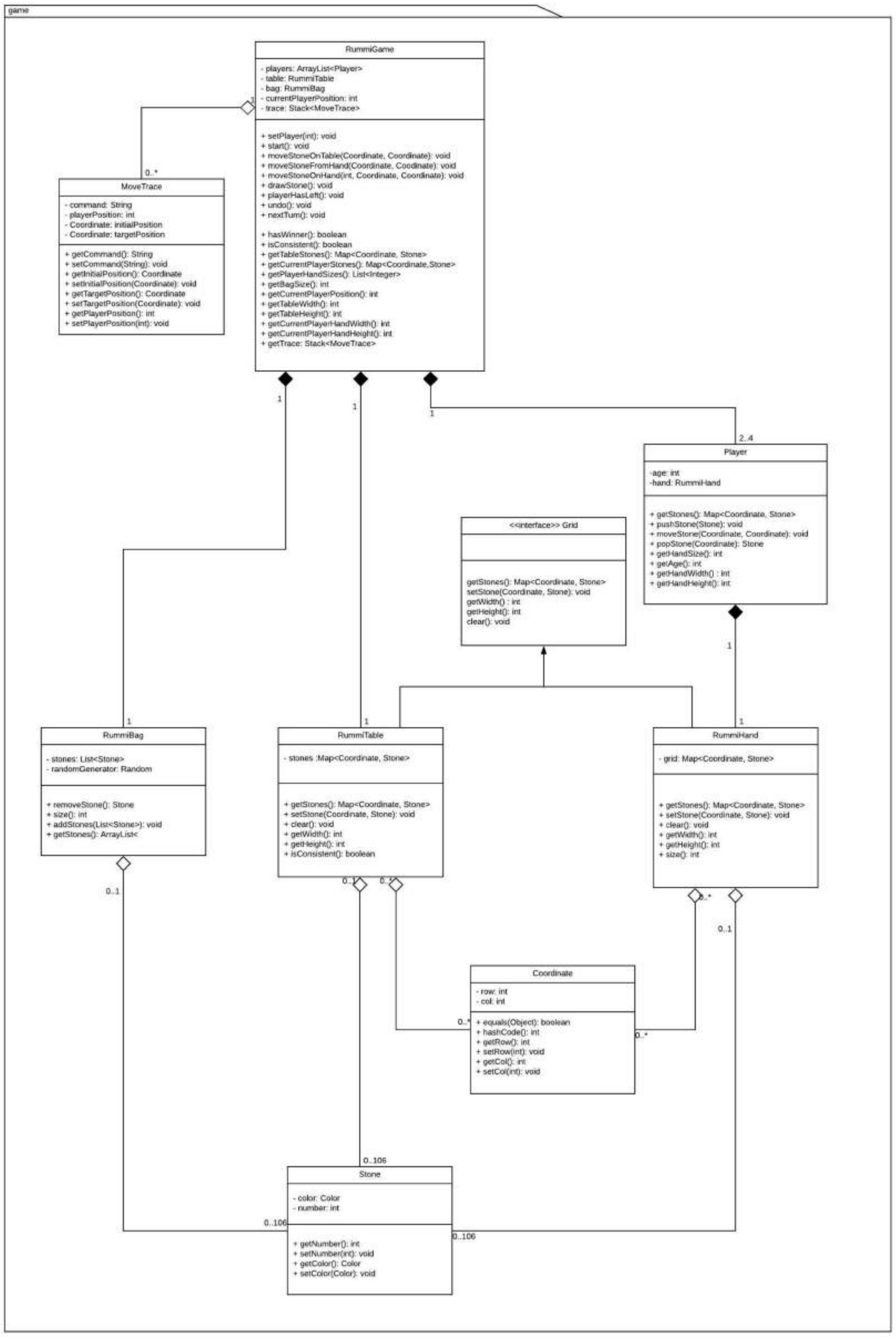

game
RummiGame
- players: ArrayList<Player>
- table: RummiTable
- bag: RummiBag
- currentPlayerPosition: int
- trace: Stack<MoveTrace>
+ setPlayer(int): void
+ start(): void
+ moveStoneOnTable(Coordinate, Coordinate): void
+ moveStoneFromHand(Coordinate, Coodinate): void
+ moveStoneOnHand(int, Coordinate, Coordinate): void
+ drawStone(): void
+ playerHasLeft(): void
+ undo(): void
+ nextTurn(): void
+ hasWinner(): boolean
+ isConsistent(): boolean
+ getTableStones(): Map<Coordinate, Stone>
+ getCurrentPlayerStones(): Map<Coordinate,Stone>
+ getPlayerHandSizes(): List<Integer>
+ getBagSize(): int
+ getCurrentPlayerPosition(): int
+ getTableWidth(): int
+ getTableHeight(): int
+ getCurrentPlayerHandWidth(): int
+ getCurrentPlayerHandHeight(): int
+ getTrace: Stack<MoveTrace>
1
0..*
MoveTrace
- command: String
- playerPosition: int
- Coordinate: initialPosition
- Coordinate: targetPosition
+ getCommand(): String
+ setCommand(String): void
+ getInitialPosition(): Coordinate
+ setInitialPosition(Coordinate): void
+ getTargetPosition(): Coordinate
+ setTargetPosition(Coordinate): void
+ getPlayerPosition(): int
+ setPlayerPosition(int): void
1
1
1
2..4
Player
-age: int
-hand: RummiHand
+ getStones(): Map<Coordinate, Stone>
+ pushStone(Stone): void
+ moveStone(Coordinate, Coordinate): void
+ popStone(Coordinate): Stone
+ getHandSize(): int
+ getAge(): int
+ getHandWidth() : int
+ getHandHeight(): int
<<interface>> Grid
getStones(): Map<Coordinate, Stone>
setStone(Coordinate, Stone): void
getWidth() : int
getHeight(): int
clear(): void
1
1
1
1
RummiBag
- stones: List<Stone>
- randomGenerator: Random
+ removeStone(): Stone
+ size(): int
+ addStones(List<Stone>): void
+ getStones(): ArrayList<
RummiTable
- stones :Map<Coordinate, Stone>
+ getStones(): Map<Coordinate, Stone>
+ setStone(Coordinate, Stone): void
+ clear(): void
+ getWidth(): int
+ getHeight(): int
+ isConsistent(): boolean
RummiHand
- grid: Map<Coordinate, Stone>
+ getStones(): Map<Coordinate, Stone>
+ setStone(Coordinate, Stone): void
+ clear(): void
+ getWidth(): int
+ getHeight(): int
+ size(): int
0..1
0..1
0..*
0..*
0..1
Coordinate
- row: int
- col: int
+ equals(Object): boolean
+ hashCode(): int
+ getRow(): int
+ setRow(int): void
+ getCol(): int
+ setCol(int): void
0..*
0..*
0..106
Stone
- color: Color
- number: int
+ getNumber(): int
+ setNumber(int): void
+ getColor(): Color
+ setColor(Color): void
0..106
0..106

Abbildung 4: Class Diagram of the game package.

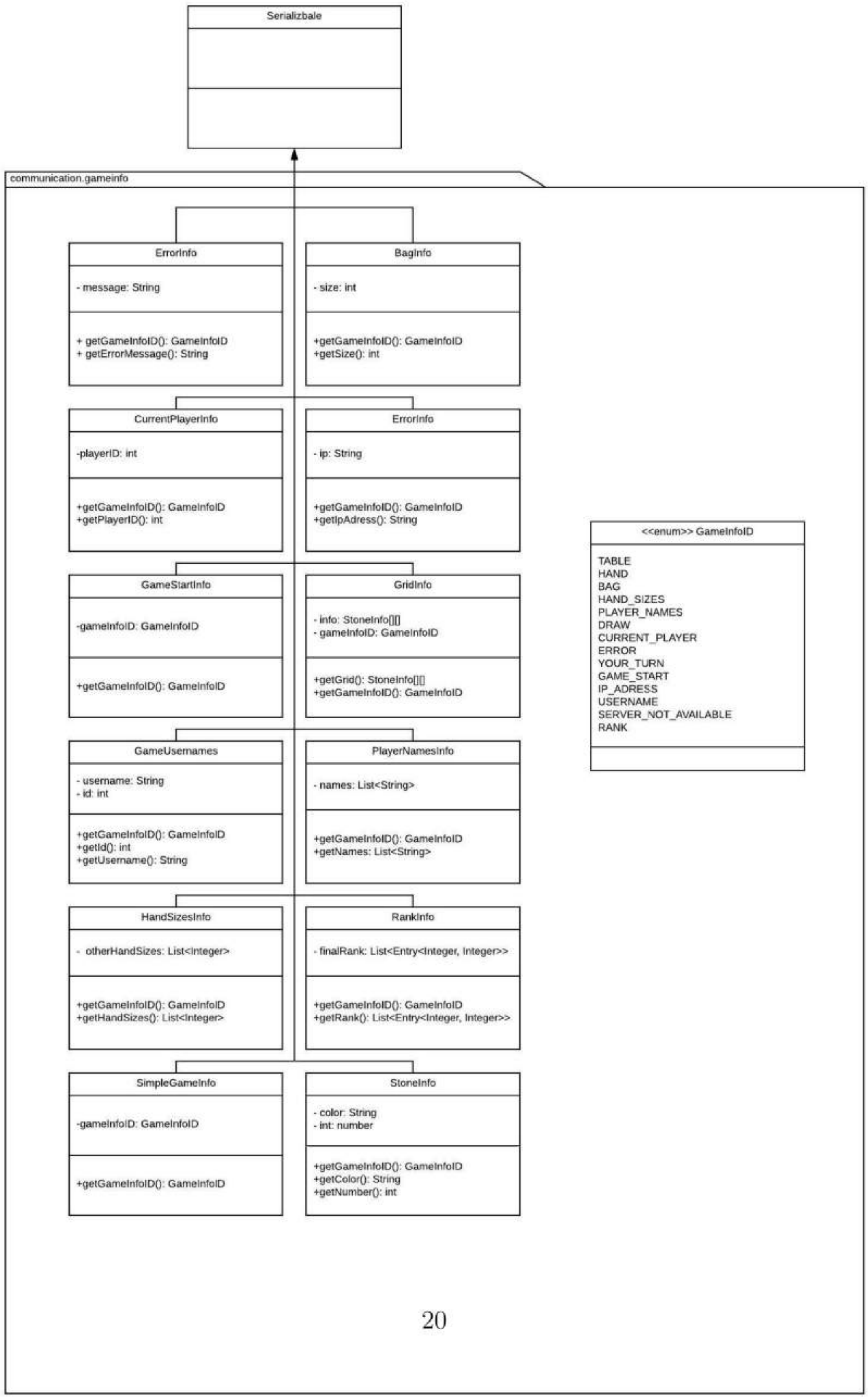


Abbildung 5: Class Diagram of the communication.gameinfo package

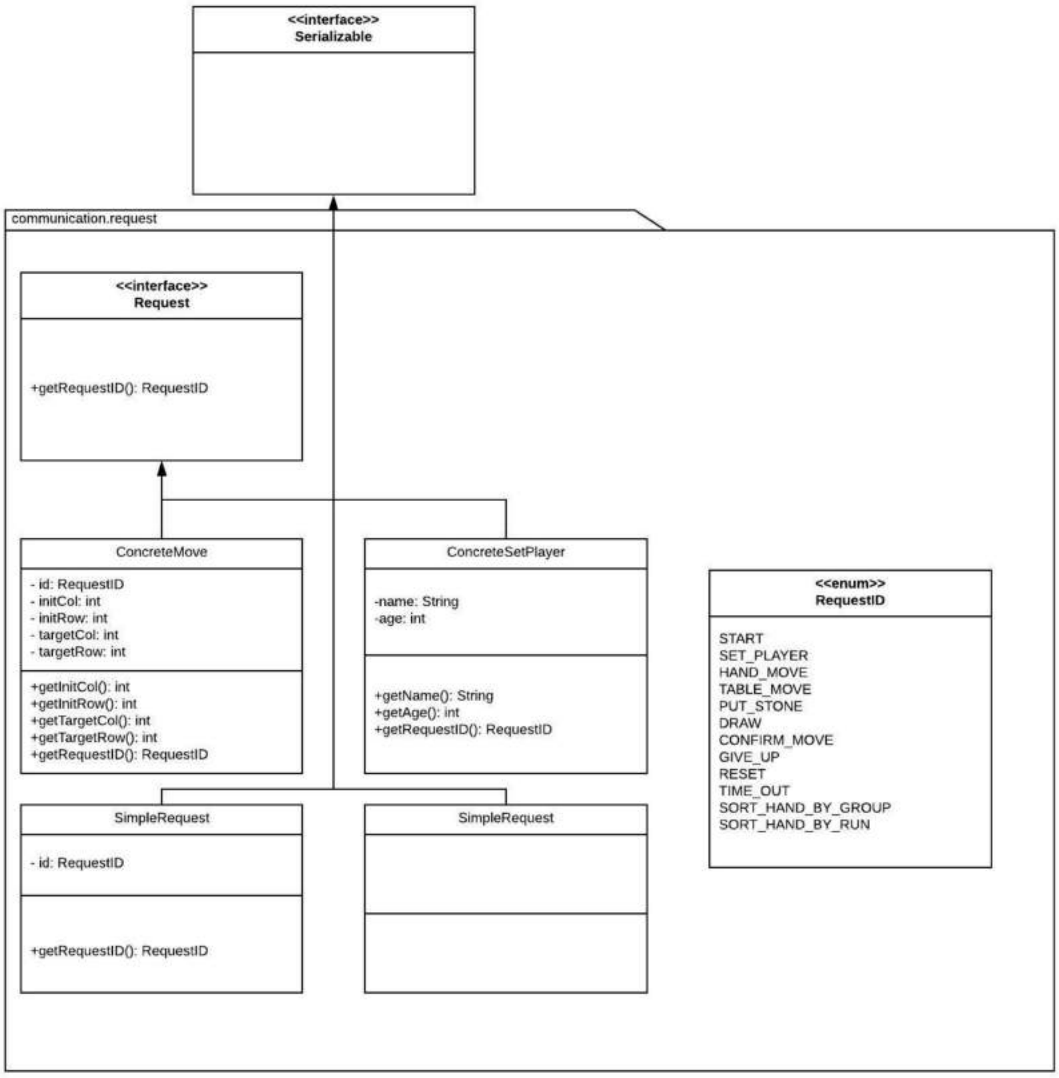


Abbildung 6: Class Diagram of the communication.request package

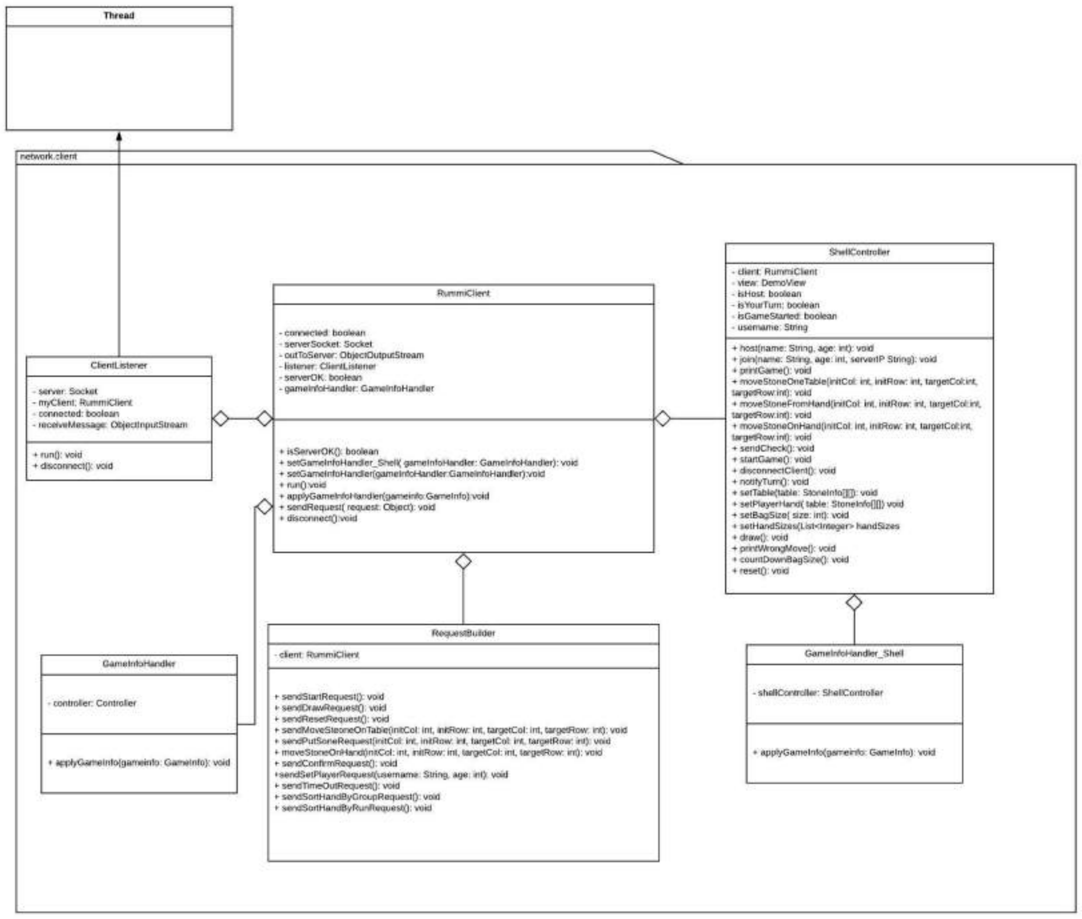

Thread
network.client
ClientListener
- server: Socket
- myClient: RummiClient
- connected: boolean
- receiveMessage: ObjectInputStream
+ run(): void
+ disconnect(): void
RummiClient
- connected: boolean
- serverSocket: Socket
- outToServer: ObjectOutputStream
- listener: ClientListener
- serverOK: boolean
- gameInfoHandler: GameInfoHandler
+ isServerOK(): boolean
+ setGameInfoHandler_Shell(gameInfoHandler: GameInfoHandler): void
+ setGameInfoHandler(gameInfoHandler:GameInfoHandler):void
+ run():void
+ applyGameInfoHandler(gameinfo:GameInfo):void
+ sendRequest(request: Object): void
+ disconnect():void
ShellController
- client: RummiClient
- view: DemoView
- isHost: boolean
- isYourTurn: boolean
- isGameStarted: boolean
- username: String
+ host(name: String, age: int): void
+ join(name: String, age: int, serverIP String): void
+ printGame(): void
+ moveStoneOneTable(initCol: int, initRow: int, targetCol:int, targetRow:int): void
+ moveStoneFromHand(initCol: int, initRow: int, targetCol:int, targetRow:int): void
+ moveStoneOnHand(initCol: int, initRow: int, targetCol:int, targetRow:int): void
+ sendCheck(): void
+ startGame(): void
+ disconnectClient(): void
+ notifyTurn(): void
+ setTable(table: StoneInfo[][]): void
+ setPlayerHand(table: StoneInfo[][]) void
+ setBagSize(size: int): void
+ setHandSizes(List<Integer> handSizes
+ draw(): void
+ printWrongMove(): void
+ countDownBagSize(): void
+ reset(): void
GameInfoHandler
- controller: Controller
+ applyGameInfo(gameinfo: GameInfo): void
RequestBuilder
- client: RummiClient
+ sendStartRequest(): void
+ sendDrawRequest(): void
+ sendResetRequest(): void
+ sendMoveSteoneOnTable(initCol: int, initRow: int, targetCol: int, targetRow: int): void
+ sendPutSoneRequest(initCol: int, initRow: int, targetCol: int, targetRow: int): void
+ moveStoneOnHand(initCol: int, initRow: int, targetCol: int, targetRow: int): void
+ sendConfirmRequest(): void
+sendSetPlayerRequest(username: String, age: int): void
+ sendTimeOutRequest(): void
+ sendSortHandByGroupRequest(): void
+ sendSortHandByRunRequest(): void
GameInfoHandler_Shell
- shellController: ShellController
+ applyGameInfo(gameinfo: GameInfo): void

Abbildung 7: Class Diagram of the network.client package

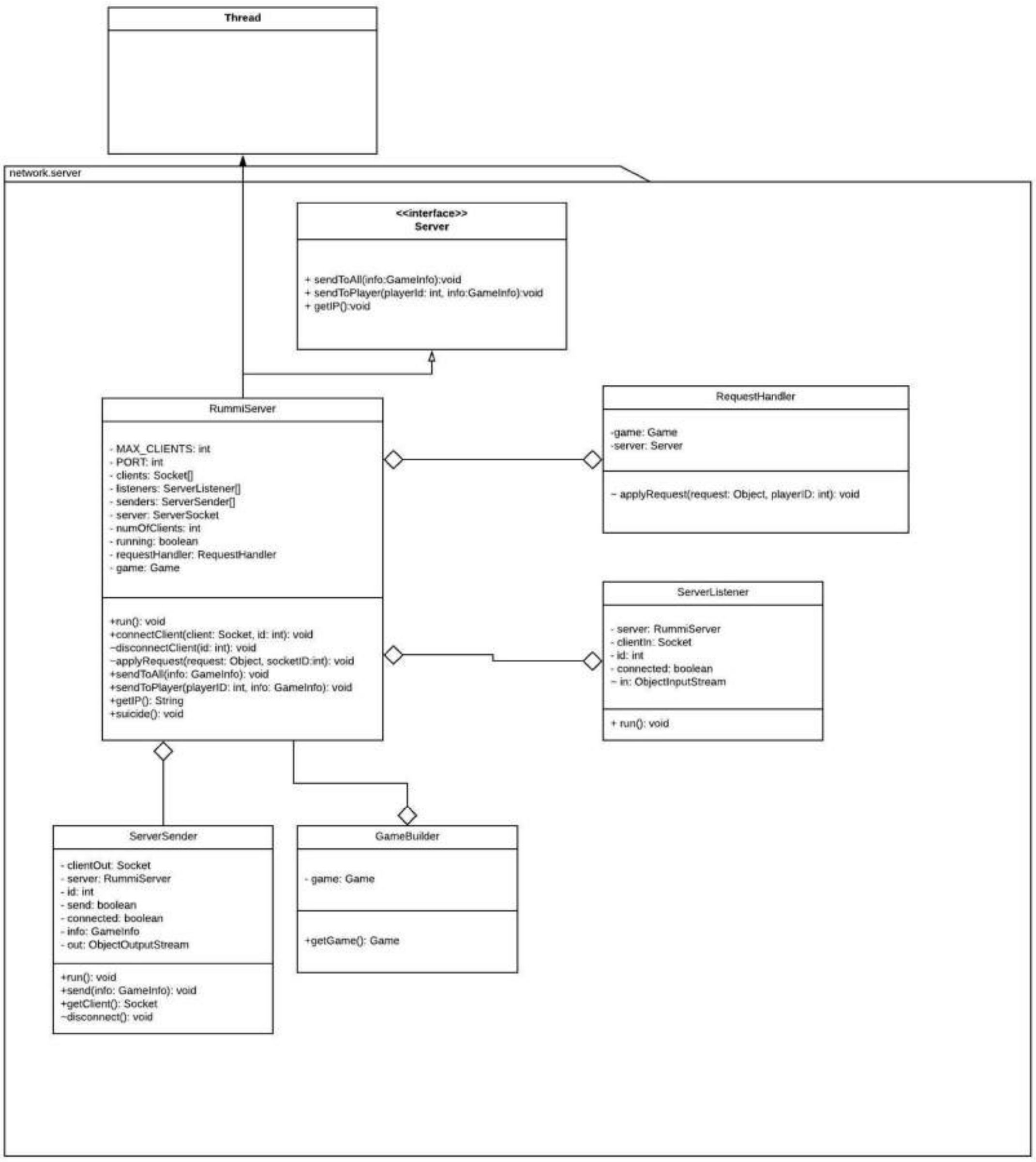


Abbildung 8: Class Diagram of the network.server package

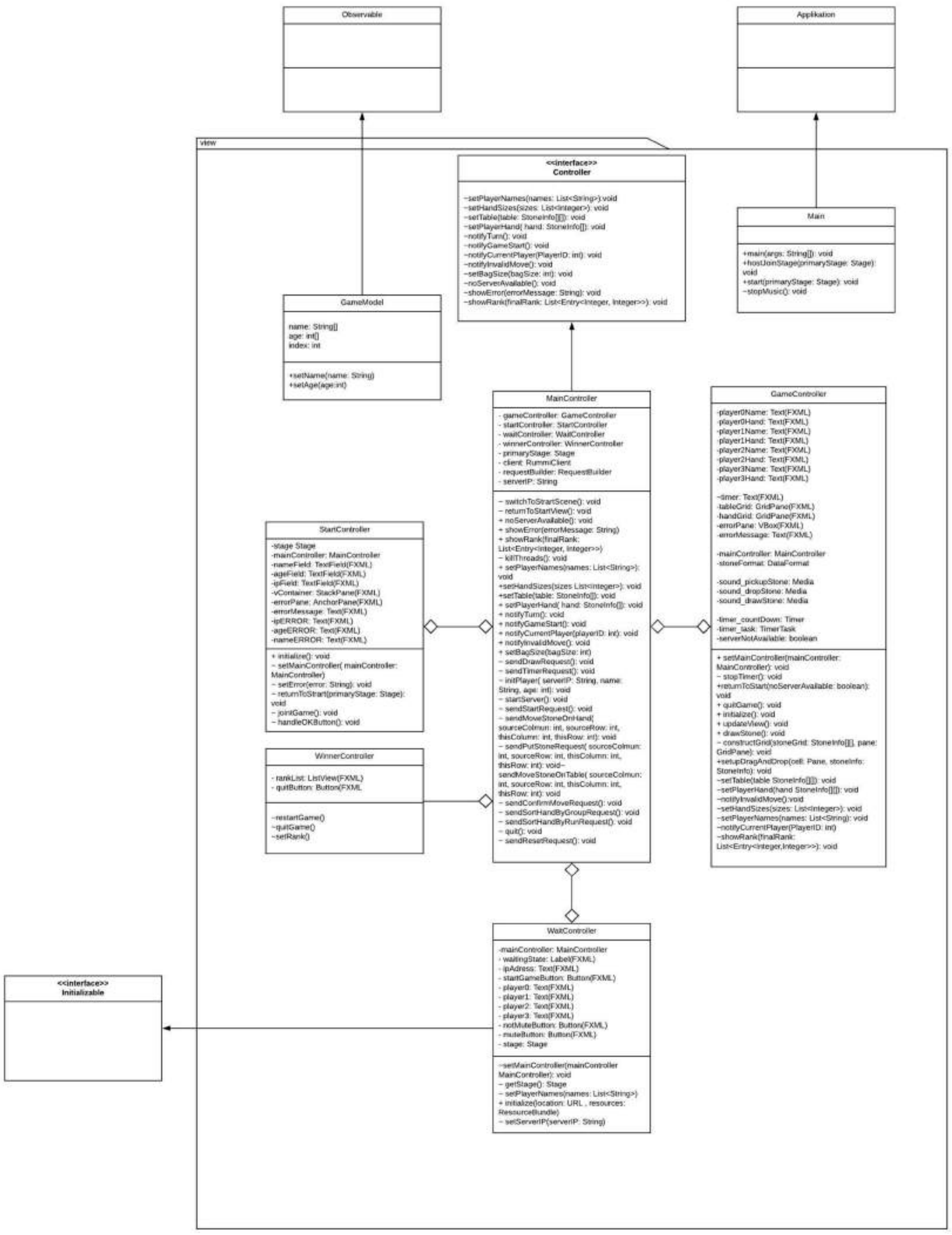

Observable
Applikation
view
<<interface>>
Controller
~setPlayerNames(names: List<String>):void
~setHandSizes(sizes: List<Integer>): void
~setTable(table: StoneInfo[][]): void
~setPlayerHand(hand: StoneInfo[]): void
~notifyTurn(): void
~notifyGameStart(): void
~notifyCurrentPlayer(PlayerID: int): void
~notifyInvalidMove(): void
~setBagSize(bagSize: int): void
~noServerAvailable(): void
~showError(errorMessage: String): void
~showRank(finalRank: List<Entry<Integer, Integer>>): void
Main
+main(args: String[]): void
+hostJoinStage(primaryStage: Stage): void
+start(primaryStage: Stage): void
~stopMusic(): void
GameModel
name: String[]
age: int[]
index: int
+setName(name: String)
+setAge(age:int)
MainController
- gameController: GameController
- startController: StartController
- waitController: WaitController
- winnerController: WinnerController
- primaryStage: Stage
- client: RummiClient
- requestBuilder: RequestBuilder
- serverIP: String
~ switchToStrartScene(): void
~ returnToStartView(): void
+ noServerAvailable(): void
+ showError(errorMessage: String)
+ showRank(finalRank: List<Entry<Integer, Integer>>)
~ killThreads(): void
+ setPlayerNames(names: List<String>): void
+setHandSizes(sizes List<Integer>): void
+setTable(table: StoneInfo[]): void
+ setPlayerHand(hand: StoneInfo[]): void
+ notifyTurn(): void
+ notifyGameStart(): void
+ notifyCurrentPlayer(playerID: int): void
+ notifyInvalidMove(): void
+ setBagSize(bagSize: int)
~ sendDrawRequest(): void
~ sendTimerRequest(): void
~ initPlayer(serverIP: String, name: String, age: int): void
~ startServer(): void
~ sendStartRequest(): void
~ sendMoveStoneOnHand(sourceColmun: int, sourceRow: int, thisColumn: int, thisRow: int): void
~ sendPutStoneRequest(sourceColmun: int, sourceRow: int, thisColumn: int, thisRow: int): void~ sendMoveStoneOnTable(sourceColmun: int, sourceRow: int, thisColumn: int, thisRow: int): void
~ sendConfirmMoveRequest(): void
~ sendSortHandByGroupRequest(): void
~ sendSortHandByRunRequest(): void
~ quit(): void
~ sendResetRequest(): void
GameController
-player0Name: Text(FXML)
-player0Hand: Text(FXML)
-player1Name: Text(FXML)
-player1Hand: Text(FXML)
-player2Name: Text(FXML)
-player2Hand: Text(FXML)
-player3Name: Text(FXML)
-player3Hand: Text(FXML)
-timer: Text(FXML)
-tableGrid: GridPane(FXML)
-handGrid: GridPane(FXML)
-errorPane: VBox(FXML)
-errorMessage: Text(FXML)
-mainController: MainController
-stoneFormat: DataFormat
-sound_pickupStone: Media
-sound_dropStone: Media
-sound_drawStone: Media
-timer_countDown: Timer
-timer_task: TimerTask
-serverNotAvailable: boolean
+ setMainController(mainController: MainController): void
~ stopTimer(): void
+returnToStart(noServerAvailable: boolean): void
+ quitGame(): void
+ initialize(): void
+ updateView(): void
+ drawStone(): void
~ constructGrid(stoneGrid: StoneInfo[][], pane: GridPane): void
+setupDragAndDrop(cell: Pane, stoneInfo: StoneInfo): void
~setTable(table StoneInfo[][]): void
~setPlayerHand(hand StoneInfo[][]): void
~notifyInvalidMove():void
~setHandSizes(sizes: List<Integer>): void
~setPlayerNames(names: List<String): void
~notifyCurrentPlayer(PlayerID: int)
~showRank(finalRank: List<Entry<Integer,Integer>>): void
StartController
-stage Stage
-mainController: MainController
-nameField: TextField(FXML)
-ageField: TextField(FXML)
-ipField: TextField(FXML)
-vContainer: StackPane(FXML)
-errorPane: AnchorPane(FXML)
-errorMessage: Text(FXML)
-ipERROR: Text(FXML)
-ageERROR: Text(FXML)
-nameERROR: Text(FXML)
+ initialize(): void
~ setMainController(mainController: MainController)
~ setError(error: String): void
~ returnToStrart(primaryStage: Stage): void
~ jointGame(): void
~ handleOKButton(): void
WinnerController
- rankList: ListView(FXML)
- quitButton: Button(FXML
~restartGame()
~quitGame()
~setRank()
WaitController
-mainController: MainController
- waitingState: Label(FXML)
- ipAdress: Text(FXML)
- startGameButton: Button(FXML)
- player0: Text(FXML)
- player1: Text(FXML)
- player2: Text(FXML)
- player3: Text(FXML)
- notMuteButton: Button(FXML)
- muteButton: Button(FXML)
- stage: Stage
~setMainController(mainController MainController): void
~ getStage(): Stage
~ setPlayerNames(names: List<String>)
+ initialize(location: URL , resources: ResourceBundle)
~ setServerIP(serverIP: String)
<<interface>>
Initializable

Abbildung 9: Class Diagram of the view package

5.3 Sequence Diagram

To make the relationships more clearly especially how classes interact with each other in a time perspective we also made a sequence diagram. (fig: 10)

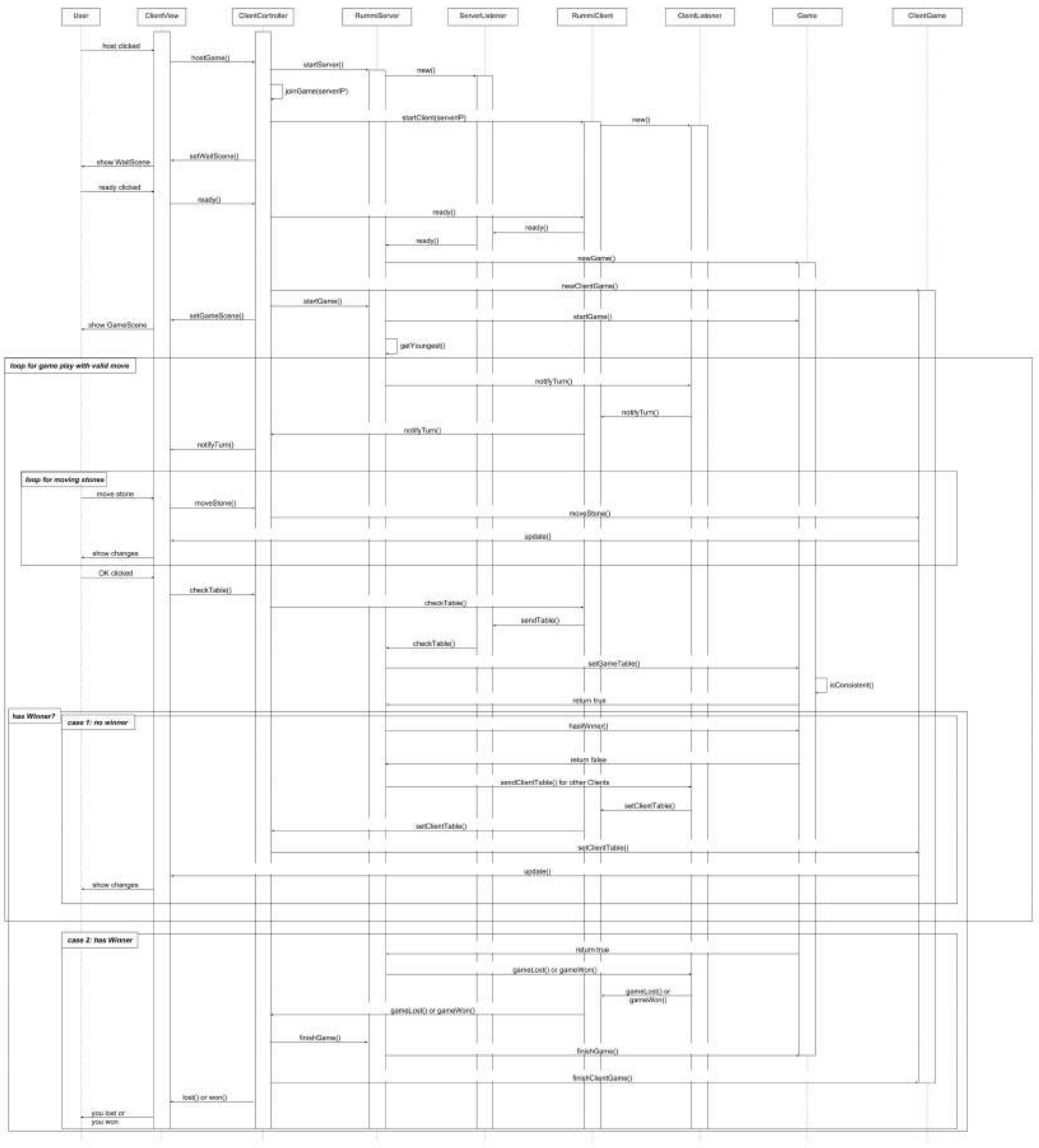


Abbildung 10: This sequence diagram should give insight how the actions are performed over time and how they interact with each other.

6 GUI

In this section we present the user interface. In the following figures you can see the different screens in the different phases of the game.

6.1 LoginPhase

The only purpose for this phase is to connect the players and asks for every information needed to start the game.

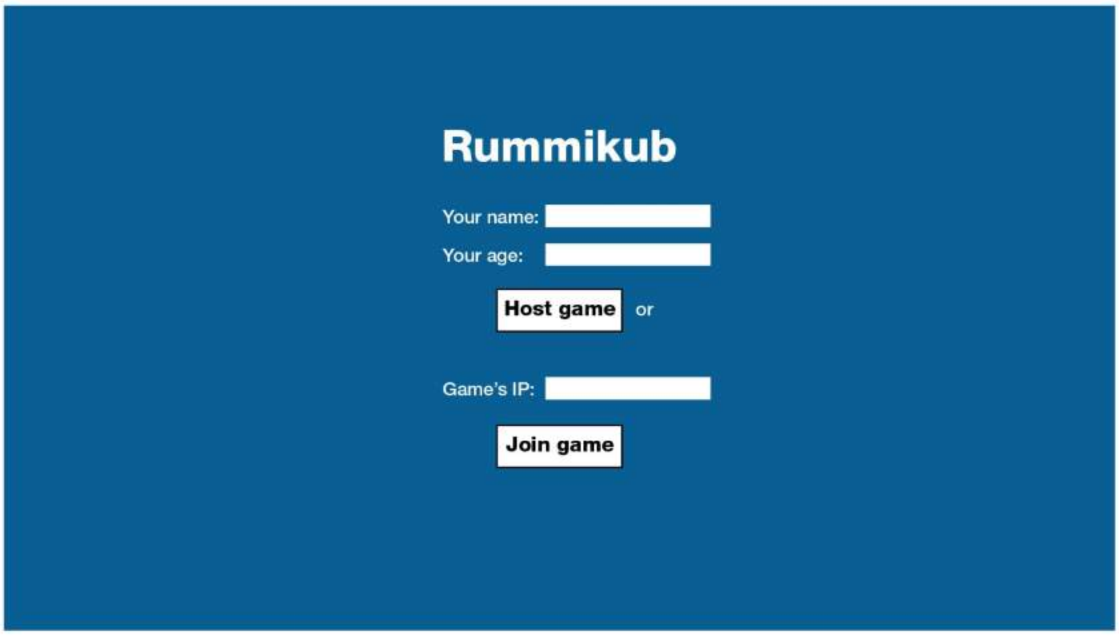


Abbildung 11: First Screen you will see after starting the program.

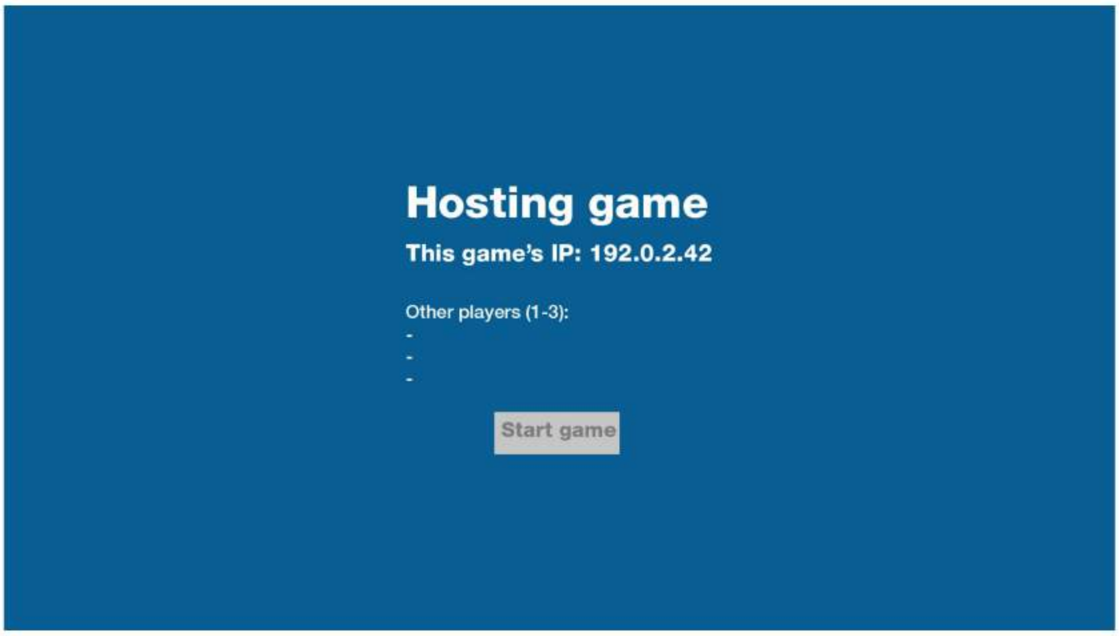


Abbildung 12: The screen you see right after clicking on host.

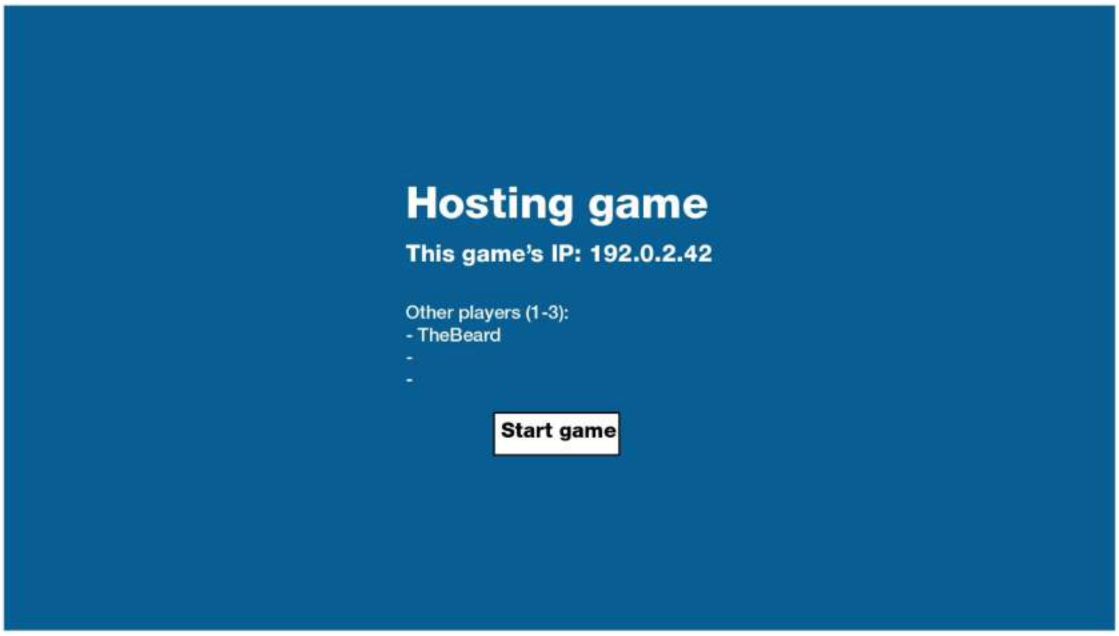


Abbildung 13: The screen the host sees when someone joined the game.

6.2 GamePhase

This phase displays the actual game.

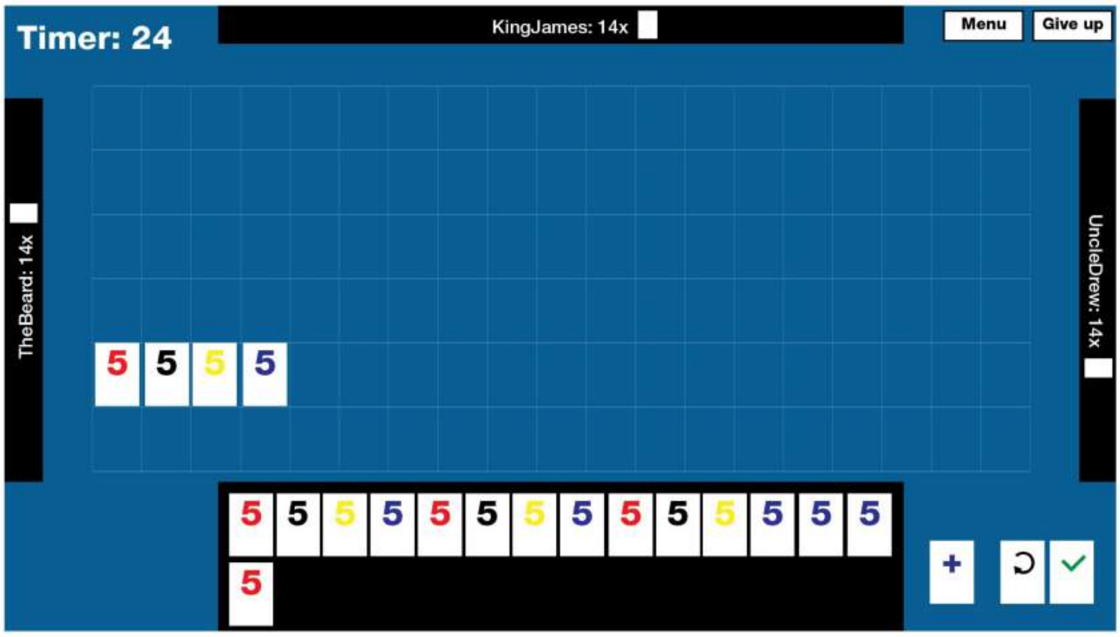


Abbildung 14: The screen you see when the host starts the game.

7 Verfication

In this section we will list all the methods used to check if our program works as planned. In each sub-chapter we briefly describe the used approach followed by a detailed documentation of our results.

7.1 Unit Testing

For UnitTesting we will use JUnit 4. It is a unit testing framework for the Java programming language. In our project we use it primarily to check the logic part of our program.

7.2 Manual Testing

In addition to JUnit testing every method/class is checked by a different team member.

8 Conclusion

Literatur

[1] D. Beyer. Praktikum sep: Java-programmierung. https://www.sosy-lab.org/Teaching/2018-WS-SEP/. Accessed: 2019-01-08.

[2] Wikipedia. Rummikub. https://de.wikipedia.org/wiki/Rummikub. Accessed: 2019-01-08.