VU Software Engineering 1

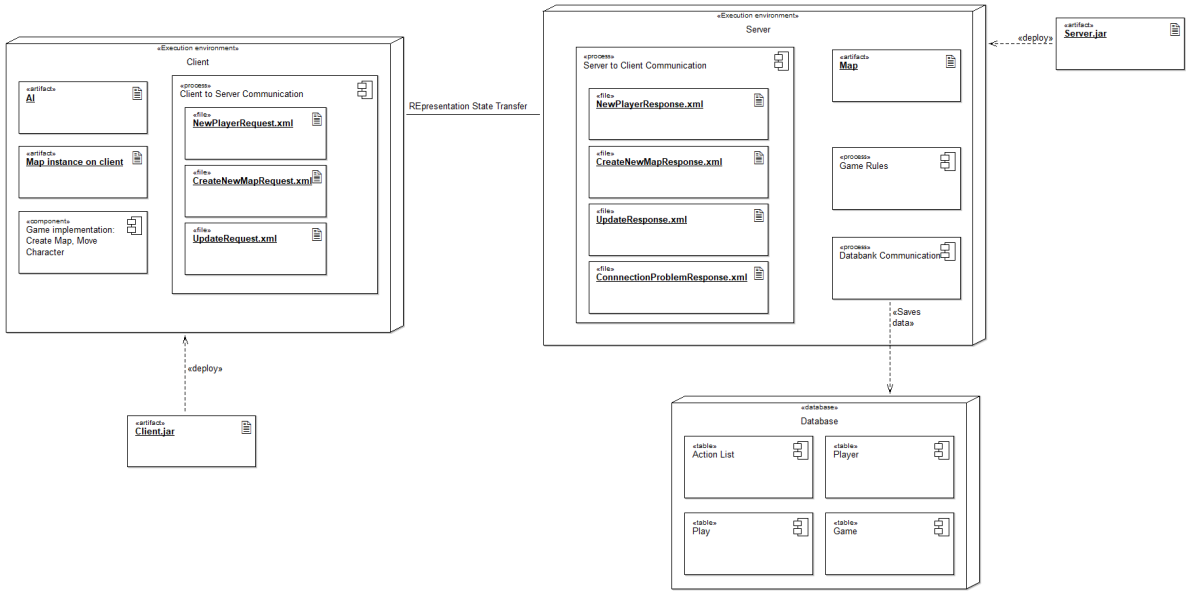
Abgabedokument

Teilaufgabe 2
(Anforderungsanalyse und Planungsphase)

Nachname, Vorname:	Nancu Razvan-Nicolae
Matrikelnummer:	01502339
E-Mail Adresse:	a01502339@unet.univie.ac.at
Datum:	08.11.2017

Rough Design / Architecture

Deployment Diagram:



The server and the client are deployed by their respective .jar files. The client has two components, the Game Implementation, where I just broadly described that it creates the map and moves the player around, and the Client to Server Communication, which sends various .xml files. The exact network will be described in a later chapter, while the deployment diagram only shows a broad overview. Two artifacts exist in the client, the AI and a map instance, which has been sent by the server after being created.

The server also has a Server to Client Communication component, with various .xml files. The two devices are connected through REST, the Representation State Transfer. The other two processes of the server are making sure the Game Rules are followed and the communication between the server and the database, in which various information is stored.

The Database is described in more detail in the last chapter, Persistence.

Use-Case Diagram:

For the rough design, I have decided to use an Use-Case Diagram, because it allows for a broad overview over the whole program and its systems. A class diagram will come in a future update of the document, in order to show the game implementation in further detail.

In the system, there will be two actors: the AI and the Player. The AI is also the Player, which is why I used a generalization. There are three systems, the Client, the Network and the Server, each taking care of a specific set of tasks. While the client takes care mostly of tasks that the AI and Player have an influence upon, including any messages that should be displayed for the human behind the screen, the server is important for anything behind the scenes, including making sure that the game works well and no rules are broken. The server decides when someone won or lost. The network simply makes it possible for the messages between the client and the server to be sent.

Firstly, the player creates a connection to the server, which in turn establishes a connection to both clients (player and enemy player) and then sends back an ID for each client. The “First connection Message” is the NewPlayerRequest and the “Connection established Message” is the NewPlayerResponse – which can be seen in more detail in the chapter about “Network Communication”. If there is a connection problem, or there is a problem generating the ID or there was a wrong data input, then the server sends a “Problem Message” called ConnectionProblemMessage. If a problem occurred, then it usually ends in an instant loss for the client where the problem happened.

The second task that takes place is creating a new map, which is created directly by the AI. The AI sends half of the map it has created to the server, through the “Create map Message”, CreateNewMapRequest. The server then checks if the rules were held correctly in “Check map generation”, so that there are 3 mountains, 4 water and 5 grass fields, no islands and so on. If the map was generated incorrectly or there was a connection problem, then the server gives the client a ConnectionProblemMessage. If the map was generated according to the rules, then the map halves are put together into one, a treasure is generated for each map half and put on a random grass tile, and then the full map is sent back to the client through “Get generated map Message”, called CreateNewMapResponse.

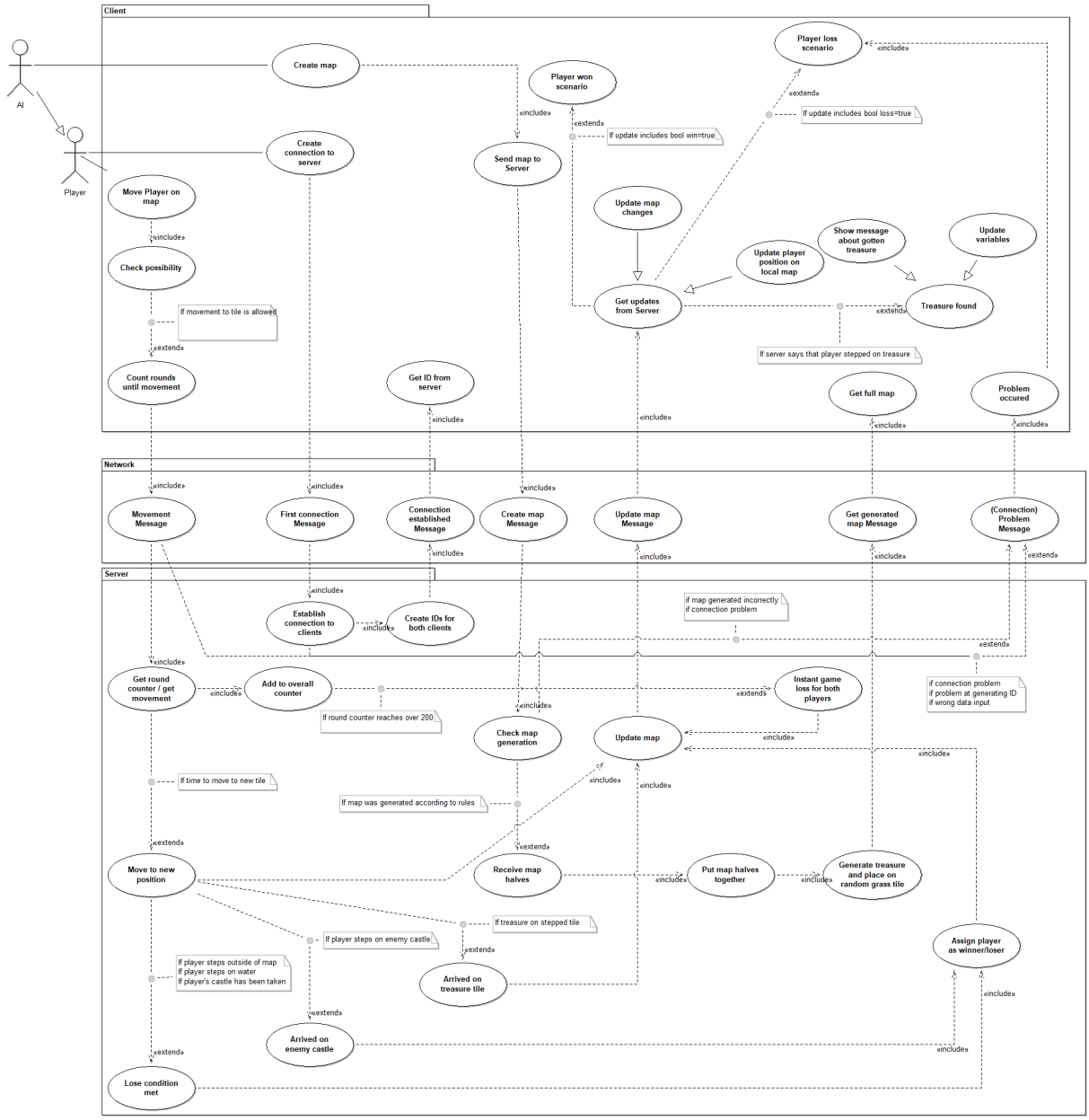
The core gameplay is the player movement across the map, controlled by the AI. The client checks if it’s possible to move to that field, and if it is allowed, then the client starts counting how many rounds it will take until the player can move to the new tile on the map. This “Movement Message” called UpdateRequest is sent to the server and has the current position, the wanted position and how many rounds it will take until the new position will be set.

Every time an UpdateRequest comes in, the server adds +1 to an overall counter, which shows how many game rounds have passed until now. If that counter reaches 200, then the game is an instant loss for both players.

If it is time to move to a new tile, then the player moves to a new position, to which the server checks for various possibilities: If the player has found the treasure, by stepping on the treasure tile on the map, met a losing condition, by either stepping outside of the map, or on water, or if the player's castle has been taken, and if the player has won by arriving on the enemy castle after finding the treasure. The winning and losing conditions are then assigned and all this information is then put together in an Update Message, called UpdateResponse, or "Update map Message" in the diagram.

The client gets the UpdateResponse and then does the following tasks: Firstly, it realizes any map changes, such as the current position of the enemy player. Secondly, it updates the player position on the map. Thirdly, it checks if the treasure has been found, which it then displays as a message to the human player and changes a boolean to "treasure=true". Lastly, the client checks if the server has sent a winning or a losing message, which in turn activates the winning or losing scenario.

Whenever a change happens, it is saved in a databank, which has not been drawn in this diagram, but is explained in more detail in the "Persistence" chapter at the end of the document.


Client
AI
Player
Create map
Player won scenario
Player loss scenario
Create connection to server
Send map to Server
Move Player on map
Update map changes
Show message about gotten treasure
Update variables
Check possibility
Update player position on local map
Get updates from Server
Treasure found
If movement to tile is allowed
If update includes bool win=true
If update includes bool loss=true
If server says that player stepped on treasure
Count rounds until movement
Get ID from server
Get full map
Problem occured
Network
Movement Message
First connection Message
Connection established Message
Create map Message
Update map Message
Get generated map Message
(Connection) Problem Message
Server
Establish connection to clients
Create IDs for both clients
if map generated incorrectly
if connection problem
Get round counter / get movement
Add to overall counter
Instant game loss for both players
if connection problem
if problem at generating ID
if wrong data input
If round counter reaches over 200
Check map generation
Update map
If time to move to new tile
If map was generated according to rules
Move to new position
Receive map halves
Put map halves together
Generate treasure and place on random grass tile
If treasure on stepped tile
If player steps on enemy castle
Assign player as winner/loser
If player steps outside of map
If player steps on water
If player's castle has been taken
Arrived on treasure tile
Arrived on enemy castle
Lose condition met
«include»
«extend»

User-Stories

- I expect the artificial intelligences to create their own maps and move across said maps without any human interaction.

- The server should be able to generate a treasure and place it randomly on the map, one for each player and on their respective halves. Neither the human player, nor the artificial intelligence, should be aware of the position of the treasure.

- I do not want to wait too long until the AI makes a move.

-I want to know whether I won or lost immediately after such an event happens.

- I want to know how long the match was, with a beginning and end time.

Business Rules

- The game is not longer than 200 rounds. This means there are 100 rounds available to each player.
“Um die Spiele für die Zuschauer spannend zu gestalten, wurde festgelegt, dass ein Spiel nicht länger als 200 Spielaktionen dauern darf” **Page 3**

- The AI may not step on a water field or outside of the map. Doing so means instant loss.
“Hierbei gilt, dass Wasser unter keinen Umständen betreten werden darf. Bewegt sich die Spielfigur einer KI in ein Wasserfeld, verliert die KI automatisch. Dies passiert auch wenn die Spielfigur mit dem gewünschten Bewegungsbefehl die Karte verlassen würde“ **Page 4**

- The AI may not think about its next move for longer than 3 seconds. Doing so means instant loss.
“ein KI für jede Aktion (Spielerbewegung, Kartengenerierung etc.) nicht mehr als 3 Sekunden Bedenkzeit zur Berechnung erhält“ **Page 3**

- The map has 8x8 tiles, and each half has at least 3 mountains, 5 grass fields and 4 water fields.
“Hierzu wird den KIs die zu verwendende quadratische Kartengröße (beispielsweise 8 x 8 Felder, die Karten sind vergleichbar mit einem kleinen Schachbrett)” **Page 4**
“jede Kartenhälfte muss mindestens 3 Bergfelder, 5 Wiesenfelder und 4 Wasserfelder beinhalten“ **Page 5**

- The AI may not generate islands, which are fields that are surrounded by water and can not be accessed by the player.
“Weiters dürfen keine Inseln generiert werden“ **Page 5**

- The treasure and the castle have to be placed on grass fields, but not on the same field. While the castle is known only to each respective player, the two treasures are hidden and only discovered either by being in the vision range of a mountain or by stepping on it.
“Burgen und Schätze dürfen nur auf Wiesenfeldern platziert werden.“ **Page 4**

- The player will not be controlled by any human interaction with the client. Only the artificial intelligence may move said player.
“Die grundlegende Spielidee ist, dass zwei KIs auf der gleichen Spielkarte eine vergleichbare Aufgabe erfüllen müssen.“ **Page 3**

Network Communication

Network Description

For the communication between the client and the server, we are going to implement REST, the Representational State Transfer. There will be three main tasks supported by this protocol (consisting of a Request and a Response respectively): The registration of the new player, sending the respective players‘ map halves and updating the game after each round. One more task for REST will be to notify the client when a problem happens.

When a player is registered, they will put in their name and matriculation number, and will receive an ID back, created by the server. The ID will make sure that no one can impersonate someone else. This will be implemented because REST does not save the state of the client, so there has to be a way to check if the right person is attempting to send new updates. From here on out, every new message will have the ID assigned by the server.

The map halves are sent the following way: The map itself will be sent as a string, which can then be decoded by the server and put it together with the other half, which in turn will be sent back as a string and decoded by the client. The string will have a number assigned for every two positions, while each number represents a specific value. Example: 01 will be grass field, 02 will be mountain, 03 will be water and 00 could be the castle (Which, by definition, is placed on a grass field already). Overall, the string sent by the player will be 64 characters long (32 fields, each field has two numbers) and the returned string will be 128 characters long, since it will be a whole map with 64 fields (8x8 tiles). There is no need to send any coordinates for the player spawn point, since the player spawns at the castle.

The most important message type sent between client and server will be the updates. The client sends at which position the player wants to move (x and y coordinate), as well as how many rounds are left until the movement takes place. The movement will be counted by the client – when the movement counter reaches 0, the server will know it should update the position of the player to the new position.

In turn, the server sends the following information back each round: Where the player is (x and y), where the enemy player is (x2 and y2), if the player has won (default 0, if won 1), if the player has lost (default 0, if lost 1) and if the player has found the treasure yet (default 0, if found 1).

The last type is the message that is sent, when a problem occurs. This will be broadly named Connection Problem Response, as it will not only hold any data input problems, but also connection problems. So if the connection fails between client and server, or if the client sends the wrong information, both errors will be respectively handled by this one message.

Requests and Responses

Client	**Server**
NewPlayerRequest(FirstName, Surname, Matnr.)	NewPlayerResponse(ID)
CreateNewMapRequest(ID, MapHalf)	CreateNewMapResponse(FullMap)
UpdateRequest(ID, CurrentPosition, NewPosition, TurnsLeft)	UpdateResponse(PositionOfPlayer, PositionOfEnemy, Treasure, Lost, Won)
ConnectionProblemRequest(ID)	ConnectionProblemResponse(Lost)

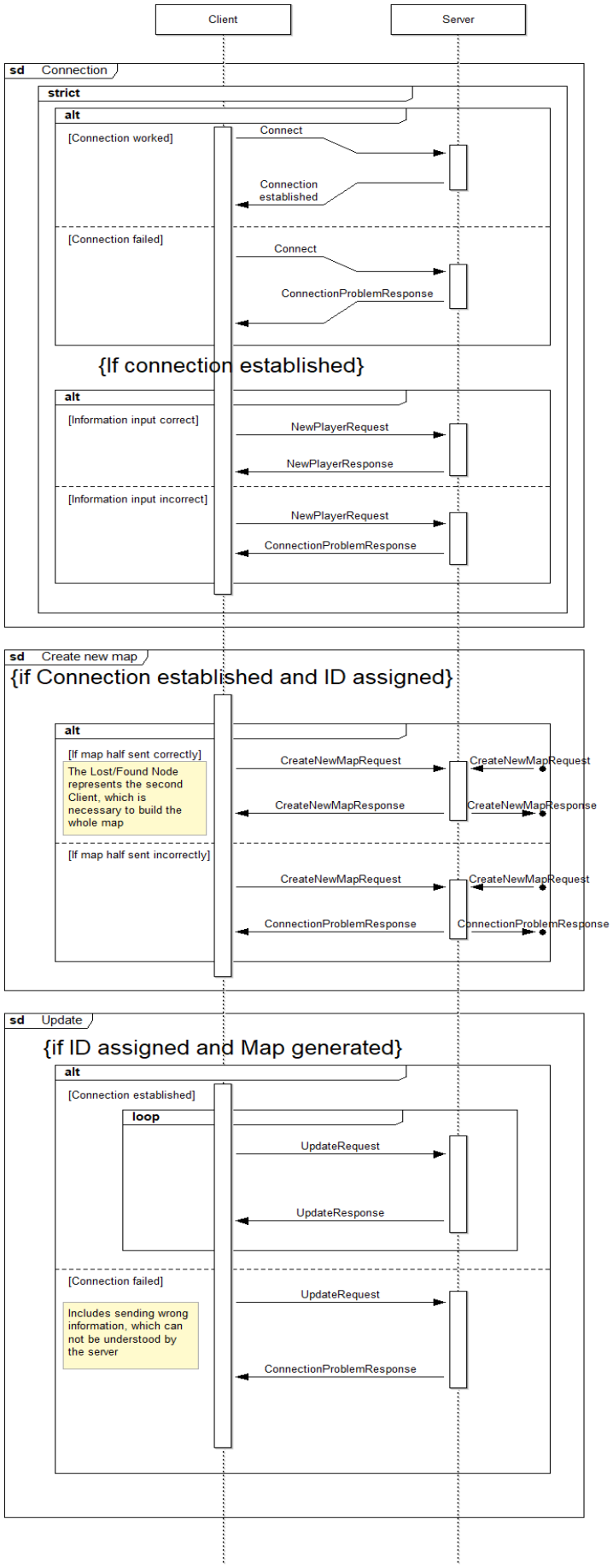
Network Communication Sequence Diagram

In this example, only *one client* and the *server* are drawn. This is because the main objective of this diagram is to show how the client and server interact with each-other, while the second client is not as relevant here, except for the *map creation*, which will be explained in more detail later.

The first task of the network is to *create a connection* between client and server. This is done in a strict order, hence the “strict” combined fragment at the beginning, which ensures it does not happen in any other order. Firstly, the client tries to connect to the server. At this point, two outcomes can happen, where the connection would either work or fail. If the connection works, then the connection is established. Otherwise, the ConnectionProblemResponsse kicks in. If the connection was established, and the information was input correctly in the NewPlayerRequest, then the server answers with a NewPlayerResponse. If the connection failed or the information sent is incorrect, then the server sends the ConnectionProblemResponse again.

The second task of the network will be to *send the map information* at the beginning of the game, in order to create the playing field. For this, there are two different alternatives: If the map half is sent correctly by both clients, then the server can send the full map in a CreateNewMapResponse. The diagram uses a Lost/Found Node as a representation of the second client, because the second client is irrelevant for this overall diagram and does not require its own Lifeline. If an error happens and the map half is sent incorrectly, then a ConnectionProblemResponse is sent to both players, but only one of the responses will contain a “loss=true” value, hence only the player who sent the information incorrectly or had the connection problem will lose the game.

The last task will be *updating the game*. This task only occurs if the client got an ID assigned and a map has been generated. If both of these worked, then there are two different alternative outcomes: If the connection failed, or the wrong information has been sent to the server, then the server answers with a ConnectionProblemResponse – but if everything works correctly, then we get into a loop where UpdateRequests and UpdateResponse messages are sent between client and server until a winner and loser has been decided to the game. Once the update phase has ended, so has the game and as such, both Lifelines of server and client end as well.


Client
Server
sd Connection
strict
alt
[Connection worked]
Connect
Connection established
[Connection failed]
Connect
ConnectionProblemResponse
{If connection established}
alt
[Information input correct]
NewPlayerRequest
NewPlayerResponse
[Information input incorrect]
NewPlayerRequest
ConnectionProblemResponse
sd Create new map
{if Connection established and ID assigned}
alt
[If map half sent correctly]
The Lost/Found Node represents the second Client, which is necessary to build the whole map
CreateNewMapRequest
CreateNewMapRequest
CreateNewMapResponse
CreateNewMapResponse
[If map half sent incorrectly]
CreateNewMapRequest
CreateNewMapRequest
ConnectionProblemResponse
ConnectionProblemResponse
sd Update
{if ID assigned and Map generated}
alt
[Connection established]
loop
UpdateRequest
UpdateResponse
[Connection failed]
Includes sending wrong information, which can not be understood by the server
UpdateRequest
ConnectionProblemResponse

Definition of Messages

Message 1: Registration of Player (NewPlayerRequest)

In this message, the human player establishes a connection between the Client and the Server. Here, the first name and the surname (family name), along with the matriculation number, are sent to the server. All of the sent messages are of type „String“. The server makes sure that the data is correct and responds with Message 2: ID Assignment.

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="NewPlayerRequest">
		<xs:complexType>
			<xs:all>
				<xs:element name="FirstName" type="xs:string" />
				<xs:element name="SurName" type="xs:string" />
				<xs:element name="MatriculationNumber" type="xs:string" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<NewPlayerRequest>
	<FirstName>Razvan</FirstName>
	<SurName>Nancu</SurName>
	<MatriculationNumber>01502339</MatriculationNumber>
</NewPlayerRequest>
```

http://<domain>:<port>/NewPlayer

Message 2: ID Assignment (NewPlayerResponse)

NewPlayerResponse assigns an ID to a player and sends it to the client. This way, the server will always know which client is currently sending a message and there will be no way to confuse the players. The ID is an integer and will mostly consist of one singular number.

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="NewPlayerResponse">
		<xs:complexType>
			<xs:all>
				<xs:element name="ID" type="xs:int" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<NewPlayerResponse>
	<ID>1</ID>
</NewPlayerResponse>
```

Message 3: Sending the Map Half (CreateNewMapRequest)

In this message, the client sends the ID which it got from the server in Message 2, and the map half which it generated. The map half will be a string and contains all important information. As long as the map was generated correctly, the client now awaits Message 4: Receiving the Map.

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="CreateNewMapRequest">
		<xs:complexType>
			<xs:all>
				<xs:element name="ID" type="xs:int" />
				<xs:element name="MapHalf" type="xs:string" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<CreateNewMapRequest>
        <ID>1</ID>
    <MapHalf>0101020301040201010201030201020301020101020203030102010201010101</MapHalf>
</CreateNewMapRequest>
```

http://<domain>:<port>/MapCreation

Message 4: Receiving the Map (CreateNewMapResponse)

The server now sends the map to the clients, which has been put together from the two map halves sent by the two competitors. It is sent as a String, which will then be processed by the client and read into a whole map. This is a better option than creating 64 objects for each single tile of the map.

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="CreateNewMapResponse">
                <xs:complexType>
                        <xs:all>
                                <xs:element name="FullMap" type="xs:string" />
                        </xs:all>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<CreateNewMapResponse>
      <FullMap>0101020301040201010201030201020301020101020203030102010201010101010203010402010102010302010203010201010202030301020102010101010</FullMap>
</CreateNewMapResponse>
```

Message 5: Updating – From Client to Server (UpdateRequest)

This message represents the core gameplay. Here, the client sends firstly the ID, so the server knows which player is currently working, then the current position, made up of two integers, X and Y (Two different objects) and the position that the AI wants to go to (also X and Y, two integer objects), followed by the amount of rounds until the movement happens (an integer, which when it reaches 0, then the server will know that the player will move now).

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="UpdateRequest">
		<xs:complexType>
			<xs:all>
				<xs:element name="ID" type="xs:int" />
				<xs:element name="PlayerPositionX" type="xs:int" />
				<xs:element name="PlayerPositionY" type="xs:int" />
				<xs:element name="WantedPositionX" type="xs:int" />
				<xs:element name="WantedPositionY" type="xs:int" />
				<xs:element name="TurnsUntilMovement" type="xs:int" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<UpdateRequest>
	<ID>1</ID>
	<PlayerPositionX>6</PlayerPositionX>
	<PlayerPositionY>1</PlayerPositionY>
	<WantedPositionX>6</WantedPositionX>
	<WantedPositionY>2</WantedPositionY>
	<TurnsUntilMovement>4</TurnsUntilMovement>
</UpdateRequest>
```

http://<domain>:<port>/Updates

Message 6: Updating – From Server to Client (UpdateResponse)

The server responds to every Update Request with, firstly, the current position of the player, which is two integer objects for the coordinate X and Y, secondly, the position of the enemy, again X and Y, then if the treasure has been found, if the player has lost, or if the player has won. These last 3 variables are booleans, which are either true or false. If the treasure has been found by that player, the server always sends true. If the player has won or lost, the server sends true to that value. Of course, if both players lose, they both would get "loss=true", which can happen if they go over 200 game rounds. Otherwise, the game will most likely always have one winner and one loser.

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="UpdateResponse">
		<xs:complexType>
			<xs:all>
				<xs:element name="PlayerPositionX" type="xs:int" />
				<xs:element name="PlayerPositionY" type="xs:int" />
				<xs:element name="EnemyPositionX" type="xs:int" />
				<xs:element name="EnemyPositionY" type="xs:int" />
				<xs:element name="Treasure" type="xs:boolean" />
				<xs:element name="Win" type="xs:boolean" />
				<xs:element name="Loss" type="xs:boolean" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<UpdateResponse>
	<PlayerPositionX>6</PlayerPositionX>
	<PlayerPositionY>1</PlayerPositionY>
	<EnemyPositionX>2</EnemyPositionX>
	<EnemyPositionY>8</EnemyPositionY>
	<Treasure>false</Treasure>
	<Win>false</Win>
	<Loss>false</Loss>
</UpdateResponse>
```

Message 7: Errors (ConnectionProblemResponse)

This message tells the player that they have encountered a problem, either connection or a different problem, such as wrong information sent, or map falsely generated, or AI took too long to think about its next move. This response will always send one object, which is a boolean that indicates that the player has lost. (loss=true)

Example Schema:

```
<?xml version="1.0" encoding="UTF-0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="ConnectionProblemResponse">
		<xs:complexType>
			<xs:all>
				<xs:element name="Loss" type="xs:boolean" />
			</xs:all>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConnectionProblemResponse>
	<Loss>true</Loss>
</ConnectionProblemResponse>
```

Example of Game Process

For this example, I have used the system under following assumptions: No player had any connection problems and all data inputs were correct. I decided that in the third round, the first player stepped in water and as such immediately lost the game. This way, I can show how the end-game takes place. In here, in comparison to the sequence diagram above, both clients are shown. For simplicity sake, the fact that each message takes an amount of time to be sent will not be drawn in this graph, but only mentioned here. As such, the graph only shows which messages generally come first and are sent last, which is more than enough to demonstrate the game flow.

Firstly, both clients send a NewPlayerRequest with their first name, their surname and their matriculation number. In both cases, they get a NewPlayerResponse back from the server with their new IDs, which they will use in later messages.

Secondly, a new map is generated, firstly by both clients calling CreateNewMapRequest and sending their ID and their half of the map, shown as a String which is then decrypted by the server. In the example, I did not input a whole 64 character long string, but rather the first four tiles (which are 8 characters long). The server answers with a CreateNewMapResponse and the full map, which consists of a 128 character long String.

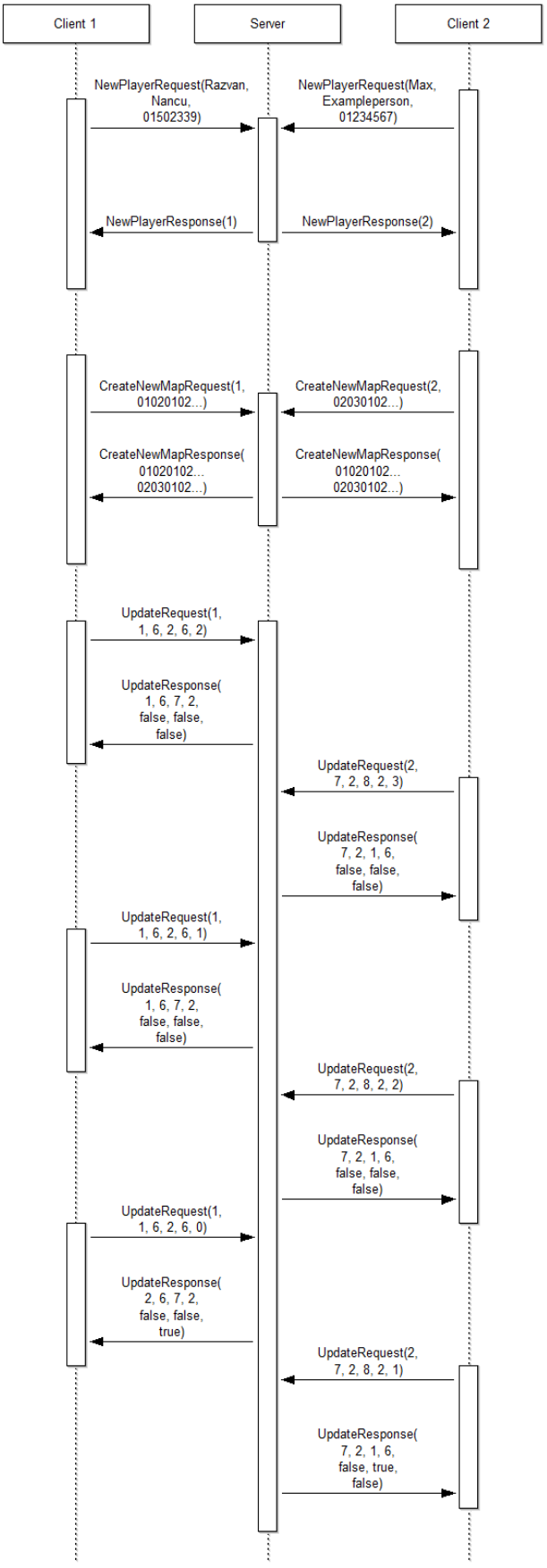
Now the game is ready to go:

First player sends an UpdateRequest with the following objects in the message: An ID, which is 1, the current position which is an X and a Y integer each, then the position that the player wants to go to, which is again an X and a Y integer, and how many turns are left until it is time to move, which in this example is “2 rounds left”. When the counter reaches “0 rounds left” the server will take the wanted position of the player and move him there.

The UpdateResponse has the following message: Firstly the current position of the player, which is an X and Y integer, then the position of the enemy, which is also an X and Y integer, and then three booleans: treasure found, won game and lost game, which are all set to false, since the game just started. The starting coordinates were set when creating the map, since the spawning point is at the castle.

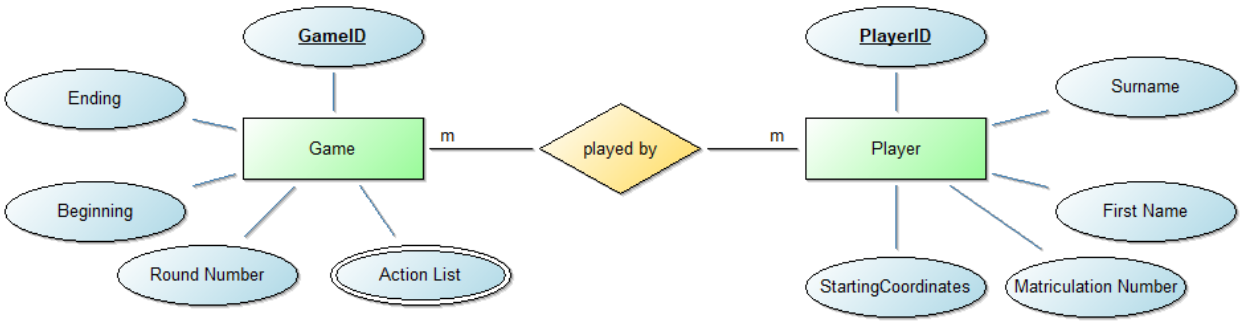
The next player’s UpdateResponse follows the same schema. Afterwards, the first player’s counter in UpdateResponse has went down by 1, so there is only one turn left until the player moves to the new position. Same happens with the other player.

Lastly, the game ends, as the UpdateRequest counter to move to tile “2/6” has reached 0 and 2/6 is a water tile (as told at the beginning of this subchapter). As such, the UpdateResponse from the server updates the coordinates of the player and sets the last boolean, which is the “lost game” boolean, to true. That player now lost the game. The other player makes an UpdateRequest which is irrelevant to the server, as the server automatically sends an UpdateResponse with the “won game” boolean set to true. This way, if the counter of the second player would have reached 0, in a hypothetical scenario, and that player would have also stepped on a water tile, the server would have ignored that move, since the first player already lost the game.


Client 1
Server
Client 2
NewPlayerRequest(Razvan, Nancu, 01502339)
NewPlayerRequest(Max, Exampleperson, 01234567)
NewPlayerResponse(1)
NewPlayerResponse(2)
CreateNewMapRequest(1, 01020102...)
CreateNewMapRequest(2, 02030102...)
CreateNewMapResponse(01020102... 02030102...)
CreateNewMapResponse(01020102... 02030102...)
UpdateRequest(1, 1, 6, 2, 6, 2)
UpdateResponse(1, 6, 7, 2, false, false, false)
UpdateRequest(2, 7, 2, 8, 2, 3)
UpdateResponse(7, 2, 1, 6, false, false, false)
UpdateRequest(1, 1, 6, 2, 6, 1)
UpdateResponse(1, 6, 7, 2, false, false, false)
UpdateRequest(2, 7, 2, 8, 2, 2)
UpdateResponse(7, 2, 1, 6, false, false, false)
UpdateRequest(1, 1, 6, 2, 6, 0)
UpdateResponse(2, 6, 7, 2, false, false, true)
UpdateRequest(2, 7, 2, 8, 2, 1)
UpdateResponse(7, 2, 1, 6, false, true, false)

Persistence

The data bank will make sure to save various information from the games which take place on the server.



Every game will have an unique Primary Key, Game ID, an integer, which will clearly specify what game was played. The game will also have a Beginning and an Ending Time, both being times in the data bank, and how many rounds the game was (Round Number), an integer. The Round Number should be set at the end of the game and shows how many rounds passed until the game ended. The action list is a multi-value attribute and it saves every move that happened during the match.

For the Action List, a new table is created, having a GameID, an ActionID and an Action. The ActionID represents the round number, basically, and the action represents what move the player did in that round. There are 5 options for the Action: Up, Down, Left, Right or Stay. Stay means that the player did not move that round, most likely because it waited for the time to move. The Game ID is a foreign key, which is the primary key of the Game table. The primary key for the Action List will consist of both, Game ID and Action ID – as such, every move is identifiable and unique. I did not add which player did which move, because it is easy to deduct from the fact that a player will always have either an odd or an even number of rounds, which can be selected from the Action ID column.

The Player will have an unique Primary Key, called Player ID, an integer, then a Surname and First Name, which are Strings (or char(30), as I have chosen them for the table), a matriculation number, which is also a string of 8 chars, since that is how long a matriculation number at the university of Vienna can get (currently, and most likely for the next couple hundreds of years, due to the newly added zero at the beginning of the string). The player also saves the starting coordinates – this way, we can always track a full gameplay due to having the coordinates of both players at the beginning of the game and can just track the moves through the “Action List” table.

I have chosen an m:m relationship because a player can play multiple games and a game is played by two players. I assume that all games and each participant would be saved in one big data bank and not many data banks would be created for each single game. Since it’s

multiple to multiple relationship, I had to add a “Play” table, which consists of three values: The Game ID, the First Player ID and the Second Player ID, all of which are integers and unique. This way, we can always know which game was played by which players.

The following are the **CREATE TABLE** instructions for SQL:

```
CREATE TABLE Game(
        GameID integer NOT NULL,
        Beginning time,
        Ending time,
        RoundNumber integer,
        Map varchar(128);
        PRIMARY KEY(GameID),
        CHECK(RoundNumber>=0),
        CHECK(RoundNumber<=200)
)

CREATE TABLE Player(
        PlayerID integer NOT NULL,
        Surname char(30),
        FirstName char(30),
        MatriculationNumber char(9),
        StartingCoordinates char(3),
        PRIMARY KEY(PlayerID)
)

CREATE TABLE ActionList(
        GameID integer NOT NULL,
        ActionID integer NOT NULL,
        Action char(5), /*UP, DOWN, LEFT, RIGHT, STAY*/
        PRIMARY KEY(GameID, ActionID),
        FOREIGN KEY(GameID) REFERENCES Game ON DELETE CASCADE ON UPDATE
CASCADE
)

CREATE TABLE Play(
        GameID integer NOT NULL,
        PlayerID1 integer NOT NULL,
        PlayerID2 integer NOT NULL,
        PRIMARY KEY(GameID, PlayerID1, PlayerID2),
        FOREIGN KEY(GameID) REFERENCES Game,
        FOREIGN KEY(PlayerID1) REFERENCES Player,
        FOREIGN KEY(PlayerID2) REFERENCES Player,
        CHECK(PlayerID1!=PlayerID2)
)
```

The following are the **inserted values**, 10 values for each table:

Values for the “Game” table:

```
INSERT INTO Game
VALUES(1, 13:00:00, 14:00:00, 120)
INSERT INTO Game
VALUES(2, 14:05:00, 14:30:00, 50)
INSERT INTO Game
VALUES(3, 15:00:00, 16:30:00, 200)
INSERT INTO Game
VALUES(4, 13:00:00, 13:10:00, 20)
INSERT INTO Game
VALUES(5, 13:40:00, 15:00:00, 198)
INSERT INTO Game
VALUES(6, 15:12:00, 15:58:00, 142)
INSERT INTO Game
VALUES(7, 16:00:00, 16:23:00, 60)
INSERT INTO Game
VALUES(8, 13:00:00, 14:01:00, 121)
INSERT INTO Game
VALUES(9, 14:05:00, 14:50:00, 138)
INSERT INTO Game
VALUES(10, 15:00:00, 16:00:00, 170)
```

Values for the “Player” table:

```
INSERT INTO Player
VALUES(1, 'Razvan', 'Nancu', '01502339', '1/6')
INSERT INTO Player
VALUES(2, 'Bumi', 'Earth', '01502457', '7/2')
INSERT INTO Player
VALUES(3, 'Heph', 'Hades', '01525639', '5/8')
INSERT INTO Player
VALUES(4, 'Razvan', 'Theimmortal', '11502339', '1/6')
INSERT INTO Player
VALUES(5, 'Jimmy', 'Thedude', '01202637', '8/8')
INSERT INTO Player
VALUES(6, 'Iroh', 'General', '01509784', '6/6')
INSERT INTO Player
VALUES(7, 'The', 'Persson', '01523439', '3/1')
INSERT INTO Player
VALUES(8, 'Ima', 'Typin', '01123339', '8/1')
INSERT INTO Player
VALUES(9, 'Bored', 'Persson', '01502339', '5/5')
INSERT INTO Player
VALUES(10, 'End', 'Ofthelist', '01502339', '2/6')
```

Values for the “ActionList” table:

```
INSERT INTO ActionList
VALUES(1, 1, 'STAY')
INSERT INTO ActionList
VALUES(1, 2, 'STAY')
INSERT INTO ActionList
VALUES(1, 3, 'UP')
INSERT INTO ActionList
VALUES(2, 1, 'STAY')
INSERT INTO ActionList
VALUES(2, 10, 'RIGHT')
INSERT INTO ActionList
VALUES(2, 100, 'LEFT')
INSERT INTO ActionList
VALUES(3, 5, 'DOWN')
INSERT INTO ActionList
VALUES(8, 1, 'STAY')
INSERT INTO ActionList
VALUES(10, 32, 'UP')
INSERT INTO ActionList
VALUES(4, 20, 'STAY')
```

Values for the “Play” table:

```
INSERT INTO Play
VALUES(1,1,2)
INSERT INTO Play
VALUES(2,1,5)
INSERT INTO Play
VALUES(3,3,8)
INSERT INTO Play
VALUES(4,1,4)
INSERT INTO Play
VALUES(5,10,6)
INSERT INTO Play
VALUES(6,2,10)
INSERT INTO Play
VALUES(7,5,7)
INSERT INTO Play
VALUES(8,7,9)
INSERT INTO Play
VALUES(9,3,6)
INSERT INTO Play
VALUES(10,1,10)
```

Database Implementation

The Database has been created and coded, but not implemented into the Controller. Time constraints do not allow for all of the methods to be debugged in time to ensure a flawless execution of the code. Still, I will briefly describe how it functions and what methods were made.

In the Game table, there are five values saved, the ID of the game, the time when it started and ended, thee number of rounds the game had and the map. The Player table had an ID, a first and a surname, a matriculation number and starting coordinates. The Play table had a game and two players and the Action List table had an ID, an action and the game ID to which the action belonged.

HibernateMain.java would prepare some functions with which different statements could be executed into the database, such as setting a new player (and an ID), or setting the map for a new game, or the amount of rounds for a game.

Here is a snippet of the setNewPlayer function:

```
public void setNewPlayer(String firstName, String surName, String matriculationNumber, int playerID) {
                Session session = HibernateUtil.getSessionFactory().openSession();
                session.beginTransaction();
                Player player = new Player();
                player.setFirstName(firstName);
                player.setSurName(surName);
                player.setMatriculationNumber(matriculationNumber);
                player.setPlayerID(playerID);
                session.saveOrUpdate(player);
                session.getTransaction().commit();
        }
```

Business Rules

There are 10 functions implementing, checking various rules.

The first rule is the round counter, which checks that the game has not gone on for over 200 rounds. If it has, then both players lost.

```
public boolean roundCount(int rounds) {

        if(rounds>=200) return false;

        return true;

}
```

One of the bigger functions implemented, spanning multiple business rules, is the checkMapGeneration() function. It consists of the following parts:
It checks if the map length was correct, so not bigger than 4 and 8

```
if(mapHalf.length != 4 || mapHalf[0].length != 8) return false;
```

Afterwards, it checks how many grass, water and mountain fields there are, and that there are not too many.

```
int GCount=0; int MCount=0; int WCount=0;
for(int i = 0; i<4; i++) {

        for(int j = 0; j<8; j++) {

                if(mapHalf[i][j].contentEquals("0G")) { GCount++;}

                else if(mapHalf[i][j].contentEquals("0M")) { MCount++;}

                else if(mapHalf[i][j].contentEquals("0W")) { WCount++;}

        }

}

//Check if there are enough of each type of Field

if(GCount<=5) return false;

if(MCount<=3) return false;

if(WCount<=4) return false;
```

The next part of the function makes sure that there are not too many water fields on each side. The following code snippet demonstrates how the function works on the upper side, where not more than 3 water fields are allowed:

```
int WCountSide=0;

for(int i = 0; i<8; i++) {

        if(mapHalf[0][i].contentEquals("0W")) WCountSide++;

        if(WCountSide>3) return false;

}
```

The last part of the map generation check process tries to see if there are any islands or not on the generated map. The method of checking is quite long in my implementation and it warranted a function of its own. The way it works is basically: a clone of the map half is generated and we change the first grass, mountain or castle field that can be found into a different value, such as “00”. Then, every field that is a mountain or grass that is next to a “00” field is changed also in “00”. This way, we change all grass and mountain fields into “00”s, leaving only water. We do this process twice, incrementing and decrementing two times. This way, there can be no fields left unchecked even if they are next to a “00” field.
In the end, the program checks if there are any “0G” or “0M” fields left – and if there are, then there were islands generated, since the only fields not transformed are those that weren’t next to a “00” field. We can deduct that any field that wasn’t next to a “00” field was separated by water from the rest of the map.

Additionally, I thought about implementing a method that beholds all if-statements used, since I had to copy and paste them once – this way we could save a bit of space. Time constraints dictated this to be left out for this version of the program.

Following snippets show how the method works.
Firstly, we change the first field into “00”. The field may not be water:

```
if((cloneHalf[i][j].contentEquals("0C") || cloneHalf[i][j].contentEquals("PC") || cloneHalf[i]
[j].contentEquals("0G") || cloneHalf[i][j].contentEquals("0M"))) {cloneHalf[i][j] = "00";}

        //If [0][0] wasn't viable, go through map until viable one found

        else {  int i2=0,j2=0;

                for(boolean until00 = false; until00!=true;) {

                        if(cloneHalf[i2][j2].contentEquals("0C") || cloneHalf[i2][j2].contentEquals("PC") ||
cloneHalf[i2][j2].contentEquals("0G") || cloneHalf[i2][j2].contentEquals("0M")|| cloneHalf[i2]
[j2].contentEquals("00")) {

                                cloneHalf[i2][j2] = "00";

                                until00=true;

                        }

                        else {i2++;}

                        if(i2==3) {i2=0;j2++;}}}
```

The following snippet shows one if-statement that changes the field if there is another “00” nearby:

```
if(i==0 && j>0 && j<7) {

if((cloneHalf[i][j].contentEquals("0C") || cloneHalf[i][j].contentEquals("PC") || cloneHalf[i]
[j].contentEquals("0G") || cloneHalf[i][j].contentEquals("0M")) &&

(cloneHalf[i][j-1].contentEquals("00")||cloneHalf[i+1][j].contentEquals("00")||cloneHalf[i]
[j+1].contentEquals("00"))) {cloneHalf[i][j] = "00";}}
```

If there were any fields left out (that did not become “00”), return true as in “Islands? True”

```
for(int i=0; i<4; i++) {

        for(int j=0; j<8; j++) {

                if(cloneHalf[i][j].contentEquals("0C") || cloneHalf[i][j].contentEquals("PC") || cloneHalf[i]
[j].contentEquals("0G") || cloneHalf[i][j].contentEquals("0M")) return true;}}
```

The next business rule, implemented as checkSpawnOfTreasureAndCastle() checks whether both, castle and treasure, have been generated for both players. If one weren’t generated, then it could either mean generating it failed, or one of the generated features overshadowed the other for some peculiar reason (Which would not happen usually, during map configuration, since it has been set the way, such that the treasure is generated only on grass fields.

```
boolean TreasureExists = false, CastleExists = false;

for(int i = 0; i < 4; i++) {for(int j = 0; j<8; j++) {

                    if(map[i][j].contentEquals("0C")||map[i][j].contentEquals("PC")) CastleExists=true;

                    if(map[i][j].contentEquals("0T")) TreasureExists=true;}}

if(CastleExists==false||TreasureExists==false) return false;
```

The next business rule checks, whether the player moves only one field distance. If not, then "false" is returned, as such losing the game.

```
public boolean checkFieldMovement(int posX, int posY, int wantedX, int wantedY) {

if((posX-1==wantedX||posX+1==wantedX||posX==wantedX)&&(posY-1==wantedY||posY+1==wantedY||
posY==wantedY)) return true;

return false;}
```

Following, this business rule checks that the amount of rounds until the next movement is not set too short or too long. The following snippet shows what happens if the player is on a grass field. The same happens if the player is on a mountain and wants to move somewhere else. We basically check what type of field the player wants to move on and then check whether the amount of turns left sent by the client was correct or not. This function is only calculated if the player has changed its wanted position and wants to move somewhere else.

```
if(map[posX][posY].contentEquals("PG")) {

if(map[wantedX][wantedY].contentEquals("0G") || map[wantedX][wantedY].contentEquals("0W"))
{if(turnsLeft==2) return true;

        else return false;}

        if(map[wantedX][wantedY].contentEquals("0M")) {

        if(turnsLeft==3) return true;

        else return false;}}
```

Next method makes sure that the player is not on water or outside the map.

```
if(positionX < 0 || positionX > map.length || positionY < 0 || positionY > map[0].length) return false;

                    if(map[positionX][positionY].contentEquals("PW") || map[positionX]
[positionY].contentEquals("0W")) return false;
```

if either of those happens, then the player loses the game.

The next method checks that the player has touched his treasure tile. If he did, then the „T" is changed to „G", since the treasure does not exist there anymore.

```
if(posX==treasureX && posY==treasureY) {
                    map[posX][posY].replace("T", "G");
                    return true;
                }
```

Another method checks if the winning conditions have been met, firstly by checking if the treasure was already taken and then if the player is now on the enemy castle position.

```
if(treasureStatus==true && posX==enemyCX && posY==enemyCY) {return true;}
```

The last business rule implemented checks, that if the player is on a mountain, and there is treasure in any of the surrounding fields, then the treasure is shown.

```
if(map[positionX][positionY].contentEquals("PM") && (map[positionX-1][positionY-1].contentEquals("0T") ||
map[positionX-1][positionY].contentEquals("0T") ||map[positionX-1][positionY+1].contentEquals("0T") ||
map[positionX][positionY-1].contentEquals("0T") || map[positionX][positionY+1].contentEquals("0T") ||
map[positionX+1][positionY-1].contentEquals("0T") || map[positionX+1][positionY].contentEquals("0T") ||
map[positionX+1][positionY+1].contentEquals("0T"))) return true;
```

What basically happens in this method is, that every field around the player is checked and if any of those has a treasure, then „true“ is sent.

Network Communication

The whole network communication is being processed by a controller, in the Controller.java class. There are 4 functions to it, registering a new player, putting map halves together, updating the map and game and handling any problems.

Since I used REST, I did not need to do any marshalling/ unmarshalling of the sent information. Any commented parts in this part of the documentations are things that have not been fully tested and as such not implemented due to time constraints.

First method implemented is the New Player Response.

```
NewPlayerResponse banana = new NewPlayerResponse();
        int playerID = s.getHighestID()+1;
        s.setHighestID(playerID);
        if(playerNumber == 2) {
                s.setIdPlayer2(playerID);
                idP2=playerID;
                playerNumber++;
                //h.setNewGame(1);
                //h.setPlay(1, s.getIdPlayer1(), s.getIdPlayer2());
```

```
        } //AT END OF MATCH SET PLAYERNUM TO 0

        else if(playerNumber == 1) {

                s.setIdPlayer1(playerID);

                idP1=playerID;

                playerNumber++;

        }

        banana.setID(playerID);

//h.setNewPlayer(newPlayerReq.getFirstName(), newPlayerReq.getSecondName(),
Integer.toString(newPlayerReq.getMatrikelnummer()), playerID);

        return banana;
```

It is relatively straight-forward. We get the highest ID yet and add 1 to it. Then we make it the player ID. Added a playerNumber value outside the method in order to control to which player the values are assigned to. Since there are only 2 players, playerNumber will only go up to 2 and then reset to 0 at the end of the match.

Creating a new map was a bit more tricky. What I basically do is send the first player that sends information to sleep and wait until the 2nd player sends their map half. Then, after the server puts the halves together, wakes up the thread and send the full map back to both players.

```
while(waitingP1!=false) {

        try {

                Thread.sleep(1000);

        } catch (InterruptedException e) {

                e.printStackTrace();

        }banana.setFullMap(s.arrayToString(s.getMap()));
```

In the Update function, the UpdateResponse, we set the round counter plus 1 and then we update the player position with the following function, which is executed inside the server:

```
public void updatePlayerPosition(int posX, int posY, int oldX, int oldY){

        map[oldX][oldY] = "0" + map[oldX][oldY].substring(1);

        map[posX][posY] = "P" + map[posX][posY].substring(1);

}
```

The next part of the updating process checks the rules, making sure that none were broken. This part is also a function in the server, which handles all these requests in the program.

```
public boolean checkRules(int id, BusinessRules bs, Server s, int posX, int posY, int wantedX, int wantedY, int turnsLeft) {

	if(bs.roundCount(getRoundCounter()) == false) return false;

	if(bs.checkPlayerNotOnWaterOrOutsideMap(posX, posY, s.getMap()) == false) return false;

	if(bs.checkFieldMovement(posX, posY, wantedX, wantedY) == false) return false;



	if(turnsLeft != 0) {

		if(id==getIdPlayer1() && (posX != getPositionXPlayer1() || posY != getPositionYPlayer1() )) {

			if(bs.checkRoundsUntilMove(posX, posY, wantedX, wantedY, turnsLeft, s.getMap()) == false) return false;

		}

		if(id==getIdPlayer2() && (posX != getPositionXPlayer2() || posY != getPositionYPlayer2())) {

			if(bs.checkRoundsUntilMove(posX, posY, wantedX, wantedY, turnsLeft, s.getMap()) == false) return false;

		}

	}

	if(turnsLeft == 0) {

		if(id==getIdPlayer1()) setPositionPlayer1(posX, posY);

		if(id==getIdPlayer2()) setPositionPlayer2(posX, posY);

	}

	return true;

}
```

If the player did something bad, then they lost and set their loss to “true”. The next parts, which happen if the player didn’t lose, is updating their and the enemy position, then checking if they are on a mountain or if they touched the treasure, and checking if they won. Once all these were done, the update is sent back.

The problem response only happens if the CreateMapResponse sends a response back to the player where the map is “null” - to which the client asks why it is null and the server responds with who lost and who won (as such answering why it was null: because someone failed to pass the map configuration).

There is also another function, NewMapForTests, which is a method where one of the sent map halves is hard-coded into the function and no second thread is created. This was done because the unit tests required to be started twice, otherwise sending a null-pointer-exception. In order to circumvent this problem, I just hard-coded one of the two responses. The original method, NewMap, works perfectly well when two clients send their map halves.

Logging Part 2

I have implemented logging in various parts of the program, each saying whether every part of the program ran flawlessly or not. I have left out any hibernate or spring messages unless they were either fatal or error messages, since those just clutter up the log file.
Setting up map:

```
2017-12-08 18:28:32,491[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Second Thread for new map active.

2017-12-08 18:28:32,497[hh:mm:ss:SS] [main] INFO  server.Server - Starting to prepare map!

2017-12-08 18:28:32,497[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Length was correct!

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Grass Fields: 21 Mountain Fields: 5 Water Fields: 5

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Water Fields Upper: 0

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Water Fields Down: 2

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Water Fields Left: 0

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Water Fields Right: 0

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Checking for islands:

2017-12-08 18:28:32,498[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - First tile was not converted to 00

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - Position 0/0 has been converted to 00 successfully.

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses - 00 00 00 00 00 00 00 00

 00 00 0W 00 00 00 00 00

 00 0W 00 00 0W 00 00 00
```

```
 00 0W 00 0W 00 00 00 00


2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Island check done.

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Length was correct!

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Grass Fields: 21 Mountain Fields: 5 Water Fields: 5

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Water Fields Upper: 0

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Water Fields Down: 2

2017-12-08 18:28:32,499[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Water Fields Left: 0

2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Water Fields Right: 0

2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Checking for islands:

2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
First tile was not converted to 00

2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Position 0/0 has been converted to 00 successfully.

2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
00 00 00 00 00 00 00 00

 00 00 0W 00 00 00 00 00

 00 0W 00 00 0W 00 00 00

 00 0W 00 0W 00 00 00 00


2017-12-08 18:28:32,500[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Island check done.

2017-12-08 18:28:32,502[hh:mm:ss:SS] [main] INFO  server.MapConfiguration - Map
genration Player1: true

2017-12-08 18:28:32,502[hh:mm:ss:SS] [main] INFO  server.MapConfiguration - Map
genration Player2: true

2017-12-08 18:28:32,503[hh:mm:ss:SS] [main] INFO  server.MapConfiguration - Spawn
done.
```

```
2017-12-08 18:28:32,503[hh:mm:ss:SS] [main] INFO  server.MapConfiguration - Spawn
done.

2017-12-08 18:28:32,503[hh:mm:ss:SS] [main] INFO  server.MapConfiguration -
Starting treasure spawn.

2017-12-08 18:28:32,505[hh:mm:ss:SS] [main] INFO  server.MapConfiguration -
Treasure spawn done.

2017-12-08 18:28:32,505[hh:mm:ss:SS] [main] INFO  server.MapConfiguration -
Treasure spawn done.

2017-12-08 18:28:32,505[hh:mm:ss:SS] [main] INFO  server.MapConfiguration - Map
config done.

2017-12-08 18:28:32,506[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Setting up whole map

2017-12-08 18:28:32,506[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Done setting up map
```

I put loggers telling us when the map preparation starts, if the map generation of the players was correct and when the configuration was done. Should be relatively straight-forward. After every important check, we get to know if it is done, such as the spawning, treasure spawning, island checking and so on. The program also prints out the map for the island checks, showing that the 00 has spread correctly to all parts of the map halves.

The second part which is logged is when an update happens to the player position. Here is a small part of the logged file, stating what happens during rounds 1 and 2:

```
2017-12-08 18:28:32,508[hh:mm:ss:SS] [main] INFO  server.Server - Updated Player position
successfully.

2017-12-08 18:28:32,508[hh:mm:ss:SS] [main] INFO  server.Server - Starting to check rules!

2017-12-08 18:28:32,508[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Currently at round number 1. The limit was not reached yet.

2017-12-08 18:28:32,508[hh:mm:ss:SS] [main] INFO  server.Server - Rules checked
successfully.

2017-12-08 18:28:32,509[hh:mm:ss:SS] [main] INFO  server.Server - Updated Player position
successfully.

2017-12-08 18:28:32,509[hh:mm:ss:SS] [main] INFO  server.Server - Starting to check rules!

2017-12-08 18:28:32,509[hh:mm:ss:SS] [main] INFO  server.ControllerTestResponses -
Currently at round number 2. The limit was not reached yet.

2017-12-08 18:28:32,509[hh:mm:ss:SS] [main] INFO  server.Server - Rules checked
successfully.
```

2017-12-08 18:28:32,509[hh:mm:ss:SS] [main] INFO server.Server - Updated Player position successfully.

We can see the server gets instructions from ControllerTestResponses

These are the most important logged parts, since they cover the main part of the game.

Error-Handling Part 2

I have used the classic form of error handling only in HybernateMain.java, since things could go wrong if the data base would not run correctly. Otherwise, I mostly used if statements on most parts of the program (mostly due to time constraints too).

The following snippet is the part used in the hybernate example:

```
try {
        // Create the SessionFactory from hibernate.cfg.xml
        return new Configuration().configure().buildSessionFactory();
} catch (Throwable ex) {
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
}
```

My REST implementation does not really need much in terms of verifying player input, since any wrongly sent messages are simply not accepted by the system.

The way I handled any errors about the business rules is through if statements, where if something did not go correctly, then a “false” boolean would be returned – since any mistake means a loss for the player anyway.

As an example, here is a snippet from Server checkRules(), where three different rules are checked and if one of them isn’t correctly followed, then the player loses:

```
if(bs.roundCount(getRoundCounter()) == false) return false;
if(bs.checkPlayerNotOnWaterOrOutsideMap(posX, posY, s.getMap()) == false) return false;
if(bs.checkFieldMovement(posX, posY, wantedX, wantedY) == false) return false;
```

Unit Tests Part 2

In the end, I only implemented 5 unit tests covering 84.3% of the server, excluding the client and the databank (which were not really implemented and relevant for this 2nd part of the project). Overall, 73.1% of the whole project was covered. As such, I do not believe that any further covering is really needed for the most important parts of the project.

In the following paragraphs, I will describe what classes I tested and what was not covered in said tests.

Firstly, in the ServerTest.java class, I tested the server. Most getters and setters were not particularly important to test, so some were not covered, such as setLossPlayer1(). The test itself looked through following methods: Array to String, String to Array, Check Rules, Update Player Position and Prepare Map. All of the tests worked flawlessly in this part.

Example of two such tests in more detail will be testCheckRules() and testUpdatePlayerPosition().

TestCheckRules creates a new BusinessRules instance and prepares a map. Not preparing the map would send a null pointer exception since multiple rules check the map in concordance to the player position – not having one set would mean the method wouldn't have a map to compare the results to. The s.checkRules(2, bs, s, 4, 0, 4, 1, 1); method sends an ID, a BusinessRule instance, a server, a current position X and Y, a wanted position X and Y and how many turns are left until the player can move. No errors appeared and all values were processed correctly.

TestUpdatePlayerPosition configures a map first, just as in the other method, and then takes the player and moves him to a designated place, thus updating the map.

BusinesssRulesTest.java handles all the business rules, making sure the rules wouldn't give a false result. In testCheckSpawnOfTreasureAndCastle(), for example, I implemented *assertEquals*(bs.checkSpawnOfTreasureAndCastle(map), **true**); in order to make sure that the method reads the map correctly and identifies the map, treasure and castles.

The last big part belongs to one class, but has three different tests, ControllerTest, ControllerTestResponse and ControllerTestMap.

ControllerTest just makes sure the setters are correct, then sees if the results are correct, such as
public void testGetMapHalfP1() {*assertNull*(c.getMapHalfP1());}
where the result should be null, since no map was initialized.

ControllerTestMap creates two new players and tests if the map is correctly put together. In here, instead of NewMap(), NewMapForTests() is used in the Controller class, due to otherwise having to start the test two times in order to not have a null-pointer-exception.

ControllerTestResponses is the most interesting test, since it verifies that the most important part of the program works: Getting the messages through REST and processing them. Firstly, the new players and map are initialized, after which 6 updates happen, one after the other. Each of these signifies a turn and checks if there are any null-pointer-exceptions or any other errors happening. The data put in is expected to work correctly, and the test proves it does.

VERSION 2 CHANGELOG:
-Corrected how Unit Cases were described
-Cut the last Business Rule about the database
-Updated the CREATE Statement for Game to have a Map varchar(128)
-Added Database Implementation
-Added Business Rules
-Added Network Implementation
-Added Logging for Part 2 of the project
-Added Error Handling for Part 2 of the project
-Added Unit Tests for Part 2 of the project