# VU Software Engineering 1

Task 3
(Client Implementation)


| | |
|---|---|
| **Prename, Surname:** | Yannick Koitzsch |
| **Matriculation number:** | 01301303 |
| **E-Mail:** | y.koitzsch@gmx.at |
| **Date:** | 16.01.2018 |

## Source Code (Git)

- git bundle can be found in the .zip file uploaded on moodle.
- working branch name: ‘final’

## Logging

- used framework: sl4j
- the logs are saved in a file by creating the following logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <appender name="FILE" class="ch.qos.logback.core.FileAppender">
                <file>logs.log</file>
                <append>true</append>
                <immediateFlush>true</immediateFlush>
                <encoder>
                <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
                </encoder>
        </appender>
        <root level="INFO">
                <appender-ref ref="FILE" />
        </root>
</configuration>
```

| Nr. | Description | Reasoning |
|---|---|---|
| 1 | client tries to connect to the server | in case the server expects clients to connect |
| 2 | find game with specific name | if the client doesn’t find a game, it might be due to the game name |
| 3 | client receives a message from the server | important to know if the client receives a message without problems |
| 4 | client answers the server | important to know if the client reacts properly to the server message |
| 5 | the game is over | important to know when the game is over and what happens until then or after this moment |
| | | **Not created, because...** |
| 6 | client has to sit out 1 round | this information is not as important and can be found out by inspecting the log file and realising that one player had 2 turns in a row |

## Code Snippet 1

```
public void connect(){
	try {
		socket = new Socket(host,port);
		this.reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
		this.writer = new PrintWriter(socket.getOutputStream(),true);
		logger.info("["+socket.toString()+"]" + " connect to " + host +":"+ port + "
with localport " + socket.getLocalPort());
	} catch (UnknownHostException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

## Code Snippet 2

```
public void join(String gamename){
	logger.info("["+socket.toString()+"] join new game with name [" + gamename + "]");
	sendToServer(MessageFactory.cs_join(gamename));
}
```

## Code Snippet 3

```
public void receiveMessages(){
	JAXBContext context;
	try {
		context = JAXBContext.newInstance(XMLMessage.class);
		Unmarshaller um = context.createUnmarshaller();
		XMLMessage msg;
		String xml = "";
		StringBuilder sb = new StringBuilder();
		while((xml = reader.readLine()) != null){
			sb.append(xml);
			if(xml.endsWith(XMLMessage.msgEnd)){
				msg = (XMLMessage) um.unmarshal(new StreamSource(new
StringReader(sb.toString())));
				sb.setLength(0);
				logger.info("["+socket.toString()+"]" + " received new message
of type ["+msg.getType().toString()+"]");
				check(msg);
			}
		}
	} catch (JAXBException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}

}
```

## Code Snippet 4

```
public void sendToServer(XMLMessage xml){
	logger.info("["+socket.toString()+"]" + " send message to server of type " +
xml.getType().toString());
	try {
		JAXBContext context = JAXBContext.newInstance(XMLMessage.class);
		Marshaller m = context.createMarshaller();
		m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
		m.marshal(xml, writer);
	} catch (JAXBException e) {
		e.printStackTrace();
	}
}
```

## Code Snippet 5

```
if(m.getType() == MsgType.OVER) {
	logger.info("["+socket.toString()+"]" + " game is over");
	gui.showWinner(Integer.valueOf(m.getDesc()));
	if(m.getDesc().equals("1")) {
		logger.info("["+socket.toString()+"]" + " winner is player 1");
	}
	if(m.getDesc().equals("2")) {
		logger.info("["+socket.toString()+"]" + " winner is player 2");
	}
	if(m.getDesc().equals("0")) {
		logger.info("["+socket.toString()+"]" + " no winner");
	}
}
```

## Example log file

```
132  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] connect to localhost:4444 with
localport 65468
134  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] join new game with name [test]
134  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] send message to server of type
JOIN
135  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] connect to localhost:4444 with
localport 65469
135  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] join new game with name [test]
135  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] send message to server of type
JOIN
145  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received new message of type
[INFO]
```

```
145  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] set playernumber to 1
147  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received new message of type
[INFO]
147  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] set playernumber to 2
149  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received new message of type
[RDY]
150  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] send message to server of type
MAP
153  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] generate the map for the first
time
160  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received new message of type
[RDY]
160  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] send message to server of type
MAP
162  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] generate the map for the first
time
374  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received new message of type
[MAP]
374  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] create a list of possible
chest locations
397  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received complete map from
server
397  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] calculate search path
397  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] get the nearest coordinate
which has not been visited yet
397  [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] send message to server of type
MOVE
910  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received new message of type
[MAP]
910  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] create a list of possible
chest locations
923  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received complete map from
server
923  [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] calculate search path
```

```
9029 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] send message to server of type
MOVE
9536 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received new message of type
[MOVE]
9536 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received opponent coordinates
[4/3]
9536 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] calculate search path
9536 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] send message to server of type
MOVE
10042 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received new message of type
[MOVE]
10042 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received opponent coordinates
[1/2]
10042 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] calculate the path to the
enemy castle
10042 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] send message to server of type
MOVE
11501 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] received new message of type
[OVER]
11501 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] game is over
11504 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] received new message of type
[OVER]
11504 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] game is over
14718 [Thread-1] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65468]] winner is player 2
14725 [Thread-2] INFO  socket.Client -
[Socket[addr=localhost/127.0.0.1,port=4444,localport=65469]] winner is player 2
```

## Network communication

### Description

I used a standard tcp socket implementation for the network communication. Server has a socket and the client can connect to that socket. Each client has a localport which let's the server distinguish between them. The multithreaded server can handle multiple clients at the same time (not limited to 2 clients at the same time).

### Implementation

| Nr. | Message | Description |
|---|---|---|
| 1 | join a game | message tells the server that the client is ready to join a game. A specific gamename can be passed to arrange specific matches. |
| 2 | client map | sends the ki generated map to the server |
| 3 | new move | sends the target coordinate of the ki |

### Code Snippet (receiving messages)

```
public void receiveMessages(){
	JAXBContext context;
	try {
		context = JAXBContext.newInstance(XMLMessage.class);
		Unmarshaller um = context.createUnmarshaller();
		XMLMessage msg;
		String xml = "";
		StringBuilder sb = new StringBuilder();
		while((xml = reader.readLine()) != null){
			sb.append(xml);
			if(xml.endsWith(XMLMessage.msgEnd)){
				msg = (XMLMessage) um.unmarshal(new StreamSource(new
StringReader(sb.toString())));
				sb.setLength(0);
				check(msg);
			}
		}
	} catch (JAXBException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

### Code Snippet (checking messages)

```
public void check(XMLMessage m){
	if(m.getType() == MsgType.INFO){
		cgi = new ClientGameInformation();
		cgi.setPlayerNumber(Integer.parseInt(m.getDesc()));
	}
	... etc
}
```

### Code Snippet 1 (join a game)

```
public void join(String gamename){
	logger.info("["+socket.toString()+"] join new game with name [" + gamename + "]");
	sendToServer(MessageFactory.cs_join(gamename));
}
```

```
public static XMLMessage cs_join(String gamename){
        XMLMessage m = new XMLMessage();
        m.setType(MsgType.JOIN);
        m.setDesc(gamename);
        return m;
}
```

### Code Snippet 2 (client map)

```
if(m.getType() == MsgType.RDY){
        cgi.setGameid(Integer.parseInt(m.getDesc()));
        sendToServer(MessageFactory.cs_sendMap(generateMap()));
}

public static XMLMessage cs_sendMap(Map map){
        XMLMessage m = new XMLMessage();
        m.setType(MsgType.MAP);
        m.setMap(map);
        return m;
}
```

### Code Snippet 3 (new move)

```
if(m.getType() == MsgType.MOVE) {
        cgi.setOpponentPosition(m.getCoordinate());
        sendToServer(MessageFactory.cs_move(calculateSearchChestMove()));
}

public static XMLMessage cs_move(Coordinate target){
        XMLMessage m = new XMLMessage();
        m.setType(MsgType.MOVE);
        m.setCoordinate(target);
        return m;
}
```

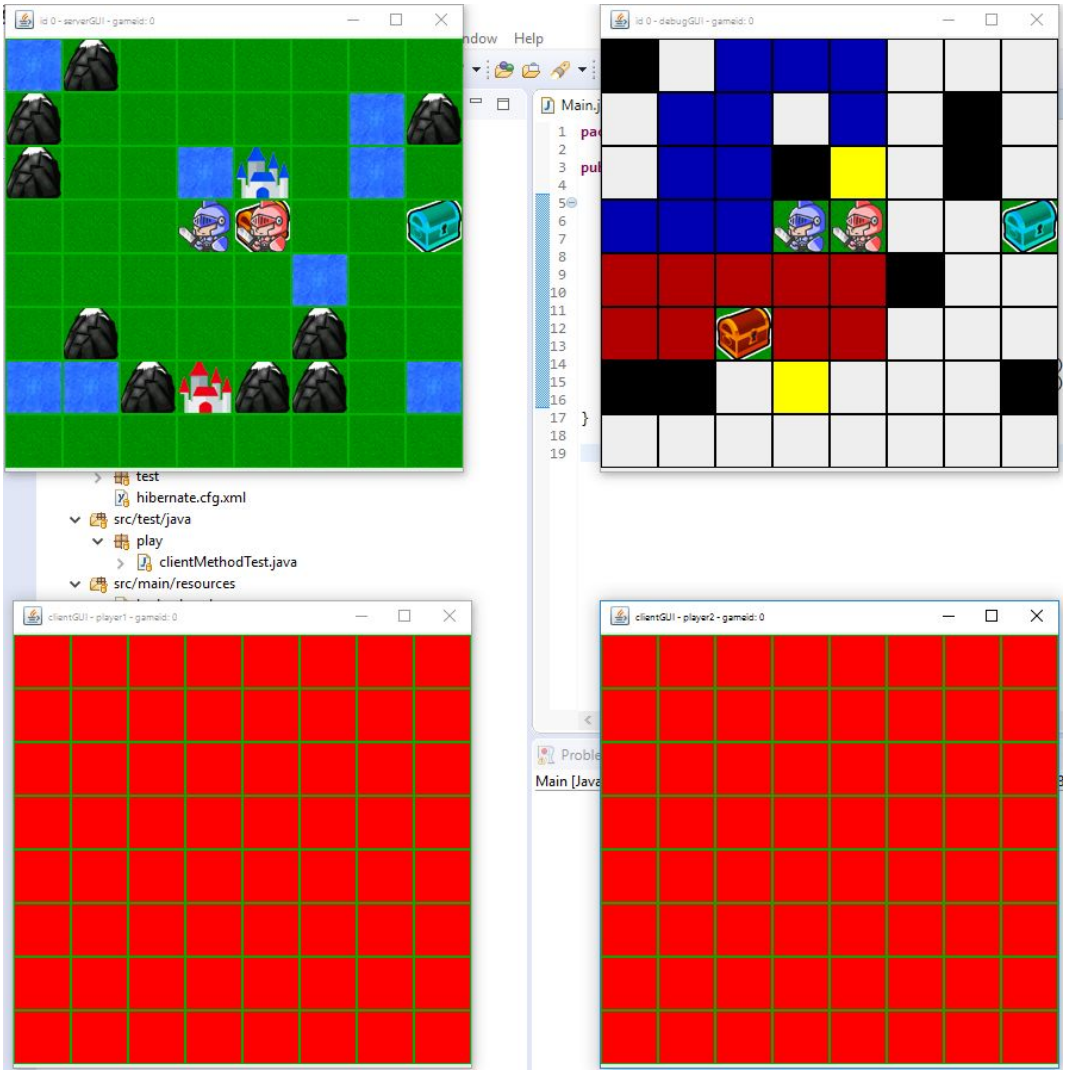
# GUI


**Description**

There are 4 user interfaces (from top left to bottom right)

- server gui
    - visualizes the complete map inclusive player, chest and castle positions
- debug gui
    - visualizes the path each KI takes and which tiles have already been visited by each KI
- player1 gui (red)
    - visualizes both players but no chests. Once the chest is found, it visualizes the chest aswell
- player2 gui (blue)
    - visualizes both players but no chests. Once the chest is found, it visualizes the chest aswell

The images show almost the start of a new game. Player 1 (red) has moved 2 fields north and player 2 has moved 1 field north.




**Description**
The images show the game when it's finished. Player 1 (red) has won and therefore the tiles of both clients turn red and therefore signal to each client who won. In the server gui (top left) it can be seen that player 1 (red) carries the chest and is on his way to the enemy castle. The debug gui shows which fields have been visited by each client.

## MVC pattern

| Nr. | Role | Classes | Description |
|---|---|---|---|
| 1 | M | ClientGameInformation | ClientGameInformation is a model which holds basic information about the current gamestate for each player |

| | | | |
|---|---|---|---|
| | | Game<br><br><br><br>World<br><br><br>Map | Game is the model which holds all informations about the current game, it includes world.<br><br>World holds information about the created world (not just map)<br><br>Map is the grid model |
| 2 | V | MapGui.class | visualizes the models and its changes server- and clientside |
| 3 | C | GameRequestHandler<br><br><br><br><br>Client | GameRequestHandler contains the business methods serverside and makes changes on the models used serverside<br><br><br>Client holds all business methods clientside and takes changes in ClientGameInformation and Map |

**Model Code Snippet (Game.class)**

```
public class Game{

	private int id;
	private World world;
	private PlayerConnection winner, loser;
	private boolean draw;
	private boolean p1HasChest;
	private boolean p2HasChest;

	Game(int id){
		this.world = new World();
		this.id = id;
	}
	//getters and setters
}
```

**Description**

This is a model on the serverside and holds all information about the game. It holds both player connections, the current world (which holds the map) and the game id.

**View Code Snippet (MapGUI.class)**

```
public void drawBasicTextures(Map map) {
	String tile;
	for(int y = 0; y < 8; y++) {
```

```
                for(int x = 0; x < 8; x++) {
                        tile = map.getXY(x, y);
                        if(tile.equals("W")) {
                                cells[x][y].setIcon(water);
                        }
                        else if(tile.equals("M")) {
                                cells[x][y].setIcon(mountain);
                        }
                        else cells[x][y].setIcon(grass);
                }
        }
}
```

**Description**

Shows the drawBasicTexture method from the MapGUI.class. cells[][] is a JLabel array. Calling this method colors all cells.

```
public void showWinner(int winner) {
        for(int y = 0; y < 8; y++) {
                for(int x = 0; x < 8; x++) {
                        try {
                                Thread.sleep(50);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                        if(winner == 1) {

                                cells[x][y].setBackground(Color.RED);
                                cells[x][y].setIcon(null);
                        }
                        else {
                                cells[x][y].setBackground(Color.BLUE);
                                cells[x][y].setIcon(null);
                        }

                }
        }
}
```

**Description**

Shows another method from the MapGui.class. Depending on the winner, it will color the cells either red or blue.

### Controller Code Snippet (Client.class)

```
public Coordinate calculateSearchChestMove() {
        Coordinate trgt;
        int playerX = cgi.getPlayerPosition().getX();
        int playerY = cgi.getPlayerPosition().getY();
```

```
        if(cgi.getPath().isEmpty()) {
                trgt = getNearestCoordinate();
                cgi.setFinalTarget(trgt);
                cgi.setPath(algoMap.findPath(playerX, playerY, trgt.getX(), trgt.getY()));

                removePathFromPossibleChestPositions(cgi.getPath());
        }
        int targetX = cgi.getPath().get(0).getX();
        int targetY = cgi.getPath().get(0).getY();
        Coordinate next = new Coordinate(targetX, targetY);
        cgi.setPlayerPosition(next);
        cgi.getPath().remove(0);
        if(cgi.getClientMap().getXY(targetX, targetY).equals("M")) {
                cgi.setWaiting(true);
        }
        return next;
}
```

**Description**

calculateSearchChestMove() method returns the coordinate the KI will move to next. It then updates the model cgi (ClientGameInformation) with this newly created information.

## Map generation

### Algorithm

It's less an algorithm but more just randomly trying to place water and mountain tiles recursively.

```
public void populate() {
        try {
                for(int x = 0; x < grid.length; x++) {
                        for(int y = 0; y < grid[0].length; y++) {
                                grid[x][y] = "G";
                        }
                }
                int randX = new Random().nextInt(8);
                int randY = new Random().nextInt(4);
                for(int i = 0; i < 3; i++) {
                        placeMountains(randX, randY);
                }

                randX = new Random().nextInt(8);
                randY = new Random().nextInt(4);
                for(int i = 0; i < 4; i++) {
                        placeWater(randX, randY);
                }
                placeFort();
        }
        catch(StackOverflowError e){
```

```
                populate(mountain, water);
        }
}
```

First the grid is getting filled with grass tiles. Then mountains, water and finally the fort.

### placeMountains(int, int) function

```
private void placeMountains(int x, int y) {
        if(grid[x][y] == "G") {
                grid[x][y] = "M";
                return;
        }
        placeMountains(new Random().nextInt(8), new Random().nextInt(4));
}
```

Take 2 random integers, check if grid[randomInteger1][randomInteger2] is a grass tile, if yes then make it a mountain tile, else repeat recursively until the desired mountain count is reached.

### placeWater(int,int) function

```
private void placeWater(int x, int y) {
        if(grid[x][y] == "G") {
                grid[x][y] = "W";
                if(hasIslands(grid)) {
                        grid[x][y] = "G";
                        placeWater(new Random().nextInt(8), new Random().nextInt(4));
                }else if(invalidBorder(grid)) {
                        grid[x][y] = "G";
                        placeWater(new Random().nextInt(8), new Random().nextInt(4));
                }
                else return;
        }
        else placeWater(new Random().nextInt(8), new Random().nextInt(4));
}
```

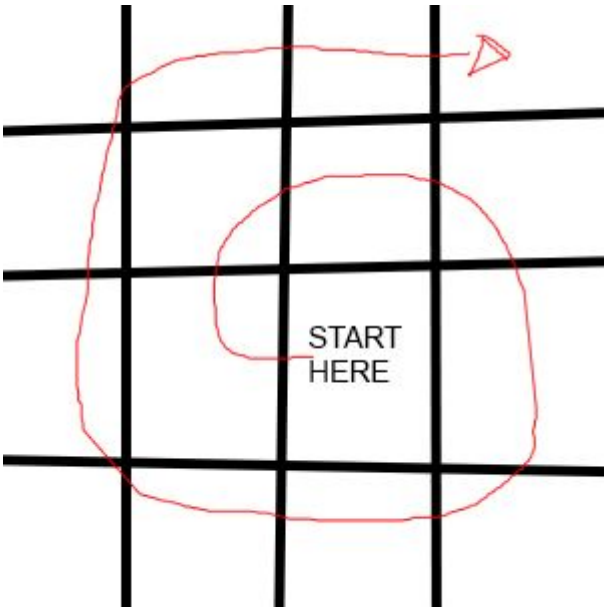
Here aswell, take 2 random integers and try to place it if the tile is grass. After placing, it will check if it generated an island or if it violated another rule (too much water on the border for example). If not, continue, else make it grass again and pick another spot.

### placeFort() function

```
public void placeFort() {
        int direction = 1;
        int counter = 0;
        int repeat = 1;
        int x = width/2;
        int y = height/2;
```

```
        for(int i = 0; i < 64; i++) {
                for(int j = 0; j < repeat; j++) {
                        if(grid[x][y] == "G") {
                                grid[x][y] = "F";
                                return;
                        }
                        switch(direction) {
                                case 1: x-=1; break;
                                case 2: y-=1; break;
                                case 3: x+=1; break;
                                case 4: y+=1; break;
                        }
                }
                counter++;
                direction++;
                if(counter % 2 == 0) repeat++;
        }
}
```

This function is the most algorithm-like. It tries to place the fort as central as possible. If the center is water or mountain, it will pick another spot and repeat in a spiral-out kinda way until it found a free grass spot.




So each generated map is completely different because the water, mountains and therefore grass, but also fort is random.

**Examples**

### Business rules for map generation

| Nr | Rule |
|---|---|
| 1 | At least 4 water, 3 mountain, 5 grass tiles |
| 2 | Water tiles must be less than half the border size (max 3 on long sides) |
| 3 | No islands |
| 4 | Must generate a 8x4 grid |

### Code Snippet 1

```
for(int i = 0; i < 3; i++) {
	placeMountains(randX, randY);
}

for(int i = 0; i < 4; i++) {
	placeWater(randX, randY);
}
```

The grid starts off with grass fields only and then places 3 mountains and 4 water tiles (the loop makes sure of that). Therefore all the remaining tiles are grass and definitely > 5.

### Code Snippet 2

```
private boolean invalidBorder(String[][] grid) {
	int count = 0;
	for(int i = 0; i < 8; i++) {
		if(grid[i][0] == "W") {
			count++;
			if(count > 3) {
				return true;
			}
		}
	}
```

```
        count = 0;
        for(int i = 0; i < 8; i++) {
                if(grid[i][3] == "W") {
                        count++;
                        if(count > 3) {
                                return true;
                        }
                }
        }
        count = 0;
        for(int i = 0; i < 4; i++) {
                if(grid[0][i] == "W") {
                        count++;
                        if(count > 1) {
                                return true;
                        }
                }
        }
        count = 0;
        for(int i = 0; i < 4; i++) {
                if(grid[7][i] == "W") {
                        count++;
                        if(count > 1) {
                                return true;
                        }
                        }
        }
        return false;
}
```

Checks the grid for invalid borders, meaning if there are too many water tiles on the border. It simply iterates through the border and counts up if there is water and if it has a too high count it returns true, meaning it's an invalid border.

### Code Snippet 3

```
private boolean hasIslands(String[][] grid) {
        String[][] grid_copy = new String[width][height];
        for(int x = 0; x < grid.length; x++) {
                for(int y = 0; y < grid[0].length; y++) {
                        grid_copy[x][y] = grid[x][y];
                }
        }
        int count = 0;
        for(int x = 0; x < grid.length; x++) {
                for(int y = 0; y < grid[0].length; y++) {
                        if(grid_copy[x][y] == "G") {
                                count++;
                                merge(x,y,grid_copy);
                        }
                }
        }
        if(count > 1) return true;
```

```
        else return false;
}

private void merge(int x, int y, String[][] grid){
        if (x < 0 || x == grid.length || y < 0 || y == grid[x].length || grid[x][y] == "W"
|| grid[x][y] == "M")
        return;

        grid[x][y] = "W";
        merge(x+1, y, grid);
        merge(x-1, y, grid);
        merge(x, y+1, grid);
        merge(x, y-1, grid);
}
```

Creates a dummy grid and checks recursively if there are islands by starting at 1 spot, marking it and then recursively continuing. This step is being repeated for each single cell. If the island count is higher than 1, it returns true, meaning there is an island not being able to reach.

### Code Snippet 4

```
public Map generateMap() {
        Map m = new Map(8,4);
        int mountainCount = 5;
        int waterCount = 4;
        m.populate(mountainCount, waterCount);
        return m;
}
```

Generates a map exactly with 8x4.

### Advantages of map generation

There are 2 advantages:

1. Fort is placed as central as possible which increases the chance of finding the chest quicker. (see placeChest() method above)
2. Mountains are not creating islands aswell even though it is allowed. I think the advantage I am getting through this, is that I save time sometimes and don't need to climb a mountain to just check 1 remaining spot for example. (hasIslands() method above)

### Responsible classes for map generation

| Class & Package | Responsibility |
|---|---|
| play.map.Map.class | creates the map, populates it, checks for business rules |

# AI

### Ideas, concept and description

The AI has basically 2 tasks in the whole game:

- move until the chest has been found
- move to the enemy castle

### AI process overview *

*after gamestart, meaning both AI are ready to move

1. save opponent starting position because this is where the enemy castle is
2. save all grass spots in an array -> chest positions (chest can only be on a grass tile)
3. pick nearest grass tile
4. move to nearest grass tile using A* algorithm.
5. while walking to nearest grass tile which has not been visited yet
6. repeat until stepping on a tile which has the hidden chest
7. walk to opponent castle using A* algorithm again

### Additional informations

When the AI moves on a mountain field, it will wait 1 round.
AI only moves 1 field each round, either up, left, right or down.

```
if(cgi.getClientMap().getXY(targetX, targetY).equals("M")) {
	cgi.setWaiting(true);
}

if(cgi.isWaiting()) {
	cgi.setWaiting(false);
	sendToServer(MessageFactory.cs_move(cgi.getPlayerPosition()));
}
```

The first condition is being called after the AI calculated its next move. If the target tile is a mountain, it will set the waiting variable to true.

The second condition is being checked in the beginning of the calculating next move method. If the waiting is true, it will just return the current location (won't move for 1 round) and then set the variable to false.

### AI Classes

| Class & Package | Description |
|---|---|
| socket.Client.class | client == player and therefore the client is handling the AI. |

| | |
|---|---|
| play.map.algo.AlgoMap.class<br>play.map.algo.Node.class | A* algorithm classes |

**How to run the .jar?**
java -jar Client_Teilaufgabe_3_Yannick_Koitzsch_01301303.jar

it works just fine but for some reason the tile images are not showing up, so it can look confusing. Images only show up running it with an IDE.