VU Software Engineering 1

Task 2
(Server Implementation)

Prename, Surname:	Yannick Koitzsch
Matriculation number:	01301303
E-Mail:	y.koitzsch@gmx.at
Date:	8.12.2017

Database realisation

As suggested I decided to go with **Hibernate** as the OR-Mapper and **SQLite** as the database.

What is being stored in the database?

- game object (players, map, winner, loser, duration, total moves)
- game_logs (anything that happens during a game)
- server_logs

Implementation:
Repository Pattern
I decided to use the repository pattern for the database/hibernate implementation. It is possible to save, delete, update, find and more...

```
public interface IRepository<T> {
	public void save(T entity);
	public Optional<T> find(Integer id);
	public List<T> findAll();
	public List<T> findAll(Predicate<T> predicate);
	public void update(T entity);
	public void delete(T entity);
	public boolean exists(T entity);
	public List<T> where(String column, String value);
	public List<T> query(String sql);
}
```

Code snippet from the RepositoryImpl.class which shows the implementation of the save method.

```
private Transaction tx = null;
private Session session = null;
@Override
public void save(T entity) {
	try{
		session = HibernateUtil.getSessionFactory().openSession();
		tx = session.beginTransaction();
		session.save(entity);
		session.flush();
		tx.commit();
	}catch(Exception e){
		e.printStackTrace();
		tx.rollback();
	}finally{
		if(session != null) session.close();
	}
}
```

In the following, I show an example of how a **game_log** is being saved.

Step 1:
First a **GameLog.class** is required with correct annotations. It is also important that every class that is being saved in the database, extends the **Model.class** which simply makes sure that there is always an ID.

```
@Entity
@Table(name="game_log")
public class GameLog extends Model{
	@Id
	@GeneratedValue
	int id;
	int game_id;
	String action;
	Timestamp timestamp;

	GameLog(){}
	public GameLog(int id, int game_id, String action) {
		this.id = id;
		this.game_id = game_id;
		this.action = action;
		this.timestamp = new Timestamp(System.currentTimeMillis());
	}

	//getters and setters
}
```

Step 2:
In the class where the gamelog should be saved, initialize and get the gamelog repository

```
IRepository<GameLog> gamelogRep = RepositoryFactory.getRepository(GameLog.class);
```

Step 3:
Whenever necessary, save the gamelog with this simple method call:

```
//examples
gamelogRep.save(new GameLog(game.getGame_id(), "gamestart"));
gamelogRep.save(new GameLog(game.getGame_id(), "player moved to “ + X + Y));
gamelogRep.save(new GameLog(game.getGame_id(), “player1 won”));
gamelogRep.save(new GameLog(game.getGame_id(), “player2 lost”));
```

It is asked to show how to determine a) the winner of a game and b) the 10th and 20th game action. So let’s start with a).
The game.class holds the winner and loser of each game:

```
@Entity
@Table(name="game")
public class Game extends Model{
        enum GameState {PREPARE, MAPX1, MAPX2, PLAY, OVER};

        @Id
        private int game_id;
        @Transient
        private GameState gameState;

        @OneToOne(fetch = FetchType.LAZY)
        private World world;

        private String winner;
        private String loser;

        Game(){}
        Game(int id) {
                gameState = GameState.PREPARE;
                this.game_id = id;
                this.world = new World(8,8);
        }
//getters and setters
```

in order to retrieve this data from the database we can do the following:

Step 1:
Make sure that the class gets the game repository.

```
IRepository<Game> gameRep = RepositoryFactory.getRepository(Game.class);
```

Step 2:
Let’s find a game with the id 1 for example and get the winner of that game.

```
Optional<Game> opt = gameRep.find(1);
Game g = opt.get();
String winner = g.getWinner();
```

and now for b)

Step 1:

```
IRepository<GameLog> gameLogRep = RepositoryFactory.getRepository(GameLog.class);
```

Step 2:
Let’s get the 10th and 20th action of game with id 5.

```
Optional<Game> opt = gameRep.find(5);
Game g = opt.get();
GameLog log = g.getLog(10).getAction();
GameLog log2 = g.getLog(20).getAction();
```

Implementation of the business rules

The 8 business rules from task 1:

1. A chest can only be picked up and used by the correctly assigned KI.
2. A game has a limit of 200 moves in total and each KI has a time limit of 3 seconds for an action.
3. Fort and chest can’t be placed on the same spot.
4. Moving on a water field is an invalid move
5. Forts and chests can only be placed on grass
6. KI generated map has to have a size of 4x8
7. Each generated map of a player must include at least 3 mountain-, 5 grass- and 4 water-fields.
8. The borders of each of the two generated maps must have less than half the water fields (max 3 on the longer sides).

The borders of each of the two generated maps must have less than half the water fields (max 3 on the longer sides).

```
int waterCount = 0;
for(int i = 0; i < 7; i++) {
	if(tiles[i][0].getType() == TileType.WATER) {
		waterCount++;
		if(waterCount > 3) {
			sendToClient(MessageFactory.defeat(),activePlayer);
			toggleActivePlayer();
			sendToClient(MessageFactory.victory(),activePlayer);
		}
	}
}
waterCount = 0;
for(int i = 0; i < 7; i++) {
	if(tiles[i][3].getType() == TileType.WATER) {
		waterCount++;
		if(waterCount > 3) {
			sendToClient(MessageFactory.defeat(),activePlayer);
			toggleActivePlayer();
			sendToClient(MessageFactory.victory(),activePlayer);
		}
	}
}
waterCount = 0;
for(int i = 0; i < 3; i++) {
	if(tiles[0][i].getType() == TileType.WATER) {
```

```
                waterCount++;
                if(waterCount > 1) {
                      sendToClient(MessageFactory.defeat(),activePlayer);
                      toggleActivePlayer();
                      sendToClient(MessageFactory.victory(),activePlayer);
               }
       }
}

waterCount = 0;
for(int i = 0; i < 3; i++) {
       if(tiles[7][i].getType() == TileType.WATER) {
                waterCount++;
                if(waterCount > 1) {
                      sendToClient(MessageFactory.defeat(),activePlayer);
                      toggleActivePlayer();
                      sendToClient(MessageFactory.victory(),activePlayer);
               }
       }
}
```

Explanation:

I iterate through all the border tiles and count the tiles with the TileType.WATER.

Is it easy to modify?

It could be easier to modify, so I will see if I can improve this.

What could I improve?

Maybe the usage of from variables to improve the maintainability of this method. Maybe instead of checking each side alone, there is a possibility to do it all together.

Each generated map of a player must include at least 3 mountain-, 5 grass- and 4 water-fields.

```
long count;
count = tileList.stream().filter(f -> f.getType() == TileType.GRASS).count();
if(count < 5) {
       System.out.println("Not enough Grass Tiles! Counted: " + count);
       sendToClient(MessageFactory.defeat(),activePlayer);
       toggleActivePlayer();
       sendToClient(MessageFactory.victory(),activePlayer);
}
count = tileList.stream().filter(f -> f.getType() == TileType.STONE).count();
if(count < 3) {
       System.out.println("Not enough Stone Tiles! Counted: " + count);
       sendToClient(MessageFactory.defeat(),activePlayer);
       toggleActivePlayer();
       sendToClient(MessageFactory.victory(),activePlayer);
}
count = tileList.stream().filter(f -> f.getType() == TileType.WATER).count();
if(count < 4) {
       System.out.println("Not enough Water Tiles! Counted: " + count);
       sendToClient(MessageFactory.defeat(),activePlayer);
       toggleActivePlayer();
       sendToClient(MessageFactory.victory(),activePlayer);
}
```

Explanation:
I iterate through all tiles and count each individual TileType.
Is it easy to modify?
Same as above. It could be easier to modify.
What could I improve?
Maybe put all checks in one function and using more variables to increase maintainability.

KI generated map has to have a size of 4x8

```
if(!(tiles.length == 8)) {
        System.out.println("Expected width: 8 but was: " + tiles[0].length);
        sendToClient(MessageFactory.defeat(),activePlayer);
        toggleActivePlayer();
        sendToClient(MessageFactory.victory(),activePlayer);
}
if(!(tiles[0].length == 4)) {
        System.out.println("Expected height: 4 but was: " + tiles[1].length);
        sendToClient(MessageFactory.defeat(),activePlayer);
        toggleActivePlayer();
        sendToClient(MessageFactory.victory(),activePlayer);
}
```

Explanation:
Simply check the width and height of the tilemap.
Is it easy to modify?
Since it’s basically only 1 line of code, yes.
What could I improve?
Not much. It’s a very simple check.

Forts can only be placed on grass

```
ArrayList<Entity> tmpEntity = new ArrayList<Entity>();
for(Tile tile: tileList) {
        tmpEntity = tile.getEntities();
        for(Entity e : tmpEntity) {
                if(e.getEntity() == EntityType.FORT) {
                        noFort = false;
                        if(tile.getType() != TileType.GRASS) {
                                System.out.println("Fort is not placed on Grass!");
                                sendToClient(MessageFactory.defeat(),activePlayer);
                                toggleActivePlayer();
                                sendToClient(MessageFactory.victory(),activePlayer);
                        }
                }
        }
}
```

Explanation:
Iterate through all tiles and find the placed fort. Then check if the TileType where the fort is placed is equal to GRASS.
Is it easy to modify?
It’s again a simple check but could probably improved when it comes to maintainability.
What could I improve?

Maybe instead of giving each tile a list of entities, give the tile a instance of a FORT entity. So it’s easy to get it and no iteration through the entitylist isrequired.

Chests can only be placed on grass and Fort and chest can’t be placed on the same spot

```
public void placeChests() {
        System.out.println("Placing Chest for P1...");
        tryToPlaceChest(0,7,0,3);
        System.out.println("Placing Chest for P2...");
        tryToPlaceChest(0,7,4,7);
}

public void tryToPlaceChest(int x1, int x2, int y1, int y2) {
        int randX = ThreadLocalRandom.current().nextInt(x1,x2+1);
        int randY = ThreadLocalRandom.current().nextInt(y1,y2+1);
        Tile t = tiles[randX][randY];
        if(t.getType() == TileType.GRASS && !t.hasFort()) {
                System.out.println("Successfully placed chest");
                tiles[t.getX()][t.getY()].getEntities().add(new Entity(EntityType.CHEST));
                return;
        }
        else {
                tryToPlaceChest(x1,x2,y1,y2);
        }
}
```

Explanation:
Player 1 map is always in the grid from 0/0 to 7/3. Player 2 map is always in the grid from 0/7 to 4/7. These coordinates are passed into the method and then the chest is tried to be placed on GRASS and NOT on the same tile with the FORT.
Is it easy to modify?
The coordinates can be modified in the beginning but that’s it. So not very much.
What could I improve?
Alot. Until next time I would like to randomize the map position so player 1 map is not always above the player 2 map. The placing chest method is using recursion which is not very efficient and will be improved as well until next time.

Each KI has a time limit of 3 seconds for an action.

```
private void checkTime() {
        double elapsed = (System.nanoTime() - startTime)/1000000000.0;
        if(elapsed > 3) {
                sendToClient(MessageFactory.defeat(),activePlayer);
                toggleActivePlayer();
                sendToClient(MessageFactory.victory(),activePlayer);
        }
}
```

Explanation:

After sending a message to the client, the server starts a timer and when the answer comes in, the method checkTime() gets called and checks if the answer was in time.

Is it easy to modify?

There is not much to be modified, except the 3 seconds.

What could I improve?

Use a variable for the 3 seconds. So it can be easily modified.

A game has a limit of 200 moves in total

```
else if(game.getGameState() == GameState.PLAY) {
        steps++;
        if(steps > 200) {
                sendToClient(MessageFactory.defeat(),activePlayer);
                toggleActivePlayer();
                sendToClient(MessageFactory.victory(),activePlayer);
        }
        else
                sendToClient(MessageFactory.move(),activePlayer);
}
```

Explanation:

When a new “round” begins, the step count is being increased by 1 with a limit up to 200.

Is it easy to modify?

Same as above. 200 can be modified. It’s a simple task

What could I improve?

Use a variable for the 200 instead. So it can be easily modified.

Moving on a water field is an invalid move.

```
String coordinates = xml.getDesc();
        int x = Integer.parseInt(coordinates.substring(0,coordinates.indexOf(",")));
        int y = Integer.parseInt(coordinates.substring(coordinates.indexOf(",")+1));
        if(game.getWorld().getTile(x, y).getType() == TileType.WATER) {
                sendToClient(MessageFactory.defeat(),activePlayer);
                toggleActivePlayer();
                sendToClient(MessageFactory.victory(),activePlayer);
        }
```

Explanation:

The target coordinates are stored in the description of the xml in the following format: “x,y”. So first the coordinates being “extracted” and then checked if there is water on those coordinates.

Is it easy to modify?

Not much, can be improved.

What could I improve?

Maybe instead of sending them as string, they can be sent as a Coordinate.class object. Which makes the code “more beautiful” getting the coordinates with instead of using String and substring etc.

A chest can only be picked up and used by the correctly assigned KI.

```
public boolean checkForPlayerChest(int x, int y) {
        if(activePlayer == p1) {
            if(y < 4) {
                  if(game.getWorld().getTile(x, y).hasChest()) {
                        return true;
                  }
                  return false;
            }
            return false;
      }
      if(activePlayer == p2) {
            if(y > 3) {
                  if(game.getWorld().getTile(x, y).hasChest()) {
                        return true;
                  }
                  return false;
            }
            return false;
      }
      return false;
}
```

Explanation:

First i check if it’s the turn of player 1 or player 2. If it’s player 1 and the current position of the player is y > 3 (which means it’s on the other players side) , it returns false. Because even if there is a chest, it’s not his own. So in case y < 4 and there is a chest, it’s 100% his chest.

Is it easy to modify?

Not much, can be improved.

What could I improve?

This is not a very good solution and will be improved until next time. Maybe a chest should have a player assigned which would make it easier.

Network communication

Everytime the server receives a new message, the message is being inspected. Examples of receving messages from the client with the MsgType.MAP, MsgType.MOVE and MsgType.RDY.

```
public void inspectMessage(XMLMessage msg){
      if(msg.getType() == expectedCommand.get(game.getGameState())) {
            if(msg.getType() == MsgType.RDY) {
                  System.out.println(activePlayer.getTempName() + " is ready");
            }

            if(msg.getType() == MsgType.MAP) {
                  System.out.println(activePlayer.getTempName() + " sent map");
                  validateMap(msg.getTiles());
            }

            if(msg.getType() == MsgType.MOVE) {
                  System.out.println(activePlayer.getTempName() + " wants to move");
                  validateMove(msg);
            }
```

```
	}
	else {
		System.out.println("ERROR: Wrong Message Type. Received: "+msg.getType() +"
Expected: "+game.getGameState());
		sendToClient(MessageFactory.defeat(),activePlayer);
		toggleActivePlayer();
		sendToClient(MessageFactory.victory(),activePlayer);
		}
	}
```

After the message being inspected and there have not been any errors, the active player is changing and the server is waiting again for a new message. If both players are done with the current gamestate (both players sent their maps or both players are ready to play) the server moves to the next step and updates the gamestate.

```
public void updateGameState() {
	System.out.println("Update GameState...");
	if(game.getGameState() == GameState.PREPARE) {
		game.setGameState(GameState.MAPX1);
	}
	else if(game.getGameState() == GameState.MAPX1) {
		game.setGameState(GameState.MAPX2);
	}
	else if(game.getGameState() == GameState.MAPX2) {
		game.setGameState(GameState.PLAY);
	}
}
```

Once this is done aswell, the server then notifies the player about the gamestate change.

```
public void notifyPlayer() {
	if(game.getGameState() == GameState.PREPARE) {
		sendToClient(MessageFactory.gamestart(),activePlayer);
	}
	else if(game.getGameState() == GameState.MAPX1) {
		sendToClient(MessageFactory.generateMap(),activePlayer);
	}
	else if(game.getGameState() == GameState.MAPX2) {
sendToClient(MessageFactory.sendCompletelMap(game.getWorld().getTiles()),activePlayer);
	}
	else if(game.getGameState() == GameState.PLAY) {
		steps++;
		if(steps > 200) {
			sendToClient(MessageFactory.defeat(),activePlayer);
			toggleActivePlayer();
			sendToClient(MessageFactory.victory(),activePlayer);
		}
		else
			sendToClient(MessageFactory.move(),activePlayer);
	}
}
```

So what's happening is that the server is going through 5 steps over and over again.

1. listen to client
2. inspect client message
3. toggle active player
4. change game state
5. notify player → back to step 1 listen to client

Logging

Useful log examples:

Example 1

After a client connects to the server I am logging this event. It's important to know when and if a client connected and what is the socket of the client. I consider this as one of the most important information.

```
LOGGER.info(clientSocket + " connected to server");
```

Example 2

The client tries to join a random game or a game with a name. This is important to know in case 2 players are connected but no game starts. It could be possible that 2 player joined a game with different names.

```
LOGGER.info(socket + " is joining random game");
or
LOGGER.info(socket + " is joining premade game with name: " + m.getDesc());
```

Example 3

When both players connected to a game and the game starts.

```
LOGGER.info("game started");
```

Example 4

Server and Clients are exchanging a lot of messages. Every time the server sends or receives a message, this should be logged and checked if there is an error.

```
LOGGER.info("received message from " + activePlayer);
```

Example 5

When the client sends a message with the wrong type, this then should be logged.

```
LOGGER.warning("received message from " + activePlayer + " with wrong message type: " +
msg.getType() + " expected: " + expectedCommand.get(game.getGameState()));
```

Content of the logfile

it basically shows the procedure of a game. First clients connect, then join a game, then the game starts. Then the game message exchanging starts etc.

```
Dez 08, 2017 9:43:17 PM socket.Server listen
INFORMATION: Socket[addr=/127.0.0.1,port=57556,localport=4444] connected to server
Dez 08, 2017 9:43:17 PM socket.Server listen
INFORMATION: Socket[addr=/127.0.0.1,port=57555,localport=4444] connected to server
Dez 08, 2017 9:43:17 PM socket.JoinGameHandler check
INFORMATION: Socket[addr=/127.0.0.1,port=57556,localport=4444] is joining premade game with
name: test
Dez 08, 2017 9:43:17 PM socket.JoinGameHandler check
INFORMATION: Socket[addr=/127.0.0.1,port=57555,localport=4444] is joining premade game with
name: test
Dez 08, 2017 9:43:17 PM play.GameRequestHandler run
INFORMATION: game started
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: received message from socket.PlayerConnection@537a2458
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: inspecting message with type RDY
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p1, now its p2 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: notify player
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: received message from socket.PlayerConnection@44552db4
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: inspecting message with type RDY
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p2, now its p1 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: notify player
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: received message from socket.PlayerConnection@537a2458
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: inspecting message with type MAP
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p1, now its p2 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p2, now its p1 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p1, now its p2 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler toggleActivePlayer
INFORMATION: Active player was p2, now its p1 turn
Dez 08, 2017 9:43:17 PM play.GameRequestHandler listen
INFORMATION: notify player

...
```