VU Software Engineering 1

Submitted document

Subtask 2
(Implementation of the Server)

Last Name, Name	Hasho Alisa
Matriculation number	01368897
E-Mail Address	a01368897@univie.ac.at
Date	08.12.2017

1. Changes and adjustments made compared to Subtask 1

As shown in the submission of Subtask 1 documentation, there has been significant adding and changing of the implementation components.

I found the amount of expectations in the given time, without any prior experience in software development with any of the required frameworks, unrealistic.
Therefore, i decided to implement everything at this stage with the frameworks i am familiar with and make the necessary adjustments in the client implementation.

- Starting with the database where i decided to use **PostgreSQL** (an object-relational database management system) as my database server, simply because that is the language i have the most experience on. Knowing that it is not file based, i would ask you to download the environment (version 10.1) and i have included a configuration file in the GIT repository with the credentials to make the running possible.

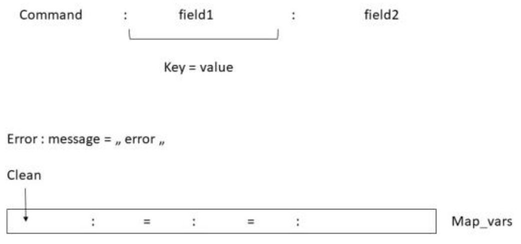
*Please set up your own user name and password after setting up the postgres database.

The database is comprised of one table called “Match History” with the attributes Player_0, Player_1, Score_0, Score_1, and Timestamp. There is a second table called “Match moves” that has an ID referring to a third table called “Client names”. This third table records names and IDs (as a serial number) every time a client registers for the game.

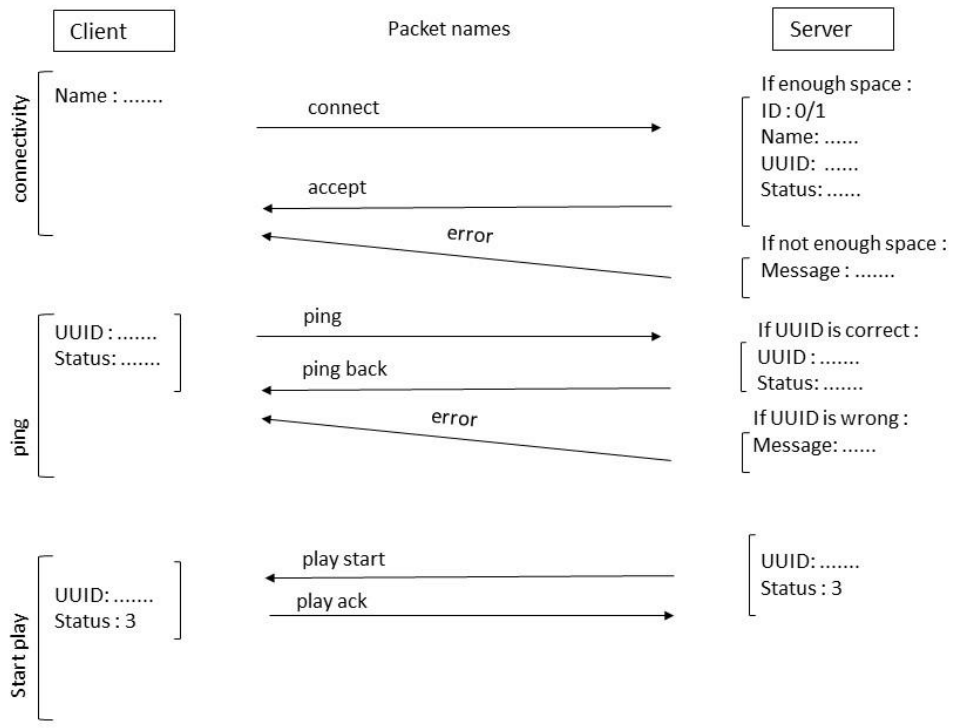
The method insert_score makes the recording of the game scores possible, regarding the business rules. Furthermore the server can record previous game winnings or loses and reach out to them with the method read_scores.

- Second, regarding the network communication, the scheme below is the core of my implementation:

The program as i conceptualized it is a number of strings going back and forth from the server to the client, sending and receiving information. It is formatted in a way that the computer can understand it quickly. The command is what embeds the requirement from the client to the server. With the fields it works somehow differently. Each field is separated by an equal and before the equal there is a key, what comes afterwards is a value. We can have infinite fields theoretically if we wanted to and everything is separated within with colons, one field to the other and commands are also separated by colons. The fields within the commands are then separated with equals (there is a key and value for each string).



Below i have also schematically described the exchange of messages between the client and the server, some examples that show the beginning of their interaction:



2. <u>The current state of development and which areas of the subtask have been fulfilled or not fulfilled</u>

I have tried to implement most of the prescribed business rules in Subtask 1. The only rules that are not yet fully implemented are the ones related to the DB, since i will transfer into a file based one in the third hand in (e.g the recording of the players actions and the possibility to query them).

In the uploaded GIT Bundle file you will find the following components:

CDS = Client dynamic storage. It states all the static variables that we don’t want to be changed in the future and it also contains the methods that treat the map:

1. random_different_numbers – the possible range
2. setup_map
3. check_move – checks the input of moves from clients
4. check_sent_map_status - checks whether the client sent the entire map
5. is_map_full - checks if the client actual sent anything
6. reverse_string - reverses sent strings
7. merge_sent_maps – merges both clients maps into a string
8. merge_sent_map_secret – same as 6 but also randomizes the location of the castle and treasure in each side of the playing field

Connection Manager is where all the connections are handled:

1. run – runs on a thread of the class connection_manager (otherwise the GUI can't be run). Here is where all the inbound connections from the client (inputs) are presented and checked.

2. The rest of the methods are sublogical parts (packets being sent) to fullfil the requirements for the game. The important methods would be:

   a) create_packet_bytes – takes a command word, the keys and the values, puts them into a string and then returns you the bytes for that string, which is what is actually transmitted with every packet sent
   b) command_ parse – parses the command word for every incoming packet to the server
   c) parse_request_map – maps all the keys to all the values of the incoming requests from the client
   d) conection_request – makes the first connection to each client

DB_Management is comprised of a number of dynamic methods that insert, read and record certain variables during the game. (Explained in section 1)

The logger: Main Panel is where the GUI is located and it is controlled by Server_Start. It shows parts of the implementation that I held as necessary to log (when was the last ping, when is the game in process, when did the game start, when the client is connecting). The logging is yet to be improved and added more functionality.

Property File reads the properties for the database and **Server Start** enables both threads for the main panel (one for the connection manager and one for the GUI).

All of the above and the rest of the methods are also commented in the source code for ease of understanding of the work sequence.

I have also implemented a test client to test the server but it is not fully functional with the requirements of the third deadline yet. It is however present in the uploaded file.

3. Instructions to execute the created implementations

Regarding the database management there is a Property file included in the GIT which makes the necessary connection and a Configuration file with the credentials.

You start the solution within the package Main server, file Server_start.java, class Server_start.

4. Logging

In order to make logging possible for this project i have used the Java default framework (java.util.logging package), that provides all objects, methods and configuration necessary to transmit log messages. Loggers are the objects that trigger log events and in this case they are created and called in the source code.
Create : Logger.getLogger(). getLogger() with a string parameter for the name of the logger.
I have determined three log levels, based on their severity:

1. Severe = error
2. Warning
3. Info = success

All my logs are stored in a text file attached in the ZIP file (Mylog.txt) as well as in the GIT Repository. I have created the static method log, which takes as parameters the integer values 0, 1 and 2 for each level (0-error / 1- success / 2- warning). I call this method from different parts in the code with classes that I thought as relevant to log. Examples:

a) Database Management: Log.log(0, "Database driver not found.");
b) Database Management: Log.log(0, "Connection to database "+db_name+" on "+db_host+" on port "+db_port+" failed. Check host and credentials.");
c) Read_scores : Log.log(0, "Reading scores from DB failed");
d) Start Game : Log.log(1, "Sent packets to start the game");
e) Connection Manager – Run : Log.log(1, "Incorrect command word sent");

I initially wanted to log in class CDS, method random_different_numbers but decided not to because it is a subroutine whose result can be tested with the call of other method.

Test client is called filippo – it prints everytime the client connects.

MyLogFile is included in the ZIP folder as an example of the log outputs from the code in different occasions. Since the server alone cannot test many of the methods the log outputs only describe functionalities as the test client connecting, maps being full or not, database successfully connected or not etc.

5. JUnit Testing

The tests implemented in the source code validate if that code results in the expected state (state testing) or executes the expected sequence of events (behavior testing). The following explains what is being tested by the implemented Junit tests:

CDSTest.java

```
➢ @Test
void testRandom_different_numbers() {
        assertTrue(CDS.random_different_numbers(4)[0] <= 3
                        && CDS.random_different_numbers(4)[1] >=0);
}
```

This method test the random_different_numbers method. This is a subroutine method used as a helper to generate the game map. The method takes a range as an argument and returns a number between the legal values. The test asserts that the values are between the legal range.

```
➢ @Test
void testIs_map_full() {

        String[]                              client1_map                                    =
{"GGGMGGGG","GGGGGMGG","GGGGGWGG","GGGGGGG"};
        CDS.client1_map = client1_map;

        String[]                              client2_map                                    =
{"GGGMGGGG","GGGGGMGG","GGGGGWGG","GGGGGGG"};
        CDS.client2_map = client2_map;

        assertEquals( true, CDS.is_map_full(0));

}
```

Is_map_full is a method used to check that the maps coming from the players are complete. We give two complete maps and assert that the result is true.

```
➢ @Test
void testCheck_move(){
        char[][] map_secret = {{'G','G','1','G','G','G','W','2'},
                                                  {'G','W','W','G','G','G','W','G'},
                                                  {'G','G','G','G','W','G','G','G'},
                                                  {'G','G','G','G','G','M','M','G'},
                                                  {'G','G','M','G','G','G','G','G'},
                                                  {'G','G','G','G','G','W','G','G'},
                                                  {'G','G','G','M','G','W','8','G'},
                                                  {'7','G','G','G','G','G','G','G'}};

        CDS.map_secret = map_secret;

        assertEquals( 1, CDS.check_move(0,0, 7));
}
```

This method is called to check if a player move is legitimate and what results as a consequence. This test asserts that when moving to the location (0,7) of our matrix we get the return code 1. Code 1 means that the user found the enemy castle and that the game is over.

Connection_ManagerTest.java

```
➢ @Test
void testCommand_parse() {
        String string_test = "command:key=value:";
        byte[] byte_test = string_test.getBytes();
        DatagramPacket      packet_test      =      new      DatagramPacket(byte_test,
byte_test.length);
```

```
        assertEquals("command", conn_manager.command_parse(packet_test));
    }
```

This is to test that the command_parse() method parses the command correctly. By asserting equality on the “command” argument with the return value, we test that the method returns the expected value.

Property_FileTest.java

- @Test

```
    void testGet_property() {
        assertEquals("localhost",Property_File.get_property("db_host"));
    }
```

Get_property is the method that extracts the PSQL database information from the config.properties file. This file holds all relevant information for connecting to a local installation of psql. We assert equality on the value of db_host.