

2013

# **SNAKES & LADDERS GAME**

# JAVA 'MINI-ASSIGNMENT'

## Vladimir Akopyan & Sanzhar Batyrov



## TABLE OF CONTENTS

Introduction .....	2
Aims: .....	2
Objectives: .....	2
Specification:.....	3
Rules of Snakes and Ladders.....	3
Design.....	4
Client: .....	4
UI:.....	4
User interface During initial design: .....	4
User interface During final implementation:.....	5
Networking: .....	9
Class relations (UML diagram) .....	11
Network Class .....	12
Functionality .....	12
Methods & Constructors.....	12
SnakesLaddersGUI Class .....	13
Functionality .....	13
Methods.....	13
• Construcor SnakesLaddersGUI() – constructs the players, initializing their locations....	13
Mirrorimageicon Class .....	15
Functionality .....	15
Methods.....	15
BoardJpanel Class.....	16
Functionality .....	16
Methods.....	16
Data Structures: .....	16
User interface Considerations .....	17
Testing.....	17
Conclusion:.....	25
Future Improvements .....	25
References: .....	25

## INTRODUCTION

### AIMS:

To design a multiplayer networked board game called snake-and-ladders using the Graphical User Interface(GUI) and Networking tools of the programming language Java.

### OBJECTIVES:

- To design a networked snakes-and-ladders game that will allow 2-5 players to simultaneously play the board game over the network;
- To write a multi-threaded server to handle client connections and to manage the state of the game (see Figure 1);
- To write a graphical client to allow each of the players to play the game.
- Design an appropriate User Interface to interact with the user, that would be user-friendly and convenient;
- Write a code for the Snakes-and-Ladders game logic.
- Build upon the previously learned knowledge and skills in order to improve our programming skills in general.
- Understand the concepts used within the project.

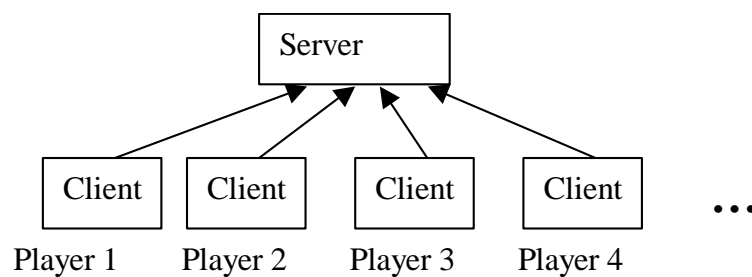


Figure 1: Multiple clients (one per player) connecting to the server. [3]

## **SPECIFICATION:**

A class to display the board (on the client), *SnakesAndLaddersGUI*, has already been written for us and we are expected to use this class in our code.

1. Multi-threaded server and associated client must be developed. This is for the game to be played over the network, for multiple players (in our case, from 2-5). They should be able to communicate effectively. All the messages and commands coming from the server should be delivered to all the clients properly.
2. The graphical user interface must be designed using Layout Managers. All the necessary buttons and panels must be available for the game to run ideally.
3. The game should run according to the logic of the game. This corresponds to the rules for the game, which are as follows:

## **RULES OF SNAKES AND LADDERS**

- 1) Every player has a counter
- 2) At the start of the game all the counters are on square 0 (off the board).
- 3) Each player takes turns in rolling a dice.
- 4) The player's counter advances the number of squares shown on the dice.
- 5) If the counter finishes on the bottom of a ladder it moves to the top of the ladder.
- 6) If the counter finishes on the top of a snake it moves to the bottom of the snake.
- 7) If a player reaches square 100 they win.
- 8) If the current position plus the dice number exceeds 100, they don't move (e.g. on square 98 – rolls a six – player doesn't move) [3]

# DESIGN

## CLIENT:

### UI:

The Client consists of two main components: the user interface, and the networking. There is also data storage for position's of player's pegs.

### User interface During initial design:

Asks for server's IP address  
Connects to the server


Clear once the player has connected  
Display list of connected players when Admin player connects

Should be replaced with a pop-up window later in development


Server ip:   1

100 99 98 97 96 95 94 93 92 91  
81 82 83 84 85 86 87 88 89 90  
80 79 78 77 76 75 74 73 72 71  
61 62 63 64 65 66 67 68 69 70  
60 59 58 57 56 55 54 53 52 51  
41 42 43 44 45 46 47 48 49 50  
40 39 38 37 36 35 34 33 32 31  
21 22 23 24 25 26 27 28 29 30  
20 19 18 17 16 15 14 13 12 11  
2 3 4 5 6 7 8 9 10

Please Enter server IP in the correct Format! 5

Your colour is:  2

Player 2 is rolling the dice 3

Your Dice 4 

Roll

Notifies player of his piece

Notifies the player about the flow of the game, i.e. who's turn, if the needs to roll, etc.

Replace with a picture if the development time allows/deemed worthwhile

Indicate when the button is inactive by 'greying out'

The user interface can be redesigned during implementation to improve usability and code elegance. However is it worthwhile to discuss high-level overview.

Section 2 informs player of his Number and colour.

Section three notifies player of what's currently happening and when it's his turn

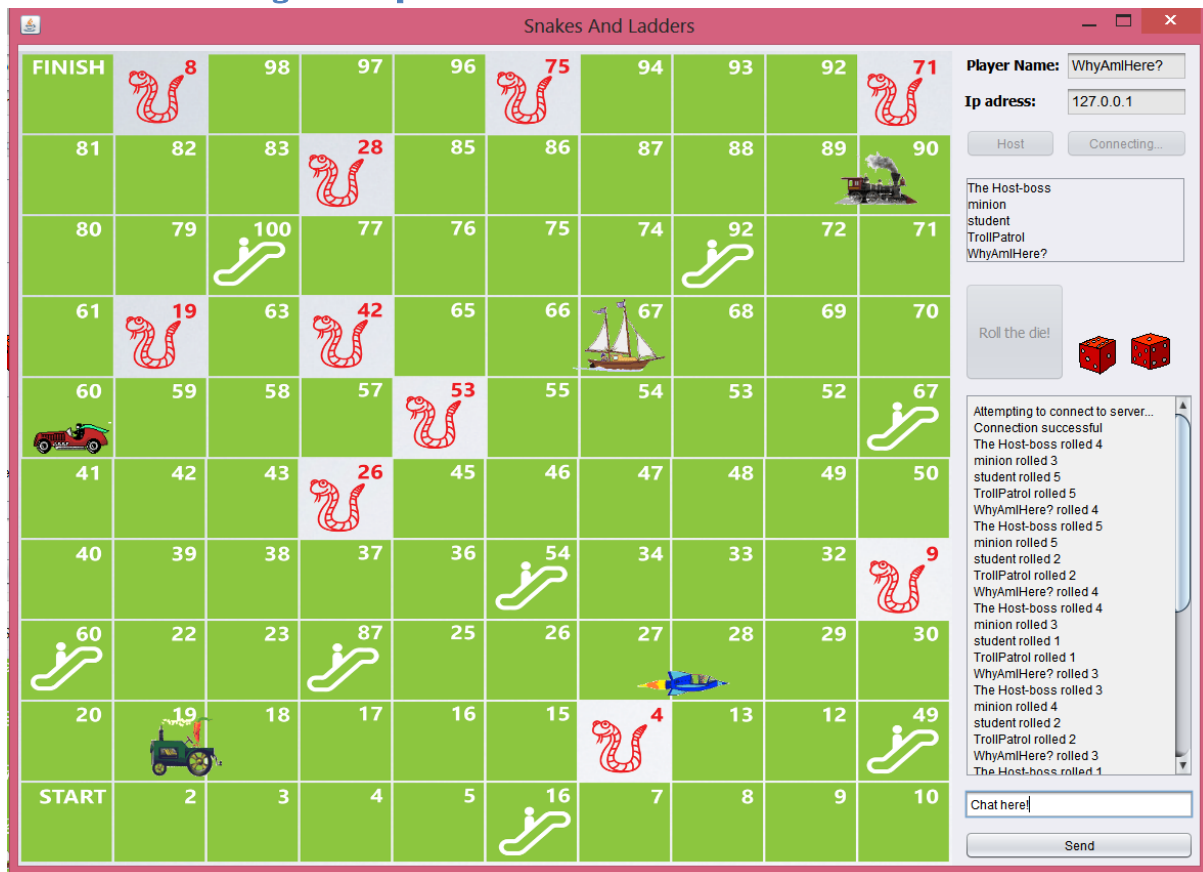
Section 4 contains the number from his previous roll, as well as the roll button. The button should have some indication of when it is active or inactive. One way to achieve this is through 'gray out' convention.

Section 1 is could be implemented as drawn, or as a pop-up window displayed at application launch. After connection is successful, it should be cleared and replaced with a list of all players present on the server.

Section 5 can be replaced by pup-up if only used for critical information and errors.

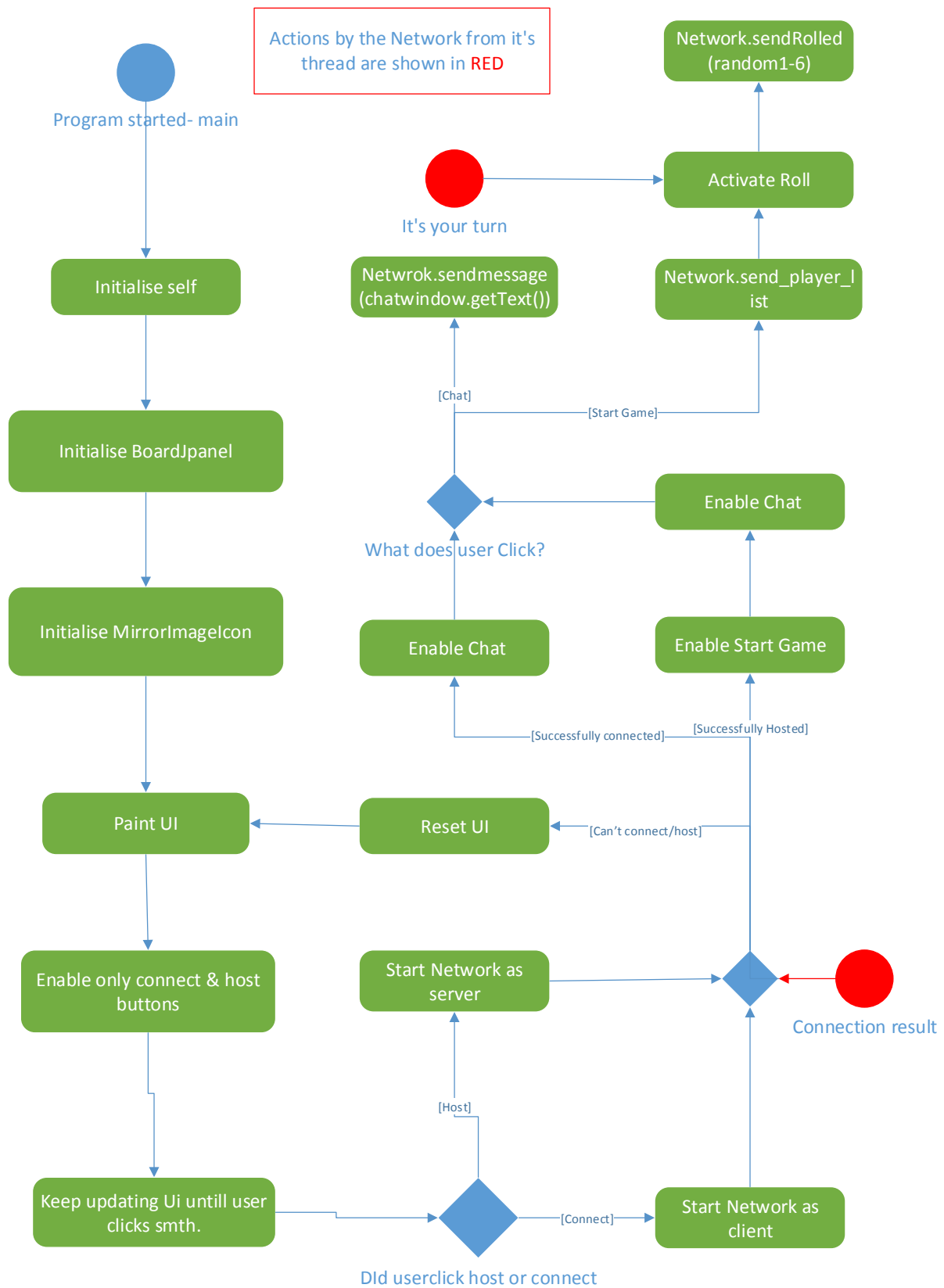
Player should be able to enter his name at the beginning of the game.

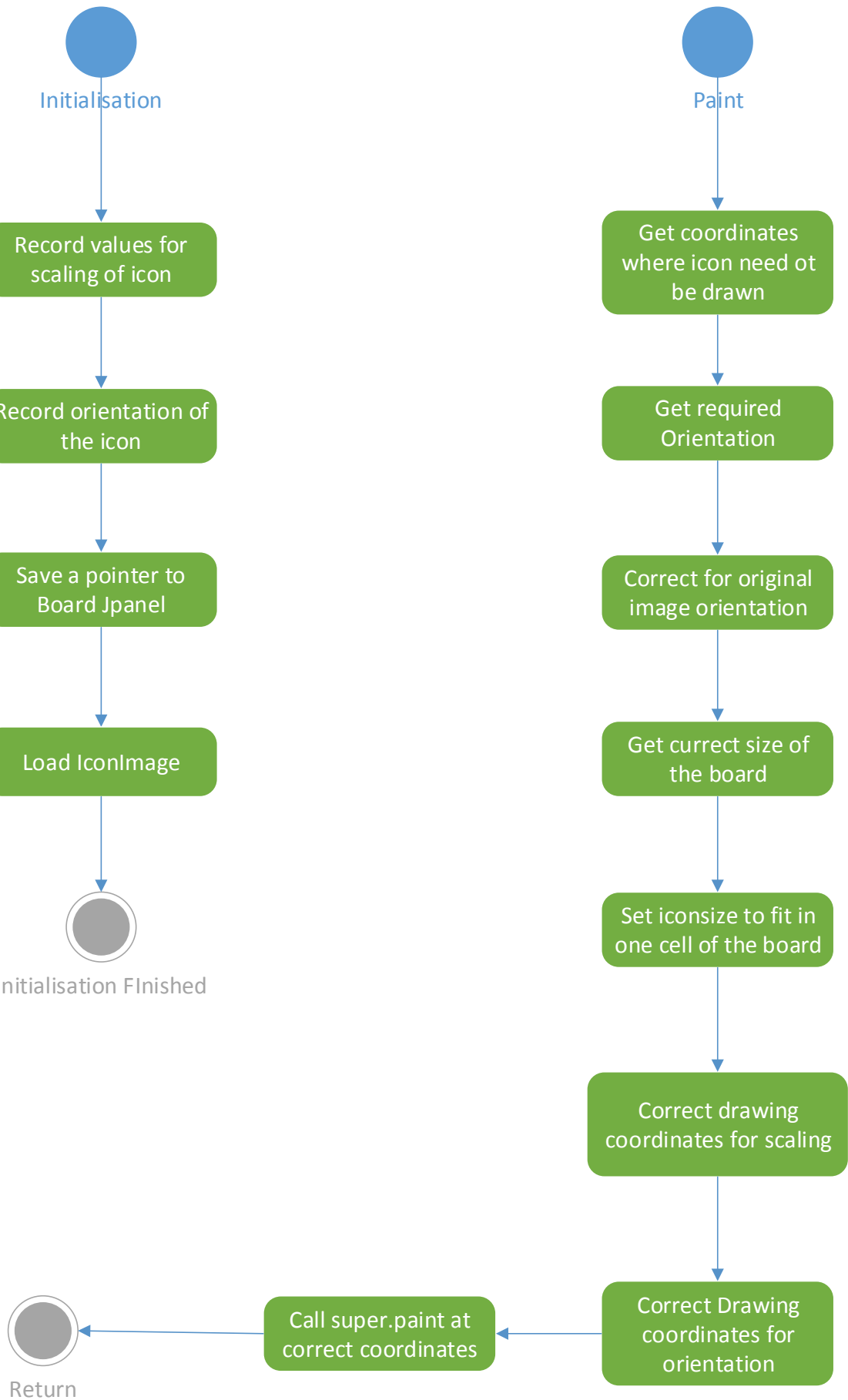
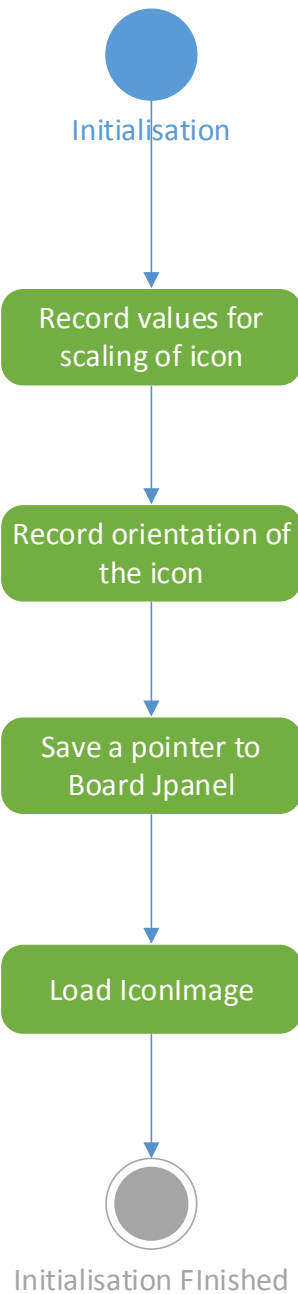
## User interface During final implementation:



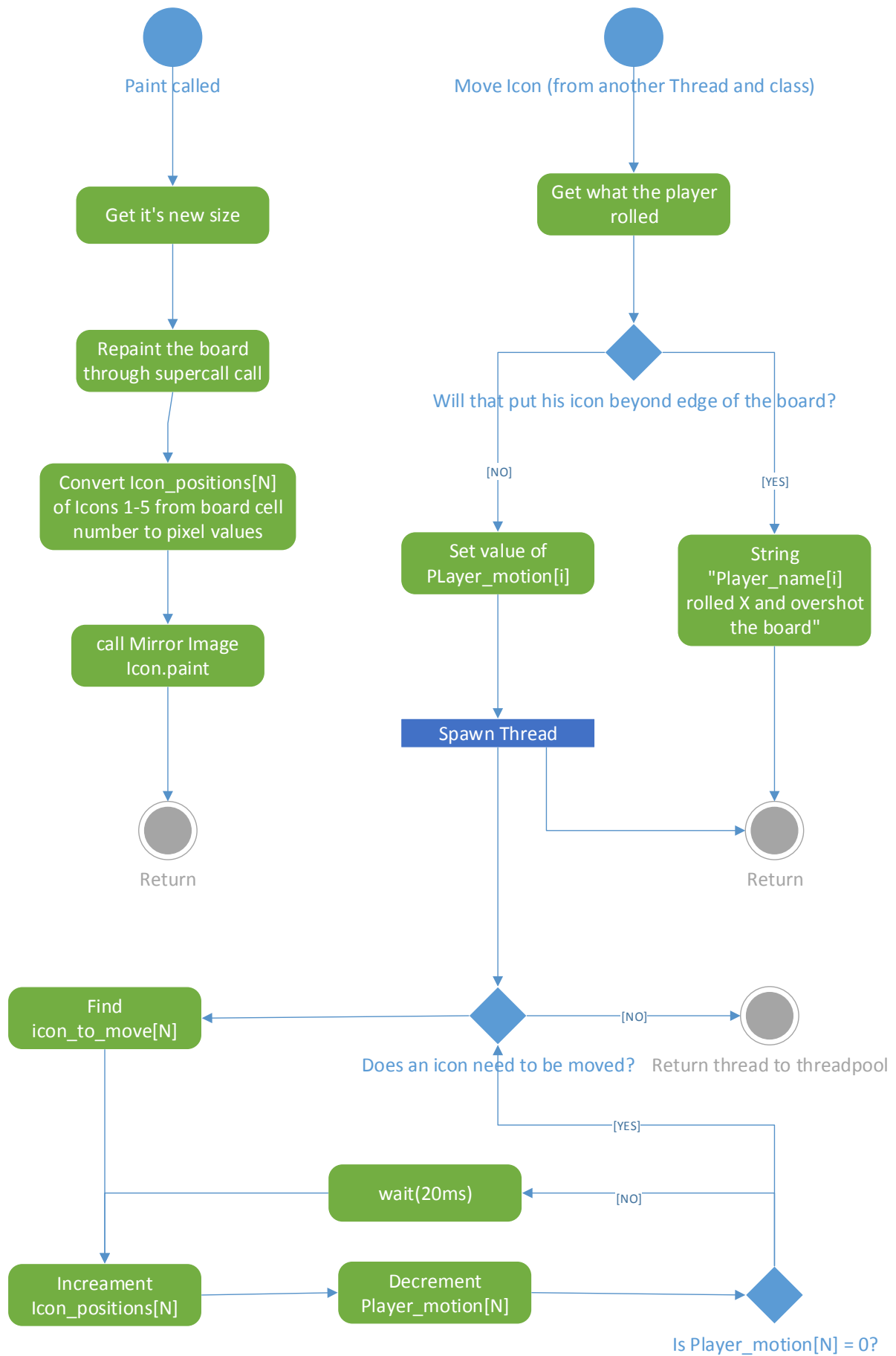
In the real world functionality of UI is more complex than just displaying; it involves interacting with a lot of other classes, as illustrated by the following UML diagrams, one for overall Ui behaviour, one for board behaviour, and one for functionality of icon drawing class. :

A print function is not displayed due to it's simplicity and lack of connection to other parts of Ui code. It is however tightly connected to network code as it implements chat functionality.









## Networking:

Network communication is done in separate thread. As network communication can take unspecified amount of time, doing this in the main sequence would freeze up the UI for arbitrary amount of time. The networking code must account for **exceptions**: Timeout, corrupt packets, and packet loss.

The following pseudo-code describes Network functionality:

*Connect to server and establish handshake*

*While(game is active)*

*Read which player's turn it is*

*If the turn belongs to this player*

*Tell player to roll the dice (through UI)*

*Send the roll to the server*

*Receive updated state of the board*

*Display updated state of the board (UI code)*

*Else*

*Receive updated state of the board*

*Display updated state of the board (UI code)*

*Receive winning player*

*Announce winner (UI code)*

*Disconnect from server/close sockets*

UI code should be executed automatically by a separate thread as soon as communication data is received.

During every communication with the server received data must be validated. If corrupted packet is received, it should be requested again. A timeout must be used, and in case of missing packet, it should be requested again.

Following is a functional description of the server-side networking:

*From main: Create 1 thread*

*From thread: Wait for connection*

*Handshake and grant user admin rights*

*Receive how many players are required*

*Create corresponding number of threads*

*Wait for connection or timeout*

*Kill unused threads if timeout was reached*

*Announce to players: Number of players and their names*

*Announce to players: game start*

*Loop*

*Announce to players: Player X turn*

*Wait for dice roll*

*If he Dc's/doesn't roll, kick his ass*

*Pass roll to game logic*

*Announce to players: new board data*

*X is next player*

*Repeat until a player reaches end of game (100<sup>th</sup> square)*

*Announce winner*

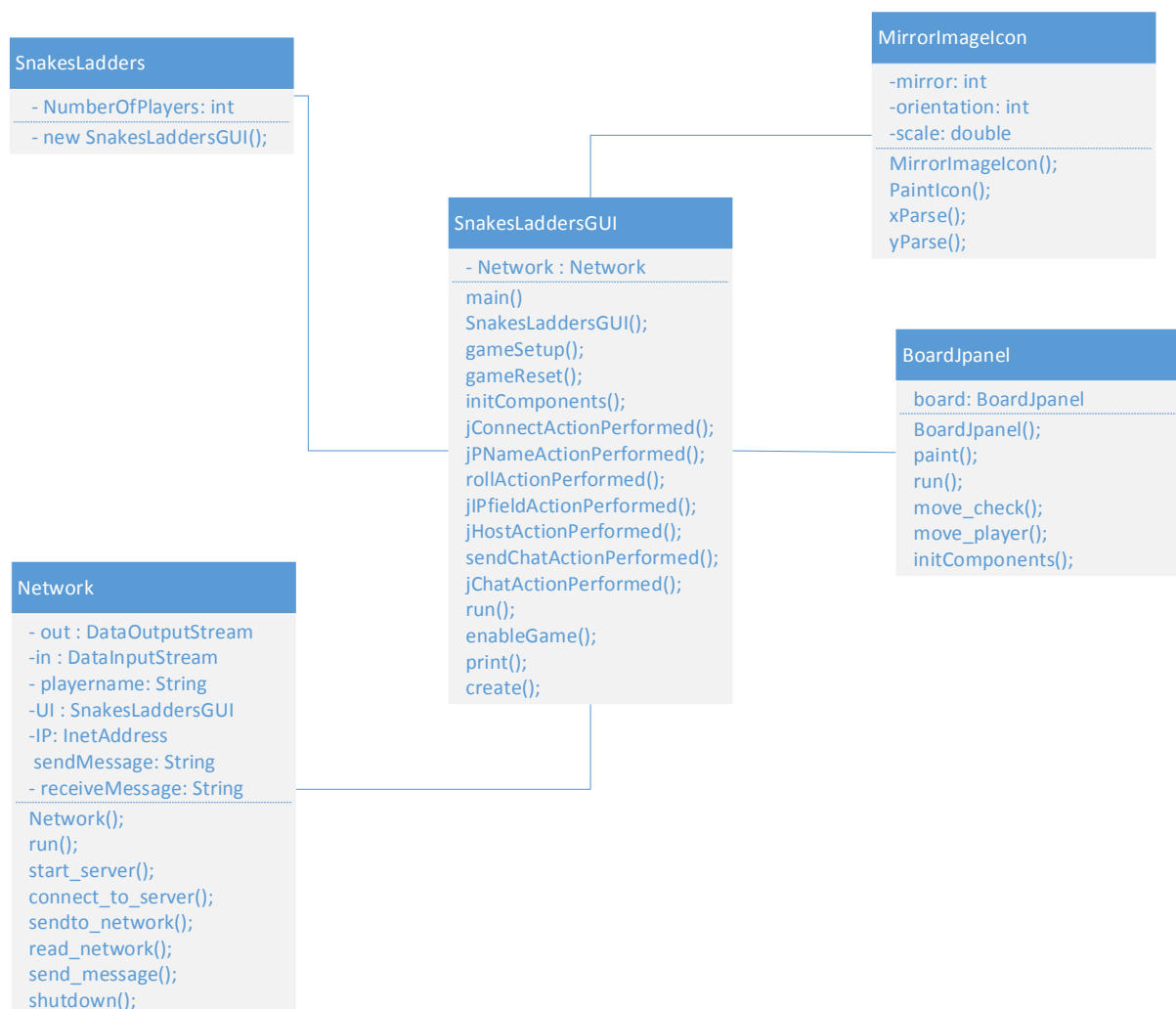
*End\_Game*

*Disconnect players/close sockets*

## CLASS RELATIONS (UML DIAGRAM)

A class diagram is used to show the existence of classes and their logical relations within the system [1]. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behaviour it has rather than detailing the methods for achieving operations. They are most useful in illustrating relationships between classes and interfaces. [2]

In the following figure, we can observe how our classes are associated with each other. The instance fields and methods of each class are shown on the diagram:



## NETWORK CLASS

The class maintains the input-output connection between users (server and multiple clients) during the whole game. The necessary functions have been implemented in order to make the class fully functional. That is an important class for multiplayer games.

### Functionality

The functionality of both Server and Client classes are combined within this class. For this type of multiplayer game, the network class is always active: the server accepts multiple clients and during the whole game this class provides two-way input-output communication between users.

### Methods & Constructors

- Constructor **Network(DataInputStream in, String player\_name)** - to be used from inside to create listener.

It has 2 parameters: one is for assigning the Input stream of type DataInputStream, and the second one is for the player name that is connecting, of type String. As it is constructor, it does not return anything.

- Constructor **Network(SnakesLaddersGUI User\_interface,String player\_name, InetAddress Server\_IP)** - initializes Network class in Client mode.

It has 3 parameters: 1.User interface object of class SnakesLaddersGUI (for using this class); 2. player name of type String; 3. IP of the server of type InetAddress.

As it is constructor, it does not return anything.

- Constructor **Network(SnakesLaddersGUI User\_interface,String player\_name)** - initializes Network class in Server mode.

It has 2 parameters: 1.User interface object of class SnakesLaddersGUI (for using this class); 2. player name of type String.

As it is constructor, it does not return anything.

- Method **run()** – either executes the method **start\_server()** or **connect\_to\_server()**;

It has no arguments and does not return anything.

- Method **start\_server()** – accepts connecting clients and creates new thread for each of them.

It has no arguments and does not return anything.

- Method **connect\_to\_server()** – connect to the server.

It has no arguments and does not return anything.

- Method **read\_network()** – reads the incoming messages or commands, through chat panel.

It has no arguments and does not return anything.

- Method **send\_message(String message)** – sends message that is assigned to the argument message.

It has 1 parameter called message of type String and does not return anything.

- Method **shutdown(String message)** - terminates the game.

It has 1 parameter of type String and does not return anything.

## **SNAKESLADDERSGUI CLASS**

This is the main class, where the main part of the program creates new object of Graphical User Interface. The class contains all the settings for GUI and game settings as well (as it part of GUI).

### **Functionality**

The functionality of the class is quite broad: it creates new GUI object, handles all the components of GUI, layouts of the components, logic of the game and game settings.

### **Methods**

- Construcor **SnakesLaddersGUI()** – constructs the players, initializing their locations.

It has no parameters and does not return anything.

- Method **gameSetup (int m, String n, int par1)** – creates first player array with the name user entered.

It has 3 parameters: string type for setting the name and the integer type for setting the IP address. The method does not return anything.

- Method **gameReset()** – resets the game.

It has no parameters and does not return anything.

- Method **initComponents()** - initilizes the graphical components (including buttons, different panels).

It has no parameters and does not return anything.

- Method **jConnectActionPerformed()** – for client to connect to the server when pressing the button Connect.

It has 1 parameter of type java.awt.event and does not return anything.

- Method **rollActionPerformed()** – for client to roll the dice to the server when pressing the button Roll.

It has 1 parameter of type `java.awt.event` and does not return anything.

- Method **jHostActionPerformed()** – for server to start hosting when pressing the button Host.

It has 1 parameter of type `java.awt.event` and does not return anything.

- Method **sendChatActionPerformed()** – for client to roll the dice to the server when pressing the button Roll.

It has 1 parameter of type `java.awt.event` and does not return anything.

- Method **enable\_game(Boolean setting)** – enables game by setting Boolean variable.

It has 1 parameter of type `Boolean` and does not return anything.

- Method **print(String message)** - prints messages into the chat panel on a function call.

It has 1 parameter of type `String` and does not return anything.

- Method **run()** - displays the GUI.

It has no parameters and does not return anything.

- Method **enable\_game(String Input)**

It takes a string received through the network and performs appropriate action.

## MIRRORIMAGEICON CLASS

The class deals with positioning the icons.

### Functionality

The class controls the correctness of icon movement along xy-coordinates.

### Methods

- Constructor **MirrorImageIcon(String filename, boolean left\_facing, int scaler)** – deals with the orientation of the icon.

It has 3 parameters: string, Boolean and integer types.

- Method **paintIcon()** – paints image at designated coordinates, and scales it to take 1/10 of the board in width and high.

It has 5 arguments: 1 of type Component(java.Component), 1 of type 2dGraphics and integer types for x, y coordinates and the coefficient of flipping.

- Method **xParse(int x)** – it parses the input coordinates and outputs coordinates corrected scaling and orientation of the icon.

It has 1 parameter of type integer and return value calculated.

- Method **yParse(int y)** – it parses the input coordinates and outputs coordinates corrected scaling and orientation of the icon.

It has 1 parameter of type integer and return value calculated.



## BOARDJPANEL CLASS

The class handles the board graphics setup.

### Functionality

The class controls the movement of the icons and sets the board.

### Methods

- Method **BoardJpanel()** – initialize the graphical picture for Board panel. It has no parameters and does not return anything.
- Method **paint(Graphics g)** – positions the board and the player icons on the board, paint them.

It has 1 parameter of type graphics and does not return anything.

- Method **position(Graphics2D g, int inputPosition, MirrorImageIcon peg )** – adjusts positions. It has 3 parameters for adjusting purposes and does not return anything.
- Method **run()** – moves the icon on the board. It has no parameters and does not return anything.
- Method **moveCheck(int Player\_number)** – checks if the player moved.

It has 1 parameter of type integer for specific player and does not return anything.

- Method **move\_player(int Player\_number, int Player\_rolled)** – moves player if rolled.

It has 2 parameters of type integer for specific player and the status of rolling, and does not return anything.

### DATA STRUCTURES:

Atomic Integer Array is used for determining location of player Icons.

Network class contains the functionality of both client and server; moreover there were implemented additional methods and constructors for the convenience. With this purpose, there were implemented 4 Network constructors, that were stored as an array of 4 constructors. This was necessary in order to be able to handle all these functionalities within one class. Additionally, the design of the class was improved by storing them into array.

## USER INTERFACE CONSIDERATIONS

The user interface part was analyzed thoroughly during the designing process, and significant amount of time was spent on the improvement of this part. Overall, we think that the User Interface was delivered with high quality:

- The necessary buttons were implemented and tested to be convenient for the user. The buttons go gray when they are inactive, e.g. after the player has already rolled.
- The application window is scalable to any resolution and the pieces are dynamically located and resized when the window is scaled. This allows the user to play in a window size perfect for their system and preference.
- The 4 board was re-designed.
- The chatting option is available between users.
- The necessary animations were designed, like rolling the dice.
- Feedbacks are provided when the user locates the mouse at a specific button.

All these considerations were made in order to approach to a perfect real game, meanwhile improving the design and making the user-friendly interface.

Overall, we think that the user interface was well done and the necessary considerations have been made.

## TESTING

### *Testing strategy:*

Testing is the one of the most important parts of the design process for any system and it was crucial for us as well.

Since we are handling many different components of the game that should be integrated together into one single game application, the testing process should have been carried very thoroughly and systematically. This saves lots of efforts and time, making the process much more efficient. We made extensive use of unit testing, where every method was tested with all inputs it is expected to process, separately so that the behaviours of them are clear to the developer; this allows the developer to change the codes flexibly at a later stage, during the integration. In fact, that is the simple strategy, but it is really useful to test and fully understand how the specific part works. On the other hand, the integration is not just the sum of its' parts, it is something where the parts must be consistent with each other. For this purpose, we started off with writing the pseudo-code, which allows us to see the whole picture of the system. Then, we divided the whole system into subparts. (Figure 2.). At this stage, we decided that one of us will be handling the networking part and another is in charge of game logic and user interface parts. Each of us worked on their parts, communicating where the integration is involved and testing the codes systematically.

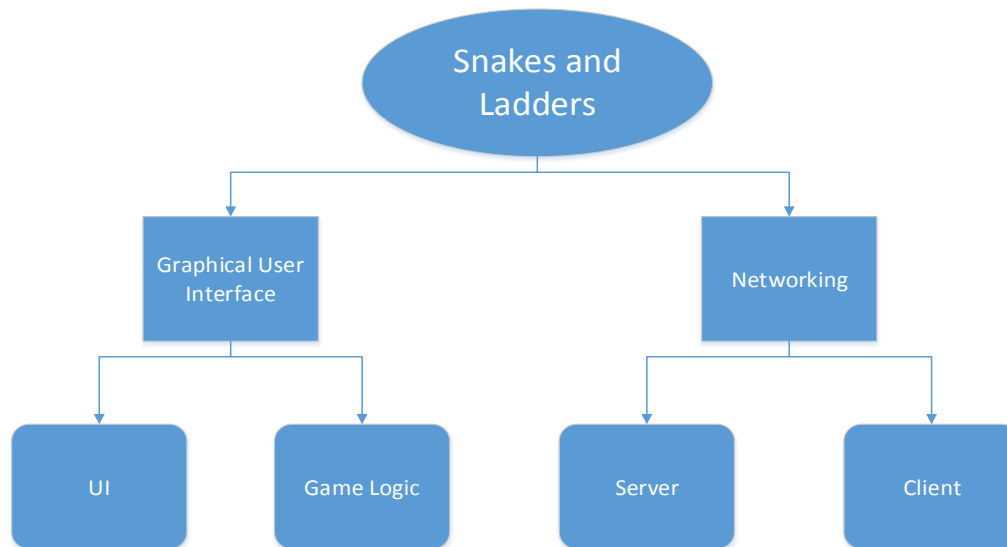


Figure 2

Thus, the main logic of our integration strategy was:

1. To clarify for ourselves what is required from us, trying to have better idea of the whole picture ( making plans, drawing diagrams, writing pseudo-codes were helpful here);
2. Divide the system into subparts and start coding bit by bit: as we already clarified what is needed, we tried to make the coding as flexible as possible. The understandings of the logics of the operations were important here;
3. The working bits should have been tested standalone, than integrated with the classes provided for us (or, with the relevant part). Test again and see if it works after integration. It is natural to have some errors, but if the developer understood the behavior of the program, he will probably be able to figure out the issue and correct as it is required.

More details about the testing and its results will be discussed in the next paragraph.

### Testing:

For practical reasons, only the most important tests are included in this document.

### **Network**

Aims: To create a multithreaded server for the game, that can accept multiple clients.

### **Method:**

1. The first step to be done was to write a simple console-based server-client program, where the server creates socket and waits the connection from the client and client connects to server and sends a simple string message “Hello”. Here is how the console output looked:

```

8
9 public TestClient(){//Constructor
10     try{
11         //Try to connect to server at IP address/port combination
12         Socket soc=new Socket("127.0.0.1",PORT);
13         //Now get the 'streams' - you use these to send message:
14         out=new PrintWriter(soc.getOutputStream(),true);
15         in=new BufferedReader(new InputStreamReader(soc.getInputStream()));
16         out.println("Hello!\n");
17         out.close();
18     }
19     catch(Exception e){
20         e.printStackTrace(); //Displays Error if things go wrong....
21     }
22 }
23
24 public static void main(String args[]){
25     new TestClient(); //Makes object (calls constructor)
26 }
27 }
28

```

Problems Javadoc Declaration Console

<terminated> TestServer [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Apr 2  
Hello!

Result: as expected, the server waits for connection and the client connects to the server successfully. Then, the client outputs “Hello” message to the server and we observe that on the console.

- Understand how it is better to do a multithreaded networking and attempt to do it. At this stage, we programmed the server so that it creates a new thread whenever it accepts new client. Thus, each new client is run on a new thread, where they should be able to output messages to servers. The logic on Server side was:

*Server accepts client*

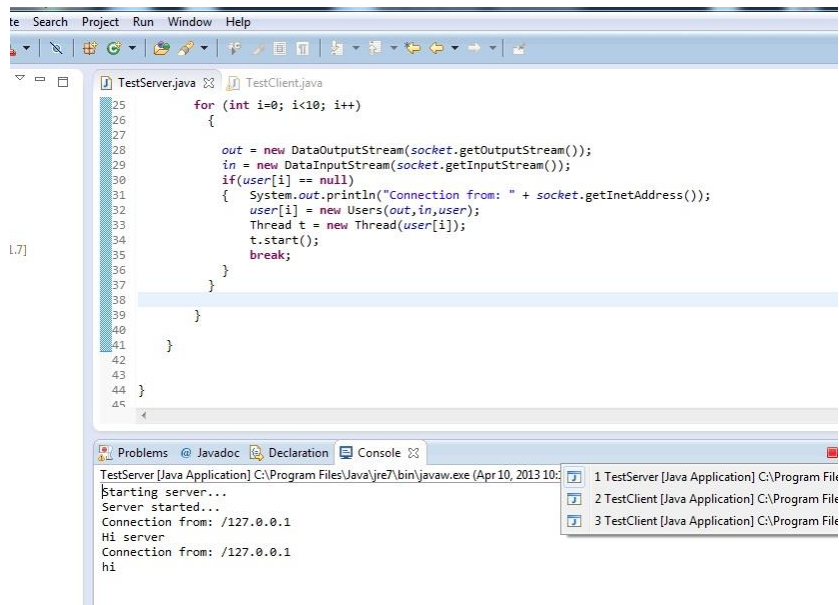
*Loops i from 0 to 5*

*If user with index i is connected*

*Creates thread for user using constructor*

Within the constructor mentioned in the pseudo-code above, the server accepts messages from the clients. Clients should be able to output messages to the server. All the messages are printed on the console.

Results: after first running there was an error: *for* loop was typed in the wrong place and we got one message several times, but that was a syntax error. After correcting this, the program worked as expected: multiple threads are created, which means multiple clients are connected to the server. Also, server inputs messages from the clients. Here is the screenshot for this part:

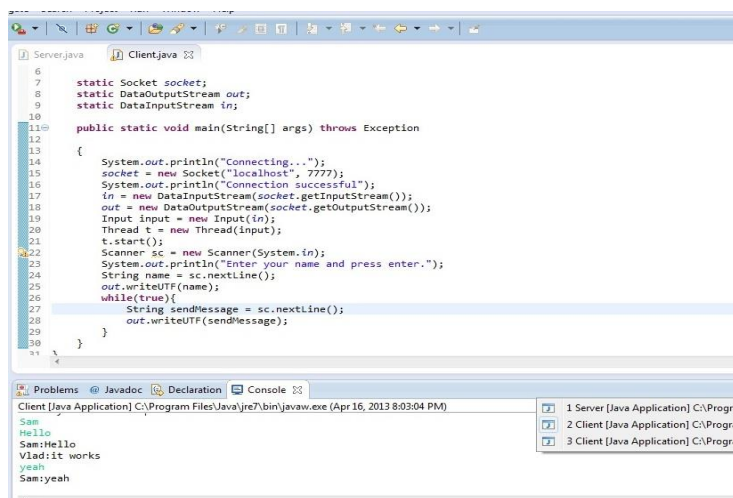


- At this stage the aim was to improve the last code a bit, so that the server always sends the incoming messages back to all the clients. Thus, the program should work as console-based chat.

The logic of sending the messages to all clients is similar to the previous method, in the manner that it loops through the players that exist on the network and server sends incoming messages to these clients:

*Input string message*  
*Loops i from 0 to 5*  
*If user with index i is connected*  
*Output the string message*

The following is the screenshot for this chat code:



Result: as predicted, after adding several lines of codes (pseudo-code provided above), our program behaves like a console based chat application.

## Board

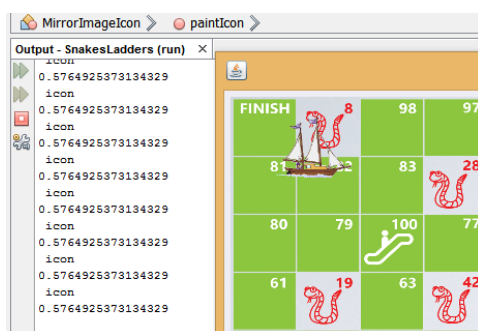
Aim: to create a good user interface.

The user interface and the layout within the applet, the layout of panels were tuned to have a good and comfortable interface.

Here is how it looks:



The size of the window is also designed to be re-scalable, which makes it comfortable for the user. The sizes of all the graphical objects change accordingly, and stay on their positions. The screenshot is provided here:

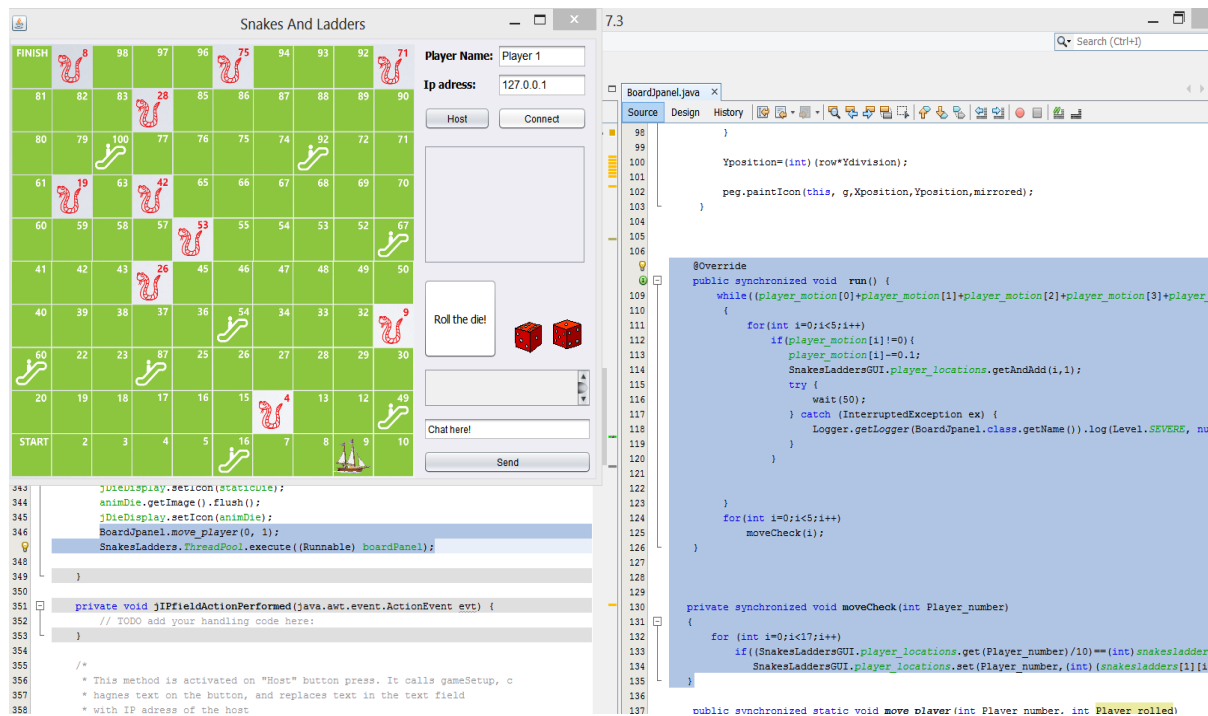


## Testing the movement of graphical objects on the board

**Aims:** Moving the objects (like ship in this example) on the board when the player rolls.

For this part, the graphical objects of each player are assigned to the method that checks the movement of the player, which was caused by another method that rolls the dice. Thus, the object moves following the roll action of the player.

We can observe this scenario from the following screenshot:



**Results:** as we can see, the ship objects on the board changes its position with the change of the location in the program, the aim was achieved successfully.

## Integration: Network + GUI

**Aims:** At this stage, we were to start integrating the multithreaded networking code we have with the *SnakesLaddersGUI* class. The task is quite tricky.

Changes that the integration process might involve:

- Changing the thread controlling code caused by changing some codes in client-server (not the logic but syntaxes);
- Instead of console, the inputs and outputs should be printed to the panel on the applet;
- Additional functions in GUI class and some modifications.

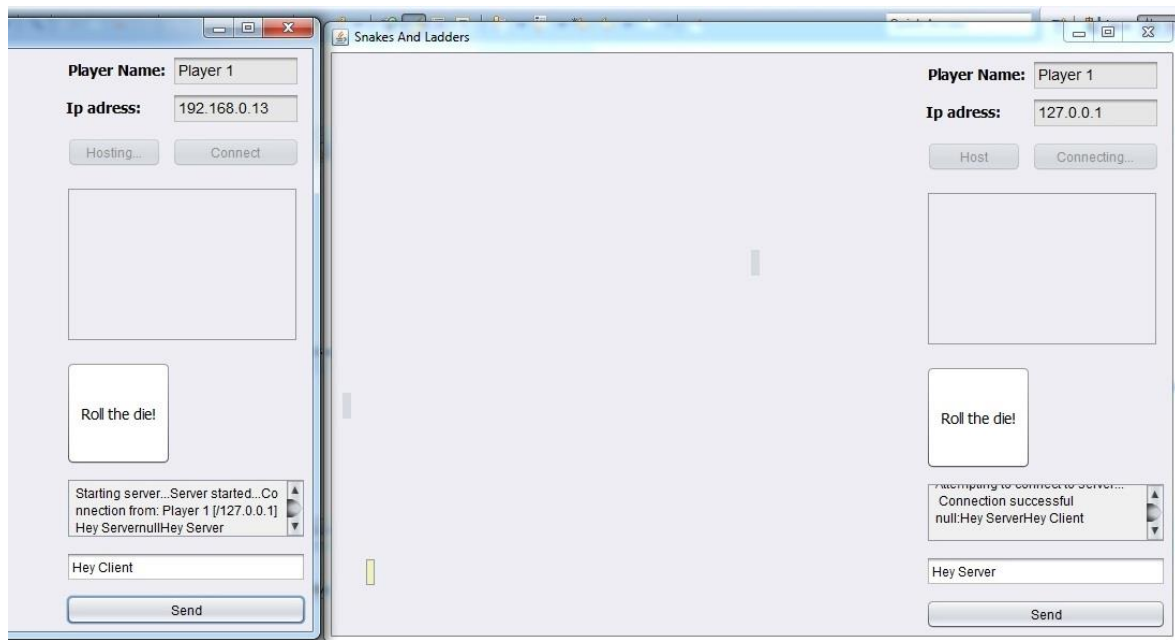
The changes mentioned have been made:

Thread controlling method changed in the sense that the method called "*ThreadPool.Execute()*" is used now, and it works in the same manner as the normal thread start; also, some parameters like *SnakesLadder UI* were added where necessary and the printing messages now goes to the UI panel using the method. Some modifications to the GUI class was made in order to be able to use the same button called 'SEND' to run 2-way communication.

### **Results:**

The actions above were applied and we run the program to test. Server-client connections were successful within GUI, however the input-output messages were not printing on the panel as it was expected. The IO-paths were investigated and we observed that the messages are transmitted in both direction, but the problem was that the UI declarations were not properly defined. After fixing this problem, we successfully could get messages on the panel of the applet.

The screenshot for successful result is as follows:





The program was tested up to maximum number of connection with all players chatting and playing the game.

Following is a screenshot:



5 clients simultaneously connected to the server, and the testing was largely successful.

Only error revealed is that the server displays the rolling player as null due to an error in assignment.

## **CONCLUSION:**

Overall, the project gave us a lot of practical experience in terms of understanding the concepts of object-oriented programming, particularly programming on Java language and making a good design. The project implies different aspects of programming such as networking programming, multithreading and Graphical User Interface, these are important to know for the programmer, and these fundamental concepts should be understood well. By doing this project, our understanding on these concepts improved significantly; moreover, we gained a valuable practical experience on programming in general. Another useful point here is that the integration process was involved as well, and this helped us to improve our skills in its' own way. By writing the report and documentation, we learn and keep improving the document writing skills.

The specification requirements were met, and the program works as predicted and perfectly.

## **FUTURE IMPROVEMENTS**

Networking is done in a separate thread for each client. At State 0, server awaits for connection. The first client to connect is granted status of an admin. He may determine number of players that will play the game. The Sever starts the game when number of players is reached, or when timeout is reached.

Networking must implement following exceptions: Corrupt Packets, connection timeout, player lost.

Validation of packets must be implemented and corrupted or lost packets should be requested again.

## **REFERENCES:**

[1] Booch G., Maksimchuk R., Engel M.W., Young B., Conallen J. and Houston K., (2007), Object-Oriented Analysis and Design with Applications (3rd Edition), Boston: Pearson Education.

[2] [http://www.sparxsystems.com/resources/uml2\\_tutorial/uml2\\_classdiagram.html](http://www.sparxsystems.com/resources/uml2_tutorial/uml2_classdiagram.html).

- EE2E1 Lecture Notes.

- EE2E1 Assignment Handout.

Gif images included with the game are included to demonstrate game functionality and aer distributed under GNU licence.