Java assignment report

Snakes and Ladders Game

Student name: Zhanibek Bakin

Student ID: 1502403

Submission date: 30.01.2017

Contents

Introduction

Objectives

The main objective of this assignment is to design and program a multiplayer Snakes and Ladders game, which can be played through local network connection(LAN). The idea is the create two separate programs, one for server, another for client. Then connect them using networking techniques introduced by Java.

Specification

The game needs to be designed such that 2, 3 or 4 players can play it simultaneously. This networking game requires to have a server, which handles connection and game logic. Also, it requires a client, which connects to server and includes the Graphical User Interface of the game.

First, the server is started. As it needs players to connect, it will be in a waiting mode. That is, looping through statements of the initial phase (connection phase). When clients are connected, the server creates a new Thread for each player. Maximum of 4 threads can run concurrently. Next phase is the beginning of the game. If the required number of players is connected, the server can transfer to the next stage by pressing the start button. Afterwards, the first player can press the rolling button. Other players should wait for their turn. Server controls the turn of the players and sends specific message for client which enables the rolling button. This button activates the method, which requests the next position on the board. In effect of this, server generates random dice number, calculates the next position and sends it to all clients. The turn is passed to the next player. The process continues until one of the players gets the position of 100 on the board. Then all connections are closed and game is over.

Design

Design Considerations

This Snakes and Ladders game is written in Java language. Java is the object-oriented programming language with high level features. One of the huge advantages of Java is that it is portable to almost any computer system. It is easy to program, automatically allocates the memory and collects garbage. Java has a multithreading capability which will allow the server to run several players concurrently. Lastly, networking feature is inherited, which makes it easy to create networking applications.

Constraints

Some disadvantages of Java are slow performance and high memory consumption compared to C and C++. However, for current application, these disadvantages are negligible.

Class for the graphical user interface was provided. It holds functions such as ‘setNumberOfPlayers()’ and ‘setPosition()’. First sets the number of players on the game. Second allows to manipulate the position of each player by passing the integer values of player and position from 0 to 100 of the board.

Class Responsibility Collaborator(CRC) cards

There are 4 classes in this game. These are shown on the following tables(cards).

Server	
Connect clients Create Player objects Send messages Display messages Throw dice Follow Snakes and Ladders logic Listen to button events Start the game Send Position to client	Player

Client	
Connect to server Process messages Send messages Display messages Set GUI Send roll requests	SnakesAndLaddersGUI Player Server

Player	
Get i/o streams from clients Receive player numbers Track turn of clients Receive requests from client Receive and process messages	Client Server

SnakesAndLaddersGUI	
Set number of players Set players' positions	Client

Figure 1. CRC cards

Top cell contains the class name. Left column includes Responsibilities of that class. Right column includes the Collaborators of that class. Collaborators are other classes which are needed to fulfill the functionalities listed in Responsibilities column.

Use case view

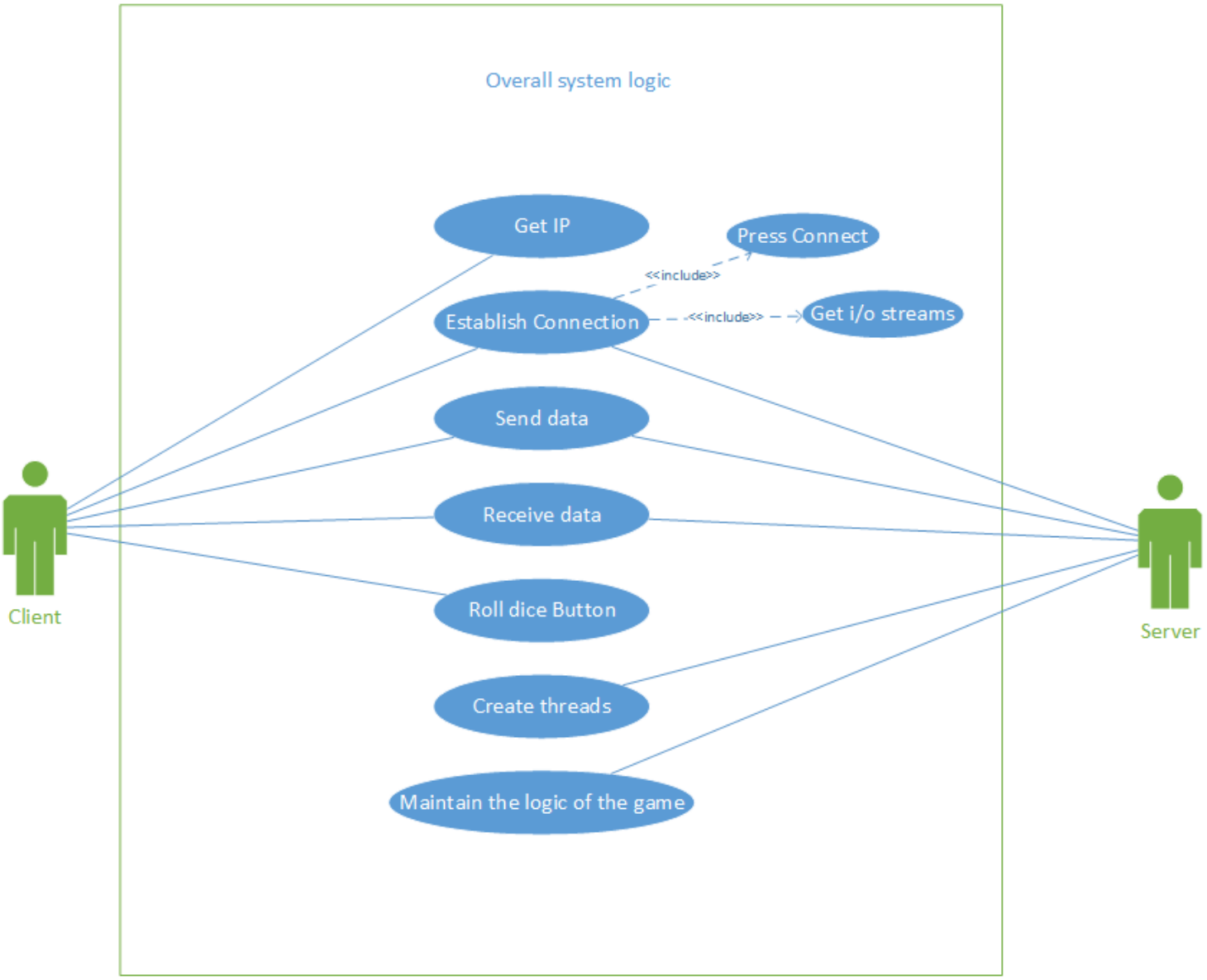

Figure 2. Use-case diagram of the system

Use case descriptions

This section describes the use-case diagram from actor's point of view. Use-cases are given in *italics* and actors are underlined.

Server: User starts server application. Server waits for clients to *establish connection* and *create threads* for each of the connected clients. If two or more clients are connected, server *sends* "Can roll" message to the first client. Then server expects to *receive* "Rolling the dice" message.

When the message is obtained, server invokes *rollLogic*() method, which calculates new position and *maintains the logic of the game*.

Client: User starts the client application. The Client obtains the IP address from the user. After pressing *Connect* button, client tries to *establish connection*. If the IP address is correct, the client *receives* a player number from the server. Then, it waits for message "Can roll", which sets the turn of the client. Hence, it can press *Roll dice* button. By pressing the button, client *sends* message "Rolling the dice" to server.

Interaction diagrams

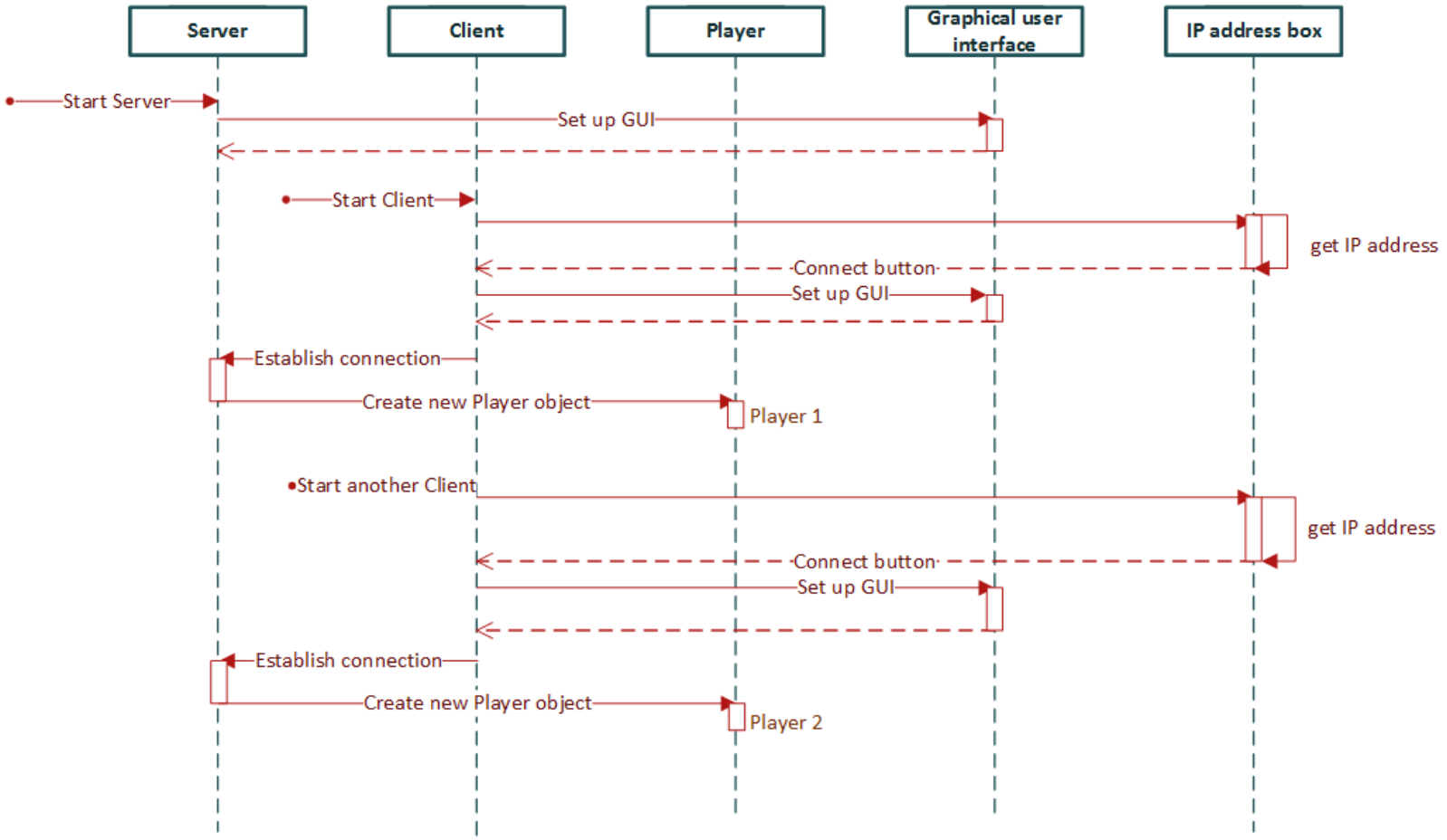

Figure 3. Sequence diagram for creating players and threads

There are 2 types of interaction diagrams – sequence and collaboration diagrams. Both show how the objects interact with each other. The Sequence diagram has vertical lines from each object, which represent timeline from top to bottom. Objects are placed horizontally. In **Figure 3** those objects are classes, except IP address box. Objects could be any instances, which affect the other objects in any way.

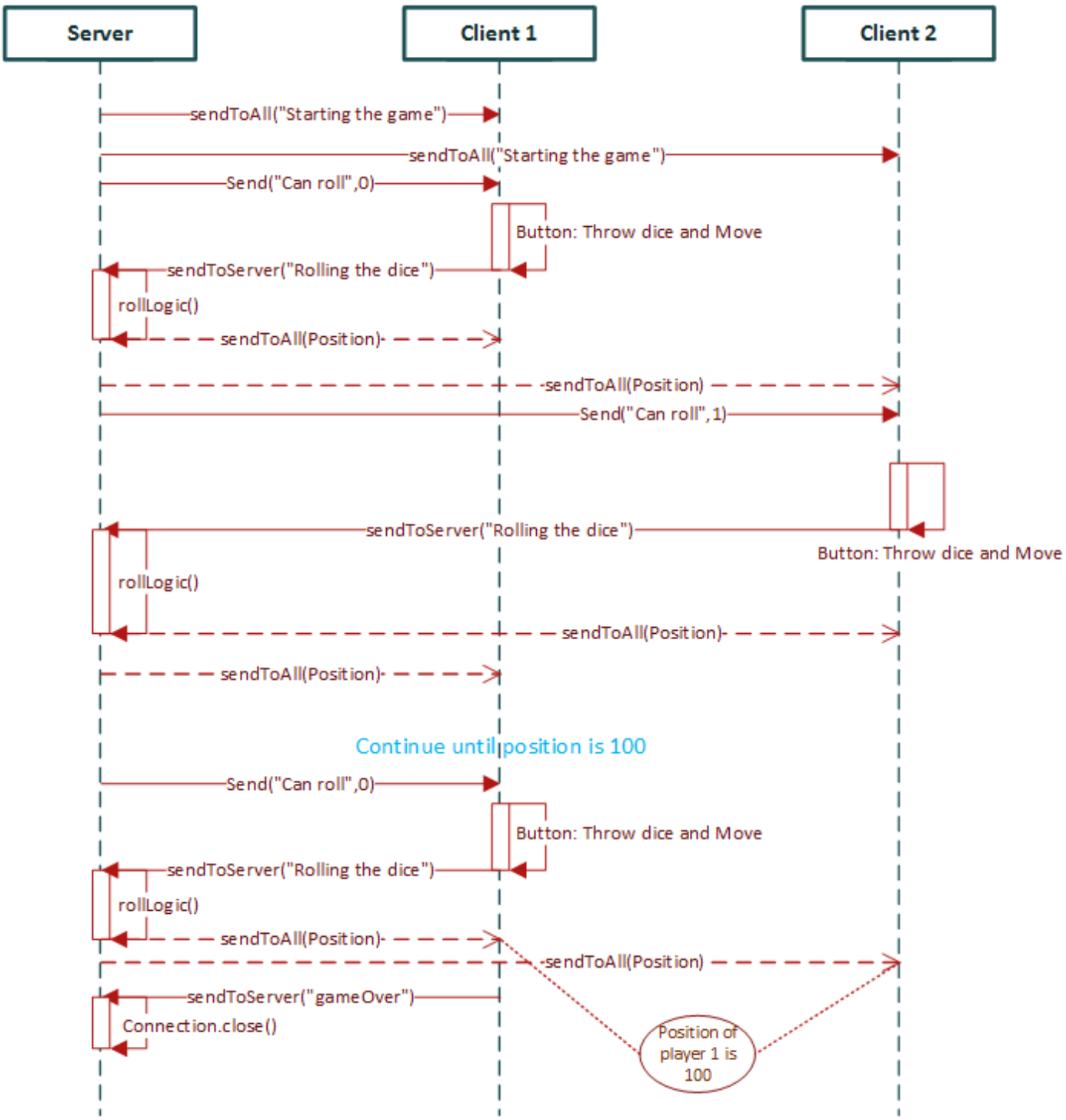

Figure 4. Sequence diagram for the game

Figure 4 illustrates the whole game process which is played by 2 users.

It can be seen, that Server initiates the sequence of events by sending “Starting the game” message to all clients. Additionally, it is responsible for tracking the turn of the players. Therefore, server sends “Can roll” message to the first player. Thereby, player 1 requests the next position. It outputs the “Rolling the dice” message when user presses the button. Server generates random number, adds it to current position of that player, performs logic of ladders and snaked and sends this position to all clients. The turn is passed to the next player. This process repeats until client receive 100. Lastly, client sends “Game Over” message and all connections are terminated.

Collaboration diagrams show the objects and their interactions which are numbered in order of execution.

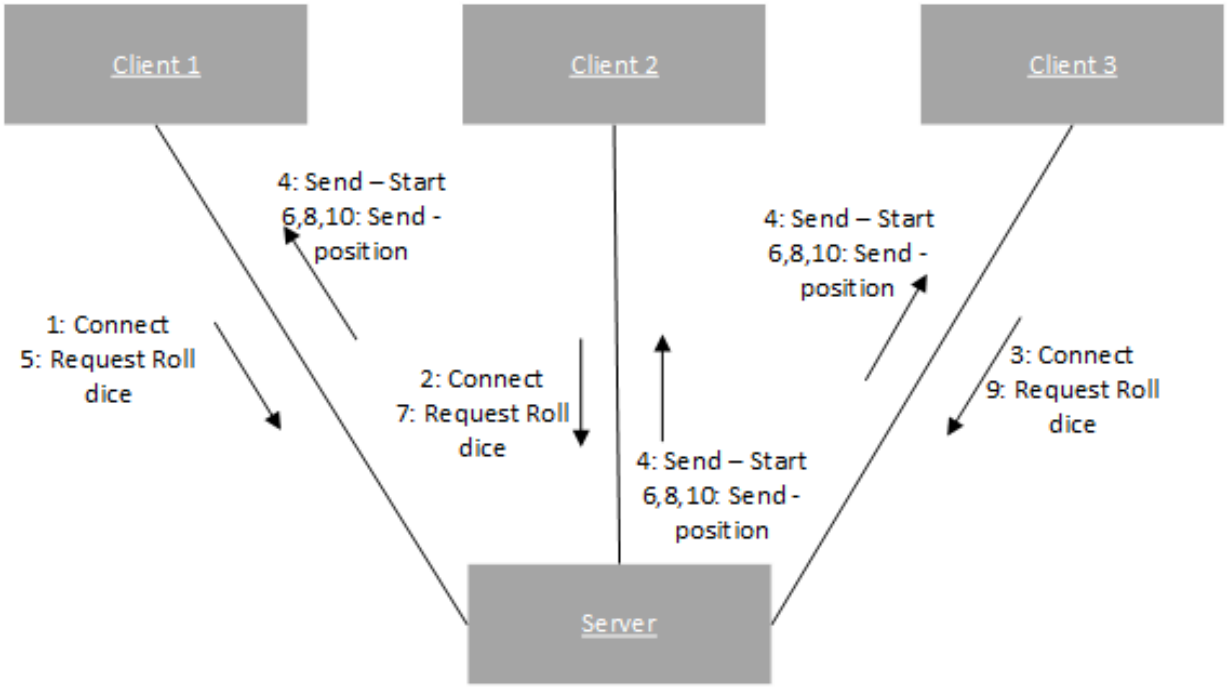

Figure 5. Collaboration diagram – first steps

Figure 5 shows the interaction between server and 3 clients. The sequence of events can be traced by following the numbers. Due to the cyclical nature of the game, **Figure 5** only illustrates 10 steps. Steps 6,8 and 10 are execute concurrently.

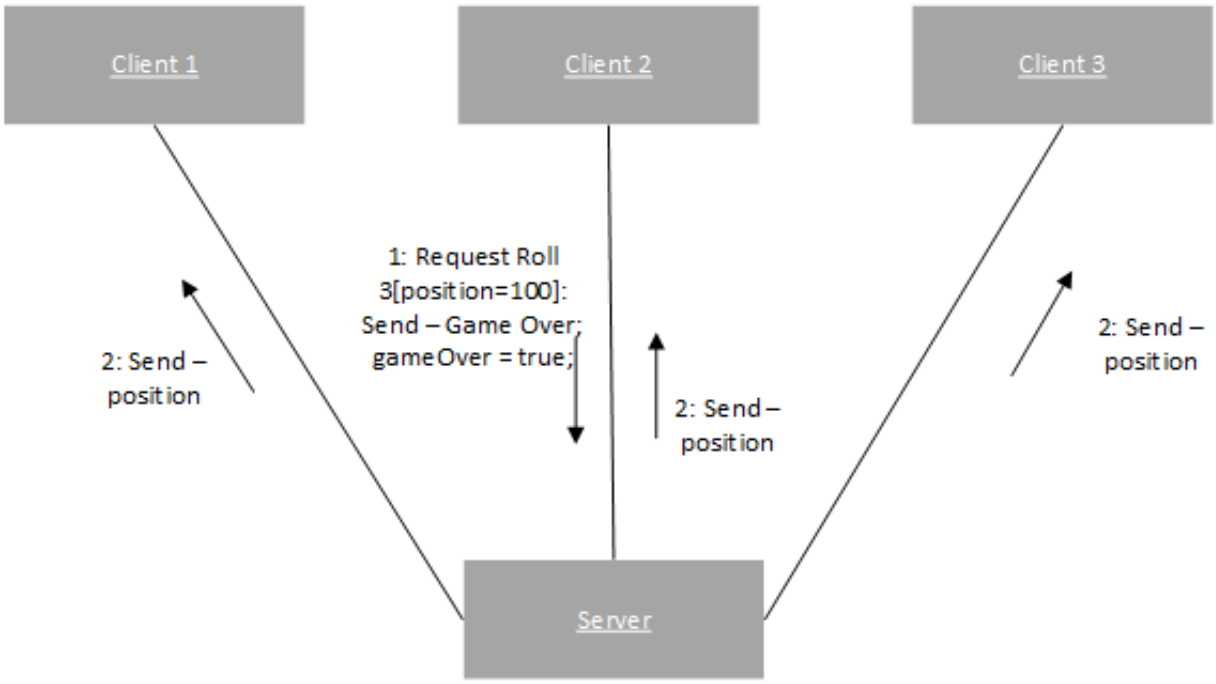

Figure 6. Collaboration diagram – final step

However, the final step is presented in **Figure 6**. When the position is 100, server sends position to all clients. In this case, Player 2 come to finish first and sends message “Game Over” to the server.

State-chart diagram

This kind of diagrams illustrate states transitions between states. **Figure 7** illustrates how the game runs from server perspective.

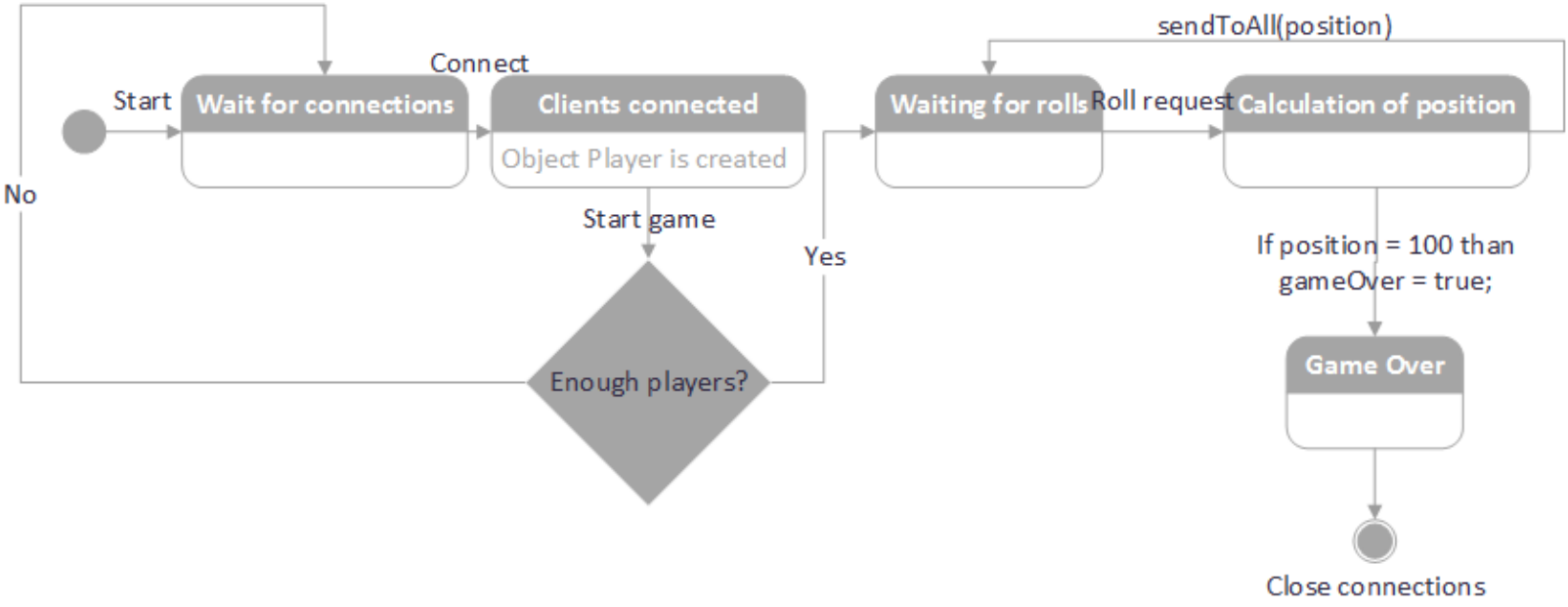

Figure 7. Server State-chart diagram

First the user starts server. Server waits for connections. Next, by clicking the start button, server checks if the game can be started. If no, it continues to listen for incoming players. Otherwise, the game is started. Now server waits for the client to request the roll dice operation. Next state is calculation of position. From this state, server directly sends the position to all players. Then it checks if the position is 100. If this is the case, the game is finished and connections are closed.

Program Structure

Class relationships

This section provides reader with 2 class diagrams – server class diagram and client class diagram. Furthermore, it explains classes, their relationships and methods.

Figure 8. Class diagram of Server and Player

There are two relationships between these classes. First is the inner class relationship (line and inscribed in circle plus). Player class is created inside the Server. Hence, it is nested class. Next relationship is association. That is, Server uses instances of class Player. It can use as many as needed(0..*).

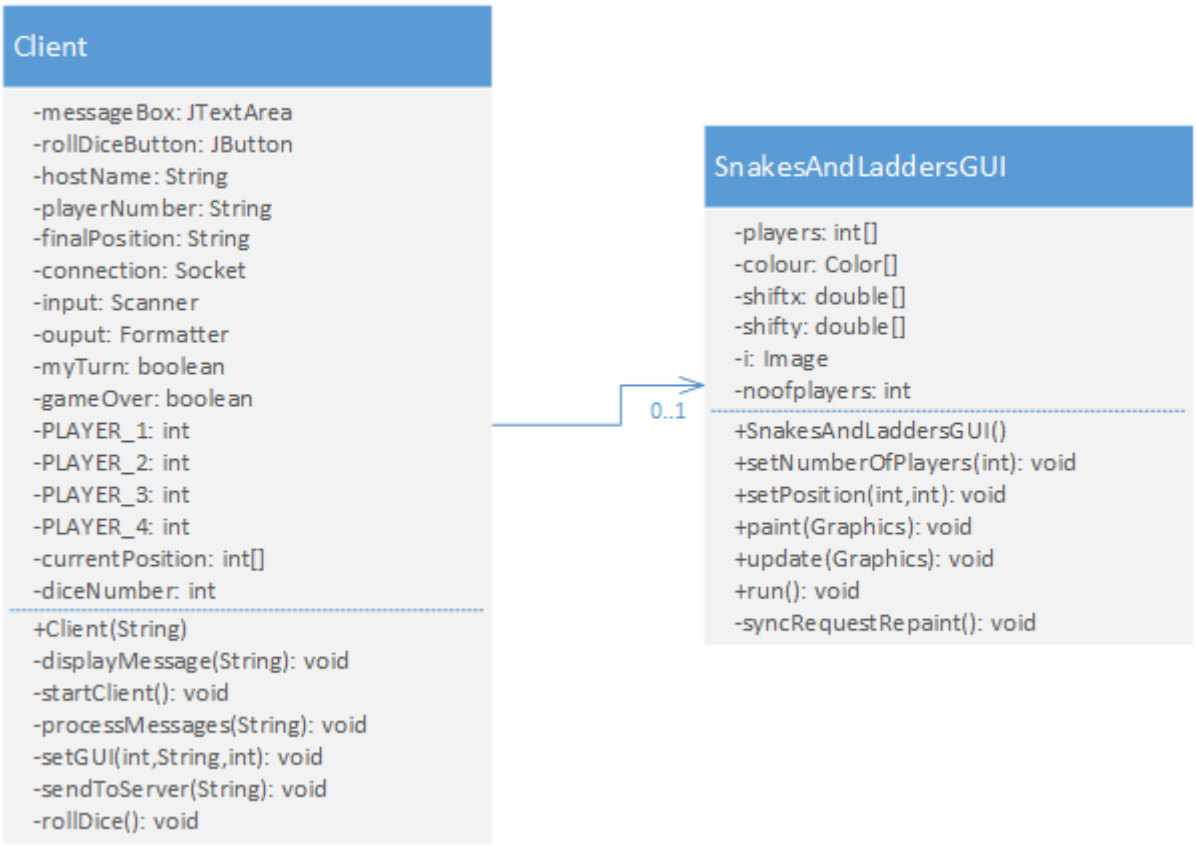

Figure 9. Class diagram of Client and GUI

The client and GUI has association relationship. Client can have maximum of one instance of SnakesAndLaddersGUI(0..1).

Design specification of Class methods

Server Class comprises of 6 methods including constructor. First method, *displayMessage*(), passes incoming messages and sets them into textArea. *sendToAll*() is used to output the messages to all connected clients. Next method, *send*(), passes both message and player number to send a message to one client at a time. If a specific message is received from client, rollLogic() method is invoked. It generates random number and sends it to all clients. Inside this method, snakedLaddersLogic() is called. This last method gets value of current position and returns a different value depending on the position of snakes and ladders on the board.

Player class only includes its constructor and run() methods. Constructor is responsible for establishing the connection with clients using input and output streams. When a new thread is started, run() method is invoked. It controls the execution of thread.

As for the client, it consists of 7 methods. Client() – receives the IP address from the user and builds a user interface for client class. displayMessage() method, similar to server's method, prints the text of the message into messageBox. startClient() method uses IP address from the user and tries to connect to server. If the connection is successful, then client waits for incoming

messages. Once the input stream detects message, method processMessage() is called. Depending on the message, client performs necessary operations. Method setGUI() gets 3 parameters – player number, position and dice number. When this method is invoked, it processes the movement of the tokens on the board. sendToServer() outputs strings to server. rollDice() sends "Rolling the dice" message to server when the roll button is pressed if it is his turn.

Two main methods of SnakesAndLaddersGUI class are setNumberOfPlayers() and setPosition() were explained in Constrains section.

Implementation and testing

The implementation of the game is very straightforward. The user starts the application by running test classes inside Eclipse. Test classes are ServerTest and ClientTest. They just create a new application of the Server and Client classes in their main functions. First, Server is started.

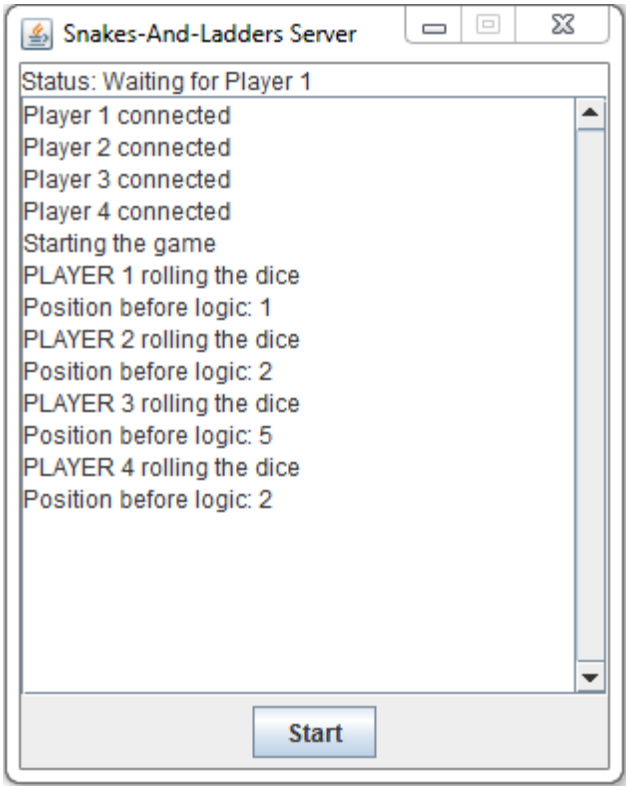

Figure 10. Server GUI

Server shows exactly what is happening throughout the execution. Status text area indicates the state of the game. Message box includes appended messages which can be traced at any time by scrolling the pane.

Figure 11. Client GUI

Thereafter, Client class is started in a same manner. When IP address is entered into the small dialog window, Client window will appear(**Figure 11**). It includes 3 main attributes - game board, message box and the button. The GUI is very simple and straightforward. The user will easily track the position, see the colour of his token, and identify his turn.

Testing was accomplished by means of debugging mode. To identify any issues in the design, breakpoint was placed on uncertain parts of the code. Hence, when running the application from the debugging mode, the execution is paused at these locations. Then, the programmer can check every line and see if it was executed or ignored.

Discussion

During the development of the game, several issues were encountered. One of them is trying to the send one message to all players. When client requested the rolling of the dice, the output.format() method was sending the message only to one client, who requested the message. This problem was solved by creating a separate method, which loops through all clients and sends each of them a message.

Data is sent to all clients using the following statement:

```
for (int i = 0; i < 4; i++)
{
            players[i].output.format(message); // send message to player[i]
            players[i].output.flush(); // flush the output
} // end for
```

where “player[i]” is the object of class Player, “i” is the number from 0 to 3(for 4 players).

Second issue was misunderstanding of how the methods of Scanner and Formatter classes work.

```
if(input.hasNextLine()) // waits until the new message is received
```

This line of code is the blocking statement. Therefore, it does not allow to continue execution if there is no message at the input. The next line might be confusing if the rules of sending and receiving data using Formatter and Scanner are unknown.

```
sendToAll("Position of Player "+ player + "\n" + position[PLAYER_1] + "\n" +
diceNumber+ "\n");
```

Due to the fact that reading of messages is accomplished with input.nextLine() method, the messages should be sent line by line. This line of code sends 3 messages. First is “Position of Player 1”(variable `player` is 1 is this case). Next message is the position of player 1. Hence, dice number is the last message.

In the end, the writer of the report came to conclusion to first thoroughly understand the theory and concepts of the build in classes and methods.

Conclusions

Given report is intended to qualified reader, who knows the object-oriented programming concepts. In future, the report could be explained in more detail, so that inexperienced programmer could analyse it as well. It terms of the design, additional functionality could be added in future. For instance, requesting the name of user and displaying it to all clients.

To sum up, the game was successfully written in Java language, tested on different computer platforms and all requirements were met. Connections were established and the game followed

the correct logic. In terms of UML, Use case, interaction, class and State-chart diagrams were included. Illustrations are clear. Explanations are clear and concise.

Code listing

Server

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.*;
import java.io.*;
import java.util.*;

// server class for game Snakes and Ladders
public class Server extends JFrame {

	private Player[] players; // array of players of instance Player
	private JPanel panel; // panel to hold button
	private JTextArea textArea, statusArea; // for displaying messages
	private JButton startButton; // start the game button
	private JScrollPane scroll; // scroll Pane needed to follow the textArea
	private ServerSocket server; // server socket to connect with players
	private String clientMessage; // receive messages from clients
	private final static int PLAYER_1 =  0; // constant for first player
	private final static int PLAYER_2 =  1; // constant for second player
	private final static int PLAYER_3 =  2; // constant for third player
	private final static int PLAYER_4 =  3; // constant for fourth player
	private int playersConnected = 0; // number of players connected
	private int diceNumber = 0; // dice number
	private int[] position; // position of players
	private boolean canStart = false; // Start the Game
	private boolean Turn_1 = false; // Turn of Player 1
	private boolean Turn_2 = false; // Turn of Player 2
	private boolean Turn_3 = false; // Turn of Player 3
	private boolean Turn_4 = false; //k Turn of Player 4
	private boolean gameOver = false; // game over


	// set up GUI for server
	public Server()
	{
		super("Snakes-And-Ladders Server"); // set title of window
		position = new int[4]; // initialising array position
		players = new Player[4]; // create array of players
		panel = new JPanel(); // create JPanel
		startButton = new JButton("Start"); // create start button
		textArea = new JTextArea(); // create JTextArea to output processing
messages
		statusArea = new JTextArea(); // create JTextArea to display
status(state) of the game

		setSize(300,380); // set size of window
```

```
            getContentPane().add(textArea, BorderLayout.CENTER); // add textArea
to CENTER of the fame
            getContentPane().add(statusArea, BorderLayout.NORTH); // add status
Area to NORTH of the frame
            getContentPane().add(panel, BorderLayout.SOUTH); // add panel to
SOUTH of the frame
            panel.add(startButton, BorderLayout.SOUTH); // add start button to
JFrame
            scroll = new JScrollPane(textArea,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS, JScrollPane.HORIZONTAL_SCROLLBAR_NEVER); //
create scroll pane
            getContentPane().add(scroll); // add scrollPane to the frame

            startButton.addActionListener(new ActionListener() { // add action
listener to listen to button clicks
                  public void actionPerformed(ActionEvent e) {
                        if (playersConnected > 1) // need 2 or more players to
start the game
                        {
                              if(!canStart) // do this only once
                              {
                                    displayMessage("Starting the game\n"); //
display message
                                    try
                                    {
                                          sendToAll("Starting the game\n"); //
send message to all players
                                    } // end try
                                    catch (Exception exception) // catch any
exception if error occurs
                                    {
                                          exception.printStackTrace(); //
prints throwable to standard error stream
                                    } // end catch
                                    Turn_1 = true; // set Turn_1 to true
                                    statusArea.setText("Status: Waiting for
Player 1"); // display the message in the statusArea
                              }
                              canStart = true; // set canStart to true
                        } // end if
                        else
                        {
                              displayMessage("Not enough players to start\n");
// display message
                        }
                  } // end method actionPerformed
            }); // close action listener

            statusArea.setText("Status: Server awaiting connections"); // set
status area
            setResizable(false); // set non resizable frame
            textArea.setEditable(false); // set non-editable textArea
            statusArea.setEditable(false); // set non-editable statusArea
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // exit application
when closing window
```

```
		setVisible(true); // show window
		try
		{
			server = new ServerSocket(15024,4); // set up server socket
		} // end try
		catch(IOException ioException) // catch i/o exception
		{
			ioException.printStackTrace(); // prints throwable to standard
error stream
			System.exit(1); // terminates running Java Virtual Machine
		} // end catch

		// creates player instances of each connection
		for(int i = 0; i < players.length; i++)
		{
			try // wait for connection, create Player
			{
				players[i] = new Player(server.accept(),i); // create
new Player objects when accepting connections
				new Thread(players[i]).start(); // execute player
runnable
				playersConnected++; // count connected players
			} // end try
			catch(IOException ioException) // catch i/o exception
			{
				ioException.printStackTrace(); // prints throwable to
standard error stream
				System.exit(1); // terminates running Java Virtual
Machine
			} // end catch
		} // end for
	} // end constructor


	private void displayMessage(final String message) // method to display
messages on textArea
	{
		// display message from event-dispatch thread of execution
		SwingUtilities.invokeLater(
			new Runnable()
			{
				public void run() // updates textArea
				{
					textArea.append(message); // add message
				} // end method run
			} // end inner class
		); // end call to swingUtilities
	} // end method displayMessage

	private void sendToAll(String message) // send message to all clients
	{
		for (int i = 0; i < 4; i++){
			players[i].output.format(message); // send message to player[i]
			players[i].output.flush(); // flush the output
		} // end for
```

```
	} // end method sendToAll

	private void send(String message, int playerNumber) // method to send a
particular player
	{
		players[playerNumber].output.format(message); // send message to
specific player
		players[playerNumber].output.flush(); // flush the output
	} // end method send

	// private inner class Player that manages each Player
	private class Player implements Runnable
	{
		private int playerNumber; // player number
		private Socket connection; // connection to client
		private Scanner input; // input from client
		private Formatter output; // output to client
		private int rollingPlayer; // player who rolls
		public Player(Socket socket, int number) // Player constructor
		{
			playerNumber = number + 1; // number starts with 0, First
player is Player 1
			connection = socket; // represents connection with client
			try // get streams from sockets
			{
				input = new Scanner(connection.getInputStream()); // get
input stream
				output = new Formatter(connection.getOutputStream()); //
get output stream
			} // end try
			catch(IOException ioException) // catch i/o exception
			{
				ioException.printStackTrace(); // prints throwable to
standard error stream
				System.exit(1); // terminates running Java Virtual
Machine
			} // end catch
		} // end Player constructor

		// control thread's execution
		public void run()
		{
			boolean sendFlag = false; // set flag to false
				while(true)
				{	// sends the playerNumber only once
					if (!sendFlag){
						displayMessage("Player " + playerNumber + "
connected\n"); // display message
						output.format("Player number is\n" +
playerNumber +"\n"); // send player's number
						output.flush(); // flush the output
						sendFlag = true; // set flag to true
					}//end if
				try
				{
```

```
                                if(!canStart) // check if the game can be started
                                {
                                        try {
                                                Thread.sleep(1000); // sleep for 1
second
                                        } catch (InterruptedException e) { // catch
interrupted exception
                                                e.printStackTrace(); // prints
throwable to standard error stream
                                        }//end catch
                                        continue; // go back to the beginning of
while loop
                                } // end if

                                        // send "can roll" message a particular
client depending on which turn it is
                                        if(Turn_1)
                                        {
                                                send("Can roll\n",PLAYER_1); // send
to player 1
                                                Turn_1 = false; // set turn 1 to
false
                                        } //end if
                                        else if (Turn_2)
                                        {
                                                send("Can roll\n",PLAYER_2); // send
to player 2
                                                Turn_2 = false; // set turn 2 to
false
                                        } // end else if
                                        else if (Turn_3)
                                        {
                                                send("Can roll\n",PLAYER_3); // send
to player 3
                                                Turn_3 = false; // set turn 3 to
false
                                        } // end else if
                                        else if (Turn_4)
                                        {
                                                send("Can roll\n",PLAYER_4); // send
to player 4
                                                Turn_4 = false; // set turn 4 to
false
                                        } // end else if

                                if(input.hasNextLine()) // waits until the new
message is received
                                        clientMessage = input.nextLine(); // get
message
                                if(clientMessage.equals("Rolling the dice"))
                                {
                                        rollingPlayer = input.nextInt(); // get the
next line of the message, which is the player number who is rolling
                                        rollLogic(rollingPlayer); // process the
rolling of the dice and send the position to the client
```

```
                                }
                                else if(clientMessage.equals("Game Over"))
                                {
                                        gameOver = true; // set gameOver to true
                                }
                        } // end try
                        catch(Exception e) // catch any exception
                        {
                                e.printStackTrace(); // prints throwable to
standard error stream
                        } // end catch
                        finally
                        {
                                try // try to close connection if the game is over
                                {
                                        if(gameOver)
                                        {
                                                connection.close(); // close
connection to client
                                                displayMessage("Player " +
playerNumber + " is disconnected\n"); // display message
                                                break;
                                        } // end if
                                } // end try
                                catch(IOException ioe) // catch i/o exception
                                {
                                        ioe.printStackTrace(); // prints throwable
to standard error stream
                                        System.exit(1);  // terminates running Java
Virtual Machine
                                } // end catch
                        } // end finally
                } // end while
        } // run
    } // end inner class Player
        private void rollLogic(int player)
        {
                switch(player) // cases for each player
                {
                case 1:
                                Turn_2 = true; // set turn 2 to true
                                statusArea.setText("Status: Waiting for Player
2"); // prints message in status area
                                diceNumber = (int)(Math.random()*6) + 1; //
generate random number from 1 to 6
                                displayMessage("PLAYER 1 rolling the dice\n"); //
display message
                                position[PLAYER_1] += diceNumber; // update
position
                                if(position[PLAYER_1] > 100)
                                {
                                        position[PLAYER_1] -= diceNumber; // stay
in the same place if the going out of the board
                                }
```

```
                                displayMessage("Position before logic: " +
Integer.toString(position[PLAYER_1]) + "\n"); // display message
                                position[PLAYER_1] =
snakesLadderLogic(position[PLAYER_1]); // perform ladders and snakes transition and
update position
                                if(position[PLAYER_1] == 100)
                                        statusArea.setText("Player 1 WON - Game
Over"); // display the winner in statusArea
                                sendToAll("Position of Player "+ player + "\n" +
position[PLAYER_1] + "\n" + diceNumber+ "\n"); // send the position and dice number
to all clients
                                break;
                        case 2: // control the turn of the players depending on the
number of players playing
                                if(playersConnected == 2)
                                {
                                        Turn_1 = true; // set turn 1 to true
                                        statusArea.setText("Status: Waiting for
Player 1"); // prints message in status area
                                } // end if
                                else if (playersConnected == 3 || playersConnected
== 4)
                                {
                                        Turn_3 = true; // set turn 3 to true
                                        statusArea.setText("Status: Waiting for
Player 3"); // prints message in status area
                                } // end else if
                                diceNumber = (int)(Math.random()*6) + 1; //
generate random number from 1 to 6
                                displayMessage("PLAYER 2 rolling the dice\n"); //
display message
                                position[PLAYER_2] += diceNumber; // update
position
                                if(position[PLAYER_2] > 100)
                                {
                                        position[PLAYER_2] -= diceNumber; // stay
in the same place if the going out of the board
                                }
                                displayMessage("Position before logic: "+
Integer.toString(position[PLAYER_2])+"\n"); // display message
                                position[PLAYER_2] =
snakesLadderLogic(position[PLAYER_2]); // perform ladders and snakes transition and
update position
                                if(position[PLAYER_2] == 100)
                                        statusArea.setText("Player 2 WON - Game
Over"); // display the winner in statusArea
                                sendToAll("Position of Player "+ player + "\n" +
position[PLAYER_2] + "\n" + diceNumber+ "\n"); // send the position of player 1 and
dice number to all clients
                                break;
                        case 3: // control the turn of the players depending on the
number of players playing
                                if(playersConnected == 3)
                                {
                                        Turn_1 = true; // set turn 1 to true
```

```
                                        statusArea.setText("Status: Waiting for
Player 1"); // prints message in status area
                                }
                                else if (playersConnected == 4)
                                {
                                        Turn_4 = true; // set turn 4 to true
                                        statusArea.setText("Status: Waiting for
Player 4"); // prints message in status area
                                }

                                diceNumber = (int)(Math.random()*6) + 1; //
generate random number from 1 to 6
                                displayMessage("PLAYER 3 rolling the dice\n"); //
display message
                                position[PLAYER_3] += diceNumber; // update
position
                                if(position[PLAYER_3] > 100)
                                {
                                        position[PLAYER_3] -= diceNumber; // stay
in the same place if the going out of the board
                                }
                                displayMessage("Position before logic: " +
Integer.toString(position[PLAYER_3])+"\n"); // display message
                                position[PLAYER_3] =
snakesLadderLogic(position[PLAYER_3]); // perform ladders and snakes transition and
update position
                                if(position[PLAYER_3] == 100)
                                        statusArea.setText("Player 3 WON - Game
Over"); // display the winner in statusArea
                                sendToAll("Position of Player "+ player + "\n" +
position[PLAYER_3] + "\n"+ diceNumber+ "\n"); // send the position of player 1 and
dice number to all clients
                                break;
                        case 4: Turn_1 = true; // set turn 1 to true
                                statusArea.setText("Status: Waiting for Player
1"); // prints message in status area
                                diceNumber = (int)(Math.random()*6) + 1; //
generate random number from 1 to 6
                                displayMessage("PLAYER 4 rolling the dice\n"); //
display message
                                position[PLAYER_4] += diceNumber; // update
position
                                if(position[PLAYER_4] > 100)
                                {
                                        position[PLAYER_4] -= diceNumber; // stay
in the same place if the going out of the board
                                }
                                displayMessage("Position before logic: " +
Integer.toString(position[PLAYER_4])+"\n"); // display message
                                position[PLAYER_4] =
snakesLadderLogic(position[PLAYER_4]); // perform ladders and snakes transition and
update position
                                if(position[PLAYER_4] == 100)
                                        statusArea.setText("Player 4 WON - Game
Over"); // display the winner in statusArea
```

```
                                    sendToAll("Position of Player "+ player + "\n" +
position[PLAYER_4] + "\n"+ diceNumber+ "\n"); // send the position of player 1 and
dice number to all clients
                                    break;
                        } // end switch
                } // end method rollLogic

                private int snakesLadderLogic(int position)
                {
                        // change the position of the token depending on the snakes and
ladders logic
                        int p = position;
                        switch(position)
                        {
                                case 1: p = 38;break;
                                case 6: p = 16;break;
                                case 11: p = 49;break;
                                case 14: p = 4;break;
                                case 21: p = 60;break;
                                case 24: p = 87;break;
                                case 31: p = 9;break;
                                case 35: p = 54;break;
                                case 44: p = 26;break;
                                case 51: p = 67;break;
                                case 56: p = 53;break;
                                case 62: p = 19;break;
                                case 64: p = 42;break;
                                case 73: p = 92;break;
                                case 78: p = 100;break;
                                case 84: p = 28;break;
                                case 91: p = 71;break;
                                case 95: p = 75;break;
                                case 98: p = 80;break;
                        } // end switch
                        return p;
                        } // end class snakesLadderLogic
        } // end class server
```

Server Test

```
public class ServerTest {

        public static void main(String[] args) {

                Server application = new Server(); // create new server application
                } // end main
} // end class
```

Client

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.Socket;
import java.net.InetAddress;
import java.io.IOException;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;
import java.util.Formatter;
import java.util.Scanner;

public class Client extends JFrame{

	private SnakesAndLaddersGUI gui; // given instance
	private JTextArea messageBox; // message box
	private JButton rollDiceButton; // button to roll the dice
	private String hostIP; // host IP for server
	private String playerNumber; // player number
	private String finalPosition = ""; // position after the logic
	private Socket connection; // connection to server
	private Scanner input; // input from server
	private Formatter output; // output to server
	private boolean myTurn = false; // my turn
	private boolean gameOver = false; // game is over
	private final static int PLAYER_1 = 0; // constant for first player
	private final static int PLAYER_2 = 1; // constant for second player
	private final static int PLAYER_3 = 2; // constant for third player
	private final static int PLAYER_4 = 3; // constant for fourth player
	private int currentPosition[]; // holds the current position of players
	private int diceNumber; // holds the dice number received from server

	public Client(String host)
	{
		super("Snake-and-Ladders Client"); // set Title of frame
		getContentPane().setLayout(null); // set layout to null
		hostIP = host; // get host address
		gui = new SnakesAndLaddersGUI(); // create new object gui of
SnakesAndLaddersGUI
		gui.setBounds(10, 48, 464, 365); // set bounds
		getContentPane().add(gui); // add gui to frame

		rollDiceButton = new JButton("Throw dice and Move"); // create new
button
		rollDiceButton.addActionListener(new ActionListener() { // add action
listener to the button
			public void actionPerformed(ActionEvent e) {
				try
				{
					rollDice(); // invokes method roll
```

```
                        } // end try
                        catch (InterruptedException ie) // catch interrupted
exception
                        {
                              ie.printStackTrace(); // prints throwable to
standard error stream
                        } // end catch
                  }
            });
            rollDiceButton.setBounds(114, 424, 258, 23); // set bounds for roll
button
            getContentPane().add(rollDiceButton); // add button to frame

            messageBox = new JTextArea(); // create messageBox of type JTextArea
            messageBox.setBounds(10, 17, 464, 20); // set bounds for the
messageBox
            getContentPane().add(messageBox); // add messageBox to frame
            messageBox.setColumns(10); // set 10 columns of messageBox

            messageBox.setEditable(false); // set non editable box

            setSize(500,490); // set size of window
      setVisible(true); // show window
      setResizable(false); // enable non resizable window

            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // // exit
application when closing window


            startClient(); // start the client
      } // end client constructor


      private void displayMessage(final String message) // method to display
messages on messageBox
      {
            // display message from event-dispatch thread of execution
            SwingUtilities.invokeLater(
                        new Runnable()
                        {
                              public void run() // updates outputArea
                              {
                                    messageBox.setText(message); // add message
                              } // end method run
                        } // end inner class
                  ); // end call to swingUtilities
      } // end method displayMessage

      // start the client thread
      private void startClient()
      {
            try // connect to server, get streams and start output thread
            {
                  // make connection to server
```

```
                    connection = new Socket(InetAddress.getByName(hostIP),15024);
// create new socket
                    input = new Scanner(connection.getInputStream()); // get input
stream from socket
                    output = new Formatter(connection.getOutputStream()); // get
output stream from socket
                    currentPosition = new int[4]; // create array of 4 integers
called currentPosition
                    gui.setNumberOfPlayers(4); // set number of players
                    for(int i = 0; i < 4; i++) // loop through each player
                    {
                          gui.setPosition(i,0); // set the position of each player
on the board to 0
                          currentPosition[i]= 0; // set currentPosition of each
player to 0
                    } // end for

                    while(true)
                    {
                          if(input.hasNextLine()) // wait for the next message
                          {
                                processMessages(input.nextLine()); // process the
messages from server
                          } // end if
                    } // end while

             } // end try
             catch(IOException ioe) // catch interrupted exception
             {
                    ioe.printStackTrace(); // prints throwable to standard error
stream
                    displayMessage("Connection not established."); // display
message
             } // end catch
      } // end method startClient

      // process incoming messages
      private void processMessages(String message)
      {

                    if(message.equals("Player number is"))
                    {
                                playerNumber = input.nextLine(); // gets the
number of player
                                switch(playerNumber)
                                { // displays player number and colour of token on
messageBox
                                      case "1": displayMessage("Player "+
playerNumber + ": Blue token");break;
                                      case "2": displayMessage("Player "+
playerNumber + ": Green token");break;
                                      case "3": displayMessage("Player "+
playerNumber + ": Red token");break;
                                      case "4": displayMessage("Player "+
playerNumber + ": Yellow token");break;
```

```
                                } // end switch
                        } // end if
                        else if(message.equals("Starting the game"))
                        {
                                switch(playerNumber)
                                { // display player number and start the game message
                                        case "1": displayMessage("Player "+ playerNumber +
": Blue token - Starting the game!");break;
                                        case "2": displayMessage("Player "+ playerNumber +
": Green token - Starting the game!");break;
                                        case "3": displayMessage("Player "+ playerNumber +
": Red token - Starting the game!");break;
                                        case "4": displayMessage("Player "+ playerNumber +
": Yellow token - Starting the game!");break;
                                } // end switch
                        }
                        else if(message.equals("Can roll")) // by receiving this
message client can roll the dice
                        {
                                if(!gameOver) // can roll until game is over
                                {
                                        myTurn = true; // set my turn to true
                                        switch(playerNumber)
                                        { // display the turn of the player
                                        case "1": displayMessage("Player "+ playerNumber +
": Blue token - Its your turn");break;
                                        case "2": displayMessage("Player "+ playerNumber +
": Green token - Its your turn");break;
                                        case "3": displayMessage("Player "+ playerNumber +
": Red token - Its your turn");break;
                                        case "4": displayMessage("Player "+ playerNumber +
": Yellow token - Its your turn");break;
                                        } // end switch
                                } // end if
                        } // end else if
                        else if(message.equals("Position of Player 1"))
                        {
                                finalPosition = input.nextLine(); // get final position
from server
                                diceNumber = input.nextInt(); // get dice number from
server
                                setGUI(PLAYER_1,finalPosition,diceNumber); // invoke
setGUI method
                        } // end else if
                        else if(message.equals("Position of Player 2"))
                        {
                                finalPosition = input.nextLine(); // get final position
from server
                                diceNumber = input.nextInt(); // get dice number from
server
                                setGUI(PLAYER_2, finalPosition,diceNumber); // invoke
setGUI method
                        } // end else if
                        else if(message.equals("Position of Player 3"))
                        {
```

```
                        finalPosition = input.nextLine(); // get final position
from server
                        diceNumber = input.nextInt(); // get dice number from
server
                        setGUI(PLAYER_3,finalPosition,diceNumber); // invoke
setGUI method
                } // end else if
                else if(message.equals("Position of Player 4"))
                {
                        finalPosition = input.nextLine(); // get final position
from server
                        diceNumber = input.nextInt(); // get dice number from
server
                        setGUI(PLAYER_4,finalPosition,diceNumber); // invoke
setGUI method
                } // end else if
        } // end method processMessage

        // update GUI of players after they rolled
        private void setGUI(int player, String finalPosition, int diceNumber)
        {
                int intPosition; // initialise intPosition
                intPosition = Integer.parseInt(finalPosition); // convert from string
to integer
                try
                {
                        Thread.sleep(300); // wait for 300 ms
                        if(intPosition > 0 && intPosition < 100)
                        {
                                if((currentPosition[player] + diceNumber) < 100)
                                {
                                        for(int i = currentPosition[player]; i <
currentPosition[player] + diceNumber; i++) // loop through the different positions
to show the movement
                                        {
                                                gui.setPosition(player, i); // set new
position i for player
                                                Thread.sleep(200); // wait for 200 ms
                                        } // end for
                                } // end if
                                gui.setPosition(player, intPosition); // set final
position of movement
                                currentPosition[player] = intPosition; // update the
current position
                        } // end if
                        else if(intPosition == 100)
                        {
                                for(int i = currentPosition[player]; i <
currentPosition[player] + diceNumber; i++)
                                {
                                        gui.setPosition(player, i); // set new position i
for player
                                        Thread.sleep(200); // wait for 200 ms
                                } // end for
```

```
                        gui.setPosition(player, intPosition); // set final
position of movement
                        displayMessage("PLAYER "+ (player + 1) +" WON! - GAME
OVER"); // display the winner and game over
                        gameOver = true; // set gameOver to true
                        sendToServer("Game Over\n"); // send message to server
                    } // end else if
             } // end try
             catch(Exception e) // catch any exception
      {
             e.printStackTrace(); // prints throwable to standard error stream
      } // end catch
      } // end method setGUI

      private void sendToServer(String message) // send messages to server
      {
             output.format(message); // output the message to server
             output.flush(); // flush the output
      } // end method sendToServer

      private void rollDice() throws InterruptedException
      {
             if(myTurn) // check for my turn
             {
                    switch(playerNumber)
                    { // display waiting message
                        case "1": displayMessage("Player "+ playerNumber + ":
Blue token - Wait for your turn");break;
                        case "2": displayMessage("Player "+ playerNumber + ":
Green token - Wait for your turn");break;
                        case "3": displayMessage("Player "+ playerNumber + ":
Red token - Wait for your turn");break;
                        case "4": displayMessage("Player "+ playerNumber + ":
Yellow token - Wait for your turn");break;
                    } // end switch
                    int intPlayerNumber = Integer.parseInt(playerNumber); //
convert from string to integer
                    myTurn = false; // set my turn to false
                    sendToServer("Rolling the dice\n" + intPlayerNumber + "\n"); //
send rolling dice message to server
             } // end if
      } // end method rollDice
} // end class Client
```

ClientTest

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.UIManager;

public class ClientTest extends JFrame {
```

```
	public static void main(String[] args) {

		UIManager.put("OptionPane.okButtonText", "Connect"); // changes the
OK button to Connect button
		String ip = JOptionPane.showInputDialog("Enter server's IP address");
// input dialog window to enter the IP address
		Client application = new Client(ip); // create new client application
	} // end main
} // end class test
```