Project 03: FoilMaker Game Client

Release date: 10/14/2016
Due date: 11/04/2016

Goals

1. Ability to use Java Networking API to facilitate communication between a server and client application over the network
2. Implement a simple application layer network protocol
3. String manipulation
4. Understand and use the Model-View-Controller (MVC) architecture pattern for GUI programming
5. GUI programming with Java Swing

Table of Contents

1. Introduction

In this project you are going to learn and use the Networking API in Java. This is a vital component technology in the Java platform stack as communication between applications over the network is inevitable in modern software. In fact, the vast majority of applications we use today interact with a remote service in one way or another. Understanding how to have two independent programs connect with each other over the network using a client/server model is an important skill.

We will learn this skill by creating a simple version of a popular game called Psych. Psych is a simple multiplayer party game based on the idea of outwitting people by fooling

them into picking fake or incorrect words for a given description. The version we will implement, allows the players to be dispersed anywhere in the world, as long as they have an internet connection and can run a Java program. A version of this game is available as an app for both the Android and iOS platforms, with the name Psych! – you are strongly encouraged to download this free app and play a round or two to get a sense of how the game works.

For the purposes of this project, the game will work as follows. The game has two separate programs that play the role of the server and the client, respectively. For this project, you will be given the server component, and will have to implement the Client program. In future projects, you will implement the Server.

From the Client's perspective: When the Client is first launched, the user has an option to either login to the server with an existing user and password combination, or register a new user and password combination. The Server is designed to remember previously created user and password combinations in a file (more details below). Once a user is created, and logged into the game, the player has an the option to either start a new game as the leader of the game, or join an existing game.

If the user chooses to start a new game, the server creates a special token for this new game and sends that to the leader. The leader shares this token with any other player that wants to join that game. If the user chooses instead to join an existing game, they need to enter the token for that game. Each new user that joins a game is displayed to the leader who can start playing the game when all players have joined.

The game proceeds in rounds. In each round the server picks a description for an uncommon word and sends it to each of the participants in the game (the leader and all other players that have joined the game). Each participant displays this description and allows the player to enter a suggested word for this description. Once a participant enters their word, it is sent to the server. The server collects the suggestions from each participants, and sends them all, along with the correct answer that it already knows, to every participant (the suggestions are in randomized order). Each participant shows these suggestions and the real answer to the player who then tries to pick the correct answer. Each participants selection is send to the server. When the server receives selections from every participant, and assigns scores to each participant based upon their selections. Players get points for picking the correct answer, and also if another player was fooled into picking their fake suggestion instead of the correct answer. The game goes on until the server runs out of words or the players decide to quit.

For this project, your assignment is to write a Java Swing GUI- based application for the FoilMaker! Client. When running your program, you will first start the Server program supplied to you. Each client that runs connects to this server and follows the steps of the game outlined above. The communication between the server and the client will follow a protocol that is provided to you (described below). This protocol specifies the contents of the messages between the clients and the servers.

2. Client Implementation

In this section we have an outline of wireframes for the client GUI and corresponding commands in the protocol. You should use Java Swing API to implement the GUI. You are strongly advised to structure the Client using the Model-View-Controller pattern described in the Appendix.

The protocol has the following generic format.

Request (Client to server)	<COMMAND>--<PARAM1>--<PARAM2>--…
Response (Server to client)	RESPONSE--<COMMAND>--<STATUS>--<TOKEN1>--<TOKEN2>-- …

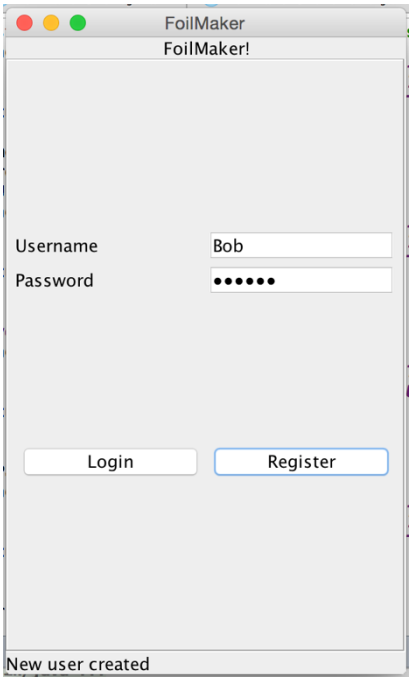
The "--" is used in this protocol to separate the various parts of each message. The allowed values for the Command, and other tokens of these messages are described below. The constants used in these message are defined in the FoilMakerNetworkProtocol.java file.

2.1 Register New User

Users need to be registered with the server before they can play the game. Once registered, the user credentials (i.e. username and password) are saved by the server in the "UserDatabase" file so that the details will not be lost even if the server is restarted. The Username or Password cannot be empty. The client should enable the user to enter a login and password which are then sent to the server in a "CREATENEWUSER" message. Any error messages should be displayed to the client. If the user creation is successful, an appropriate message should be shown to the user.

Hint: you may want to lookup the JTextField and JPasswordField classes.

Request	CREATENEWUSER--<username>--<password> E.g. CREATENEWUSER--Bob--bob123
Response	RESPONSE--CREATENEWUSER--<STATUS>--
Status	Meaning
INVALIDMESSAGEFORMAT	Request does not comply with the format given above
INVALIDUSERNAME	Username empty
INVALIDUSERPASSWORD	Password empty
USERALREADYEXISTS	User already exists in the user store
SUCCESS	User created in the user store successfully



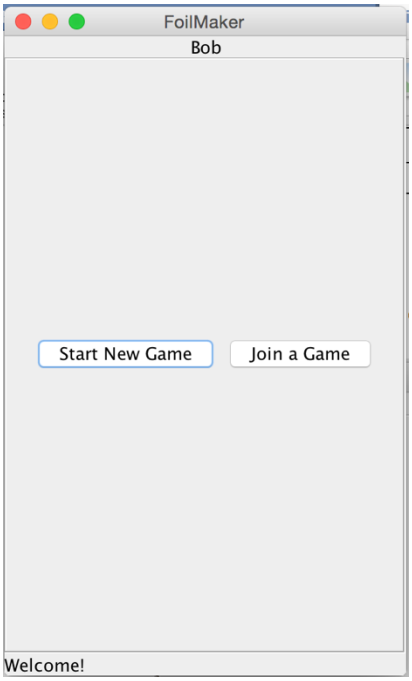
2.2 User Login

A registered user should log into the system before playing the game. The client allows the user to enter a login name and password, which it sends to the server using "LOGIN" message. Any error messages from the sever should be displayed to the user. If the login is successful, the server will provide a unique token (session cookie). The cookie is saved in the client and will be sent in future messages to the server to show that the message is from a logged-in client.

Request	LOGIN--<username>--<password> E.g. LOGIN--Bob--bob123
Response	RESPONSE--LOGIN--<STATUS>--
Status	Meaning
INVALIDMESSAGEFORMAT	Request does not comply with the format given above
UNKNOWNUSER	Invalid username
INVALIDUSERPASSWORD	Invalid password (User not authenticated)
USERALREADYLOGGEDIN	User already logged in
SUCCESS	User logged into the system successfully. In this case the user receives a unique token (session cookie) from the server, which she has to use in subsequent requests to the server. E.g. RESPONSE--LOGIN--SUCCESS--XXuKQrVDw

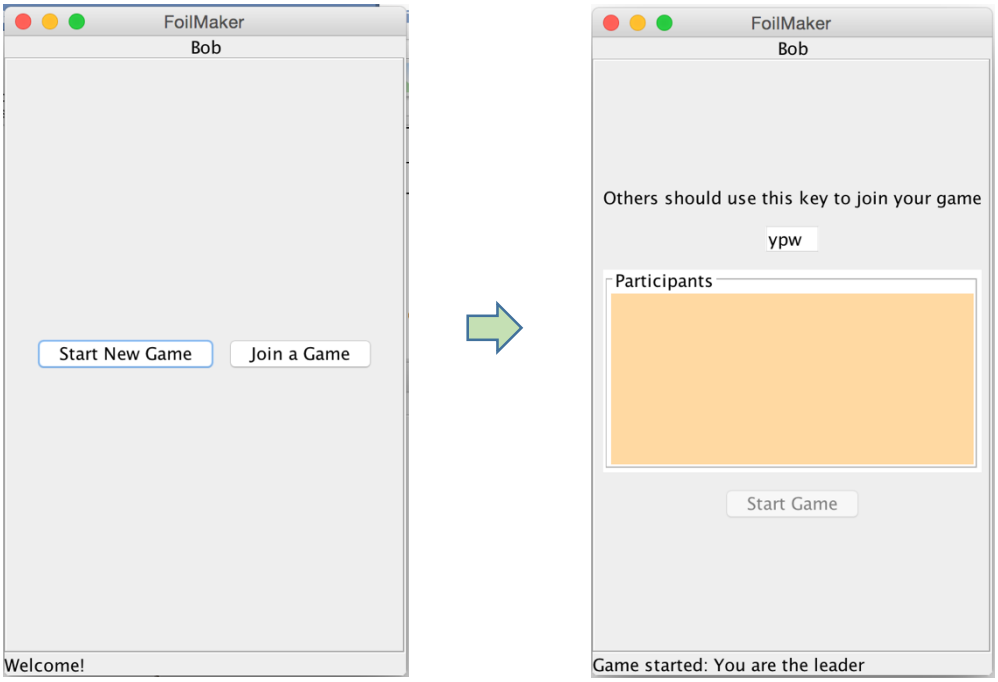
2.3 Play Game

Once a user logs into the system successfully she can either start a new game or join an existing game.



2.4 Start a New Game

A user that starts a new game becomes the leader of that game session. The <user token> field below is the session cookie for this user received when the user logged in to the server. If successful in creating a new game, the server will respond with the game token (three character string). This token should be saved by the client and will be used in subsequent requests. The leader view should now display this game token, and a list of participants that have joined this game. The leader shares this token (outside the game) with any other users that want to join the game.

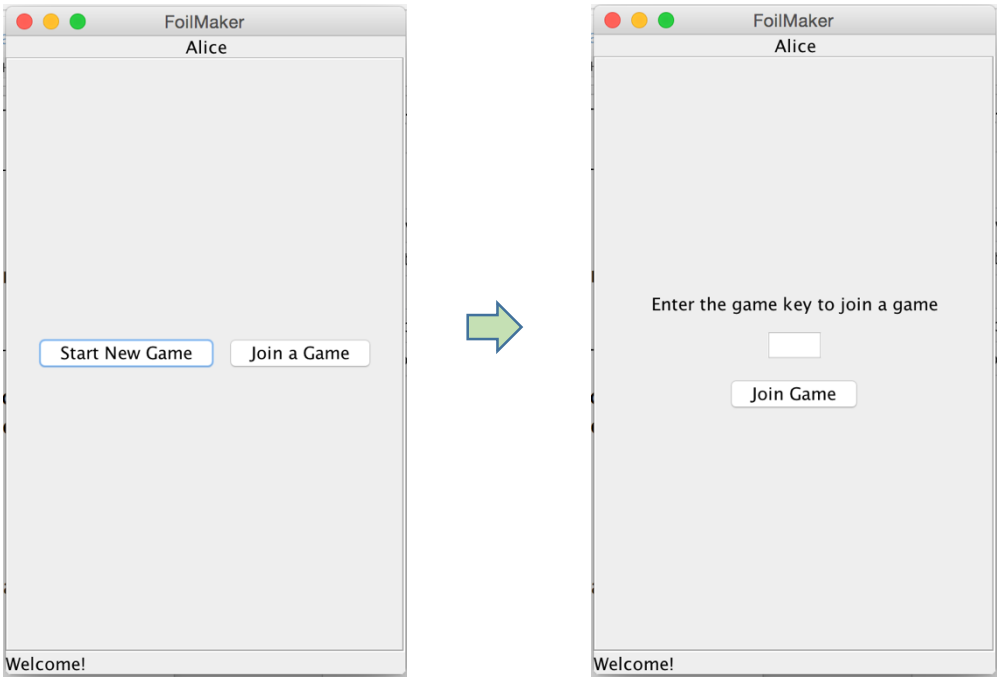


Request	STARTNEWGAME--<user token> E.g. STARTNEWGAME--XXuKQrVDw
Response	RESPONSE--STARTNEWGAME --<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	Invalid user token
FAILURE	User already playing or server failed to create a new game session due to an internal error.
SUCCESS	A new game session created successfully. In this case the user receives a game token that uniquely identifies the newly created game session. A user willing to join this game (see next section) will need to send this token to the server in her request to join. E.g. RESPONSE--STARTNEWGAME--SUCCESS--ypw

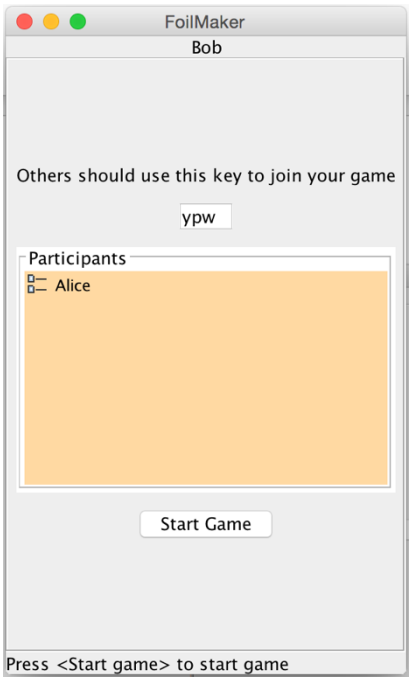
The leader should wait for others (at least one other player) to join before she starts the game. Once at least one other participant has joined, the "Start Game" button on the leader's view becomes enabled allow the leader to start the game. When another user joins the game, the leader is informed by the server of the new participant in the game (see below). You should update the leader's view (see the figure above) as and when new players join in.

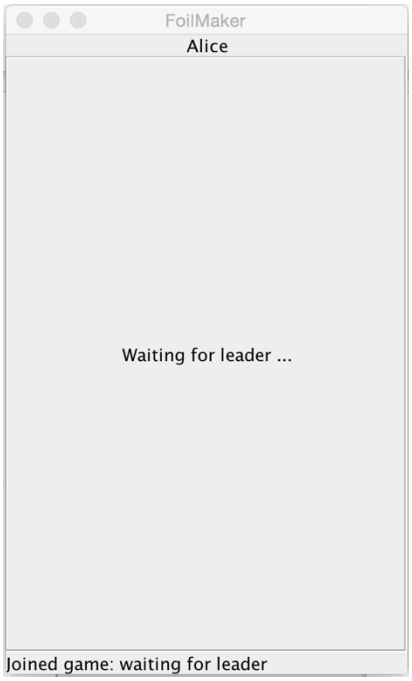
2.5 Join a Game

Player trying to join a game should wait for the leader to start the game. The leader will share with each such player the game key either verbally, or through some other means (text message, snapchat etc. – not using this game itself). The clients sends a "JOINGAME" message to the server with the player's login token, and the game key. Any error messages should be displayed to the user. If the joining is successful, the player's view should change to the view below indicating "Waiting for leader …". This means that we are waiting for the leader to launch the game. For each user that successfully joins a game, the leader of the game receives a "NEWPARTICIPANT" message with the name and score of the new user. This user should be added to the list of participants in the leader's view.



Request	JOINGAME--<user token>--<game token> E.g. JOINGAME--aHm]lUhlLg--ypw
Response	RESPONSE--JOINGAME--<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	Invalid user token
GAMEKEYNOTFOUND	Invalid game token
FAILURE	User already playing the game
SUCCESS	A new game session created successfully. In this case the leader also receives a separate notification for the newly joined player as follows. NEWPARTICIPANT--<username>--<score> Here <username> is the new player's name and <score> is her cumulative score (if the user dropped out and joins in again). E.g. NEWPARTICIPANT--Alice--0





2.6 Launch Game

When the leader is ready to start the game, she clicks on the "Start Game" button. This causes the leader's client to send a "ALLPARTICIPANTSHAVEJOINED" message to the server. Any error messages from the server should be displayed to the leader's view.

Request	ALLPARTICIPANTSHAVEJOINED--<user token>--<game token> E.g. ALLPARTICIPANTSHAVEJOINED--XXuKQrVDw--ypw
Response	RESPONSE--ALLPARTICIPANTSHAVEJOINED --<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	Invalid user token
INVALIDGAMETOKEN	Invalid game token
USERNOTGAMELEADER	User already playing the game

When the server managed to launch the game successfully it will not send a SUCCESS code in this case rather it will start the game.

The game consists of several rounds. Each round has the following Steps:

- The server sends the new description to each participant
- Each participants displays the description and waits for its user to enter a suggested word for this description which is sent to the server
- The server waits for a suggestion from each participant. Once they are all received, it adds the correct answer, scrambles the suggestions and the correct answer and sends them to each participant.
- Each participant waits for the suggestions from the server, and then shows them to the user. Once the user picks a suggestion it is send to the server

- The server waits for each participant to send its choice, then scores each participant and sends a message with all participant names and scores to each participant. After this it also sends to each player either the word for the next round in a “NEWWORD” message, or a “GAMEOVER” message when there are no more words.
- Each participant displays the names and scores of each participant in the game and waits for the user to be ready to play the next round. When the player is ready, the client shows the word for the next round

Server sends the first round’s word, and the correct answer to each participant’s client (so that the client application can prevent a user from using it as a suggestion) according to the following format.

NEWGAMEWORD--<question>--<correct answer>
E.g. NEWGAMEWORD--A group of zebras--a dazzle

Each client receives this command from the server so that client application can show it on its GUI as follows.

2.7 Send Player’s Suggestion

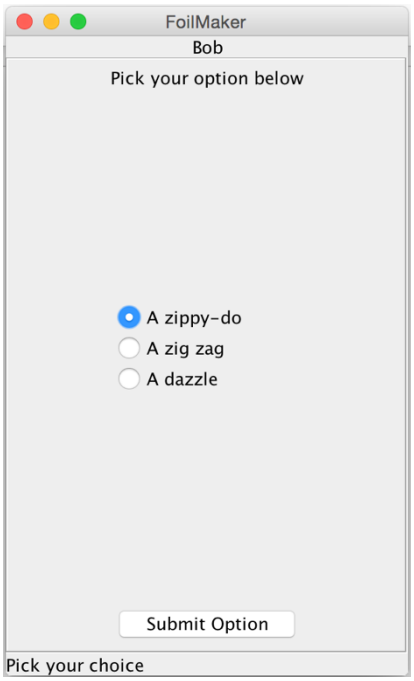
Each client can send her suggestion as follows.

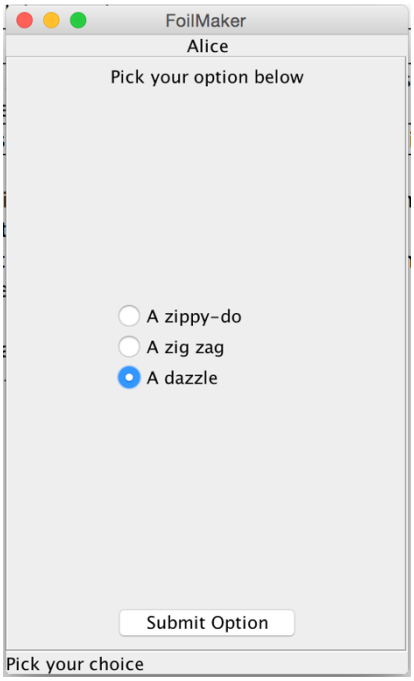
Request	PLAYERSUGGESTION--<user token>--<game token>--<suggestion> E.g. PLAYERSUGGESTION--XXuKQrVDw--ypw--A zig zag
Response	RESPONSE--PLAYERSUGGESTION--<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	Invalid user token

INVALIDGAMETOKEN	Invalid game token
UNEXPECTEDMESSAGETYPE	A suggestion was sent when a different message was expected by the server
INVALIDMESSAGEFORMAT	Message format is not according to what is given above

Once the server receives suggestions from all the players it will send them along with the correct answer to each client so that it can show them to the user. The user is going to pick an answer from this list in the next step. Furthermore if there are N players (including leader) in a session there will be N + 1 answer options on this list.

ROUNDOPTIONS--<answer 1>--<answer 2>--<answer 3>-- …
E.g. ROUNDOPTIONS--A zippy-do--A zig zag--A dazzle





2.8 Send Player's Choice

Once the user picks her choice you can send it to the server as follows. Once the server receives all the answers and finishes processing them it will send the results to each client as given in the next section.

Request	PLAYERCHOICE--<user token>--<game token>--<choice> E.g. PLAYERCHOICE--XXuKQrVDw--ypw--A dazzle
Response	RESPONSE--PLAYERCHOICE--<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	Invalid user token
INVALIDGAMETOKEN	Invalid game token
UNEXPECTEDMESSAGETYPE	A suggestion was sent when a different message was expected by the server
INVALIDMESSAGEFORMAT	Message format is not according to what is given above

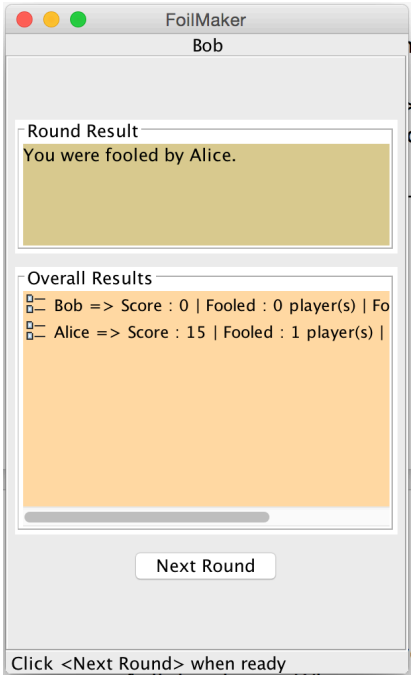
2.9 Receive Results

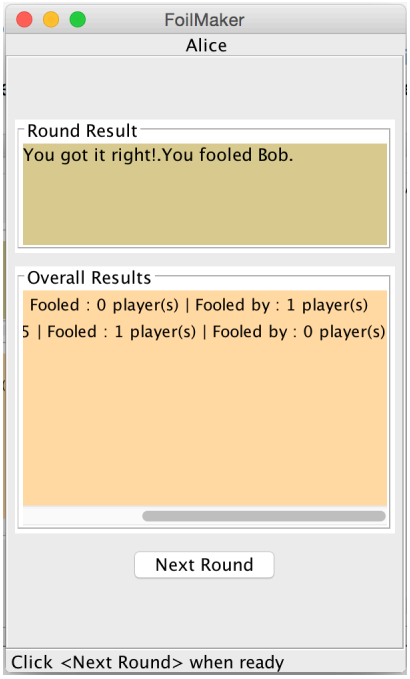
This is the format of the message that includes overall results of the game so far.

ROUNDRESULT--<Player1>--<Message for player 1>--<Cumulative score of player 1>--<Number of players fooled by player 1>--<Number of times player 1 was fooled by the others> …

E.g. ROUNDRESULT--Bob--You were fooled by Alice.--0--0--1--Alice--You got it right!.You fooled Bob.--15--1—0

Please note that each player is going to get the same message that includes overall results of all the players. When you display results you can show the message that is intended to a given player separately and overall results of each play in tabular form (see below).

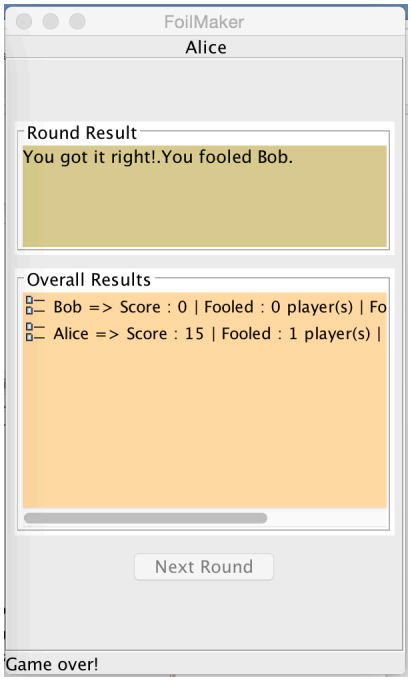




2.10 Go to Next Round

The server sends next question (NEWGAMEWORD message) as soon as it has sent results to all the clients (ROUNDRESULT command). So the clients don't have to send a special message to move on to the next round.

However the server sends GAMEOVER-- command as it runs out of questions. Clients can react to this command appropriately and quit. For instance disable the "Next Round" button and set status message as follows.



2.11 Quit Game

The client application should be able to quit the game whenever the user wishes. You can either have a button to do that or just simply close the GUI. When the client quits it should send LOGOUT-- command to the server as follows.

Request	LOGOUT--
Response	RESPONSE--LOGOUT--<STATUS>--
Status	Meaning
USERNOTLOGGEDIN	User currently not logged in
SUCCESS	User logged out of the system successfully

3. How to Run the Server

You can test your client application against the given server implementation. Please make sure the given files, *FoilMaker.student.jar, UserDatabase,* and *WordleDeck* reside in the same directory. The given user store (UserDatabase) already includes 2 users (Alice and Bob). The wordle file includes one question and answer. Run the following command in a terminal to launch the server. Your clients will have to create network connections with this server using the the port number that you specify below and the IP address of the machine on which this server is run.

```
% java -jar FoilMaker.student.jar <port number>
```

4. Grading Rubric

1. Proper use of socket API for communication (this includes socket creation, stream reader, stream writer, etc) – 15 %
2. Implementation – 60 %
   2.1 Register New User – 5 %
   2.2 User Login - 5 %
   2.3 Play Game - 5 %
   2.4 Start a New Game - 5 %
   2.5 Join a Game - 5 %
   2.6 Launch Game - 5 %
   2.7 Send Player's Suggestion - 5 %
   2.8 Send Player's Choice - 5 %
   2.9 Receive Results - 10 %
   2.10 Go to Next Round - 5 %
   2.11 Quit Game - 5 %
3. Use of MVC architecture – 10 %
4. Look and feel of the GUI – 15 %

Appendix A

Introduction to Java Networking API

In this project you need to use *Stream Sockets* in your client program to *reliably* communicate with the game server. Implicitly this means you will be using TCP as the transport layer protocol for point-to-point communication with the server. You need to know 2 things in order to talk to a remote application over the network.

1. The IP address of the computer that hosts the server
2. Port number assigned to the server application. Think of this as an identifier assigned to the application by the host operating system.

In essence an IP address and port number is just enough to uniquely identify a remote application. Once you know those two parameters you can connect to the remote server and start communicating as follows.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

String serverIP = "localhost";
int serverPort = 50000;

try {
    // Connect to server
    Socket socket = new Socket(serverIP, serverPort);

    // Create data writer
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

    // Create data reader
    InputStreamReader isr = new InputStreamReader(socket.getInputStream());
    BufferedReader in = new BufferedReader(isr);

    // Send message to server
    out.println("Message to server");

    // Read server response
    String serverMessage = in.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

Appendix B

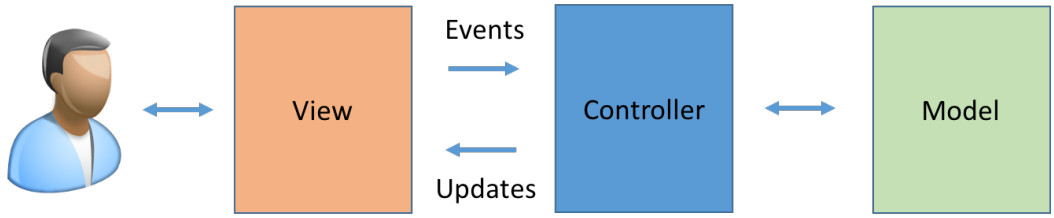
Sample Swing GUI

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class SampleGUI extends JFrame {
    /* Main panel */
    JPanel mainPanel = new JPanel();
    CardLayout layout = new CardLayout();

    /* Panel 1 */
    JPanel panelFirst = new JPanel();
    JButton buttonOne = new JButton("Switch to Panel 2");

    /* Panel 2 */
    JPanel panelSecond = new JPanel();
    JButton buttonSecond = new JButton("Switch to Panel 1");

    public SampleGUI() {
        /* Add button to panel 1 */
        panelFirst.add(buttonOne);
        panelFirst.setBackground(Color.CYAN);

        /* Add button to panel 2 */
        panelSecond.add(buttonSecond);
        panelSecond.setBackground(Color.ORANGE);

        /* Add panel 1 and panel 2 to main panel */
        mainPanel.setLayout(layout);
        mainPanel.add(panelFirst, "1");
        mainPanel.add(panelSecond, "2");

        /* Add action listener to button 1 on panel 1 */
        buttonOne.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
                layout.show(mainPanel, "2"); /* Show panel 2 when button clicked */
            }
        });

        /* Add action listener to button 2 on panel 2 */
        buttonSecond.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
                layout.show(mainPanel, "1"); /* Show panel 1 when button clicked */
            }
        });

        /* Set up frame */
        /* Add main panel to GUI frame */
        add(mainPanel);

        /* Set dimensions */
        setLocation(200, 200);
```

```
        setMinimumSize(new Dimension(200, 200));

        /* Dispose frame when close button is pressed */
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        /* Show panel 1 initially */
        pack();
        layout.show(mainPanel, "1");
        setVisible(true);
    }

    public static void main(String[] args) {
        new SampleGUI(); /* Show GUI */
    }
}
```

Appendix C

MVC Architecture



MVC is a common programming methodology specially used for GUIs. The basic idea is to separate concerns and this figure depicts how components in this architecture interact with each other. The *model* keeps the data. It maintains data structures that reflects the internal state of the whole application. The *controller* manipulates data in the model as per user interactions (for instance clicking on a button in a calculator application). Based on changes in the model it sends updates to the *view* that in turn shows them to the user (for instance result of a calculation in calculator application).

In this project you may have 3 different classes to represent MVC components where the model class captures data (questions, word suggestions, etc. received from the server), the view class implements Java GUI classes, and the controller implements network programming bits, application protocol, etc.