

UNIVERSITY OF BIRMINGHAM

MSC SOFTWARE WORKSHOP GROUP PROJECT

Super Magic Fairy Fun Quiz

Authors

Ashton HILLS
Ningyuan SUN
Andrew TALBOT
Richard WARNER
Edmond WONG

Team ATHENS

Supervisor

Alexandros EVANGELIDIS

19 March, 2018



School of Computer Science
University of Birmingham
Birmingham, B15 2TT

Contents

1	Introduction	1
2	Team Organisation	1
3	Super Magic Fairy Fun Quiz: A Brief Description	2
3.1	How to Play	2
3.2	Scoring System	2
3.3	Ranking System	2
4	System Design	3
4.1	Introduction	3
4.2	Communication Protocols	3
4.3	client Package	4
4.4	database Package	6
4.5	general Package	8
4.6	guiSwing Package	8
4.7	server Package	9
5	Test Plan	11
6	Evaluation and Conclusion	21
A	Project Diary	21
A.1	Tuesday 20 th February, 2018 (15:00 - 16:00)	21
A.2	Friday 23 rd February, 2018 (15:00 - 16:30)	22
A.3	Monday 26 th February, 2018 (15:00 - 16:30)	23
A.4	Tuesday 27 th February, 2018 (15:20 - 16:00)	24
A.5	Thursday 1 st March, 2018 (14:00 - 14:45)	24
A.6	Friday 2 nd March, 2018 (14:00 - 18:30)	25
A.7	Tuesday 6 th March, 2018 (11:00 - 12:00)	25
A.8	Tuesday 6 th March, 2018 (14:00 - 14:45)	26
A.9	Saturday 10 th March, 2018 (20:00 - 20:45)	27
A.10	Tuesday 13 th March, 2018 (14:00 - 15:00)	27
A.11	Thursday 15 th March, 2018 (15:00 - 15:30)	28
B	System Requirements	30
C	User Stories	33
D	Class Diagrams	36
D.1	Original Class Diagram	36
D.2	Final Class Diagrams	37
D.2.1	Client and GUI Class Diagram	37
D.2.2	Database and Server Class Diagram	38
D.2.3	Server and Client Diagram	39
E	Database ER Diagram	40

F	GUI Development	40
F.1	Rationale for the Prototypes	40
F.2	Paper GUI	41
F.3	GUI Prototype	43
G	Contribution Statement	44



1 Introduction

Hardworking Computer Science students at the University of Birmingham have always wanted a fun, competitive way to revise. Introducing: Super Magic Fairy Fun Quiz (SMFFQ). SMFFQ is a multi-player, competitive quiz game, whose questions are categorised into topics based on the modules in the MSc Computer Science course at the University of Birmingham. The title was inspired by the incredible analogy Uday Reddy[1] gave about the underlying “magic” of recursion. Users compete against each other in a first-come, first-serve quiz, where the answer options are limited to only one per player. Once one answer is taken, no other player can choose it. Each player in a game must race to take the correct answer as quick as possible! iop’

2 Team Organisation

At the very first meeting, the team was in agreement on the desire to design and build a game for the project. Several ambitious ideas were pitched, but after careful consideration of the time allowed and the coding capabilities of the team, it was decided that a multi-player quiz game would be within the scope, but still have enough complexity to provide a challenge to the team, and thus be an appropriate project to pursue.

The opportunity was taken for team members to introduce themselves, as not everyone was previously acquainted. Individual strengths, weaknesses and backgrounds and interests were discussed, and this information was later used when tasking different aspects of the project.

Following this, the project was broken down into five subsections:

- **Graphical User Interface:** Overseen by Richard.
- **Database:** Overseen by Ashton.
- **Client:** Overseen by Andrew.
- **Server:** Overseen by Ningyuan.
- **Communication Protocol:** Overseen by Edmond.

Richard was eager to work on the Graphical User Interface, as GUI development was an interest of his. Ashton knew she would be dealing heavily with databases in her job in September, and took this opportunity to gain a greater understanding of using them in line with a program. It was clear that Ningyuan and Andrew were both very competent programmers, and they took on the task of building the team’s server and client subsystems. Edmond’s exceptional organisational skills were quickly recognised; alongside developing the communication protocol, he took on a leadership role within the team, organising meetings and acting as the point of contact between team and tutor.

During the first meeting, it became apparent that the team wanted to make an attempt to follow a more formal software engineering approach. Due to the relatively short time constraints of the project, as well as this being everyone’s first formal introduction to software development project, the team decided to use a Scrum method (the reader should note that the team did not have a product owner). Team members took turns as the Scrum master of the group, who was able to choose (with input from all team members) what to focus on during small sprints between meetings. This made working on the project very flexible, and, due to the short breaks between meetings, meant that individuals could adapt their programming very quickly to adjust as the project progressed. To reference how much our project progressed and changed as we went. Please refer to our initial first cut class diagram, see D.1, which displays our initial



thoughts on how the project would look. We can compare this to our finalized Class Diagram, see D.2.1, which has heavily developed from the first.

The team worked really well together for the duration of the project. Communication and organization were extremely efficient, and team morale was always high. Individuals were quick to work together to overcome any challenges or blocks met while developing the system, and everyone contributed significantly and fairly to the overall project.

3 Super Magic Fairy Fun Quiz: A Brief Description

3.1 How to Play

This system allows up to four players to play against each other as they race to score the most points in a quiz of ten questions based on a selected topic in Computer Science. As each question and set of answers are displayed, a timer of 15 seconds begins to count down, during which time the players must select the answer that they think is correct.

However, there is a twist: Once one player selects an answer, that answer is then no longer available for selection by another player. The remaining players must race to select what they believe to be the correct answer, before it is claimed by another player and before the timer is up.

3.2 Scoring System

The amount of points a player can score is proportional not only to how many correct answers versus incorrect answers he or she selects, but also how *quickly* he or she selects them. A player will gain more points for answering correctly in less time, but players must be careful, as answering incorrectly will subtract points from their score.

Every quiz game consists of ten rounds of questions. Once one question has been displayed and the timer started, the game then waits for any of the players to respond. Upon receiving an initial response, the remaining players either select their answers or abstain from answering at all. Once the time is up, points are allocated as follows:

- A player who answered correctly is awarded $100 * \frac{k}{15}$ points, where k is the time remaining for that question. This means that the quicker they answer correctly, the more points they will score.
- Any players who select an incorrect answer are docked 50 points from their total score for that quiz.
- Any players who choose *not* to answer (whether that is because they are unsure of the answer, or they can see that another player has already claimed the correct answer) will be neither awarded nor docked points.

At the end of all ten rounds, the players receive bonus points according to where they came overall amongst their competitors. The player in 1st place is given 200 bonus points, 2nd place receives a bonus of 100, 3rd receives 50 and 4th receives 25. These points are added to their total scores for that quiz, and then this is added to their overall cumulative score which determines their 'rank'.

3.3 Ranking System

Each player's rank is determined by their total score across all games they have ever played:



1. `whatiscomputer?` (0 to 49 points)
2. `noob` (50 to 149 points)
3. `ramibless` (150 to 299 points)
4. `SQL: Still Quite Lamé` (300 to 499 points)
5. `slightly less noobie` (500 to 999 points)
6. `De-bugsy Malone` (1000 to 1749 points)
7. `Java is love` (1750 to 2999 points)
8. `freakin sick` (3000 to 4999 points)
9. `dat pro Masters lyfe` (5000 to 7999 points)
10. `Manfred` (8000+ points)

4 System Design

4.1 Introduction

The system design used a client-server architecture, while also using the topic-based publish-subscribe system in order to implement lobbies [2].

4.2 Communication Protocols

These protocols are inner classes in the `Server` class and `Client` class. They communicate using serialized `Message` objects. These objects contain different data structures which are needed for various purposes which are detailed above. `Server` and `Client` deal with different requests based on the “header” of each message, represented as an `int`. The message headers that the server will receive are:

- `CONNECT` = 0;
- `REGISTER` = 1;
- `DISCONNECT` = 2;
- `PLAY` = 3;
- `CHANGE_INFO` = 4;
- `ANSWER` = 5;
- `QUIT` = 6;
- `LEADERBOARD` = 7;
- `LOGOUT` = 8;

The message headers that the client will receive are:

- `GOOD_SIGN_IN` = 0;



- BAD_SIGN_IN = 1;
- GOOD_SIGN_UP = 2;
- BAD_SIGN_UP = 3;
- GOOD_UPDATE = 4;
- BAD_UPDATE = 5;
- QUEUEING = 6;
- PLAYER_JOINED = 7;
- PLAYER_QUIT = 8;
- LOBBY_COUNTDOWN = 9;
- GAME_ENDED = 10;
- GAME_COUNTDOWN = 11;
- Q_AND_A = 12;
- FIRST_PLAYER_CLAIMED = 13;
- PLAYER_CLAIMED = 14;
- OPTION_TAKEN = 15;
- UPDATE = 16;
- INTERRUPTION = 17;
- RESULTS = 18;
- LEADERBOARD = 19;
- LOGOUT = 20;

The details of each protocol are discussed below.

4.3 client Package

Inside the `client` package, there is a single class named `Client` containing all functionality for the client, including listening for messages from the server, writing messages to the server and providing checks on user input such as registering a new valid username.

The client is built with the intentions of sending information to the server, and to also receive information from the server and then handle it accordingly. One of the aims of the project was to ensure that the source code is modular, thus `Client` only deals with communication with `Server`, and any information about the state of the program is stored in the `GameModel` class, which is the Model in the Model-Observer pattern that was implemented.

The client is required to run two processes concurrently: it must be listening to the server constantly to receive information, and it must also send information to the server. For the purposes of this project, it seemed appropriate to extend the `Thread` class, rather than implement `Runnable`, as the superclass of the client was not required. The main thread is what is used to send messages to the server, and all of the code which is used to listen to the Server is stored in



the `run` method, which is executed by the new thread created in the constructor of the `Client` object.

The client sends and receives messages following the same protocol as the server. The only method of communication used is the sending of `Serializable Message` objects through an `ObjectInputStream` and an `ObjectOutputStream`. We ensure that the only object that we send to the Server is the `Message` object. However, we use the method `writeObject()` when sending an object. This throws a `NotSerializableException`, which is caught, flagging whenever there is an attempt to send a non `Serializable` object.

A `while` loop is used to check whether the client is still running; if it is, then it will be constantly listening to the server for incoming messages. When a message is received, it is converted to a `Message` object, which is then passed through to the `ComProtocolClient` which is an inner class in `Client`, provided by Edmond. While listening to the server, this is placed in a try and catch block, catching an `IOException`, so that if the server unexpectedly disconnects, then users can be notified through a `JFrame` in the GUI, and be able to close the program through this `JFrame`. We do this to ensure that the user is not left with an unresponsive GUI.

These are the message headers that the client listens for and then handles depending on the header, as seen in the previous Communication Protocol section. Due to the Model-View approach to the project that was taken in the GUI, most of these protocols will update the data in the `GameModel`, and do nothing else. Sometimes the Client must notify the `ViewDriver` to create `JFrames`, that can appear regardless of the data in the `GameModel`, which is important for example with error messages.

- `GOOD_SIGN_IN` - When this header is received, the Client updates the data in the `GameModel` with the user that is sent in the message.
- `BAD_SIGN_IN` - When this header is received the client notifies the `ViewDriver` to say that the login has failed.
- `GOOD_SIGN_UP` - When this header is received, the Client notifies the `ViewDriver` to tell the user that they have signed up successfully.
- `BAD_SIGN_UP` - When this header is received, the Client notifies the `ViewDriver` to tell the user that the registration has failed.
- `GOOD_UPDATE` - When this header is received, the Client notifies the `GameModel` to take the user to the Main Menu as they have successfully updated their player information (avatar, username, password).
- `BAD_UPDATE` - When this header is received, the Client notifies the `ViewDriver` that the update to their player information has unfortunately failed. For example, when they attempt to change their username to something that is already in the database.
- `QUEUEING` - When this header is received, the Client updates the data in the `GameModel` adding the users that are already in the lobby, including the user of this instance of the Client too.
- `PLAYER_JOINED` - When this header is received, and the user in the message is not the user with this instance of the Client, then the Client updates the data in the `GameModel` with all of the users in the lobby, we receive this message when a user joins the lobby. However, if the user in the message is the user in the instance of the client, then the Client updates the `GameModel` to add the user to the appropriate panel in the GUI, which in this case is the `GameLobbyQueue`.



- **PLAYER_QUIT** - When this header is received, and the user in the message is not the user with this instance of the Client, then the Client updates the data in the **GameModel** with all of the users in the lobby, we receive this message when a user leaves the lobby. However, if the user in the message is the user in the instance of the client, then the Client updates the **ViewDriver** to update the view in the GUI, this is a way of notifying the user that they have successfully left the lobby, and will not start the game.
- **LOBBY_COUNTDOWN** - When this header is called, the Client updates the **GameModel** to inform them that the lobby that the user is in will begin, and that question will start shortly. The Gui should update based on this update in the information.
- **GAME_ENDED** - When this header is received, the Client updates the data in the **GameModel**, to handle when the game ends.
- **GAME_COUNTDOWN** - When this header is received, the Client updates the **GameModel**, so that there is data so the GUI can update so they know that the game is about to begin.
- **Q_AND_A** - When this header is received, the Client notifies the **GameModel** with data so that the GUI can know that the game has either started and the game must update with the new question to be displayed.
- **FIRST_PLAYER_CLAIMED** - When this header is received, the Client updates the data in the **GameModel**, with the relevant information about the first user to claim a button, so that the GUI can be updated accordingly.
- **PLAYER_CLAIMED** - When this header is received, the Client updates the data in the **GameModel**, with the relevant information about the next user to claim a button, so that the GUI can be updated accordingly.
- **OPTION_TAKEN** - When this header is received, the Client When this header is received, the option that the client was attempting to take had already been claimed by another player.
- **UPDATE** - When this header is received, the Client notifies the **GameModel** with the current scores and players in the game.
- **RESULTS** - When this header is received, the Client notifies the **GameModel** with the scores which are then displayed by the GUI.
- **LEADERBOARD** - When this header is received, the GUI will update to the leader board screen and populate it with all of the Users from the array of Users in the message.
- **LOGOUT** - When this header is received, the server will have just removed this current user from the active users array list. The GUI then updates to go back to the welcome page.

4.4 database Package

The system's database contains four tables: **question_bank**, **answer_choices**, **topics** and **users**. The first three are pre-populated with data that is to be used by the system to construct and run quizzes for the players. The final table, **users**, is populated and consistently updated via the Java code, as more people register as players, play games, earn points, and change their own account details such as their username, password or avatar.



The development of the database started somewhat intuitively - a quiz game would have been able to pull from a bank of questions and corresponding answers. The first table, **question.bank**, has four attributes. The **question_id** column acts as the table's primary key. The table then has attributes for the actual question (**question**), the topic under which that question falls (**topic**), and the level of difficulty of that question (**level**). Each question in this table has four corresponding answer choices, all of which are stored in the **answer.choices** table.

The **answer.choices** table has a primary key which is stored in the **answer_id** attribute. The actual answer is then stored in the **answer** attribute, and the third attribute, **correct**, is a boolean which is set to **TRUE** if that answer is the correct answer to the question to which it corresponds, or **FALSE** if it is incorrect. The final attribute is **question_fk**, which acts as a foreign key referencing **question_id** in the **question.bank** table.

The third table, **topics**, is smaller, storing only the different quiz topics (**topic**), along with each primary key, stored in **topics_id**, and a shorthand version of the topic, stored in **abbreviation**. The abbreviations of each topic have no whitespace and are therefore able to be used as attribute names in the **users** table (see below). For instance, the topic "Nature Inspired Search and Optimisation" is abbreviated to "niso", and this abbreviation can be found as an attribute title in **users**.

Originally, the **topics** table did not seem necessary, until it was decided that users should be able to view not only their overall score of all the points they have collected cumulatively, but also that they should have the ability to see their individual high scores for a particular quiz topic, so that they could see which topics they excelled at and which ones they still perhaps needed to revise more. Passing the topic name (a String with whitespace) through the method in the Java required some way of converting this into a valid attribute name in the database. It was deemed best to do this in the database itself, instead of hard-code a conversion method into the Java. This way, if it was required that a new topic of questions be added to the system, only the database would have to be updated, and not the Java as well.

The final table contains all information pertaining to individual users of the system. Consequently, the **users** table has many attributes: **user_id** (primary key), **username**, **password**, **total_score** (the user's overall cumulative score), **rank** (an integer between 1 and 10 that links to a String title for that rank in the Java), **avatar** (an integer between 1 and 10 that links to an image avatar that is displayed via the GUI to the user), **niso**, **sw**, **ds**, **cs**, **ai**, **db**, and **os** (the last seven of which are all topic abbreviations and contain the user's high scores for each topic).

The **users** table is updated via the Java code every time a new user registers to play the game. They select a username and password, and, provided both of these are valid, a new entry in the table is created for them, filling all of the remaining attributes with their default values. Scores and rank are then continuously updated as the player takes quizzes and earns more and more points. Additionally, they are able to change their username, password, and/or avatar as they wish.

The database is able to be accessed by the system's implementation of the JDBC API. The database package in the system contains a JDBC class, which houses a method to open a connection with the database. Also in this package is the **Queries** class which implements **QueriesInterface**, containing many methods to query the database, which are called by the system's server. Each method constructs and executes a **PreparedStatement** and either returns a boolean confirmation, or data from a table in the database. The system relies heavily on the database to register or verify users, construct leaderboards and create actual games for users to play. Because of this, **PreparedStatement**s were favoured over regular **Statements** in order to help prevent against SQL injection attacks, as well as increase the overall efficiency of



the program due to `PreparedStatement`s being pre-compiled and able to be stored as cache if they are repeatedly used.

The reader may refer to Appendix E to see an Entity Relation diagram for a visual representation of the database and its relations.

4.5 general Package

The `general` package contains three widely used objects in the application: `User`, `Message`, and `Question`.

The `User` class is used for referring to a specific user. Note that only relevant information of that user is passed over during various points of the game. The database stores the true information of each user, and the user class simply pulls information from the database (such as username, rank, avatar, etc.) and uses that information during the game. The information of a specific user is changed when a game is finished (i.e. the database is updated with the user's new scores and new rank, if applicable,) as well as when the user goes onto the "Options" menu to change their own information. The field variables in the user class are as follows: `String username`, `String password`, `int status`, `int rank`, `int avatar`, `int totalScore`, and `Map<String, Integer> topicHighScores`.

The `Message` class facilitates communication between the client and the server. This class contains three different constructors, which are used at different times in the application, based on the message headers. The field variables are: `int header`, `User user`, `String text`, `String question`, `String[] options`, `int answer`, `User[] queueing`, and `Map<User, Integer> scores`. It is important to note that every `Message` object will be instantiated with a header. Otherwise, it will not be possible for the client or server to process the message and to call the relevant methods. This object is needed as it is required to send many different types of information between the server and the client (e.g. `Strings`, arrays of `String`, `User` objects, etc.).

Finally, the `Question` class is used for managing the questions that the server sends. An object of this class is constructed from a `String` variable to store the body of a question in the quiz, a `String` array for the four corresponding answer options, and lastly, an integer number for storing the index to the correct answer in the array.

4.6 guiSwing Package

The `guiSwing` package contains all of the functional components of the Graphical User Interface - written with `Swing` as per the brief. It uses the `Observer` pattern so that the clients have matching views for the most part. Exceptions to this include the `EditInfoPanel`, login/register panels and the `answerPanel` (which is the panel displaying the buttons within the game) within `gamePanel`, which shuffles answers differently per client. This package aims to implement the `Model-Observer` pattern in which the `Model` receives information data about the state of the program, and the classes which implement `Observer` in the GUI update according to this information.

`ViewDriver` is the main class within the `guiSwing` package, and initialises the frame and panels that the GUI uses. It creates a new instance of `GameModel` referring to the current client, and updates the view using `update()` as necessary, depending on the current state of the `GameModel`. `ViewDriver` also displays most of the client error messages in a graphical manner.

`GameModel` is the `Observable` class, which is being updated by the client when the client receives information about the state of the program within the server. When the field variables in the `GameModel` class are updated, we call the methods `setChanged()`, and `notifyObservers()`, so that the current `Observers` of the `GameModel` will execute their `update` methods.



WelcomePanel is the initial view players are greeted with, where the player can choose either to login, register or exit the application.

RegisterPanel allows the user to create a new user with a default avatar. First it checks that username, password and verify are not empty and then checks that password and verify match before sending a message to the database to add the new player. If the database returns an error message the current **ViewDriver** will create a new error pane informing the player of the problem.

MainMenuPanel allows the player to choose the topic they wish to play and provides a number of other useful methods including editing their player information, accessing help, exiting the application, logging out and changing the background of the application.

EditInfoPanel allows the user to change their current username, password and avatar and updates the database accordingly. These changes are reflected elsewhere in the game, e.g. the game panel and the leaderboard.

LeaderboardPanel is the panel that players can view by pressing the **LEADERBOARD** button on the **MainMenuPanel**. On the pressing of this button, the client for this GUI requests information about the current state of the Leaderboards. This information is displayed through a **JTabbedPane**, which has 8 different tabs. The initial tab is the one referring to the cumulative scores for all game types for each user, thus referred to as the Global Leaderboard. Then the remaining seven panels are used to display the information for each game type scores only. Each individual Leaderboard panel is generated through a method **makeLeaderboard()**, so that it is not necessary to hard code each Leaderboard panel.

HelpMenuPanel displays information in a non-editable **JTextArea** on a **JScrollPane** regarding how to use the application and how it works.

PlayerPanel is an extended **JPanel** which contains the current user information such as their username, rank, total score or high Score, and avatar. This is slotted into other panels such as the **LobbyQueuePanel** and **GamePanel** for easy access to the variables and reducing redundant code.

GamePanel is the where the game action happens. It is divided into three panels including **playersPanel**, **questionPanel** and **answerPanel**. The **PlayersPanel** holds all **PlayerPanel** objects participating in the game as well as their scores. Both the **questionPanel** and **answerPanel** get their information from the database but the **answerPanel** displays the answers in a different order per client and properly updates which answer has been selected and made unavailable by which player.

GameScorePanel displays the players and their scores from highest to lowest after the current round of the game has ended.

4.7 server Package

The **server** package provides several useful methods to ensure that all the components in the system can cooperate properly. In general, the server is responsible for connecting the client and database, holding the game information, and maintaining the general running of the whole program.

Server is the main body of the server. It includes a thread pool, thread lock, some static field variables that are accessible to all the threads, and some static methods to operate on these field variables. When a client connects to the server, the server will start a new runnable object **ServerThread** to handle all of the incoming messages.

ServerThread contains part of the message handler methods covering **CONNECT**, **REGISTER**, **DISCONNECT**, **LOGOUT**, **PLAY**, **CHANGE_INFO**, **ANSWER**, **LEADERBOARD**, and **QUIT** message. After the establishment of the connection, **ServerThread** will start its listening loop. This loop



will not stop until the user logs out, the user disconnects, or an exception is thrown inside `ServerThread`.

- **Sign-in:** `Client` will send a `CONNECT` message containing the username and password. Once `Server` receives the message, `Server` will query the database to check if the username and password are correct. Only when they are both correct can the user log-in successfully, and `Server` will send a `GOOD_SIGN_IN` message. Otherwise, a `BAD_SIGN_IN` message will be sent to `Client`.
- **Sign-up:** `Client` will send a `REGISTER` message containing the username and password. `Server` analyses the message and starts querying the database to check if the username already exists. If the username is available for use by a new user, `Server` will send a `GOOD_SIGN_UP` message to the client. Otherwise, it will send a `BAD_SIGN_UP` message.
- **Logout:** In this case, there are two kinds of messages that the user can send to the server: `LOGOUT` and `DISCONNECT`. `Server` receives them and then removes this user from `activeUserList` and `gameLobbyList`. No message is sent back to the client. The server simply stops the listening loop and closes the corresponding socket if the message type is `DISCONNECT`, or maintains the socket connection if the message type is `LOGOUT`.
- **Change Information:** The `CHANGE_INFO` message signals that user wants to change their own information. A user is only allowed to change their username, password and avatar. Once the server detects the message, it will query and update the database. User verification is not needed here, because users cannot change their information without being already logged in.
- **Play:** Users send the `PLAY` message to find a match with their desired topic. When it receives this message, the server will call the message handler in the `Lobby` class to set this specific user to matching status, and the server will try to find a suitable lobby for this user. If there is no suitable lobby found, a lobby will start for this specific user.
- **Quit:** If the server receives a `QUIT` message, it will call the message handler in the `Lobby` class to remove this specific user from the lobby.

Above is the main duty of `ServerThread`; it constantly listens to the client and handles some of the messages itself. However, when it comes to the messages that relate to the lobby or game, the `Lobby` class will take charge, which is made of a thread lock, a timer, some static field variables as shared memory and many methods to handle different kind of incoming messages. It *cannot* listen to the client; instead, it completely depends on the `ServerThread` to call the method in the `Lobby` class. In this case, other than the timer thread no new thread is needed to handle the game lobby, as the `ServerThread` is reused.

- **User Joins Lobby:** When `ServerThread` detects the `QUIT` message, it will call the method inside the `Lobby` class trying to put this specific user into a suitable lobby. For each lobby, it will judge if the players number has reached the maximum, if the lobby has started the game, if the lobbys topic is equal to the users topic and if the user has already inside the lobby. Only when these four conditions are satisfied the user can join this lobby. Then `Lobby` class will send a `QUEUEING` message to this user telling the current players in this lobby and another `PLAYER_JOINED` message will be broadcast to all the other players telling that a new player has joined. And this users status will be set to `MATCHED`.



- **User Leaves Lobby:** When `ServerThread` detects the `QUIT` message, it will call the method inside `Lobby` to remove this user from the lobby. After the user is removed, a `PLAYER_QUIT` message will be broadcast. If the last user in the lobby has left, then this `Lobby` object will destroy itself.
- **Game Starts Mechanism:** After each player has joined, a method from the `Lobby` class will be called to check if the game should start. The game will start when there are four players joined. When game starts, a five second countdown will start, and a `GAME_COUNTDOWN` message will be broadcast in the lobby before throwing the first question. During this five second countdown, `Lobby` class will also initialize its game kernel, setting everything to default and sending a query to the database to get ten questions and their corresponding answer choices.
- **Game Mechanism:** Game will last for ten rounds. In each round, the mechanism is the same. First of all, server will broadcast `Q_AND_A` message to all the players and start a 15s countdown. And the `ServerThread` of each player will listen to their own masters answer. The first `ServerThread` that detects the players answer will call the `acceptChoice` method inside the `Lobby` class. The method will remember the the players choice ,broadcast a `FIRST_PLAYER_CLAIMED` message to all the other players. Any other answer comes after that will be remembered but a `PLAYER_CLAIMED` message will be broadcast to all the other players. If all the players have given their choices before the timer finishes, this round will automatically end. After this round is over, `Lobby` will reject any coming answer and starts to calculate the players score so far. Then `Lobby` will broadcast an `UPDATE` message that contains the current game score to notify clients that this round is over. It will break for 5s before the next round starts.
- **Game Ends Mechanism:** After ten rounds, the game is over. A `RESULTS` message containing the total score of each user will broadcast to all the players in the lobby. All the users status will be reset to `IDLE`, and `Server` will destroy this lobby after that.

5 Test Plan

Testing was always a consideration throughout the development of the application. As such, all team members individually tested their subsystems as they developed them, using main methods and JUnit tests (where applicable), and consistently discussed as a group different scenarios and use cases which could potentially affect or even break the system. The team chose to use JUnit 5 as its main form of white-box testing, as it was a form of testing that all team members were familiar with from the Software Workshop coursework assignments.

Once the team was confident with functioning of the smaller units within the system, it was time to begin integrating different components and testing them as a whole. Here is a more specific layout of the team's Test Plan: Note that the first table is for White Box testing, and the second is for the Black Box testing of the GUI.

Test Plan - White Box Testing				
<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
database				
Queries.isCorrect-Answer	answer is correct	true	true	pass
Queries.isCorrect-Answer	answer is incorrect	false	false	pass



Subsystem	Test Case	Expected	Actual	Pass/Fail
Queries.registerUser	username and password valid	true	true	pass
Queries.registerUser	username already exists	false	false	pass
Queries.checkUserNameAvailable	username is available	true	true	pass
Queries.checkUserNameAvailable	username is not available	false	false	pass
Queries.verifyUser	both username and password correct	true	true	pass
Queries.verifyUser	username incorrect	false	false	pass
Queries.verifyUser	password correct	false	false	pass
Queries.verifyUser	both username and password incorrect	false	false	pass
Queries.verifyUser	check username matching is not case sensitive	true	true	pass
Queries.verifyUser	check password matching is case sensitive	false	false	pass
Queries.changeUsername	change to another valid username	true	true	pass
Queries.changeUsername	check username matching is not case sensitive	true	true	pass
Queries.changeUsername	check username is stored with the casing desired by the user	true	true	pass
Queries.changePassword	change to another valid password	true	true	pass
Queries.changePassword	check username matching is not case sensitive	true	true	pass
Queries.changeAvatar	change to another avatar	true	true	pass
Queries.changeAvatar	check username matching is not case sensitive	true	true	pass
Queries.updateScoreAndRank	update overall score only	false	false	pass
Queries.updateScoreAndRank	update overall score and topic high score	false	false	pass
Queries.updateScoreAndRank	update overall score and rank only	true	true	pass
Queries.updateScoreAndRank	update overall score, topic high score and rank	true	true	pass
Queries.getTotalScore	normal case	123	123	pass
Queries.getTotalScore	check username matching is not case sensitive	0	0	pass
Queries.getTopicHighScore	normal case	5	5	pass
Queries.getTopicHighScore	check username matching is not case sensitive	5	5	pass



Subsystem	Test Case	Expected	Actual	Pass/Fail
Queries.getRank	normal case	2	2	pass
Queries.getRank	check username matching is not case sensitive	1	1	pass
Queries.topicTo-Column	normal case	“niso”	“niso”	pass
Queries.getTopic-Leaderboard	leaderboard of 3 rows	“1 2 username3 2 72 5 Username2 3 33 9 username4 9 2”	“1 2 username3 2 72 5 Username2 3 33 9 username4 9 2”	pass
Queries.getMain-Leaderboard	leaderboard of 3 rows	“1 5 Username2 3 10002 5 username5 5 5013 2 username3 2 222”	“1 5 Username2 3 10002 5 username5 5 5013 2 username3 2 222”	pass
Queries.createQuiz	create quiz of 5 random questions	N/A since random	test to be designed this week	N/A
client				
client.Client	CONNECT message sent	message header CONNECT with correct User	message header CONNECT with correct User	pass
client.Client	REGISTER message sent	message header REGISTER with correct User	message header REGISTER with correct User	pass
client.Client	LOGOUT message sent	message header LOGOUT with correct User	message header LOGOUT with correct User	pass
client.Client	DISCONNECT message sent	message header DISCONNECT with correct User	message header DISCONNECT with correct User	pass
client.Client	PLAY message sent	message header PLAY with correct User	message header PLAY with correct User	pass
client.Client	QUIT message sent	message header QUIT with correct User	message header QUIT with correct User	pass



<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
client.Client	CHANGE_INFO message sent	message header CHANGE- _INFO with correct User	message header CHANGE- _INFO with correct User	pass
client.Client	LEADERBOARD message sent	message header LEADER- BOARD with correct User	message header LEADER- BOARD with correct User	pass
server				
ServerThread	login successfully	GOOD_SIGN- _IN	GOOD_SIGN- _IN	pass
ServerThread	login unsuccessfully: wrong username	BAD_SIGN_IN	BAD_SIGN_IN	pass
ServerThread	login unsuccessfully: wrong password	BAD_SIGN_IN	BAD_SIGN_IN	pass
ServerThread	login unsuccessfully: al- ready logged in	BAD_SIGN_IN	BAD_SIGN_IN	pass
ServerThread	register successfully	GOOD_SIGN- _UP	GOOD_SIGN- _UP	pass
ServerThread	register unsuccessfully	BAD_SIGN_UP	BAD_SIGN_UP	pass
ServerThread	client disconnects dur- ing: before login	Socket Closed Exception	Socket Closed Exception	pass
ServerThread	client disconnects dur- ing: after login	Socket Closed Exception	Socket Closed Exception	pass
ServerThread	logout successfully	DISCONNECT	DISCONNECT	pass
Lobby	1 player joins lobby	QUEUEING	QUEUEING	pass
Lobby	2 players join lobby	QUEUEING and PLAY- ERJOINED	QUEUEING and PLAY- ERJOINED	pass
Lobby	1 player leaves lobby	PLAYER_QUIT	PLAYER_QUIT	pass
Lobby	2 players leave lobby	PLAYER_QUIT	PLAYER_QUIT	pass
Lobby	start of a game (not everyone answers every question): 0s: Start: GAME_COUNTDOWN	GAME_- COUNTDOWN	GAME_- COUNTDOWN	pass
Lobby	start of a round 1 (not everyone answers every question): 5s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	end of a round 1 (not everyone answers every question): 20s: UPDATE	UPDATE	UPDATE	pass



<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
Lobby	start of a round 2 (not everyone answers every question): 25s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	end of a round 2 (not everyone answers every question): 40s: UPDATE	UPDATE	UPDATE	pass
Lobby	start of a round 3 (not everyone answers the last question): 45s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	end of a game (not everyone answers the last question): 60s: RESULT	RESULT	RESULT	pass
Lobby	start of a game: 0s: Start: GAME_COUNTDOWN	GAME_-COUNTDOWN	GAME_-COUNTDOWN	pass
Lobby	start of a round 1: 5s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	during round 1: (Player A answers) 10s: FIRST_PLAYER_CLAIMED	FIRST-_PLAYER-_CLAIMED	FIRST-_PLAYER-_CLAIMED	pass
Lobby	end of a round 1: 20s: UPDATE	UPDATE	UPDATE	pass
Lobby	start of a round 2: 25s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	during round 2: (Player B answers) 35s: FIRST_PLAYER_CLAIMED	FIRST-_PLAYER-_CLAIMED	FIRST-_PLAYER-_CLAIMED	pass
Lobby	end of a round 2: 40s: UPDATE	UPDATE	UPDATE	pass
Lobby	start of a round 3 (not everyone answers the last question): 45s: Q_AND_A	Q_AND_A	Q_AND_A	pass
Lobby	during round 3: (Player A answers) 50s: FIRST_PLAYER_CLAIMED	FIRST-_PLAYER-_CLAIMED	FIRST-_PLAYER-_CLAIMED	pass
Lobby	during round 3: (Player B answers) 55s: PLAYER_CLAIMED	PLAYER-_CLAIMED	_PLAYER-_CLAIMED	pass
Lobby	end of a game: 55s: RESULT	RESULT	RESULT	pass
Lobby	client disconnects during: lobby	Socket Closed Exception and PLAYER_QUIT	Socket Closed Exception and PLAYER_QUIT	pass



<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
Lobby	client disconnects during: lobby countdown	Socket Closed Exception and QUEUEING	Socket Closed Exception and QUEUEING	pass
Lobby	client disconnects during: two person game	Socket Closed Exception and PLAYER_QUIT and RESULT	Socket Closed Exception and PLAYER_QUIT and RESULT	pass
Lobby	client disconnects during: more than two person game	Socket Closed Exception and PLAYER_QUIT	Socket Closed Exception and PLAYER_QUIT	pass
Lobby	clients press play button simultaneously	Join in the same lobby	Join in the same lobby	pass
guiSwing				
GameModel	should update the state of the Model accordingly	true	true	pass
GameModel.set-BooleansFalse	should update the panel booleans to false	true	true	pass
GameModel.enter-LoginMenu	should set the panels booleans to false, but set loginMenu to true	true	true	pass
GameModel.enter-WelcomePage	should update the panel booleans to false	true	true	pass
GameModel.enter-MainMenu	should set panels booleans to false, but mainMenu to true, and set user to the user parameter	true	true	pass
GameModel.enter-RegisterMenu	boolean false passed - should set all panel booleans to false except for registerMenu and invalidUsername	true	true	pass
GameModel.enter-RegisterMenu	boolean true passed - should set all panel booleans to false except for registerMenu	true	true	pass
GameModel.enter-EditInfoMenu	set all panel booleans to false except for edit-Menu	true	true	pass
GameModel.enter-HelpMenu	set all panel booleans to false except for help-Menu	true	true	pass



<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
GameModel.loginUser	should set all panel booleans to false except for mainMenu, and set user to the user passed	true	true	pass
GameModel.enter-LobbyQueue	set all panel booleans to false except for lobby-Menu, all users in user array should be added to usersInLobby with score 0 - empty array	true	true	pass
GameModel.enter-LobbyQueue	set all panel booleans to false except for lobby-Menu, all users in user array should be added to usersInLobby with score 0 - single array	true	true	pass
GameModel.enter-LobbyQueue	set all panel booleans to false except for lobby-Menu, all users in user array should be added to usersInLobby with score 0 - single array	true	true	pass
GameModel.remove-UserFromLobby	should remove User from the usersInLobby Map	true	true	pass
GameModel.user-JoinedLobby	all users in the User array should be added to the map of usersInLobby	true	true	pass
GameModel.enter-GameScreen	all panel booleans set to false except gameMenu, and removedButton set to -1	true	true	pass
GameModel.end-GameLobby	all panel booleans set to false except main-Menu and usersInLobby an empty map	true	true	pass
GameModel.enter-Leaderboards	all panel booleans set to false except for leader-board, and theleaders field set to the leaders array - empty array	true	true	pass
GameModel.enter-Leaderboards	all panel booleans set to false except for leader-board, and theleaders field set to the leaders array - single array	true	true	pass



<i>Subsystem</i>	<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
GameModel.enter-Leaderboards	all panel booleans set to false except for leader-board, and the leaders field set to the leaders array - multiple array	true	true	pass
GameModel.start-Countdown	all panel booleans false, except the countdown set to true	true	true	pass
GameModel.set-Background	all panel booleans set false, set the background field variable	true	true	pass
End of White Box Test Plan				

Test Plan - Black Box (GUI) Testing			
<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
the first viewed page	a page with a title and buttons for Login, Register and Leave functions	the frame is as expected	pass
a dynamic background on the welcome page	a moving background on the welcome page	a static background	fail
music on the welcome page	relaxing music to play on opening the welcome page	no music plays	fail
the login page should have a back button	when pressed the welcome page appears	the welcome page appears as expected	pass
pressing the Quit button	the frame should close	the frame closes as expected	pass
pressing the Login button	the frame should update to a new Login page	the frame updates as expected	pass
a failed login	a new frame will appear and notify the user as to why they cannot login	a new frame appears as expected	pass
being able to type on the login page	we are able to type into the login field and password field, with password field not displaying the characters	the frame updates as expected	pass
pressing the Register button	the frame should update to a new Register page	the frame updates as expected	pass
the register page should have a back button	when pressed the welcome page appears	the welcome page appears as expected	pass
incorrect registration details	a new frame will appear notifying the user as to why they were unable to register	a frame appears notifying the user how to correctly format a username and password, and as to how their registration may have failed	pass



Test Case	Expected	Actual	Pass/Fail
a main menu should have buttons for playing a game, viewing leaderboards, editing players, help, quit and logout	all the buttons present	all the buttons as expected	pass
a ComboBox to choose the type of game	the user can choose the game type from a ComboBox in the page	the user can choose game type as expected	pass
a change Background button	we expect a new frame to appear on pressing this	a new frame appears	pass
pressing apply on the new frame spawned with a chosen background	the background should change on the frame to the new background only even in other panels	the frame updates as expected	pass
pressing the play button	a lobby page must appear on the frame	a lobby page appears, but there are some weird results, some "artefacts" are appearing on various panels and needs to be fixed	fail
all players join the lobby	the top panel on the frame will update displaying a get ready message to all users	the frame updates as expected	pass
press the back button on the lobby	the frame will update to display the main menu, and for other user the user who left will be removed from the lobby	the frame updates as expected	pass
the game starts	the frame will update displaying the first question and four buttons with answers printed on them, and players on the side	the frame updates as expected	pass
pressing an answer button	the frame will update with the button clicked as yellow, in addition the button will add the text "claimed by you" at the end, and the users avatar	Most of this updates as expected, however the users avatar does not appear	fail



Test Case	Expected	Actual	Pass/Fail
another user presses an answer button	the frame will update with the button clicked as cyan, in addition the button will add the text "claimed by juser;" at the end, and the users avatar	Most of this updates as expected, however the users avatar does not appear	fail
the game finishes	the frame will update to display all of the users with a results page, with the winner attached a gold medal	the frame updates as expected	pass
pressing the back button on the results page	the frame will update to display the main menu	the frame updates as expected	pass
pressing the leaderboard button	a leaderboard button should appear on the frame displaying the top 10 users	the frame updates as expected	pass
leaderboard display tabs	when pressing tabs on the leaderboard, it will display the leaderboard for that type	the frame updates as expected	pass
pressing the leaderboard back button	the frame should update to the Main Menu page again	the frame updates as expected	pass
pressing the edit button	the frame should update to the edit page	the frame updates as expected	pass
the edit page	the edit page will have 3 text fields allowing the user to change their username and password, and a JComboBox to display and change avatars	the new frame appears as expected	pass
press change info button on edit info page	a new frame will appear to notify the user that these changes are final and ask them to press a button if they want to go ahead	the new frame appears as expected	pass
pressing the help button	the frame to update with some text displaying information on how to play	the frame updates as expected	pass
pressing the back button on the help page	the frame to update and display the main menu	the frame updates as expected	pass
pressing the quit button	the frame should close	the frame closes	pass



<i>Test Case</i>	<i>Expected</i>	<i>Actual</i>	<i>Pass/Fail</i>
pressing the logout button	the frame should update to the welcome page	the frame updates as expected	pass
End of Black Box (GUI) Test Plan			

6 Evaluation and Conclusion

The team was enthusiastic from the start (and some times perhaps a little too ambitious). From the start of the project, every member has contributed a great deal, playing to each individuals strengths. As you can see from our test plan (see Section 5), the application still has room for improvement, however the current state of this game is in a solid state where it would perform sufficiently in the majority of cases. We have a clean and simple GUI, presenting relevant information to the user effectively. The game itself presents questions and a scoring system based on time.

Possible features that we could implement to expand on this include:

- Allow users to add their own questions for each topic.
- Expanding on the previous point, a “Create Your Own Quiz” feature would allow users to create their own quiz topic all together.
- Introduce a single-player mode which allows user’s to play, even without a human opponent.
- Give lecturers the opportunity to become administrators so they can add questions to quizzes and see the full table of leaderboards.
- Add the notion of levels to each question. This would be implemented by dividing the questions into different difficulties, and hence tailor each player’s quiz questions based on their experience level. This implementation can be incorporated easily as the question bank table already includes a level column (note that currently, every question is set to level 1).

These features allows the game to be scalable.

Overall, the application is a functioning, fun and competitive game. It allows users to sign up, connect, edit their own profiles, view leaderboards and compete against others in lobby based quiz games.

A Project Diary

A.1 Tuesday 20th February, 2018 (15:00 - 16:00)

All in attendance, Scrum Master: Edmond

Agenda: Decide on a project.

Minutes: All in agreement to design and build a game. Possible game ideas:

- Minecraft
- Tank-style games



- Platformer (e.g. Mario)
- Quiz-Up/Revision App
- Scrabble/Rummikub

Actions:

1. Email Alex about Minecraft to see if it is too ambitious.
2. Log into the database.
3. Finalise a game idea.

A.2 Friday 23rd February, 2018 (15:00 - 16:30)

All in attendance, Scrum Master: Ashton

Agenda:

1. Finalise a project idea (or at least a plan for the weekend and finalise by Monday).
2. Get everyone connected to the database via JDBC.
3. Get everyone on subversion with a short tutorial on it (look at Canvas).
4. Set realistic weekly targets.
5. Distribute and assign workloads.

Minutes:

1. *Project idea:* All vote against Minecraft on the grounds that it is too large a project for such a small space of time. From previous group meeting, Scrabble and Quiz-Up were both suggested. Scrabble might be too difficult once we try to implement all the different rules, and a proper valid dictionary. Quiz-Up has opportunity for good use of a database, multi-player capabilities, if we have time we could add extra features like a chatroom, points allocated based on speed of answer, connect to social media? Lots of potential with a realistic goal. Agreed to be doable within the time frame we have. All agree to do Quiz-up!
2. *Database connection:* Connected the team to the database through a Java program uploaded on svn. See svn for username and password.
3. *Subversion:* Demonstrated basic functionality of svn. Team to watch Canvas tutorial on using it over the weekend.
4. *Weekly targets:* refer to spreadsheet on Google Drive.
5. *Work distribution:*
 - Requirements - Edmond
 - Prototype class diagram - Andrew
 - User stories - Ashton
 - Draft architecture - Ningyuan



- GUI - Richard

Actions:

1. Watch svn Canvas tutorial.
2. Refer to spreadsheet for assigned jobs.
3. Think of name for game.

A.3 Monday 26th February, 2018 (15:00 - 16:30)

All in attendance, Scrum Master: Andrew

Agenda:

1. Project title: Super Magic Fairy Fun Quiz?
2. Using svn in Eclipse.
3. Go over requirements.
4. Go over architecture.
5. Go over class diagrams.
6. Assign subsystems to team members and start coding.

Minutes:

1. *User stories*: Scenarios have been created for students on MSc conversion, but what other kind of users are we going to have? Main users would be students, but game has potential to be used by anyone with an interest in Computer Science. More possible users could be lecturers or teachers in computer science who might want to use the app, compete against students, add their own quizzes. Or even students on the Advanced MSc?
2. *svn*: Everyone to use same add-on in order to connect Eclipse to Subversion (called 'Subversive'). Spent time connecting to svn, critical so we can start programming.
3. *Project title*: Stick with Super Magic Fairy Fun Quiz for now; think of new one as we build.
4. *Diagrams*: Make sure we use draw.io
5. *Single versus multi-player*: Would it be a good idea to play against an AI or for a player to train on their own in single player mode? Potentially we will add multiple AIs which players can play against, depending on the user's rank, so they can play against an AI of varying strength. AI will be a subclass of User.

Actions:

1. Ashton to complete draft user stories for alternative users for tomorrow.
2. Everyone to get their head around svn and Eclipse so that Alex can help us in tomorrow's tutorial if we still need it.
3. Add questions for quizzes to the question bank on Google Drive as we go.
4. Set up SVN on Eclipse on everyone's machine.

**A.4 Tuesday 27th February, 2018 (15:20 - 16:00)**

All in attendance, Scrum Master: Edmond

Agenda: Discuss after meeting with Alex. Project idea too simple. Need to implement some use of Threads/concurrency.

Minutes:

1. Changing idea slightly - we will now implement only a multi-player game, and ditch the AI idea from our project. Threads will be utilised to lock an answer once one user has chosen it, to prevent other players from having the same answer. Potentially could add a chat functionality to the game to implement the use of another thread if we have enough time. ArrayLists could be used here.
2. Ready to start programming! Code delegation:
 - Server - Ningyuan
 - Client - Andrew
 - Communication protocol - Edmond
 - Database - Ashton
 - GUI - Richard

Actions: We have split up the work between ourselves as above; our aim is to attempt to make an interface for our subsystem, and then try to implement some of our program also. Alex would like to see a working implementation of the GUI log-in/register and its functionality with the database for Tuesday 06/03/18.

A.5 Thursday 1st March, 2018 (14:00 - 14:45)

All in attendance, Scrum Master: Ashton

Agenda: Check status of everyone's coding. Is it all coherent. Does everything interact as intended?

Minutes:

1. *Database:* Ashton has made some good progress on the databases. We wonder if there should be a total score leaderboard and a rank leaderboard. Suggest it would be good to facilitate both.
2. *Sending objects:* Potentially use GIFs of fairies as objects? Or users could have avatars as fairies?
3. *Leaderboards:* Decided on a ranking system.
4. *Personal Testing:* Everyone to make sure they constantly test their own methods as they go. Just use main method for now to ensure everything works as intended. JUnit later on.

Actions:

1. Ed to send email to Alex asking if we need to send objects or just Strings.



2. Type up scoring system so it is clear.
3. Continue the sick work we are doing!

A.6 Friday 2nd March, 2018 (14:00 - 18:30)

All in attendance, Scrum Master: Ashton

Agenda: Double check everyone is on same page regarding scoring system, and how the game works.

Minutes: Today's meeting was more of a coding session.

1. *Database:* Ashton worked on filling up the database with data and connecting to it.
2. *GUI/Client:* Richard and Andrew discussed how the GUI will link with the client object.
3. *Communication protocol:* Edmond, Ashton, Andrew and Ningyuan discussed and finalised how the communication protocol should work between the server and the client (both ways).
4. *Server/Client:* Andrew and Ningyuan managed to get the server and client to send serialised objects to each other and read from them successfully.
5. *NOTE:* Richard had to leave early due to increasingly bad weather conditions.

Actions: Continue to develop our individual subsystems and keep testing too.

A.7 Tuesday 6th March, 2018 (11:00 - 12:00)

All in attendance, Scrum Master: Edmond

Agenda:

1. Server/lobby architecture.
2. Prepare what we will show Alex.
3. Test plan.
4. Database alterations.

Minutes:

1. *Server/lobby architecture:* Go through with Alex. Build many small servers inside the big server. Each small server represents a specific game and controls everything that relates to a game, including communication and the timer. Instead of having a new thread to control the communication stuff of the game lobby, maybe we could reuse the ServerThread (which is used to communicate with a specific client) to do the message broadcasting job in the form of triggering-action. We store the ObjectOutputStream of each player in the game lobby object. Every time when a player sends a message, the ServerThread that is responsible for listening to this thread will modify the game kernel, broadcasting messages to other players or maybe even start a new thread to start timer.



2. *Demo at meeting later:* Try to show Alex a working Server Client GUI with sign-up and log-in.
3. *Test plan:* Formalise the test plan for each subsystem in the Google Drive. Write it in plain English for now; we can re-implement the code for this later on.
4. *Database:* Altered users table to contain avatar column. We will implement Ashton's prepared statements that she has written in her package.

Actions:

1. To ask Alex later:
 - Can we use JavaFX?
 - Sever/lobby architecture
 - Can we create a ClientListenThread *first*, and then this creates a Client? We are having issues with extending our class.
2. Consider implementing a 'Forgot Password' function. This will require our users table to have an email address column, and a Java class that sends an automatically generated email to the user's corresponding email address, where the email pulls the correct password from the database to send to the user.
3. Consider building an interface to be implemented by server containing all the methods to query the database.

A.8 Tuesday 6th March, 2018 (14:00 - 14:45)

All in attendance. Scrum Master: Ashton

Agenda: Demonstrate what we have so far to Alex.

Minutes:

1. Demonstrated successful connection to database and log-in via GUI.
2. Went over what each person has done so far.
3. Asked Alex questions discussed in our group meeting earlier today - see actions below.

Actions:

1. Make username non-case-sensitive.
2. Remove properties class and replace with .properties file.
3. Convert to JavaFX?
4. Make ER diagram for database and ensure all group members have a thorough understanding of the database.
5. Look into publish-subscribe messaging pattern with regards to the complications in the architecture and threads.
6. Meeting with Alex next week at 14:30 in LG04, where he expects to be able to play our game.
7. Ashton to start taking SQL queries for ServerThread and turning them into Prepared-Statements; create a new class for all the statement methods.



A.9 Saturday 10th March, 2018 (20:00 - 20:45)

All in attendance, Scrum Master: Andrew

Agenda: Go through all the subsystems, give updates, and state what is left to do.

Minutes:

1. *GUI:* Log-in and log-out functions work. Register function still needs some work. Play function gets to the right screen, we just need to update it accordingly as other players join. Then we can start working on the game mechanics.
2. *Client:* So far, the client can login, logout, register, enter a lobby, and send an answer. The client needs to be able to handle changes of information which we can implement later. This is not a priority. The next thing for the client to do is to receive questions from the server.
3. *Server:* At the moment, the lobby has a timer function and can manage many messages from many threads. The server package also has a server monitor to see the current status of the whole server (including any details of particular lobbies). The server should now implement the register protocol, and plenty of testing too. The server should ensure that it can cooperate with all other systems.
4. *Database:* So far, the database has a .properties file which is loaded as needed, contains relevant connections to our database. It also has methods for prepared statements which the server will call. It remains to tidy up current methods, add a few more Prepared-Statement methods and also return questions and answers from database.

Actions:

1. See above for the relevant subsystems to be developed.
2. Test plan and report to begin writing as this is a substantial chunk of our mark.

A.10 Tuesday 13th March, 2018 (14:00 - 15:00)

All in attendance, Scrum Master: Ningyuan

Agenda:

1. Present our prototype application to Alex.
2. Get feedback and decide what the next steps are.

Minutes:

1. *Demonstration to Alex:* Demonstrated general server/client communication, user registration and log-in using a connection to the database, and played a demo 3-question quiz. Showed Alex the users table, and the code for checking username and password validity which is in the Client class. Explained how you score points (i.e. points for correct answers, deductions for incorrect answers and no marks for abstaining). Showed Alex the Queries class containing all our PreparedStatements. Showed our report outline via Google Drive.



2. *Feedback:* Alex is happy with what we have showed him and the direction in which we are going. See actions below for next steps.
3. *Test plan:* Alex says it is important to talk about what our testing plan before we started coding; demonstrate our awareness of mock objects, dummies, and use of different testing methods (e.g. JUnit, test stubs, GUI, etc...).

Actions:

1. Change opening photo of system.
2. Figure out which order players appear in the lobby depending on who logs in first, second, third etc. Should ideally appear in the order of people joining the game.
3. When an answer has been taken by a user, ideally change the colour of the button (instead of the text) to indicate that the button has been taken, so users can still see all the answer choices, but still know which answers are now unavailable to them.
4. Write code for displaying Leaderboard.
5. Make whether youve actually pressed a button clearer.
6. In the report, talk about how the class diagram has changed; put latest class diagram in report (not scrapped diagrams - put these in the appendix!). It is a GOOD thing that we have a diagram from before we started coding, so we can talk about how we had a plan but also how the project has changed as we progressed.
7. Download Eclipse UML Generators to create class diagrams.
8. Check password validity pattern - make it less complex.
9. Fix drop down topic selection button so that users can actually choose a topic, and its not just default set to data structures.
10. Options button: Ability for user to Change username, Change password, Change avatar (Query methods for these all already exist, just need implementing).
11. Implement protocol for log-out.
12. Backing out is an issue, not working at the moment (e.g. Ed goes into a lobby, then press back, be he cant play again). Also Nenos GUI didnt update.

A.11 Thursday 15th March, 2018 (15:00 - 15:30)

All in attendance, Scrum Master: Richard

Agenda:

1. Things we might get asked in the Viva: Avatar selection (in-game selection versus player menu).
2. GFX Prototypes (drawn - annotate; mockup - Draw.io and implement annotations)
3. Debugging: Have a button on the welcome screen which automatically logs in four users (starts three other clients) and starts a lobby or game for testing.



4. OptionsPanel - Player should be able to (at a minimum) change their username, password and (?) delete their account.
5. GamePanel - clean up GFX
 - Add player's avatar to claimed question.
 - Make sure answers are shuffled using the shuffleAnswers method
 - Use the dedicated player panel.
 - Some answers too long for the answer button (fix via JLabels assigned over buttons? Or fix via XML...Richard self-learning.)
6. LobbyQueuePanel - Clean up GFX
 - Fix ghosting panes issue on LobbyQueuePanel
 - Rename variables in a meaningful manner.
 - Javadoc and comment everything!
7. TestPlan (WhiteBox/BlackBox)
8. Testing
9. Report write-up.

Minutes:

1. *User objects*: Edited user class; added getStringRank method. Went over various constructors in user class.
2. *Log-out*: Currently Client sends disconnect message when click log-out button. It is sending the wrong message right now - must fix.
3. *Database connection*: What happens if database cant be reached? Right now we have to completely reboot client if database connection unsuccessful. Can we deal with this with a try/catch?
4. *Presentation & laptops*: Is it going to be an issue using laptops for our presentation? Will they be able to connect? If java version is not the same between two computers, might be issues with Serialisable object.
5. *Report*: People write about their own sections. Read over each others additions to check for grammar/spelling. Focus on the system as it is now, not as much on how it was developed.

Actions:

1. Ask Alex if we need a PowerPoint for the Tuesday presentation (what will the practise demo entail?) ...*UPDATE*: Yes we do need a powerpoint presentation - it is essentially a rehearsal!
2. Add avatar to User constructor, or simply create new constructor.
3. Find avatar images (of lecturers/famous computer scientists).
4. Clarify if its okay to use lecturers images with Alex.



5. Get to work on testing.
6. Client/server testing to be in one package, then database in one package and GUI on its own (make new testing packages!).
7. Test connection between laptops.
8. Put report in third person.
9. Add another protocol on server end for logout instead of disconnect.
10. Look at UML Eclipse tool to create class diagrams.
11. Potentially change home screen background to something more clear? Let Richard know if you find a good photo in 1024x768 resolution.

B System Requirements

1. Database table with questions:
 - a. The database must contain tables with questions, 4 possible options and an indication to the correct answer per topic.
 - b. The database could allow the users to add questions to the database.
2. Database table with user information:
 - a. The database must contain a table to store individuals users' (unique) usernames and passwords.
 - b. The database must allow the registration of new users.
 - c. The database must contain information about the user's rank.
 - d. The database must contain information about the user's high scores for each topic.
 - e. The database must contain information about the user's total (cumulative) score.
 - f. The database must contain information about the user's avatar.
3. Architecture:
 - a. The server must communicate to the database to retrieve and add information from all of the tables.
 - b. The clients must communicate to the server through the communication protocols stated below.
 - c. The users must be able to connect to a server.
 - d. The system must use a communication component for managing the communication protocols.
 - e. The system must have a displaying component for the GUI and the sound.
 - f. The system must have a database component which contains tables for questions in each topic and user information.
 - g. The system must have a game kernel which contains all the information in a game.
 - h. The system must have a ranking and matching component.
4. Communication Protocols:



- a. The clients must be able to establish a connection with the server and update the GUI accordingly.
- 5. Multi-player functionality:
 - a. The application must allow users to connect to one other player to compete.
 - b. The system must match players who are doing the same topic against each other.
 - c. The system should match similar players against each other.
- 6. Main menu:
 - a. The system must display a PLAY button.
 - i. The PLAY button must allow the user to then select a topic.
 - b. The system must display a LEADERBOARD button.
 - i. The leaderboard should allow the user to filter the leaderboards by rank, topic.
 - c. The system should display a PLAYER button.
 - i. This must display all information about the user to the user such as number of wins and losses, best topic, number of times played...
 - ii. This must allow a user to change their information such as their password, user-name or avatar.
 - d. The system must display a HELP button.
 - i. The HELP button should give a brief description of the game, walking the user through on how to play the game.
 - e. The system must display a QUIT button.
 - i. The QUIT button must close the connection between the client and the server.
 - f. The system could display Uday as a magic fairy on the main menu.
- 7. Quiz:
 - a. The system must allow a user to choose a topic.
 - b. The system must display only questions from that topic.
 - c. The system must implement a timer for scoring - the quicker a user answers, the more points are awarded.
 - d. The system must display the question and set of answers for a maximum of 15 seconds. The questions and set of answers will no longer display after 15 seconds are up or if all players answers within that time.
 - e. The system should display a random permutation of the 4 answer options.
 - f. The system must not allow more than one player to select the same option, based on a first come first served basis (requiring synchronised threads).
 - g. The system must use 10 randomly chosen questions per game.
 - h. The system must display how many points each player has after every question.
 - i. The system must display the final scores after the game has ended.
 - j. The system should allow for a REMATCH feature, allowing users to play the same player again in the same topic.
 - k. The system must give bonus points at the end depending on the standings.

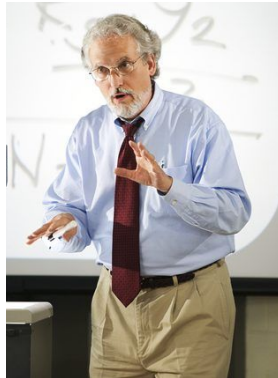


8. Miscellaneous:


- a. The system could allow users to design their own avatars.
- b. The system could allow users to choose their own avatars from a bank of preselected images such as Lecturers or Famous computer scientists.
- c. The system should implement sound effects when users are competing.
- d. The system should implement sound effects when users are navigating through the application.




C User Stories

	Christopher Peacock	
	Tagline: "Remember to scrum"	
	Background: <ul style="list-style-type: none">- Age: 50- Lecturer in computer science at the University of Birmingham.- Expert in computing but not so experienced with latest apps/smart-phone features.	Main points: <ul style="list-style-type: none">- Has lectured for many years for many different modules and year groups.- Has a huge passion for Computer Science.- Is interested in new ways to engage students, increase course satisfaction and graduate employability.
Goals: <ul style="list-style-type: none">- To incorporate new ways to engage students in the material taught in lectures.- To gain an idea of how much interest students are taking in his module.- To gain an idea of how much of his module that students are understanding.	Narrative: <p>Christopher is a widely-respected and experienced lecturer at UoB in the school of Computer Science. He teaches modules taken by a wide variety of computer science students. He has a huge love for the subject and a passion for teaching and research.</p> <p>While he always invites students to ask him any questions they might have after lectures or in his office hours, he knows some students might be too nervous or hesitant to do so. He also knows many students are unlikely to go home and immediately go over their notes after a lecture. He wants to provide them with a way to go over the lecture notes and cement their comprehension quickly.</p>	
Frustrations & Pain Points: <ul style="list-style-type: none">- Students getting poor marks on tests.- Students complaining that the module is going too fast.- Canvas.- Panopto.	<p>He is very aware of the ever-increasing prominence of technology and thinks any application that will help students to practise and revise content learned in their degree is a good idea.</p>	
Scenarios: <ul style="list-style-type: none">- Chris has just finished teaching the tricky topic of recursion to his MSc Conversion students. A QuizUp app would be a quick and easy way for students to test their knowledge, and the opportunity to compete against the lecturer would engage more players.- Chris's Software Workshop module has a class test coming up in two weeks. He knows students have expressed that they feel nervous for it, and wants to give them a way to quickly revise and test their understanding of the material to be covered on it.		



	James Jameson	
	Tagline: ‘Java is love Java is life.’”	
	Background: <ul style="list-style-type: none">- Age: 20- Studying BSc Computer Science at University of Birmingham.- Good level of experience using computers and apps.- Finalist.	Main points: <ul style="list-style-type: none">- Has a Samsung S8 phone, uses Facebook, Snapchat and Twitter.- Works part-time at the Guild of Students.- Prefers to complete any necessary group work during the day, but revise/do work alone, often with late nights in the labs.
Goals: <ul style="list-style-type: none">- To get a first in his degree.- To apply to and obtain a place on a Masters course.- To balance his studies with his part-time job.	Narrative: <p>James is a studious third year computer science student. Some of his modules he shares with students on the MSc CompSci conversion course.</p> <p>James thoroughly enjoys what he studies and works hard. He has a part-time job at the Guild and as a result is often studying late into the evenings to make up for lost time.</p> <p>James enjoys trivia and goes to pub quizzes with his housemates. He thinks a revision quiz app would be a fun way to revise for his degree and compete against his peers.</p>	
Frustrations & Pain Points: <ul style="list-style-type: none">- Noisy people in labs.- Someone beating him at his favourite subject.- Not feeling prepared for assessments.		
Scenarios: <ul style="list-style-type: none">- James has suddenly just remembered he has a test coming up in Artificial Intelligence. He wants to budget what little time he has to revise wisely, and so wants a quick way of seeing which topics he needs to revise.- James is quite a competitive person, and wants to see what the high scores are in his favourite computer science topics, so that he can work to beat them.- James’s current username is JamesJameson, but he has decided he does not like it and wants to change it to something more catchy.		

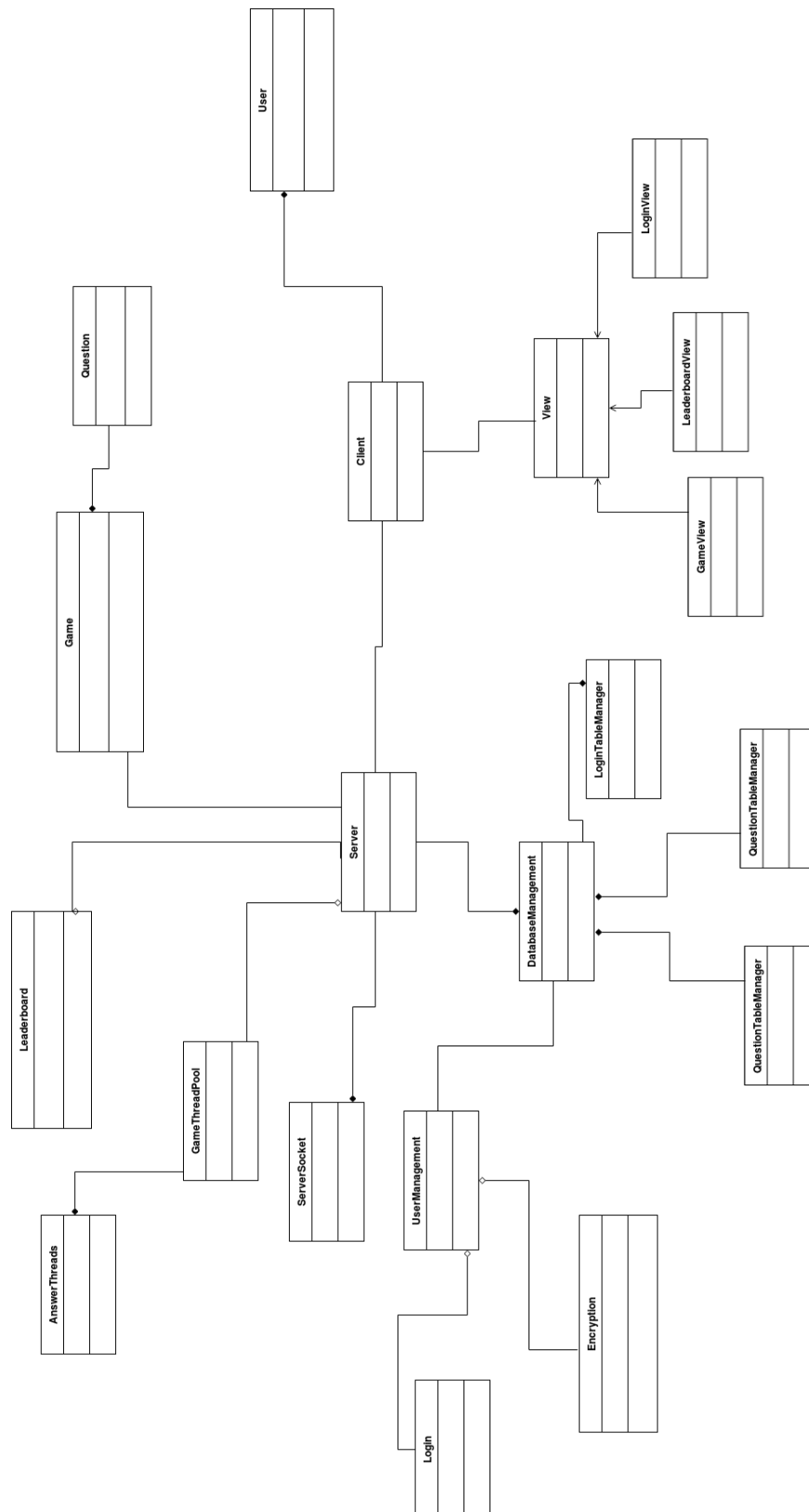


	Charlotte Wilson	
	Tagline: 'Study sesh later on?'	
	Background: <ul style="list-style-type: none">- Age: 22- Studying MSc CompSci conversion at University of Birmingham.- Plays volleyball.- Good level of experience using computers and apps.	Main points: <ul style="list-style-type: none">- Has the latest iPhone and several social media accounts.- Has several different friendship groups.- Has a busy schedule balancing Master's degree and sport.- Loves coffee and a good night out.- Revises best in groups.
Goals: <ul style="list-style-type: none">- To get a Distinction in her degree.- To get onto a graduate scheme in London.- To still make the most of her final year at university.	Narrative: <p>Charlotte is a postgraduate on the MSc Computer Science conversion course. She loves socialising and knows many people both on and outside of her course. She works better in study groups rather than on her own, and likes to go for lunch with friends or drinks after lectures with course-mates.</p> <p>Charlotte is always on the go. She either walks or gets a bus to uni, is a bit of a coffee addict, has volleyball matches every Wednesday, many of which involve her travelling on a coach for an hour, and she visits her boyfriend on the train at weekends.</p> <p>Like most young adults, Emily has a smart-phone and is very good at using it. She has Facebook, Instagram, Snapchat, and WhatsApp, along with a separate app for her university email, and Canvas. She is often updating her story and has several different group chats on the go.</p> <p>Her undergraduate degree was in mathematics, but she does not have any previous experience with actual computing or programming prior to this course. Because of this, Charlotte finds that she grasps certain concepts in her degree rather quickly, but really struggles with others.</p>	
Frustrations & Pain Points: <ul style="list-style-type: none">- Wasting time.- Not feeling prepared for assessments.- No wi-fi/internet connection.- Reading through long complicated lecture notes.- Not having enough exercises available to test her understanding of a module.		
Scenarios: <ul style="list-style-type: none">- Charlotte is on a coach headed back to Birmingham after her away volleyball match in Nottingham. It's late in the evening and too dark for her to read her notes; however, she still wants to be able to revise for her Software Workshop in-class test in two days' time.- Charlotte has got a Databases assessment coming up soon and wants to be able to quickly test her current understanding of SQL over the weekend while she is visiting her boyfriend.- Charlotte and her friend want to revise together and quickly test each other's knowledge before they do their Data Structures coursework.		



D Class Diagrams

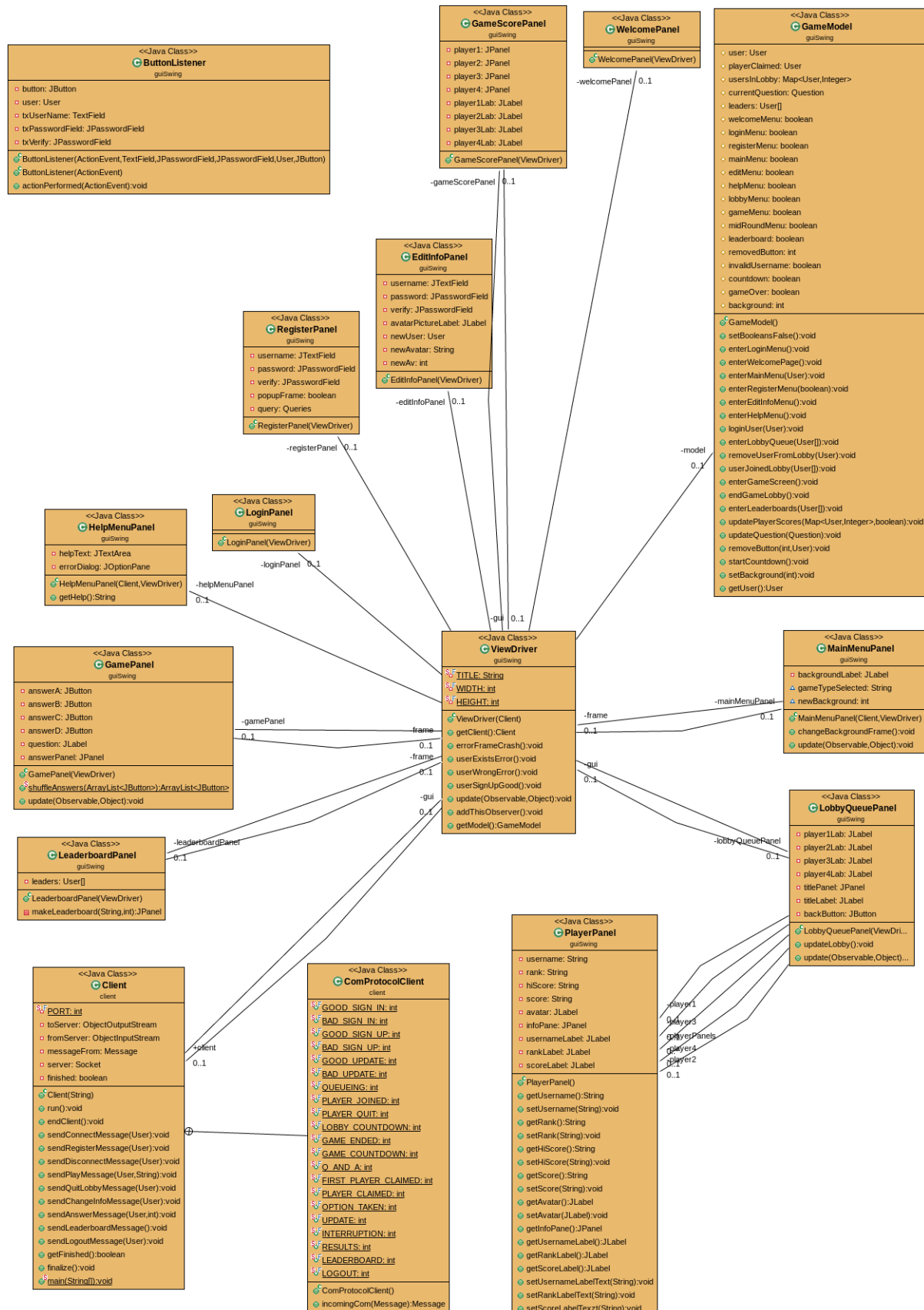
D.1 Original Class Diagram





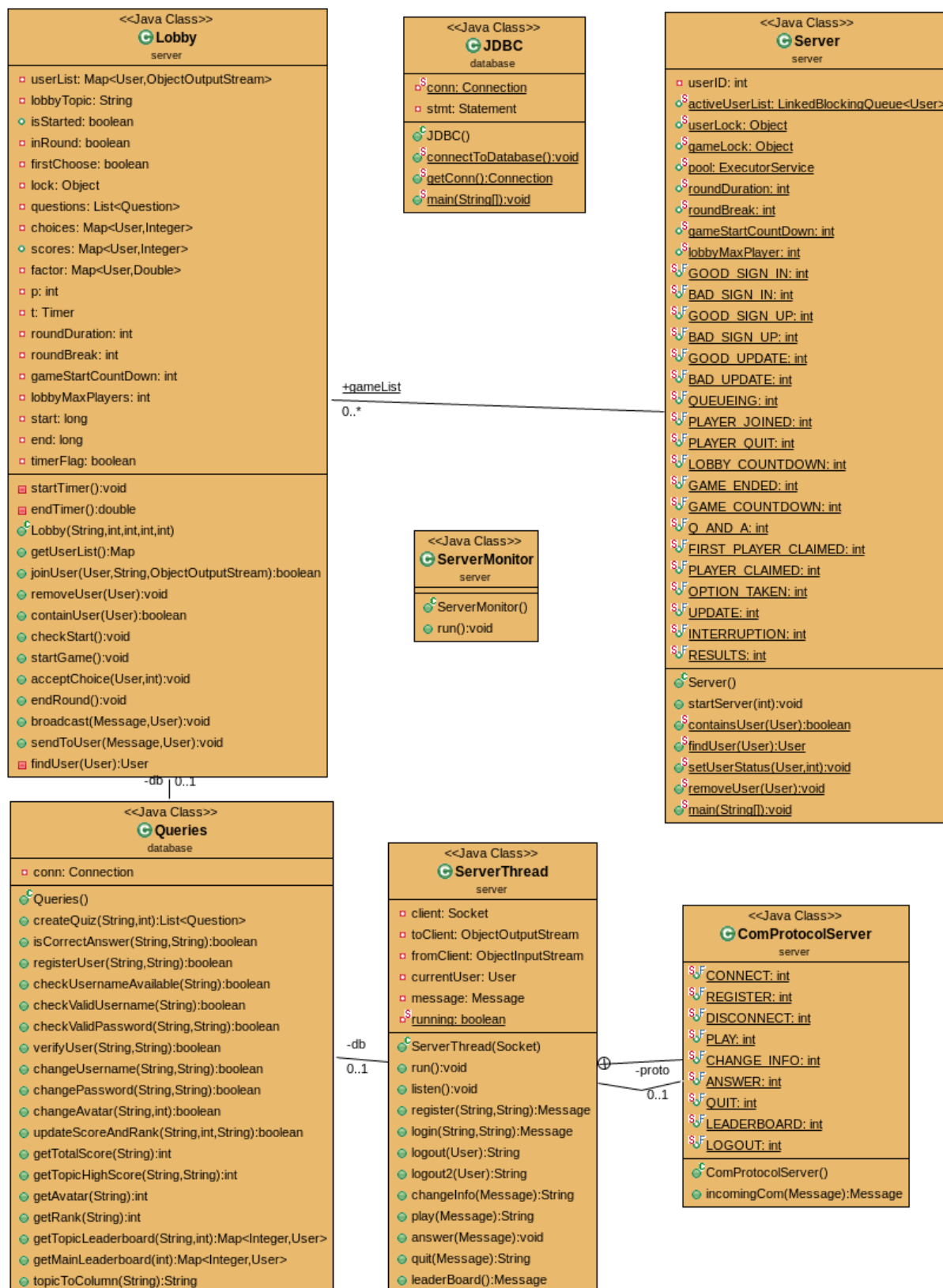
D.2 Final Class Diagrams

D.2.1 Client and GUI Class Diagram



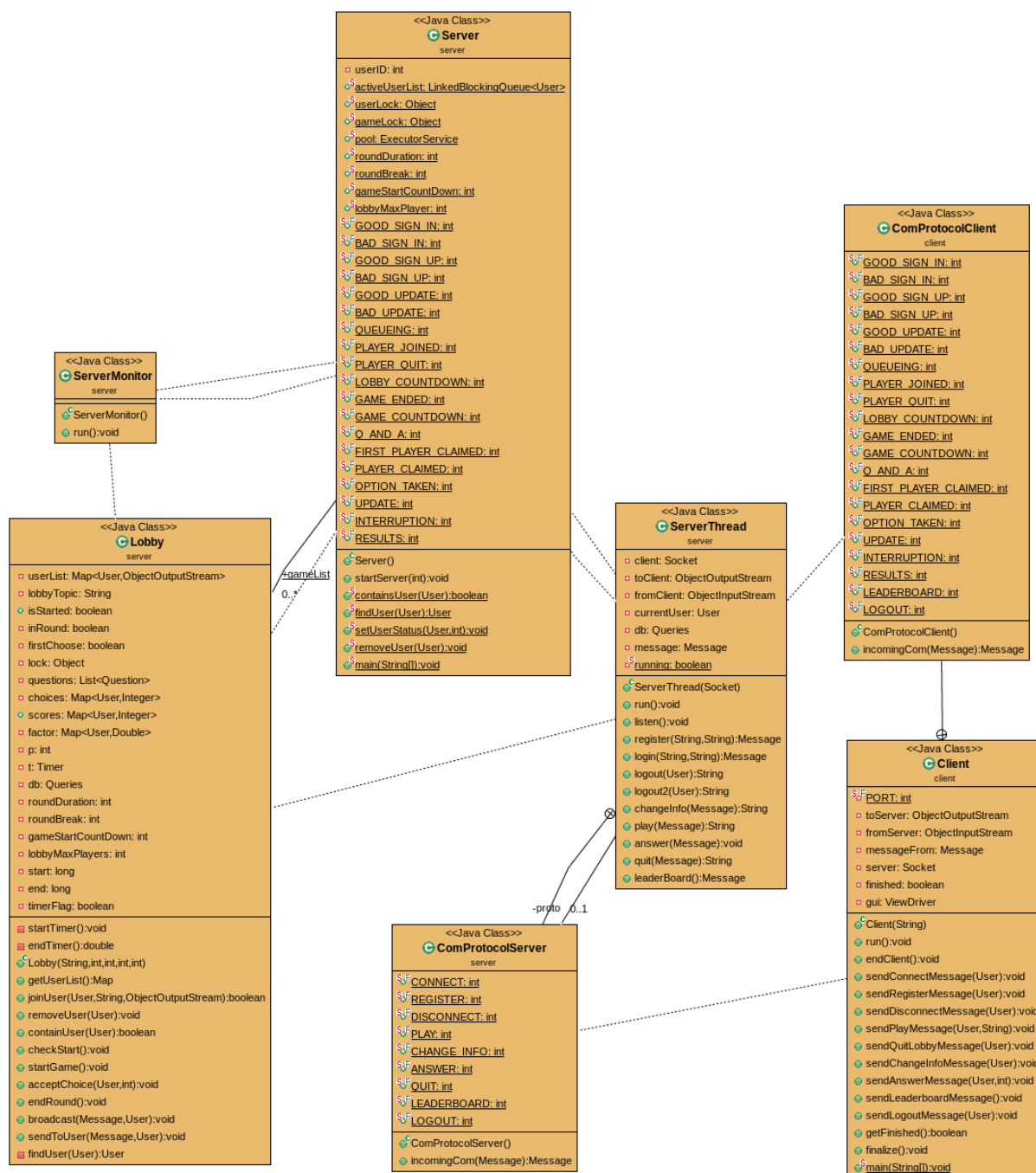


D.2.2 Database and Server Class Diagram



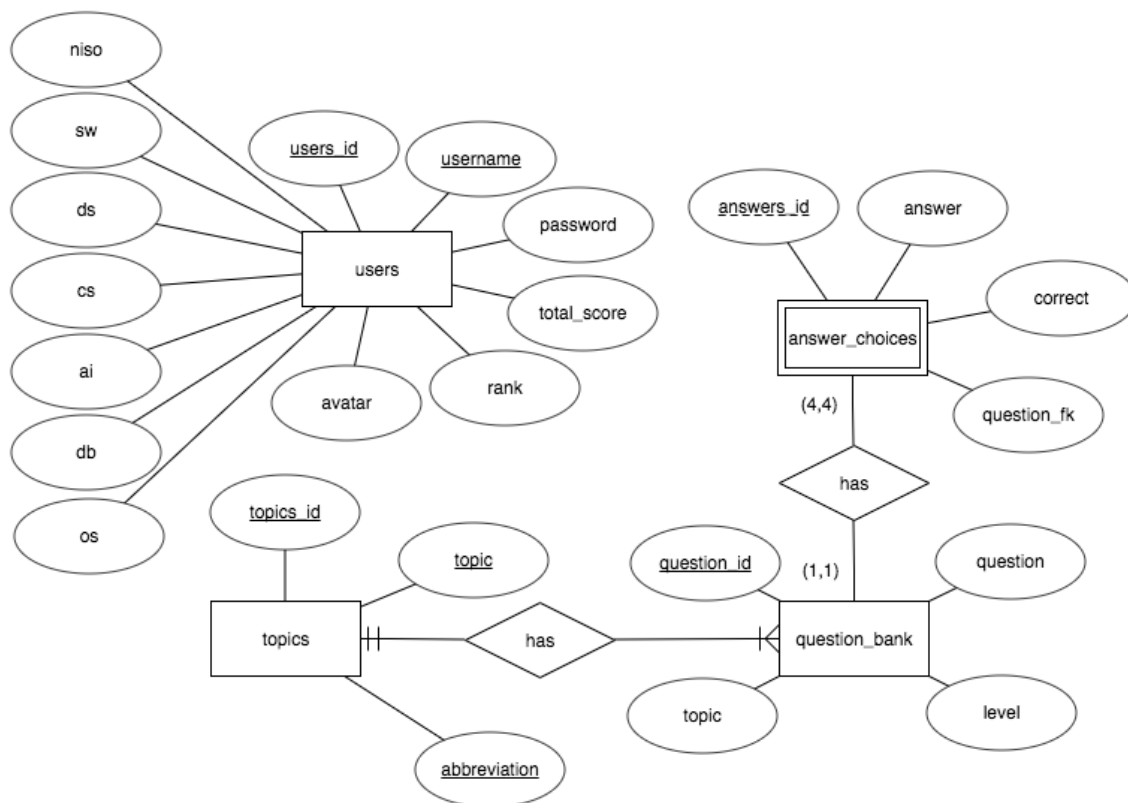


D.2.3 Server and Client Diagram





E Database ER Diagram



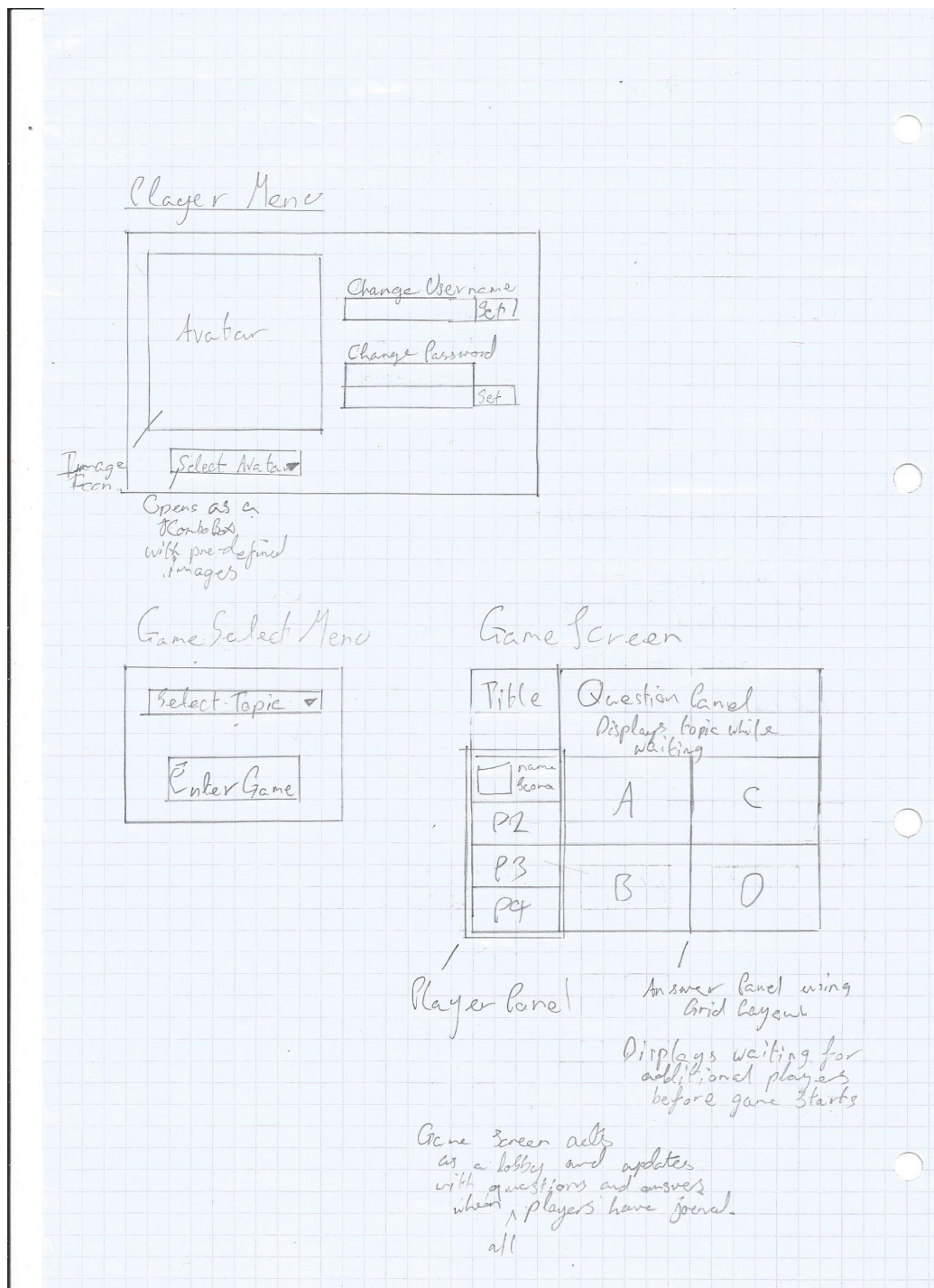
F GUI Development

F.1 Rationale for the Prototypes

PaperGui is just a quick sketch of ideas for the main functional screens of the GUI. From this we incremented ideas towards the functionality of the second prototype (modelled in Draw.io). Suggested improvements for the second prototype include having separate screens for the Welcome Panel and Login Panel (suggested by Andrew). This was due to the possibility that a new user would attempt to register a new user on the login panel, hit register, and then retype their credentials in the register panel as the information would not be passed between panels which could frustrate the user. Edmond suggested that we should use a dedicated Lobby panel and Game Panel instead of combining the two. The reason for this was provide a smoother, less ambiguous game gui where it would be obvious to players that they were no longer waiting for the game to start and that it had begun.



F.2 Paper GUI



ATHENS QUICK PROTO TYPE

Login Menu

Welcome to
Title

Username
password

Sign In Sign Up

JTextField

JPasswordField

JButton

Sign Up button redraws
frame and adds a verify
JPasswordField

Sign In button changes to
Register and Sign Up
button changes to Cancel.

Main Menu

Keyboard

Top
s

Players

Play

Player

Quit

JMessageField

JOptionPane

JButton

JButton

JOptionPane

Opens a panel
with top, s, and
game button

Player button
allows player
to change
player details

Non-editable

cell's data from
Database



F.3 GUI Prototype

MSc Computer Science Revision Quiz

○○○

Welcome to
Super Magic Fairy Revision Quiz!

Sign InSign Up

MSc Computer Science Revision Quiz

○○○

Register

Username: Password: Verify: RegisterBack

MSc Computer Science Revision Quiz

○○○

Login

Username: Password: RegisterBack

MSc Computer Science Revision Quiz

○○○

Main Menu

PlayLeaderboardPlayerHelpQuit

MSc Computer Science Revision Quiz

○○○

Leaderboard

1. PlayerName : HighScore Rank
2. PlayerName : HighScore Rank
3. PlayerName : HighScore Rank
4. PlayerName : HighScore Rank
5. PlayerName : HighScore Rank
6. PlayerName : HighScore Rank
7. PlayerName : HighScore Rank
8. PlayerName : HighScore Rank
9. PlayerName : HighScore Rank
10. PlayerName : HighScore Rank

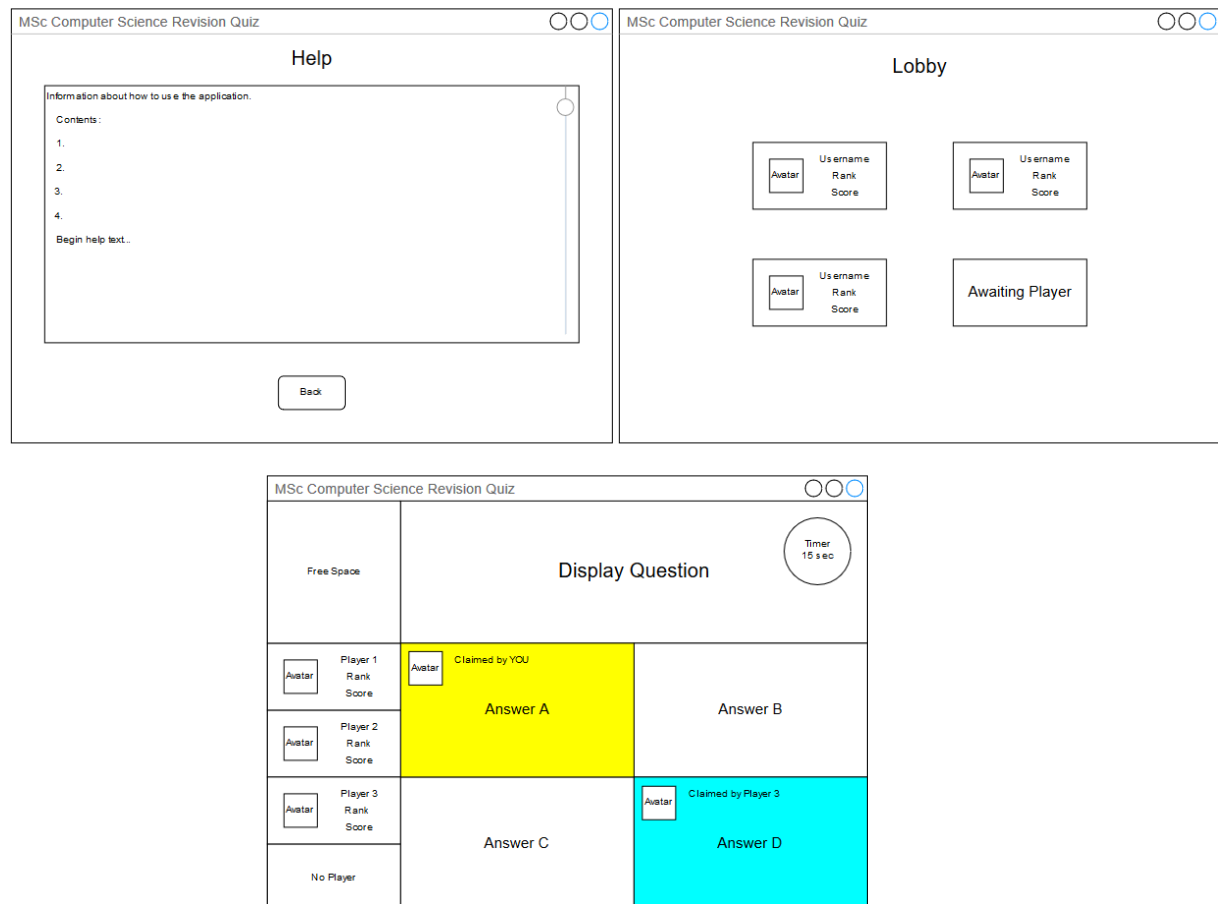
Back

MSc Computer Science Revision Quiz

○○○

Edit Player Info

Avatar
avatar_file
Username: Password: Verify: ApplyBack



G Contribution Statement

I was responsible for designing and building the tables within our database, and subsequently writing and testing Java code to enable the server to query the database, through the implementation of the JDBC API and the use of `PreparedStatement`s.

— Ashton HILLS

Cooperated with Edmond Wong, I mainly contributed to the **Server** coding task. I was responsible for providing message handling methods for different protocol messages and maintaining the running of the server.

— Ningyuan SUN

For the first part of the project, I worked on the **Client**, and how it would connect to the GUI. As this job did not take too much time, once the bulk of it was done, I assisted with the GUI.

— Andrew TALBOT

Worked the original design and prototypes of the GUI layout, configuration and display. I added some game logic. Cooperated with Parker for the revised GUI visuals and its functionality.



— Richard WARNER

For the first half of the project, I mainly took the secretarial position to ensure that the team had a clear direction from the start. I was thankful to be assigned a motivated and hard working team, so it seemed beneficial for the group that I took the organisational role. I worked mainly with Andrew and Ningyuan as my main component was the communication protocol. After the foundations of this procedure were put into place, I helped out on other areas of the project such as the GUI, the database and general debugging.

— Edmond WONG

References

- [1] Uday Reddy. Prof. uday s. reddy. <http://www.cs.bham.ac.uk/~udr/>, 2018.
- [2] wikipedia. Publish- subscribe pattern. https://en.wikipedia.org/wiki/Publishsubscribe_pattern/, 2018.