Hermes in Bytecode

A Messenger in Java

Software Workshop
Team Marathon

Viola Marku
Meng-Jung Lee
Ruoyu He
Yunxiao Zhuang
Jochen Stüber

School of Computer Science
University of Birmingham
20th March 2017

Contents

List of Figures

Statement of Contribution

All team members contributed equally.

Viola Marku Meng-Jung Lee Ruoyu He Yunxiao Zhuang Jochen Stüber

1 Introduction

Within the scope of this project, we have implemented a messenger application based on the client-server architectural paradigm. Like the messenger of gods from Greek mythology, alluded to in the title of this report, our application delivers messages in near real-time. Once users have created an account, they can log in, see other users who are online, and initiate chats with them (*à deux* or in a group). Furthermore, users can retrieve their chat histories, revisiting what has been said in the past.

Regarding architecture, model-view separation has been used to decouple client and GUI using the Observer-Observable pattern. Client and server communicate using an application protocol based on the popular open source format `JSON`. Finally, a `SQL` database is used to persist application data.

Providing an overview of the report's structure, we begin by describing the system requirements. Following that, system design is discussed based on various diagrams created. From this follows a textual specification of each application component. Then, we cover testing, laying out our test plan comprising both automated and user testing. Subsequently, we evaluate the product, discussing both benefits and concerns. Finally, we list our references and provide our team organisation report in the appendix.

2 System Requirements

During the initial project phase, we elicited a series of functional and non-functional requirements which are specified in the following.

2.1 Functional Requirements

Sign-in and sign-up

1. The user can only log into the system after creating an account.
2. The user can create an account with a unique user name, and a password.
3. The user name has to contain 4 to 12 characters (inclusive).
4. The password has to contain 8 to 20 characters (inclusive).
5. The system notifies the user if the chosen user name is already taken.
6. The system notifies the user upon successful creation of the account.
7. The system notifies the user if wrong credentials have been provided for login.
8. The user can log out from the system.

Sending and receiving messages

9. The user can see all the users that are online, excluding himself/herself, in a list of online users.
10. The system adds and removes users from the online list based on whether they are online or not.
11. The user can initiate a conversation with any online user visible in the online list.
12. The user can initiate group chats with multiple online users.

13. When a user initiates a new chat with one or more other users, a new chat is opened and displayed to all participants after the user has sent the first message.
14. Messages are delivered synchronously, i.e. immediately after sending.
15. The sender and sending time are displayed with each message of a chat.
16. If a group chat participant closes the chat or goes offline, the group chat is terminated for all participants.

Viewing chat histories

17. The user can select among and view previous chat histories.
18. A chat history is identified by the participants of the chat. All messages sent between those participants are part of the same history.
19. When a chat history has been selected for display, all messages sent in the chat are displayed in chronological order and in the same format that is used in chats.

2.2 Non-functional Requirements

Security and privacy

20. The system is secure, requiring a password to log in.
21. Passwords are entered in password fields, i.e. not displayed in the clear.
22. Passwords are not sent over the network as clear text.
23. Users can only view messages of chats that they have participated in, thus protecting privacy.

Performance

24. The system is fast, offering almost real-time messaging.
25. The system is reliable, gracefully handling, and notifying the user of, any errors or when the connection to the server cannot be established or is lost.
26. The system is scalable, creating a new server thread for each connected user.

Standards

27. The system is platform-independent, running on all systems for which a Java Virtual Machine exists.
28. The system's protocol is based on a widely used open standard.
29. The system interoperates with a SQL database.

Usability

30. The system provides consistent feedback about its state to the user.
31. The system employs a simple design that is intuitive to use, adhering to common usability conventions.
32. The system prevents errors by not allowing invalid operations and asking for confirmation when consequences may be costly.

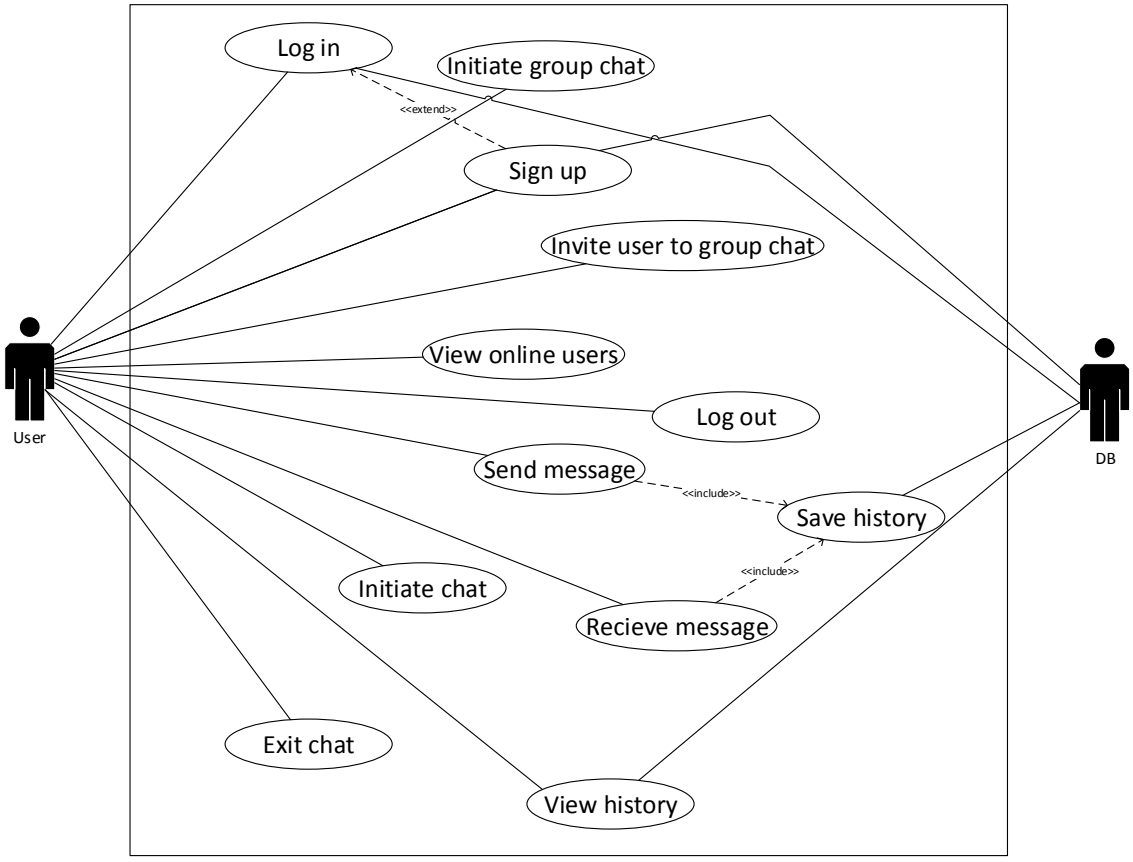

Figure 1: Use case diagram

3 System Design

Based on the specified requirements, we have designed the system using proven software engineering methodologies. All diagrams created during that phase use the Unified Modelling Language (UML). Firstly, a use case diagram was created (Figure 1), followed by activity diagrams (Figure 2 to Figure 5) which detail out the system's core use cases. Drawing on insights from those efforts, a set of interfaces was created, defining the contracts between system components. In the course of the development process, all interfaces and classes were documented in a class diagram (Figure 6). For the database, an entity-relationship model has been created (Figure 10) which is included with the database specification.

3.1 Use Case Diagram

The use case diagram (Figure 1) shows the system's use cases and their interrelationships. The user, and the database server were identified as external actors with which the system interacts.

3.2 Activity Diagrams

The interaction within the system, as well as between the system and external actors is specified in the following activity diagrams.

3.2.1 Sign-up

Firstly, the activities involved in signing up are specified in Figure 2.

3.2.2 Sign-in

Secondly, the activities involved in signing in are specified in Figure 3.

3.2.3 Send and Receive Messages

Thirdly, the activities involved in sending and receiving messages are specified in Figure 4.

3.2.4 View Chat History

Finally, the activities involved in viewing a chat history are specified in Figure 5.

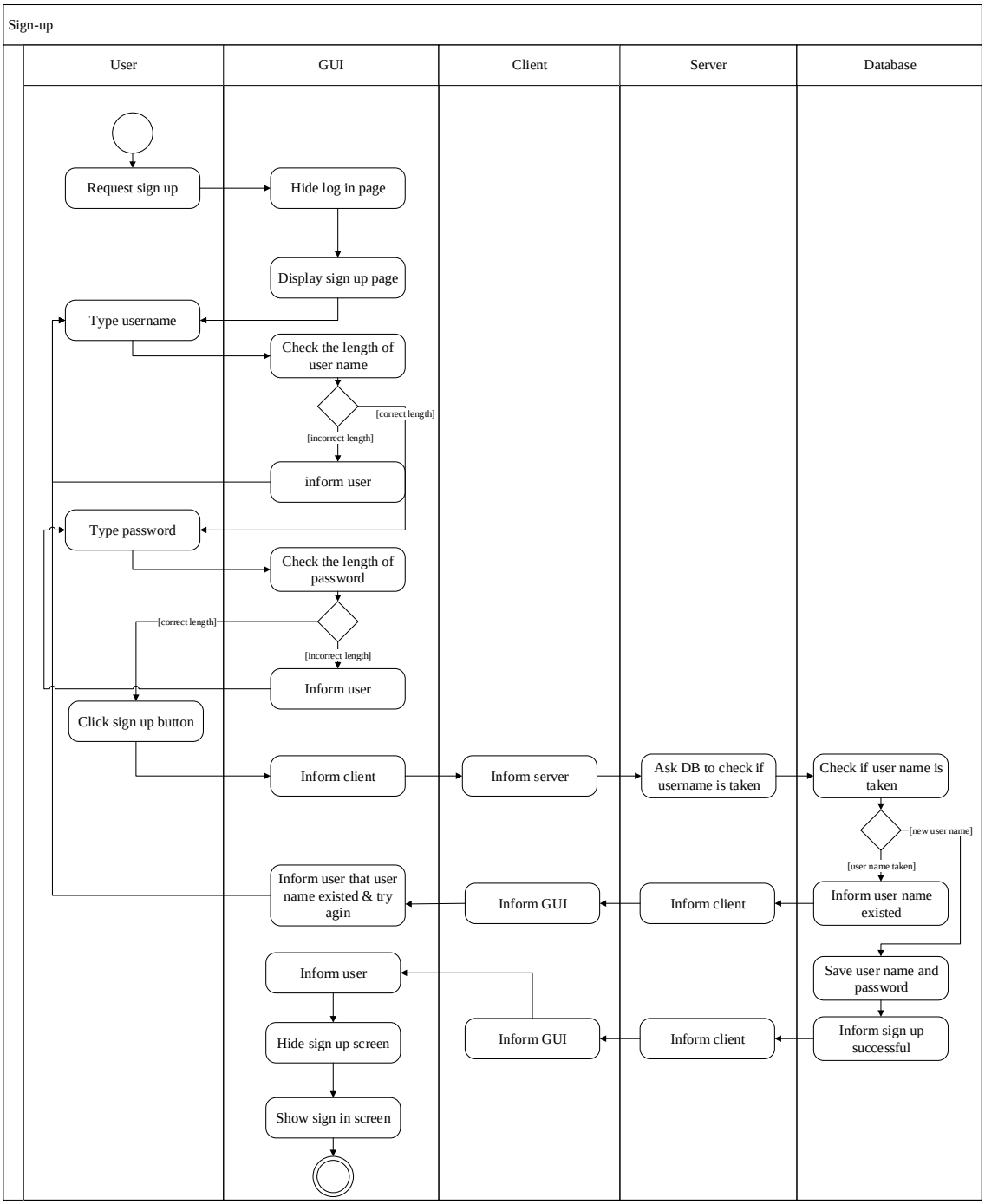

Figure 2: Activity diagram for signing up

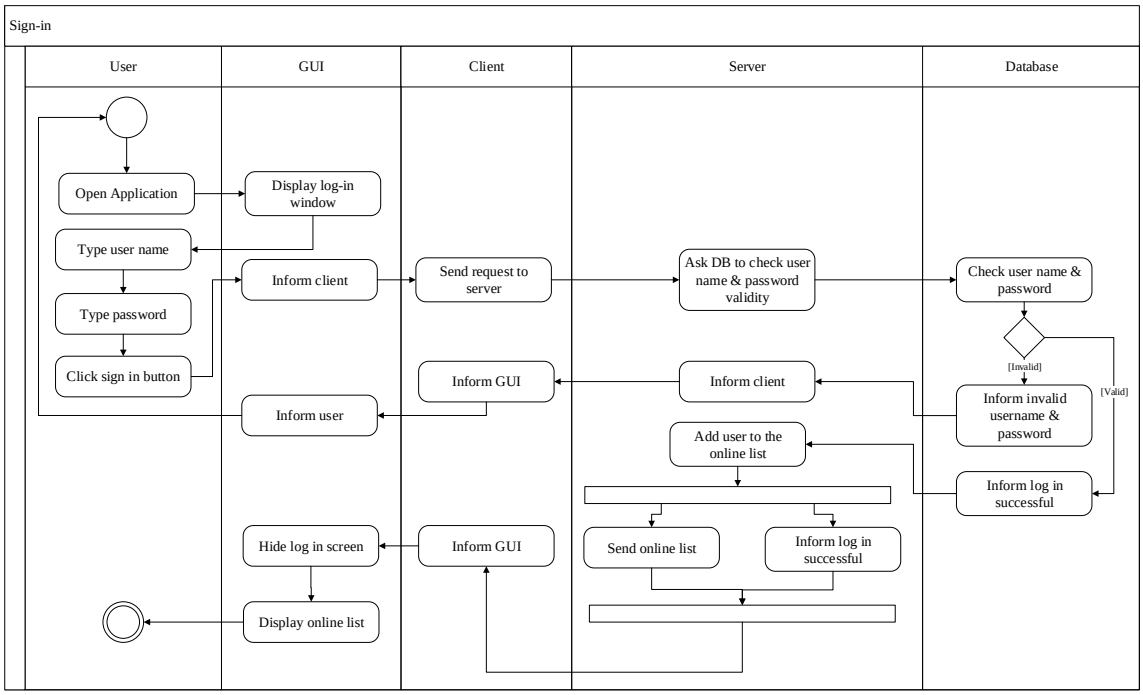

Figure 3: Activity diagram for signing in

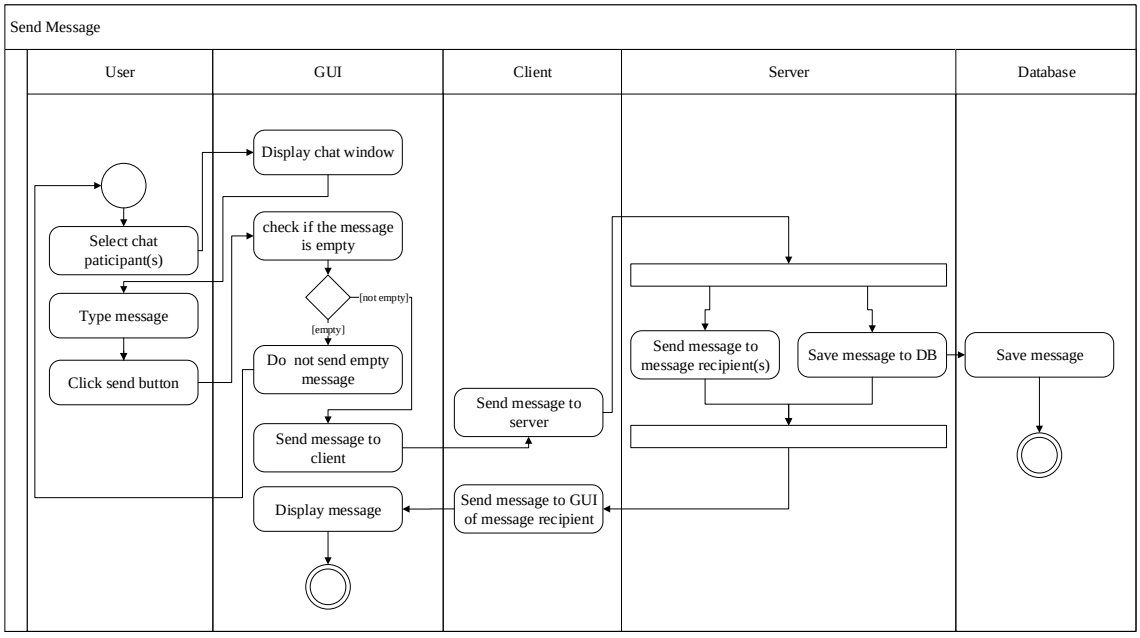

Figure 4: Activity diagram for sending and receiving messages

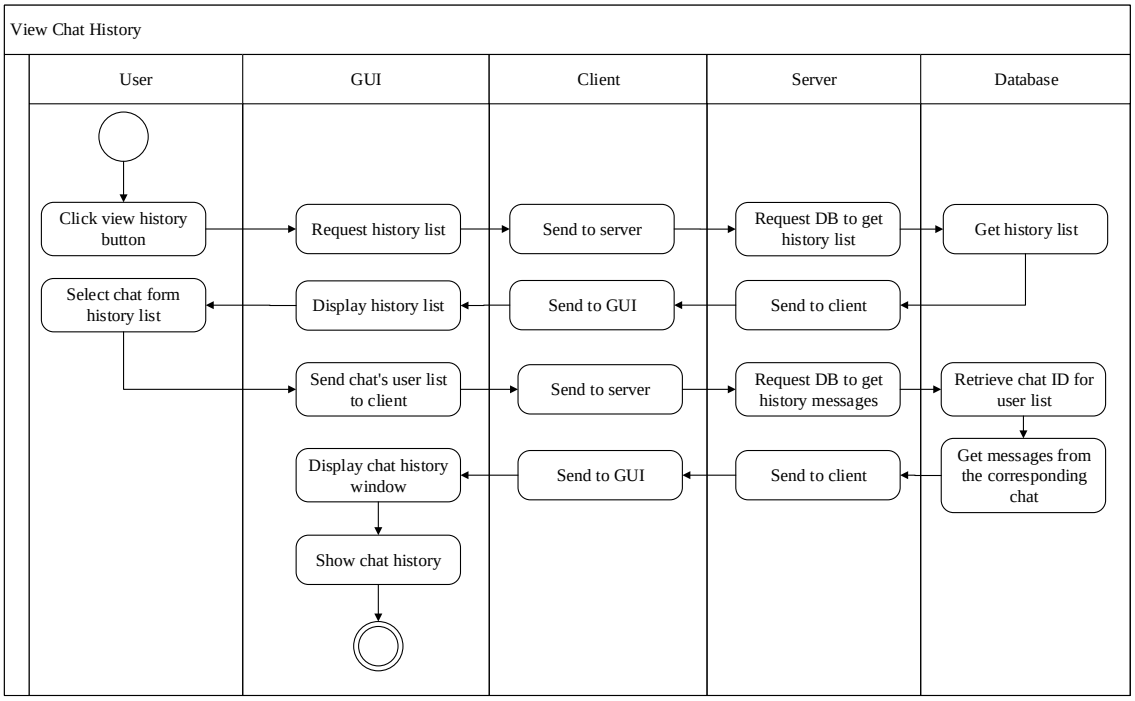

Figure 5: Activity diagram for viewing a chat history

3.3 Class Diagram

All classes and interfaces of the system are shown in the class diagram in Figure 6. For detailed reference, we recommend to view the diagram in the PDF version of this report which allows zooming into the high-resolution image.

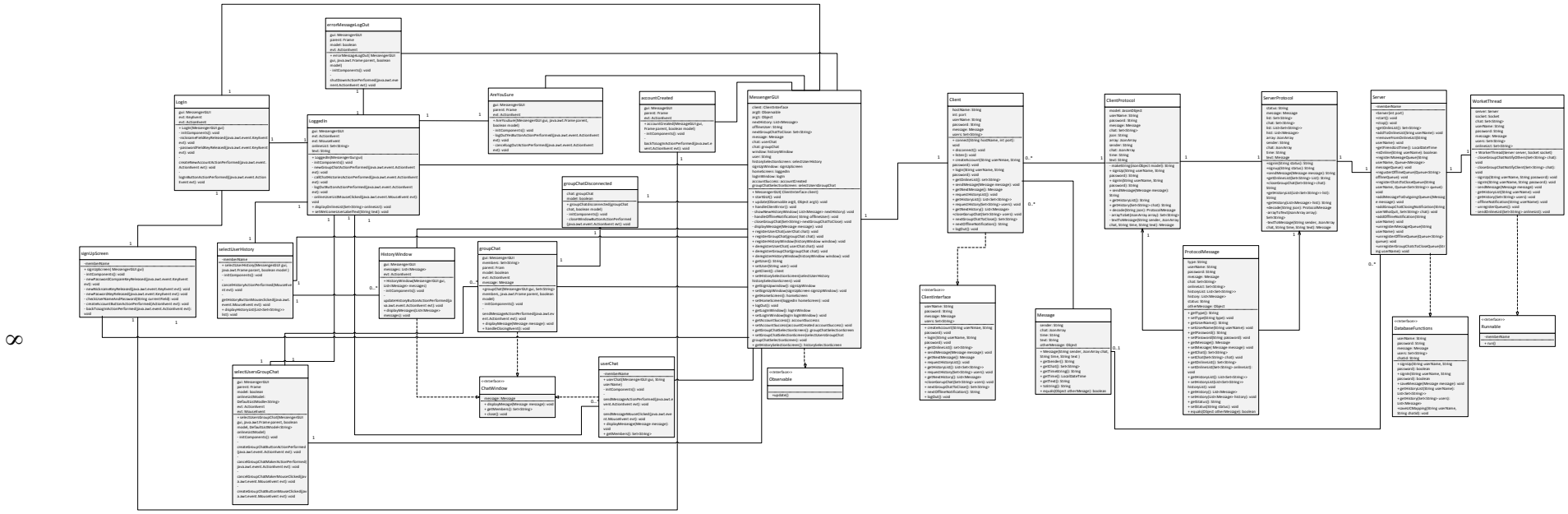

Figure 6: Class diagram

4 System Specification

Within this section, the individual system components are specified. Implementing a client-server architecture, the system can be divided into a client and server component on the first level. On the client side, model-view separation was implemented based on the Observer-Observable pattern. Hence, the client side of the application is separated into a view (the GUI), and a model (the client). On the server side, a client-facing and a database-facing component can be distinguished.

4.1 GUI

4.1.1 Overview

For our Messenger application, we decided to take inspiration from Windows' MSN design and interaction. MSN used a puristic design, based on rectangular frames, separating a home screen from individual chats. We adopted that design as a benchmark due to its simplicity, usability, and aesthetic virtues. Based on the activity diagrams, we outlined the main user interactions that occur within the application and used that information to identify a set of main windows and dialogues.

With regard to model-view separation, a class `MessengerGUI` was developed that implements `Observer` and is registered with the model. Within its `update` method, the `MessengerGUI` receives status values from the model which are defined in the interface `GUI`. That interface is not implemented as it is equivalent to the `Observer` interface. It is only used for documentation. GUI components interact with the model based on the `ClientInterface` which defines methods to send requests to the server and obtain results. Using the Observer-Observable pattern, the GUI stays responsive while client requests are being processed. The GUI runs in Swing's event-dispatch thread while the client runs in the main thread. Client requests are invoked from the event-dispatch thread but instead of blocking until a response is received, the GUI is immediately available for further user interaction. Meanwhile, the client listens for responses in the main thread and notifies the GUI once a response has arrived.

4.1.2 Interactions with the user and other components

On system start-up, the login screen is shown. The user has the option of clicking on the "Create New Account" `JButton` which will display a new frame for sign-up (see Figure 7). In the sign-up window the user is required to fill in `JFields` for user name and password. Both `JFields` have length requirements (4 to 12 characters for nickname and 8 to 20 characters for password) which, if satisfied, will enable the "Create Account" `JButton`, so that the user is able to submit the information.

When the user clicks the "Create account", the application will either display a success message if the sign-up was successful or a `JDialog` in the case where the nickname is already in use. The client notifies the GUI which case applies, based on the server response. If the nickname is already present in the database, the user can then click the "Try Again" `JButton` and input a new nickname. If the account creation was successful, the user can click the

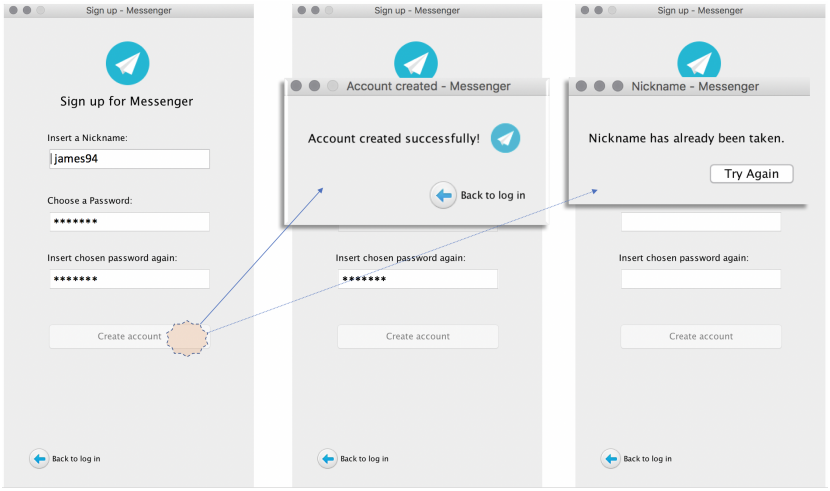

Figure 7: GUI: Creating an account

"Back to log in" `JButton` and log in.

To achieve this, the user is expected to input credentials in the login `JFrame` and click the "Log In" `JButton`. The GUI dispatches a login request to the client and if the nickname-password combination is correct, the user will be logged in, transitioning to the home screen, where the `JList` of online users is displayed. If invalid credential have been provided, the GUI displays a `JDialog`, notifying the user.

Once logged in, the user can execute a variety of tasks: initialise a single user chat, create a new group chat, retrieve any available chat history or log out from the application. Clicking on any user located on the `JList` instantiates a new chat frame.

When the user clicks the "Chat Histories" `JButton`, the client is notified to retrieve the list of the user's previous chats. When this list arrives, the GUI retrieves it from the client and displays it in a scrollable selection screen (see Figure 8). Subsequently, the user can select a particular chat from the list, and the client is notified to retrieve that chat's history. Once the history arrives, the GUI is notified, and displays them in a new `JFrame`.

Back in the home screen, if the user clicks the "New Group Chat" `JButton`, a new frame is displayed with a copy of the online users list (see Figure 9). The user can then multi-select the users he/she wishes as participants and click "Create group chat", opening a new chat window with those participants. The new chat frame contains a `JList` of the user names of the participants. A window listener records if the user closes the chat window to then notify all other participants, terminating the chat whilst informing users with a `JDialog`. If any participant goes offline, the procedure is analogous.

When the user clicks the "Log Out" button, a new `JDialog` prompts the user for confirmation to ensure that the user really wishes to log out. If the user clicks "Yes", the associated log out method is called which closes all application windows, and requests the client to log out.

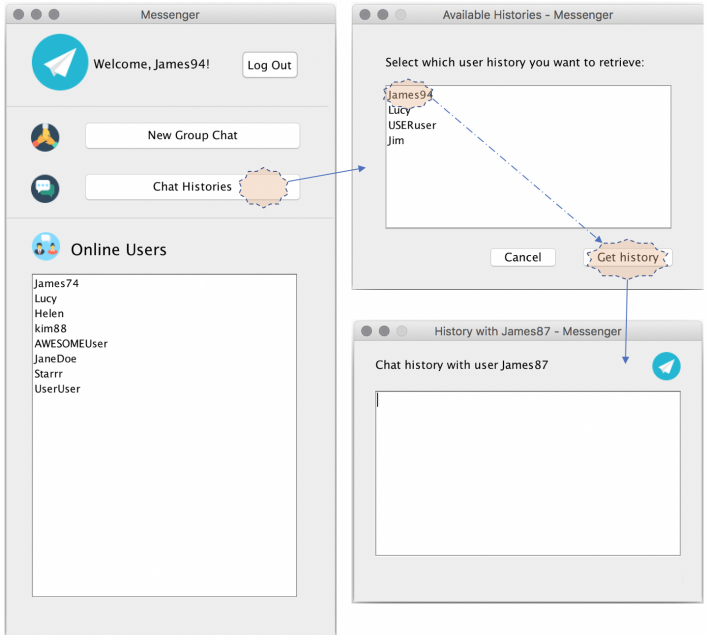

Figure 8: GUI: Retrieving a chat history

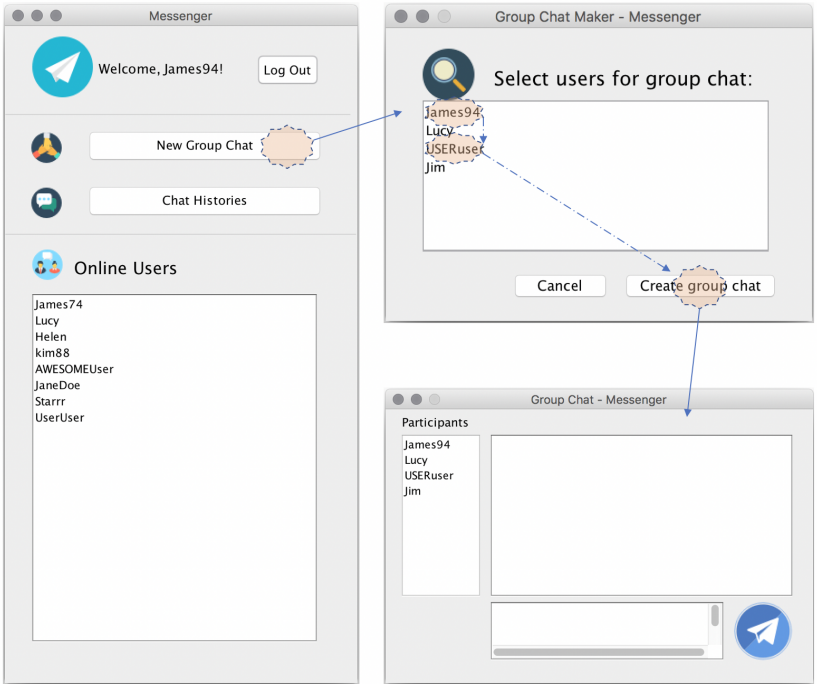

Figure 9: GUI: Creating group chats

4.2 Client

The client is responsible for mediating between GUI and server. The `ClientInterface` allows the GUI to access client services, dispatching requests and receiving results. Furthermore, the client extends `Observable`, allowing the GUI to observe its state. On the server-facing side, the `ClientProtocol` is used to construct and decode `JSON`-based protocol messages. Decoded messages are transformed into instances of `ProtocolMessage` which provides accessors for protocol data. These mechanisms provide the client with a standard way of communicating with the server.

At application start-up, the client obtains the IP address and port number of the server from the command line and uses that information to connect to the server. Once the connection is established, the client is ready to take instructions from the GUI (e.g. send messages, view chat history), transform those instructions into `JSON` format, and send them to the server. Hence, the server receives those instructions in `JSON` format and can react correspondingly.

Any information the server sends to the client is received in `JSON` format, too. In a loop, the client continuously listens for responses until the user logs out or the connection is lost. When the client receives any information from the server, the `JSON` is decoded by the `ClientProtocol`. After determining the message type, the client uses a `switch` statement to distinguish different cases and inform the GUI with the appropriate status. Where necessary, a queue data structure is used to allow the GUI to process events chronologically. When the GUI's update method is called, it retrieves the next element from the queue, and handles it appropriately.

4.3 Server

The server works as a service provider to satisfy client requests over the network. Within the application, the classes encapsulating server-side functionality are `Server` and `WorkerThread`. Database-related code is encapsulated in separate classes, accessed by the server via the `DatabaseFunctions` interface. Such functionality is described separately further below.

When the server is started, it sets up a server socket and begins listening for and accepting connections on an allocated port. In the general case, multiple clients will be trying to connect to the server at the same time. In order to handle these tasks in an efficient and scalable way, a new `WorkerThread` is allocated to each client connection. Threads are managed by a `CachedThreadPool` which reuses idle threads instead of incurring the overhead of starting a new thread, whenever possible. Such thread pool implementation may yield significant efficiency gains (Ling, Mullen, and Lin, 2000). Furthermore, `CachedThreadPool` grows dynamically with demand, thus scaling flexibly.

Mirroring behaviour in the client, the server uses a `ServerProtocol` to encode and decode `JSON`-based protocol messages. Analogous to the client, decoded messages are transformed to instances of `ProtocolMessage`. Within the `WorkerThread` class's `run` methods, the request type is determined and corresponding handler functions are called. Once a request has been processed, a response is constructed and dispatched, if required.

In order to prevent race conditions and data inconsistencies, thread-safe data structures and

synchronisation are used throughout the server. Threads communicate with each other by means of queues. Each `WorkerThread` has associated queues of tasks to process, e.g. messages to be delivered. When a user signs in, the `WorkerThread` serving it registers its queues with the server. Other threads place tasks in those queues. The concurrent data structures used for that purpose are `ConcurrentHashMap` for maps, `ConcurrentHashMap.newKeySet` for sets, and `LinkedBlockingQueue` for queues. In each case, the most efficient data structure for the requirements has been chosen based on the Java API documentation.

Finally, as indicated previously, the server is also responsible for accessing the database. When client requests require it, the `WorkerThread` calls corresponding methods defined in the `DatabaseFunctions` interface. That functionality is defined in detail further below.

4.4 Protocol

In order to facilitate communication between client and server, a well-defined protocol is required. For this project, we have designed a `JSON`-based application protocol.

We have chosen `JSON` because it is a "lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate." (json.org, 2017). Furthermore, `JSON` is language-independent and open source with a variety of free libraries available for generating and parsing. The team has worked together to discuss the information that needs to be interchanged between client and server, defining messages and their fields. The resulting protocol logic is encapsulated in classes `ClientProtocol`, `ServerProtocol`, and `ProtocolMessage`. A comprehensive textual protocol specification, comprising all message types and their fields is contained in the file `ProtocolSpecification.txt` contained in the main project folder in Subversion.

In the `ClientProtocol` and `ServerProtocol` classes, a dedicated method is available for each type of write operation, i.e. creation of a message. To read `JSON`, the `decode` method takes in a `String`, parses it and returns an instance of `ProtocolMessage` which exposes accessors for the protocol data. To implement those protocol operations, Java's standard `JSON` API `javax.json` and its reference implementation have been used.

4.5 Database

`PostgreSQL` was selected as the relational database system for this project. Familiarity with and availability of `PostgreSQL` were the main reasons for choosing this particular relational database, while other options exist. Within the database, essential application data is persisted. This includes user accounts, i.e. user names and passwords. Furthermore, messages are stored to allow viewing chat histories. Finally, users are mapped to chats to allow retrieving a chat history when provided a list of chat participants. Based on the system requirements and the third normal form, three tables have been created in the database (see Figure 10).

Class `CreateTables` contains code to create the application's tables in the database automatically. Classes `DatabaseConnector` and `SQLServer` contain the methods required to obtain database connections, to query and update tables, and to process result sets. Class `DatabaseConnector` implements the `DatabaseFunctions` interface and is used by the server

to access database-related functionality.

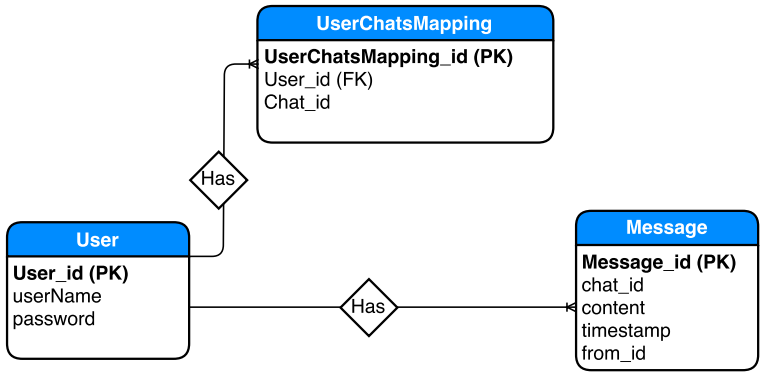

Figure 10: Entity-relationship diagram

5 Test Plan

To assure the correctness of the system, a systematic approach to testing has been adopted, based on the two pillars of automated testing and user testing.

5.1 Automated Testing

Each component of the system has been tested using automated means. Methods for automated testing that have been used are JUnit tests and test applications which execute the code units under consideration in their main method.

5.1.1 GUI

For the GUI, testing has been carried out using several test applications. For that purpose, a `DummyClientForGUI` has been developed. Application `GUITestDrive` tests successfully logging in, and successfully signing up, including retrieving the online list. `GUITestLoginFail` and `GUITestSignUpFail` test the cases where signing in and signing up fail, respectively.

5.1.2 Client

To test the client, JUnit tests have been developed. For this to be possible, a `DummyServer` and a `DummyObservable` have been developed. Using those classes, the client's `listen` method has been scrutinised, testing whether the `Observable` is updated with the correct status values based on protocol messages dispatched from the server. All `JUnit` tests for the client are contained in class `ClientTest`. The testing carried out for the client's functionality for sending protocol messages to the server is described further below in the section covering

the protocol. This is because the client's sending functionality amounts to writing protocol messages over the network.

5.1.3 Server

As for the client, the server has been tested using JUnit. For that purpose, a `DummyClient` and a `DummyDatabaseFunctions` implementation have been created. Using those classes, the server's interactions with both the client and the database are tested in class `ServerTest`.

5.1.4 Protocol

Both `ClientProtocol` and `ServerProtocol` have been comprehensively tested using JUnit. For each read and write operation, there exists a JUnit test. Those tests are contained in classes `ClientProtocolTest` and `ServerProtocolTest`.

5.1.5 Database

Finally, to test the database, an application-based approach has been adopted. Class `DataTestFunction` comprises the tests of database-related functionality.

5.2 User Testing

In addition to the automated testing described before, the application has been tested systematically by means of structured human interaction. In this manner, integration testing has been carried out. Furthermore, aspects of the system that could not be evaluated using automated means have been covered thereby. To achieve this, a comprehensive set of test cases has been developed and documented in the plan shown below. For each test case, the expected result is indicated and the final result has been documented. Iteratively, the system has been brought to the shippable state where all test cases have been passed.

Functionality	Test Case	Expected Result	Test Result
Sign-up	1. Create an account successfully. 2. Create an account with a duplicate username. 3. Create an account with username and no password. 4. Create an account with invalid username and/or password lengths. 5. Create an account without inputting username and password.	1. Account is created and user receives confirmation. 2. The GUI informs the user of duplicate username. 3. The "Create Account" button is not enabled, and feedback is given. 4. The "Create Account" button is not enabled, and feedback is given. 5. The "Create Account" button is not enabled, and feedback is given.	1. As expected 2. As expected 3. As expected 4. As expected 5. As expected
Sign-in	1. Sign in with correct username and password. 2. Sign in with incorrect username. 3. Sign in with incorrect password. 4. Sign in without inputting username. 5. Sign in without inputting password.	1. The user is correctly signed in. 2. The GUI informs of incorrect combination. 3. The GUI informs of incorrect combination. 4. The "Sign In" button is not enabled, and feedback is given. 5. The "Sign In" button is not enabled, and feedback is given.	1. As expected 2. As expected 3. As expected 4. As expected 5. As expected
View Online List	1. Check if online list is available to view. 2. Check if online list updates when a user goes offline. 3. Check if online list update when a user goes online.	1. The online list is displayed after sign-in. 2. The online list is correctly updated. 3. The online list is correctly updated.	1. As expected 2. As expected 3. As expected
Create & Close Chat	1. Create a new chat with a single participant. 2. Create a new group chat with multiple participants. 3. Create a chat that already exists. 4. Create a group chat that already exists. 5. Check if a group chat window is closed when one of the participants goes offline. 6. Check if a group chat window is closed when one of the participants closes the chat window. 7. Check if a single user chat window shows a notification when the other user goes offline. 8. Check if a chat window displays the correct username of the participant(s).	1. Single user chat is created. 2. Group chat is created with correct participants 3. A new chat window is created, messages will be appended to the same chat history. 4. A new chat window is created, messages will be appended to the same chat history. 5. The group chat window is closed, the other users receive feedback of the event. 6. The group chat window is closed, the other users receive feedback of the event. 7. The user is notified that the other user is offline, and the send message button is disabled. 8. The chat window displays the correct user name(s).	1. As expected 2. As expected 3. As expected 4. As expected 5. As expected 6. As expected 7. As expected 8. As expected
Send & Receive Messages	1. Check if a user can view the messages he/she has sent in the chat window. 2. Check if a user can send messages to the person that is participating in the chat. 3. Check if a user can send messages to everyone that participates in a group chat. 4. Check if a user can receive messages.	1. The user can view the messages that he/she has sent in the chat window. 2. The message sent by the user is delivered to the recipient and displayed in the chat window. 3. All the participants receive the user message. 4. The user receives messages from other users.	1. As expected 2. As expected 3. As expected 4. As expected

Browse History List	1. Check if the history list window displays when user clicks the button to view the history list. 2. Check if the history list shows correctly. 3. Check if each chat in the history list is shown with the participants' user names. 4. Check if history window displays when user clicks on one of the chat.	1. The history list window is displayed correctly. 2. The history list content is displayed correctly. 3. The history list content is displayed correctly. 4. The history window is displayed correctly.	1. As expected 2. As expected 3. As expected 4. As expected
Get Chat History	1. Check if the history messages of a chat are correctly delivered. 2. Check if the history messages of a group chat are correctly delivered. 3. Check if each message is shown with the user name of sender and the time at which it was sent.	1. All messages that have been sent in the chat are delivered. They are displayed in the order in which they have been added to the database. 2. History messages are delivered correctly as for user chats. 3. Messages are shown with the user name of the sender and the time at which they have been sent.	1. As expected 2. As expected 3. As expected
Log Out	1. Check if the user is logged out when clicking the "Log Out" button. 2. Check if all application windows close after log out.	1. The user is asked to confirm logging out. Upon confirmation, the client disconnects from the server. If the user does not confirm, he/she is not logged out. 2. All application windows are closed after log out.	1. As expected 2. As expected
Exceptions	1. The client is terminated or loses the connection to the server. 2. The server is terminated or looses the connection to the client. 3. The server loses the connection to the database. 4. Due to a database error, an SQL exception is thrown.	1. The user is informed, using a dialogue, that a problem has occurred. All application windows are closed, and the client-side application gracefully terminates. The server remains responsive for other clients. 2. All users are informed, using a dialogue, that a problem has occurred. All clients are terminated. All application windows are closed, and all client-side applications gracefully terminate. 3. The user is informed, using a dialogue, that a problem has occurred. The server is terminated. All connected clients are terminated. All application windows of all clients are closed, and all active applications gracefully terminate. 4. The thread which has attempted to access the database is terminated. The user is informed, using a dialogue, that a problem has occurred. All of the user's application windows are closed, and the user's client-side application gracefully terminates.	1. As expected 2. As expected 3. As expected 4. As expected

6 Evaluation

In this final section of the report, we evaluate the system along the three dimensions "functional scope", "usability", and "technical implementation".

6.1 Functional Scope

Firstly, in terms of scope, the system provides the core functionality expected from a messaging application, namely exchanging instant messages with online users. Group chats and viewing chat histories are advanced features that go beyond this baseline. For future releases, we would prioritise for implementation the sharing of attachments, support for emojis, and WhatsApp-like functionality for showing whether a user has viewed a message.

6.2 Usability

Concerning usability, the system complies consistently with widely-used usability heuristics such as Nielsen's (Nielsen, 1995). The system state is visible to the user, feedback is provided throughout the process, e.g. after successful login or in case of a lost connection. Errors are prevented, e.g. through a confirmation dialogue before logging out. The puristic design of the GUI presents a consistent look and feel. For further improvements, chat windows could be highlighted when a new message has arrived. Keyboard shortcuts for power users could be provided. Furthermore, customisation of the interface could be allowed, e.g. enlarging button sizes for those with impaired vision.

6.3 Technical Implementation

Before discussing the application's strengths, some weaknesses require consideration. Firstly, not all data exchanged between client and server is encrypted. While passwords are not sent in the clear, all other traffic is. Furthermore, when a user retrieves a chat history, all messages sent in the chat are retrieved, transmitted, and displayed. For a large chat, this may draw heavily on memory and bandwidth resources. As an improvement to this, the history could be sent in increments, requesting older messages only when the user has viewed the first increment. Further technical optimisations would be possible, e.g. integrating a logging framework or supporting Java property files.

The system's technical strengths lie, to begin with, in a consistent application of architectural and design patterns. Firstly, the client-server architecture allows for scalability and low coupling between client and server. Secondly, model-view separation based on the Observer-Observable pattern facilitates independence of the client from the GUI. Generally, the system adheres to the principle of using interfaces, not implementations. Client and GUI, as well as server and database functions interact solely based on interfaces. Implementations can thus be flexibly changed or modified without breaking other code. Finally, using the platform-independent and widely used `JSON` format for the protocol means that alternative clients or servers can be implemented in a technology-agnostic manner based on open source libraries.

References

json.org (2017). *Introducting JSON*. URL: `http://www.json.org/index.html` (visited on 03/15/2017).

Ling, Yibei, Tracy Mullen, and Xiaola Lin (2000). "Analysis of optimal thread pool size". In: *ACM SIGOPS Operating Systems Review* 34.2, pp. 42–55.

Nielsen, Jakob (1995). *10 Usability Heuristics for User Interface Design*. URL: `https://www.nngroup.com/articles/ten-usability-heuristics/` (visited on 03/17/2017).

Appendices

A Team Organisation

A.1 Task Allocation

Tasks were allocated as follows:

Team member	Task
Viola Marku	GUI, meeting minutes
Meng-Jung Lee	Server
Ruoyu He	Database
Yunxiao Zhuang	Client
Jochen Stüber	Server, point of contact for Fani

A.2 Meeting Minutes

Meeting minutes for all conducted meetings are appended below.

Meeting 1
Date: 15/02/17
Attendees: All

- Proposing new topics;
- Talking about the topic proposals;
- Decided to pick the messenger style application for ease of implementation and chances of upgrading functionalities;
- Rough split of roles, to be finalised later in the week;
- Starting to define system specification and description of desired functionalities;
- Completed a first draft of functional and non-functional requirements;
- Decided next meeting day and what to achieve.

Meeting 2
Date: 17/02/17
Attendees: All

- Started working on use case diagram;
- Class diagram;
- Started to discuss the GUI and designed a first prototype;
- Started to discuss database schema and organisation.

Meeting 3
Date: 24/02/17
Attendees: All

- Made a plan on what to start on;
- Breaking down the parts to start on;
- Started modelling each use case;
- Writing methods for GUI and client;

Meeting 4
Date: 28/02/17
Attendees: All

- We showed each other the use cases and potential methods;
- Defined interfaces;
- ER diagram for the DB;
- Defined protocols;
- Discussed DB connection protocols.

Meeting 5
Date: 01/03/17
Attendees: All

- Continuing the client gui observer/observed;
- We finished defining all the interfaces between GUI and client.

Meeting 6
Date: 06/03/17
Attendees: All

- Installed subversive and made sure it works on everyone laptop;
- Showed and finalised the final GUI design and methods;

Meeting 7
Date: 07/03/17
Attendees: Yunxiao, Meng-Jung, Ruoyu, Jochen (Viola excused due to illness)

- Specified protocol between client and server
- Specified server interface for database-related functionality
- Divided up work for server between Jasmine and Jochen

Meeting 8
Date: 17/03/17
Attendees: All

- Worked on the gui/client methods;
- Started working on the report;

Meeting 9
Date: 18/03/17
Attendees: All

- Testing and debugging;
- Continued to work on the report;
- Connected to database and tested;

Meeting 10
Date: 19/03/17
Attendees: All

- Continued testing and debugging;
- Finalised the report;
- Started planning the presentation PDF.