MSc Computer Science

Software Workshop: 06-26956

Team Project: Spring Term 2015

Team Bilbao

Anna Jackson, Jessica White, Sam Joseph
Harry Seager, Divyjyot Saraon

Contents

Description of the system

DevChat is an instant messaging system for software developers (student/professional) which allows registered users to engage in private and team text based chats.

Specification:

When DevChat is started, a user should be able to:

- log into the system with and already existing account,
- create an account using basic details,
- create an account using basic details and extra personal information

Once logged in users should be able to:

- view a list of users currently online.
- start a private chat with one of the users online.
- start a team chat with two or more of these users.
- open numerous chats, both private and team, simultaneously.
- view a list of private and team chats they have already partaken in.
- open "profile" screens of their own account and other users

If a profile screen is clicked:

- user information and account details are displayed
- sensitive information is hidden from other users (i.e. password and email)

When a chat is begun or resumed:

- a chat window should be opened on the users screen.
- if online, a chat window should open on any users screens involved in the chat.
- if the others users being chatted to are not logged in, messages should still be sent, stored to database, so that they can view the message next time they log in.
- the chat window should display who is involved in the chat.
- past messages from the chat should be loaded in to the window automatically.

When a team chat is begun or resumed extra functionality includes:

- being able to leave a team
- being able to remove a team member.
- being able to add a person to a team.
- team chat will display people included in the chat.

When a chat is closed:

- only the window selected is closed.
- messages will still be sent to the user that closed the window.

When the main screen is closed:

- all chat windows are closed.
- the user is logged out of the system and shown to be offline.

Errors will be displayed to the user when:

- they enter the wrong log in details
- they enter a password with fewer than 8 characters or which does not contain a number

The user cannot:

- access other users passwords or emails
- access any chats they have not been added to.

Whole System Design

The system uses a client-server architecture, with most classes being declared in one of three packages: `database`, `server`, `client`. Information about users' accounts, log-in credentials and chat history is stored a database. As the time required to connect to the database is much greater than the time required to query the database, a single connection to the database is established and maintained by the server through an instance of `database.Database`.

The client has two classes, `client.Client` and `client.ChatWindow`, although they each declare a number of inner classes. Both classes implement `Runnable` so that the client can respond continuously to both the server and the GUI. `client.Client` implements all functions of the client except for the actual chat, which is handled by `client.ChatWindow`. The observer classes mirror this structure, with `client.MainGUI` observing `client.Client` and either `client.PrivateChatGUI` or `client.TeamChatGUI` observing `client.ChatWindow`, depending on the status of the chat.

Additionally, a number of classes are declared in package `general` and are used by all parts of the system. These include:

- `general.Account`: information about a user (username, email, personal information etc.)
- `general.Chat`: a unique identifier for each chat, a list of participants and whether the chat is private or team
- `general.ChatMessage`: a message, the message sender, a time stamp, the chat ID
- `general.Message`: used in communication between the server and client (see protocols).

These classes facilitate both the translation of information into and out of the database and consistency of format between different parts of the system.

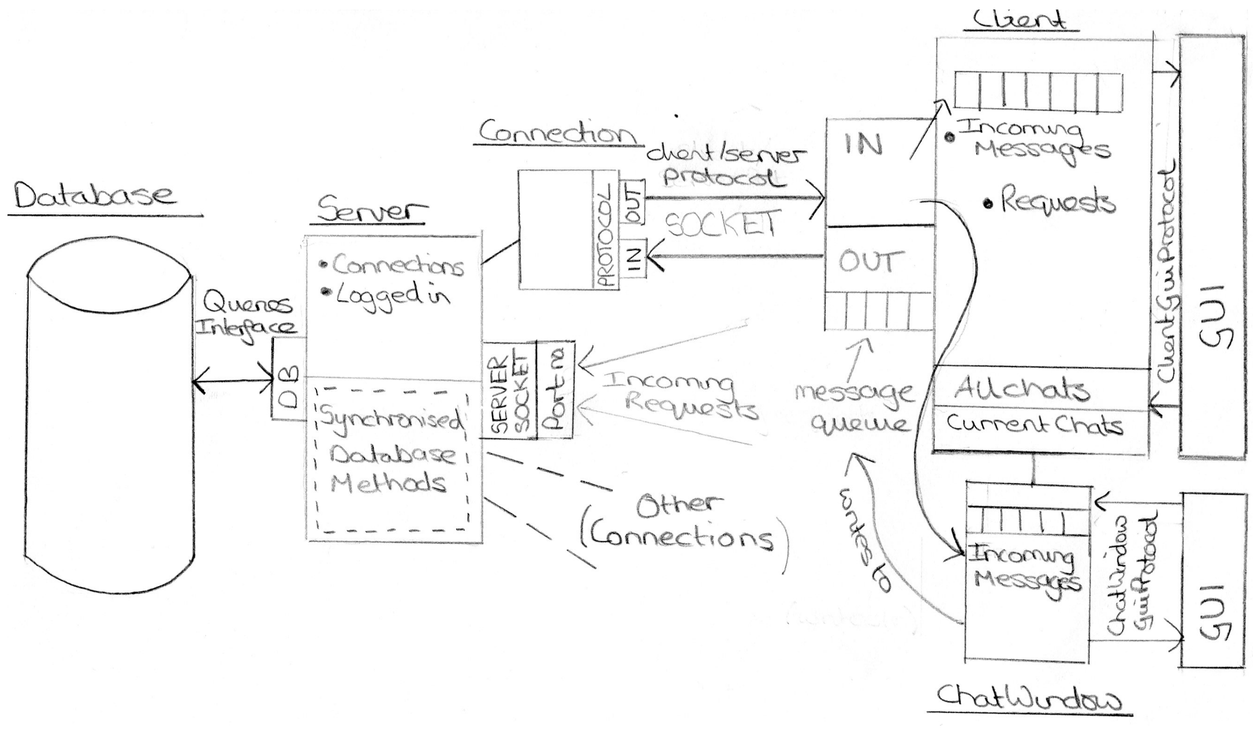


Figure 1 - Whole System Design

Protocols

There are three points in the system where different components must communicate with each other according to a fixed protocol:

DB <-- ***protocol*** --> Server <-- ***protocol*** --> Client <-- ***protocol*** --> GUI

Server/Database (see Appendix A)

An interface details which methods the server can call and what object the database will return. All `void` methods throw exceptions to indicate failure writing to the database. This allows the server to handle the exceptions wherever is most appropriate for its implementation.

Client/Server (see Appendix B)

The protocol is based around the client and server sending instances of `general.Message` between them. An object of this type has the following field variables:

- a status
- a thread and message ID
- an array of objects (may be null)

The status is of type enumeration, so that the finite number of statuses available is enforced not only by the protocol but also by the Java compiler.

Since the protocol states that only instances of `general.Message` are to be sent between the client and server, all objects read from `socket.ObjectInputStream` on both the client and server side can be safely cast to this type without any additional checks being necessary.

GUI/Client (see Appendix C)

Two interfaces, `client.ClientGUIProtocol` and `client.ChatWindowGUIProtocol`, detail the methods the observer classes can call to request information from the server and what updates the observer should expect from the client. All updates use instances of `client.GUIMessage` which has a status (enumeration) and an object. The observer classes parse the objects they are updated with by first looking at the status then casting the object to the expected type.

Database Design

The database for our project is required to store user log in credentials and account details and chat information, including participants of the chats, messages sent and details of who sent it. In addition, it needs to be constructed in such a way that the server's queries can be obtained and formatted easily and quickly. The database has therefore been arranged as follows:

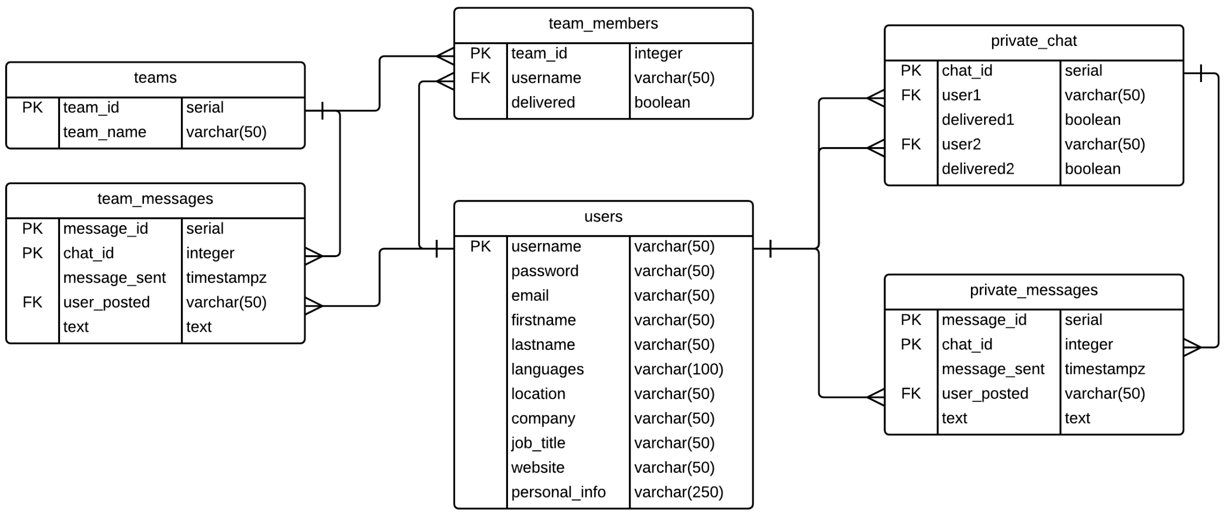


Figure 2 – Database Design

users - Contains user information relating to log in and account details. Although variable length strings are not usually good candidates as primary keys, usernames have to be unique in dev_chat so this causes no issues.
private_chat - Table of all chats between two users (private chats).

private_messages - Stores each chat message against the relevant chat_id from the private_chat table, along with the user who sent it.

teams - Reference table of teams. These are group chats containing a minimum of three users at outset (they can go below three if users are removed but will remain as team chats).

team_members - Table matching teams and the users as their team members.

team_messages - Stores each chat message against the relevant chat_id from the teams table (team_id), along with the user who sent it.

'Delivered' fields in the above ERD would record whether the user has seen all messages in the chat but this feature is not currently implemented. It has however been left in to avoid having to amend the database if we were to do so.

Text fields in the message tables have been set to the data type 'text', which has an unlimited character length, which has been chosen due to Dev Chat being aimed at software development teams. As a result, it is likely that large bodies of text could be sent in one message.

The database class utilises Oracle’s JDBC API to access the postgreSQL database. This provides a Connection, established in the constructor, through which SQL queries are passed. Query results are in turn stored as ResultSet objects.

Querying the database is done via the use of prepared statements. These allow for regularly used SQL statements to be precompiled when the initial connection to the database is established, meaning a new query does not have to be compiled again at each request. Additionally, these prepared statements can take parameters in the place of question marks within the precompiled statement, so these statements can be reused repeatedly, even when the query’s parameters are different.

Server Design

The server has two classes, `server.Server` which creates the `ServerSocket` and listens continuously for clients trying to connect, and `server.Connection`, which manages the input and output streams for each client. `server.Connection` also uses two inner classes, `Listener` and `Writer`, both of which implement `Runnable`, to continuously read from and write to the socket input and output streams. Inner classes have been used so that the threads can share resources.

When a client connects to the server, a new instance of `server.Connection` is created, allocated a session ID, and put into map `server.Server.connections` with the session ID as they key. When a client sends login details to the server, an additional entry is added into map `server.Server.loggedIn` which maps usernames to their current session ID and allows the server to identify the correct socket connection if a client requests that it send a message to a particular user.

`server.Connection` uses an instance of `server.Protocol` to parse incoming messages and process them with methods declared in `server.Server`. As there will normally be multiple instances of `server.Connection`, all methods which they can call are synchronised. The use of a separate protocol class allows changes to be made to the client/server protocol without affecting the implementation of the server.

When a user logs out, their username and session ID are removed from `server.Server.loggedIn`. When a client disconnects, either intentionally or because of an error, the exception is caught in `server.Connection`, all run methods terminated and resources released before the connection is removed from `server.Server.connections`. If a client reconnects to the server a new instance of `server.Connection` with a new session ID is created.

All errors which occur while trying to access the database are caught in `server.Protocol`, which sends an error message to the client. All errors are currently handled in the same way; however, further development of the system would ideally distinguish between different types of exception.

If an error occurs while the server is running, all run methods are terminated, all resources released, and `server.Server.connections` and `server.Server.loggedIn` cleared. The server then attempts to reestablish its connection.

Client Design

The client has two classes, `client.Client` and `client.ChatWindow`, which both extend `Observable` and implement `Runnable`. A loop in the constructor of `client.Client` stops the object from being created until a socket connection has been established. Two two inner classes, `client.Client.IncomingReader` and `client.Client.OutgoingWriter`, which both implement `Runnable` , then manage the socket connection with the server. All instances of `client.ChatWindow` are uniquely identified by a thread ID.

Some field variables in `client.Client` can only be initialised once the user has logged in, rather than in the constructor. These include a map which uses thread IDs for storing active chats, two maps which use chat and team IDs to store previous chats, and two further maps for mapping between chat/team IDs and thread IDs. `client.ChatWindow` runs in its own thread and implements a different protocol interface to `client.Client`; however, the two classes are fundamentally similar in structure and the points made below about the sending, receiving, and processing of messages apply to the two classes. Instances of `client.ChatWindow` are always created and passed to the observer by `client.Client`.

The use of different threads in the client means the following tasks can tasks can be performed continuously:
1) Read from the socket
2) Write to the socket
3) Respond to method calls by the GUI
4) Respond to messages from the server

Read from the socket
`client.Client.IncomingReader` listens continuously for messages on the socket input stream. When a message is received, `void client.Client.receiveMessage(Message message)` is called, which checks the thread ID of the message. If the threadID is equal to 0 or 1, the message is added to the queue `client.Client.incomingMessages`; otherwise, the message is put onto the queue `client.ChatWindow.incomingMessages` of the instance of `client.ChatWindow` which has that thread ID. The messages are read from these queues and handled by the run methods of the respective objects.

Write to the socket
The queue `client.Client.outgoingMessages` can be written to by all parts of the client. `void client.Client.OutgoingWriter.run()` continuously takes messages from this queue and writes them to the socket output stream. A `LinkedBlockingQueue`, from `java.util.concurrent`, has been chosen as the implementation of this queue as it is thread safe and blocks if the queue is empty. Messages sent from `client.Client` use thread ID 1. Messages from `client.ChatWindow` use the thread ID given when the class was created.

Respond to method calls by the GUI
Methods implementing `client.ClientGUIProtocol` and `client.ChatWindowGUIProtocol` are called by the observer classes. These methods create instance of `general.Message` which they add to `client.Client.outgoingMessages` to be sent to the server. The action to be taken once the server returns the requested information is known at the time the method is called; however, there is

no guarantee of when the server will respond. This issue is resolved through the use of an abstract class, `client.Request` which has one method, `void client.Request.respond(Message message)`. Every time a method sends writes a message to `client.Client.outgoingMessages`, it also creates an instance of a class extending `client.Request`, defining what the response to the message should be. This object is then put into a map of requests, with the message ID as the key.

Respond to messages from the server
The run methods in `client.Client` and `client.ChatWindow` take messages from their incoming messages queue. If the message is a response from the server, an instance of `client.Request` is retrieved from the object's request map and `void client.Request.respond(Message message)` called to perform the predetermined response. If communication has been initiated by the server, the action taken is determined by `general.Message.status`. If necessary, the message might also be forwarded to an instance of `client.ChatWindow`, e.g. receiving a private message.

In addition to these four primary tasks, the client has also been designed to handle the following: retrieving chat history, error handling, logging out and disconnecting, reestablishing lost connections.

Retrieving chat history
When a user logs in, the client requests lists of all their previous chats from the server. These are stored locally in `client.Client.allPrivateChats` and `client.Client.allTeamChats`. This information is displayed in the GUI so that the user can resume a chat they have already started and used by `client.Client` to open chat windows for previously inactive chats.

Error handling
Error messages received from the server are handled by updating the observer with an instance of `client.GUIMessage` with status `ERROR` and object `String` which details the error. Ideally, future development would also allow the system to recover from the error by resending any failed requests.

Logging out and exiting
When a user logs out, all field variables initialised at login are set to null or cleared. When a user closes the GUI, the thread running `client.Client` is interrupted. All run methods in the client are then terminated and resources released. In both cases, all chat windows are closed and `void client.ChatWindow.run()` terminated in all instances of `client.ChatWindow`. Conditions in `void client.Client.IncomingReader.receiveMessage(Message message)` mean that all messages except for confirmation of login or an account being created are discarded unless a user is logged in.

Reestablishing lost connections
If the connection to the server is lost, the run methods in `client.Client.IncomingReader` and `client.Client.OutgoingReader` are terminated. There is then an attempt every two seconds to reconnect to the server. New instances of `client.Client.IncomingReader` and `client.Client.OutgoingReader` are created once the connection has been reestablished.

GUI Design

Design Decisions

Prototypes were drawn up at an early stage and have remained largely unchanged. Focus has been on getting a working GUI, with efforts made to avoid style over substance. As such, the design focused on providing a simplistic environment where the information required is ready provided or easily reachable through recognisable user controls. ‘GridBagLayout’ was often chosen for layout management. This layout was chosen primarily for the flexibility it provides in terms of layout when creating GUI’s.

See appendix I for all final GUI prototype designs. These are also available on SVN.

Login GUI.

Upon launching DevChat the `View` class calls Client and the method `client.View.LoginGUI(Client client)` is called to display a ‘login’ GUI on screen. The login GUI asks the user to provide a username and password to select the option of creating an account. If 'Create Account' is selected, login GUI's’ visibility is set to false and `client.CreateAccountGUI(Client client)` is called. Here the user is required to provide the following information to set up an account, with some constraints:

- username (unique)
- password (longer than 8 characters and containing one number)
- email address

They have the option to provide further information such as:

- First Name
- Last Name
- Languages
- Job Title.
- Website

Upon successful login or account creation `client.MainGUI(Client client)` is launched. The main GUI comprises of four panels (see fig 3) – each that calls is own class:

1)`client.ToolbarPanelMainGUI`

This toolbar displays the DevChat logo and a 'Logout' button.

2)`client.ContactsPanel`

A `JList` was used to list all users currently online (by usernames). `client.ContactsPanel.AskForUsersOnline` is called to check which users are logged in by running `client.usersOnline()`. When `client.usersOnline()` is called, the observers are notified and the `JList` is updated using `client.ContactsPanel.update()`. `client.ContactsPanel.AskForUsersOnline` extends `java.util.TimerTask` and is set to run every 10 seconds, which after manual testing appeared to be an acceptable time length for both the program and for users.

The `JList` uses a derivative of `MULTIPLE_INTERVAL_SELECTION` allowing a user to select 1 or more contacts to start a chat with. If >1 users are selected, a Dialog box appears requesting that the user enter a 'Team Name'. The `String` team name entered and an `ArrayList` containing the selected usernames are passed into `client.makeTeam`. If one user is selected then `client.startPrivateChat` is called.

3)<u>`client.ProfilePanel`</u>

This panel displays profile information unique to the current user. A profile picture, the user's full name and company they work for are displayed. These `JLabel` components are updated when the profile button is pressed and `client.viewAccount()` is called. This retrieves the information needed as an Account object and sends it to `client.ProfilePanel.update` as a `GUIMessage`. The information for each relevant field with in the profile is called from the Account object passed using getters. If there is no information available the String "empty" appears for that field (as set in the field variables of `client.ProfilePanel`.

4)<u>`client.InboxPanel`</u>

This panel displays the previous chats the user has been involved in using a `JList` component. The methods `client.getPrivateChats()` and `client.getTeamChats()` update this `JList` by notifying observers which passes a `GUIMessage` to `client.InboxPanel.update()`. The `GUIMessage` contains a status of `ALL_PRIVATE_CHATS` and an `ArrayList<Chat>`. The `ArrayList<Chat>` is iterated through and the usernames from the `Chat` objects are added to a new `ArrayList<String>`. These are added to the `DefaultListModel<String>` used by the `JList`, and the usernames are also added to an `ArrayList<String>`.

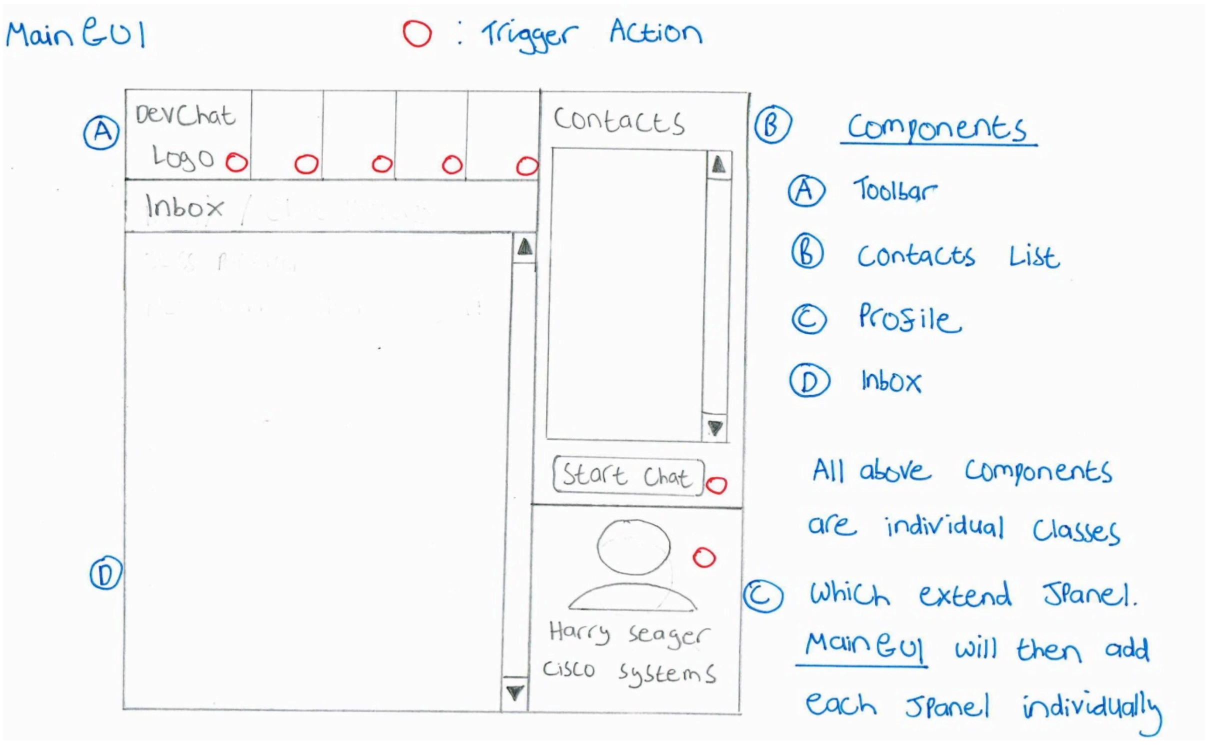


Figure 3 – Main GUI containing a toolbar, contacts list, profile and inbox.

Team Chat GUI.

The team chat GUI, shown in fig 4 is constructed in the class `client.TeamChatGUI`. It is called either when more than 1 person is selected in the contacts list to start a chat with, or when a previous "team chat" has been selected from the inbox panel.

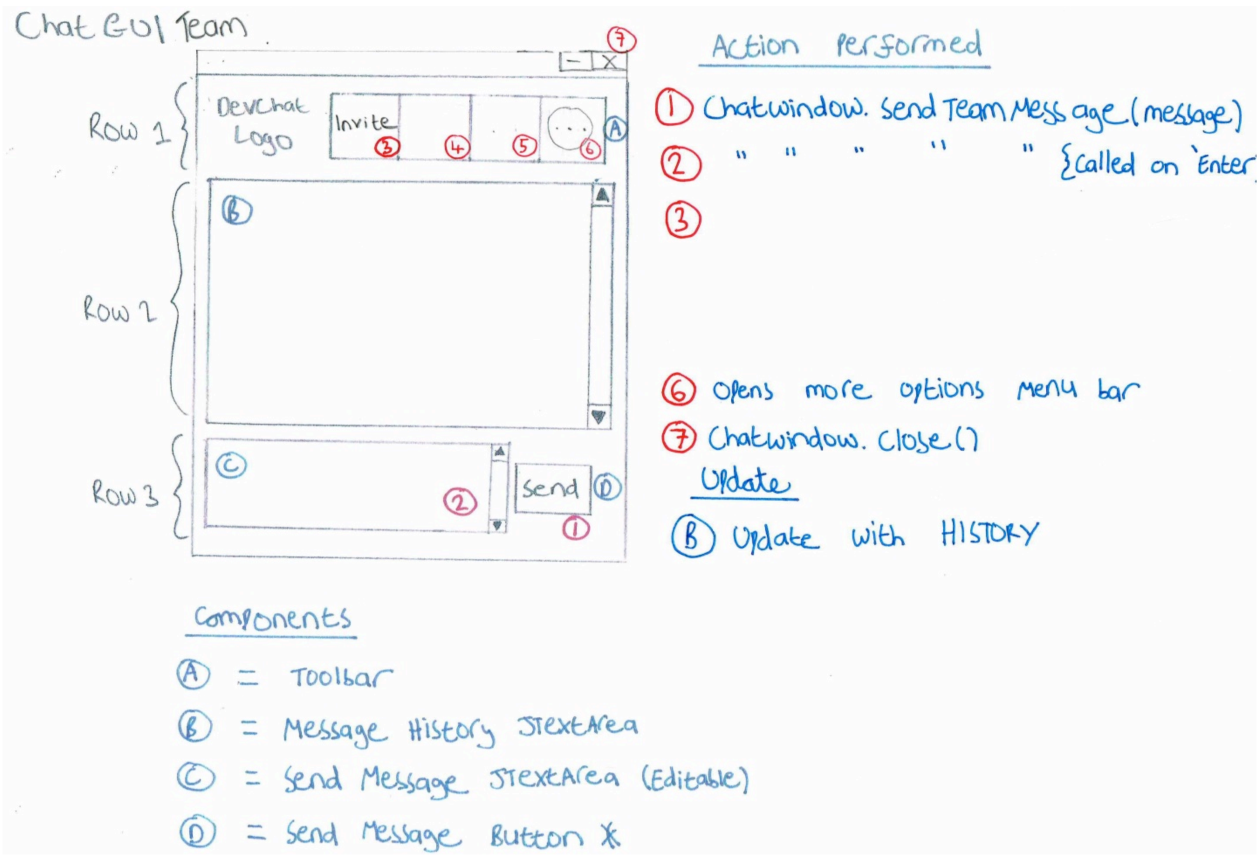

Chat GUI Team
Row 1
Row 2
Row 3
DevChat Logo
Invite
Send
Action performed
① Chatwindow. send Team Message(message)
② " " " " " {called on 'Enter'
③
⑥ Opens more options menu bar
⑦ Chatwindow. Close()
Update
Ⓑ Update with HISTORY
Components
Ⓐ = Toolbar
Ⓑ = Message History JTextArea
Ⓒ = Send Message JTextArea (Editable)
Ⓓ = Send Message Button

Figure 4 – Team Chat GUI.

Three JTextArea's are set as field variables, two of which are not editable:

- `membersList` (uneditable)
  Displays a list of who is involved in the chat.
- `mainText` (uneditable)
  Displays message history for the chat
- `sendText` (editable)
  Where the user writes the text they wish to send (which they do using the 'send' button).

Each `client.TeamChatGUI` is constructed with the observable class `client.ChatWindow`. For each `client.TeamChatGUI`, a new thread is started which has `client.ChatWindow` as an argument. This allows each chat to run in a separate thread allowing concurrent chats between multiple users.

If the chat has been previously created, a call is made to `client.getTeamChat()` in order to retrieve the needed information from the database. This is sent back in the form of `general.Message` to `ChatWindow`, which if successful creates a `GUIMessage` with the status `HISTORY` and an `ArrayList<ChatMessage>`. The observers are notified, sending the `GUIMessage` to `client.TeamChatGUI.update()`. The instances of `general.ChatMessage` are then displayed in the history text area.

In an active chat the server will inform `client.ChatWindow` of new messages. These messages get passed onto `client.TeamChatGUI` as a `GUIMessage` object with the status `DISPLAY_MESSAGE` and a String object of the message being sent. This is then added to the message history text area.

The toolbar in `client.TeamChatGUI` gives the option to 'Invite/Remove Members' and also to leave the team (`client.leaveTeam`). Selecting the 'Invite/Remove Members' button calls `client.InviteRemoveGUI`. The invite / remove GUI provides a list of all the usernames of those currently in the Team Chat.

Selecting a member and selecting 'Remove Member' calls `ChatWindow.removeTeamMember` with the selected username passed as an argument.

Members can also be added to the Team Chat using the invite / remove GUI by selecting a username from a dropdown list of all users using DevChat (uses `client.allUsers`). Once a user is selected from this dropdown list selecting 'Add' will then initiate the `ChatWindow.addTeamMember` method with the selected username again passed as an argument.

Selecting 'Leave Team' from the Team Chat toolbar runs the same method as for removing a member however the current users username is used as the argument.

Private Chat GUI

'PrivateChatGUI' is a derivative of team chat without the options to leave a team and invite/remove members:

Test Plan

The system has been tested in two ways:
- JUnit testing for the server, database, and client
- Manual testing for the GUI and whole system

JUnit Testing

All components of the system except the GUI have been tested with JUnit tests. These tests have two main purposes:
- To verify that components correctly adhere to communication protocols
- To verify that internal field variables are updated and initialised as expected

All JUnit tests are in package `tests`. To enable the checking of field variables without compromising their visibility, variables which need to be tested have been set to protected, and spies, which extend these classes and implement getters and setters, created in `tests`.

The first set of testing was performed on the server. The database was simulated with a dummy class implementing `database.Queries` providing predetermined return values. The client was simulated by writing/reading instances of `general.Message` directly to/from the socket.

The server was then used to test the database. The client was again simulated by directly reading and writing to the socket.

To test the client, another dummy database class was created and used with the actual server. To test that the GUI is updated correctly, a class `tests.TestObserver` has been created, which stores `client.GUIMessage` statuses and the objects with which it is updated in instances of `ArrayList`. Assertions are then made about the expected sequence of statuses/objects in these lists.

Implemented tests and documentation of expected behaviour can be found in appendices D, E and F. At the time of submission, all of these tests have passed.

Manual Testing

Appendix G contains a document detailing sequences of actions to be performed and the expected behaviour. At the time of submission, all expected behaviours have been observed.

Team Organisation

Role/Group	Person/People	Responsibilities
Team Lead:	Anna Jackson	Design of basic system/protocols and role / job allocations.
Secretary:	Jessica White	Keeping records for the diary, managing Facebook group, being the main point of contact
Database:	Sam Joseph	Building, populating and testing
Server:	Anna Jackson	Building and testing
Client:	Jessica White / Anna Jackson	Jessica and Anna were both involved in building the Client and ChatWindow. Between the two of them also managed testing of this side using spies and test classes
GUI:	Harry Seager / Divyjyot Saraon	Harry designed and built the GUI's. Divyjyot coded any functionality for it.

Project diary, including agendas minutes of meetings.

The full project diary/ minutes can be found in Appendix H and in the SVN file.

Below is a summary of our meetings and progress:

Date	Attendance	What was done	To do
24/02	Full Group	Discussed Specification including: Goals, team structure, architecture and protocols	Write protocols Write up our specification
26/02	Tutorial. Full Group	Discussed our specification	Answer questions from discussion
28/02	Sam,Jess	Discussed DB, Client, GUI	Listed concerns
18/02	Divyjyot, Anna	Discussed Server, Client, GUI	Listed concerns
03/03	Full Group	Agreed on the basic needs. Discussed concerns and developed answers.	Skeletons produced for GUI, Client, Server & DB
10/03	Full Group	Discussed structure and threads plus any problems that have arisen. Came up with provisional deadlines for each stage of the work.	- GUI designs. - Client: implement method stubs, Jess : Chat Window implementation. Anna : Methods. - Server/DB: Sam: update for new version of queries
12/03	Tutorial. Full Group	Discussed work so far and developed an agenda for the weekend.	**Jess:** Sort our success' and error's., add what we need to GUIMessage class, read Client - find holes, testing ChatWindow, check protocols **Divyjot:** Updates, observables, Transitions. **Harry:** Making / sorting all the frames. **Sam:** Testing **Anna:** Client - thread safe, closing things, testing
18/03	Divyjyot, Harry, Jess	Tested methods	See document
19/03	Full Test	Tested methods	See document

Evaluation

This was the first big structured piece of group work that this team has partaken in. As such there have been many lessons learnt during the venture.

First of all – the importance of protocols. The protocols played a key role in determining how the different system components would come together. As a group, the protocols were the first aspect of the project that we developed, and as such, the entire group had a basic knowledge of how each component would communicate. This proved invaluable later on, and though they had to be edited during the process, having the initial protocols in place avoided larger errors.

As the database/server/client was developed first, the protocol determining these interactions was written first. If this project was to be repeated, the client/GUI protocols would have been written at the same time. The reason this wasn't done was to ensure the backend of the system worked stably, before we considered how to let the user interact with it. However, the GUI side of the project later wanted functionality that couldn't be supported by the database/ server/ client. This may have been avoided if the client/GUI protocols had been written earlier.

As mentioned the database and server were a couple of the first developed and tested areas of the system. So far, the server has been shown to be fit for purpose. The use of threads running objects declared as inner classes means that the server responds quickly. The resources are shared efficiently in Connection and connections are easily shut down. Given more time, there could have been better structure for sending error messages to the client and the project could also have incorporated XML to send the messages.

As for the GUI, it was designed and implemented with the user in mind. The icons and interactive elements such as buttons are easily distinguishable and the error pop-up dialog boxes adequately handle error messages relayed by the client. However, there is room for improvement. Currently, there are different windows for logging in, creating an account and the MainGUI frames. These could have been put into the same window while transitioning from one frame or panel to the other. The color theme is neutral and could have been aesthetically more appealing if there had been more time.

The system as a whole fits purpose at present. It can do everything stated in the specification. The implementation of the abstract Request class added a lot of flexibility to the system. Its implementation allowed the Client to handle the unpredictable delay of server responses and the functionality of the Client can be considerably extended without major changes to the structure. It is thought that this may enhance the scalability of the product; but to what extent hasn't been tested.

The system was tested using both automated and manual testing. Nonetheless, though we tested the system using JUnits and manual tests this is an area we could have greatly improved upon. Tests could have been conducted to check the connections between components. We have conducted limited integration tests using the TestObservable spy (such as between client.Client and the GUI's and client.ChatWindow and the GUIs) but this could have been done more extensively throughout the process.

Further, as a group we conducted testing at the very end of development. With more experience (and contact time) and test-driven-development approach may have been a better way to build the system, testing each tiny component and building upon it. Such an environment would also be beneficial for paired-programming, which with switching between “teams” would have given everyone a stronger knowledge of the entire system.

In terms of future improvement, the Client could be improved by having a structure to handle errors rather than notifying the user straight away, and errors could be separated into different types. Further in terms of overall structure, though the system we have developed works, we are aware there are other ways in which we could have approached it. Depending on what is considered preferable to the customer, there could have been a greater separation of concerns, the structure could have been written in a way that produced less lines of code etc.

Appendix A : Server/Database Protocol

`public interface` **`Queries`**

This class contains methods which the database class used in the server must implement. The methods allow the server to request information from the database.

Modifier and Type	Method and Description
`void`	**`addUser`**`(java.lang.String username, int teamID)` Adds given user to the team chat record in the database
`java.util.ArrayList<java.lang.String>`	**`allUsers`**`()` Gets a list of all users present in the database
`java.util.ArrayList<java.lang.String>`	**`chatMember`**`(int chatID)` Gets list of all users listed in the database under the given private chat ID
`void`	**`createAccount`**`(general.Account account)` Inserts a new account entry into the database
`general.Chat`	**`createPrivateChat`**`(java.lang.String username1, java.lang.String username2)` Adds record of private chat between users to database and creates a populated Chat object with a new chat ID.
`general.Chat`	**`createTeam`**`(java.lang.String teamName, java.util.ArrayList<java.lang.String> usernames)` Adds record of team chat between ArrayList of users to database and creates a populated Chat object with a new chat ID.
`general.Account`	**`getAccount`**`(java.lang.String username)` Gets the account details (as an Account object) matching the username
`java.util.ArrayList<general.Chat>`	**`getChats`**`(java.lang.String username)` Gets list of all chats (as Chat objects) in the database under the given username
`java.util.ArrayList<general.ChatMessage>`	**`getHistoryPrivate`**`(int chatID)` Gets list of all chat messages matching the given private chat ID in the database
`java.util.ArrayList<general.ChatMessage>`	**`getHistoryTeam`**`(int teamID)` Gets list of all chat messages matching the given team ID in the database
`java.util.ArrayList<general.Chat>`	**`getTeams`**`(java.lang.String username)` Gets list of all team chats (as Chat objects) in the database under the given username
`void`	**`login`**`(java.lang.String username, java.lang.String password)` Checks if user has an account.
`void`	**`removeUser`**`(java.lang.String username, int teamID)` Removes given user from the team chat record in the database
`void`	**`savePrivateMessage`**`(general.ChatMessage chatMessage)` Adds record of sent chat message to database under the assigned chat ID.

`void`	**`saveTeamMessage`**`(general.ChatMessage chatMessage)` Adds record of sent chat message to database under the assigned chat ID.
`void`	**`setDeliveredPrivate`**`(int chatID, java.lang.String username)` Sets delivered boolean to true in database for the given user in the matching chat ID
`void`	**`setDeliveredTeam`**`(int teamID, java.lang.String username)` Sets delivered boolean to true in database for the given user in the matching chat ID
`java.util.ArrayList<java.lang.String>`	**`teamMembers`**`(int teamID)` Gets list of all users listed in the database under the given team chat ID
`boolean`	**`usernameFree`**`(java.lang.String username)` Checks if username chosen by new user is already taken

Appendix B : Client/Server Protocol

The protocol is based around the client and server sending instances of general.Message between them. An object of this type has the following field variables:

- a status
- a thread and message ID
- an array of objects

The status has been implemented as an enumeration so that the finite number of statuses available is enforced not only by the protocol but also by the Java compiler (reducing the chances of bugs due to naming inconsistencies). The contents of the array is determined by the message status as described in the text file. When communication is initiated by the client, the server uses an instance of server.Protocol (a field variable in each instance of server.Connection) to parse the message, process it and respond to the client accordingly. The thread and message ID of the original message are used in the response for identification on the client side. When communication is initiated by the server, a thread ID of 0 is always used, and no response from the client is expected.

Communication initiated by the client:

* Client >>> **LOGIN**, objects[0]: String {username}, objects[1]: String {password}
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **CREATE_ACCOUNT**, objects[0]: Account {new account details}
(CLIENT RESPONSIBILITIES: the account details should be complete and valid)
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **VIEW_ACCOUNT**, objects[0]: String {username}
(CLIENT RESPONSIBILITIES: the username exists)
* Server >>> SUCCESS, objects[0]: Account {requested account}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **USERS_ONLINE**, objects : none
* Server >>> SUCCESS, objects[0]: Set<String> {usernames currently connected to server}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **ALL_USERS**, objects : none
* Server >>> SUCCESS, objects[0]: Set<String> {all usernames}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **START_PRIVATE_CHAT**, objects[0]: String {username of recipient}
* Server >>> SUCCESS, objects[0]: Chat {chat details}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **MAKE_TEAM**, objects[0]: String {team name}, objects[1]: ArrayList<String> {usernames}
* Server >>> SUCCESS, objects[0]: Chat {chat details}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **VIEW_PRIVATE_HISTORY**, objects[0]: int {chat id}
* Server >>> SUCCESS, objects[0]: ArrayList<ChatMessage> {chat history}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **VIEW_TEAM_HISTORY**, objects[0]: int {team id}

* Server >>> SUCCESS, objects[0]: ArrayList<ChatMessage> {chat history}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **SEND_PRIVATE_MESSAGE**, objects[0]: ChatMessage {chat message being sent}
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **SEND_TEAM_MESSAGE**, objects[0]: ChatMessage {chat message being sent}
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **ADD_USER**, objects[0]: String {username}, objects[1]: Chat {chat to add to}
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **REMOVE_USER**, objects[0]: String {username}, objects[1]: Integer {team id}
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **GET_PRIVATE_CHATS**, objects[0]: String {username}
* Server >>> SUCCESS, objects[0] : ArrayList<Chat> {all previous private chats}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **GET_TEAM_CHATS**, objects[0]: String {username}
* Server >>> SUCCESS, objects[0] : ArrayList<Chat> {all previous team messages}
* Server >>> ERROR, objects[0] : String {error message}

* Client >>> **LOGOUT**, objects[0]: String {username};
* Server >>> SUCCESS, objects : none
* Server >>> ERROR, objects[0] : String {error message}

<u>Communication initiated by the server:</u>

* Server >>> **JOIN_CHAT**, objects[0]: Chat {details of chat}

* Server >>> **JOIN_TEAM**, objects[0]: Chat {chat to join}

* Server >>> **LEAVE_TEAM**, objects[0]: int {team id}

* Server >>> **RECEIVE_PRIVATE_ MESSAGE**, objects[0]: ChatMessage {the message sent}

* Server >>> **RECEIVE_TEAM_MESSAGE**, objects[0]: ChatMessage {the message sent}

* Server >>> **ADD_TEAM_MEMBER**, objects[0]: int {teamID}, objects[1]: String {username of person added}

* Server >>> **REMOVE_TEAM_MEMBER**, objects[0]: int {teamID}, objects[1]: String {username of person removed}

Appendix C : GUI/Client Protocol

`public interface` **`ClientGUIProtocol`**
`extends java.lang.Runnable`

The following methods can be called by the client GUI to send requests to the server. All methods are void as the server response is not immediate. The documentation indicates how the notify observers method will be called. Where an error response is not specified, the following GUIMessage is sent: status: ERROR, String {error message}

Modifier and Type	Method and Description
`void`	**`allUsers`**`()` requests a list of all user registered with the system -status: ALL_USERS, object: ArrayList<String> {all usernames}
`void`	**`createAccount`**`(Account account)` creates an account and logs the user in -status: ACCOUNT_CREATED, object: String {username} -status: ACCOUNT_CREATED_FAIL, object: String {error message}
`void`	**`getPrivateChats`**`(java.lang.String username)` requests a list of private chats the username is part of -status: ALL_PRIVATE_CHATS, object: ArrayList<Chat> {all private chats}
`void`	**`getTeamChats`**`(java.lang.String username)` requests a list of team chats the username is part of -status: ALL_TEAM_CHATS, object: ArrayList<Chat> {all team chats}
`void`	**`login`**`(java.lang.String username, java.lang.String password)` logs a user into the system GUIMessage: -status: LOGGED_IN, object: String {username} -status: LOGIN_FAIL, object: String {error message}
`void`	**`logout`**`()` requests to be logged out of the system -status: LOGOUT, object: null
`void`	**`makeTeam`**`(java.lang.String teamname, java.util.ArrayList<java.lang.String> usernames)` requests a new team chat with the teamname and usernames specified -status: MAKE_TEAM, object: ChatWindow {chat window to run}
`void`	**`resumePrivateChat`**`(Chat chat)` requests to resume an already begun private chat -status: START_PRIVATE_CHAT, object: ChatWindow {chat window to run}
`void`	**`resumeTeamChat`**`(Chat chat)` requests to resume an already begun team chat -status: MAKE_TEAM, object: ChatWindow {chat window to run}
`void`	**`run`**`()` Checks continually for incoming server messages and processes them. Reestablishes lost connections to the server.
`void`	**`startPrivateChat`**`(java.lang.String username)` requests a new private chat with username specified -status: START_PRIVATE_CHAT, object: ChatWindow {chat window to run}

void	**usersOnline**() requests a list of all users online -status: USERS_ONLINE, object: ArrayList<String> {usernames online}
void	**viewAccount**(java.lang.String username) requests the account belonging to username -status: VIEW_ACCOUNT, object: Account {account requested}

`public interface` **`ChatWindowGUIProtocol`**
`extends java.lang.Runnable`

The following methods can be called by the chat window GUIs to send requests to the server. All methods are void as the server response is not immediate. The documentation indicates how the notify observers method will be called. These updates may also be caused by another using making the same request to the server.

In the case of an error, the following GUIMessage is sent:
-status: ERROR, String {error message}

An object observing ChatWindow can also be expected to be updated with the following GUIMessage once it has been created:
-status: HISTORY, object: ArrayList<ChatMessage> {chat history}

Modifier and Type	Method and Description
void	**addTeamMember**(java.lang.String username) requests that a team member be added -status: ADD_MEMBER, object: String {username added}
void	**removeTeamMember**(java.lang.String username) requests that a team member be removed -status: REMOVE_MEMBER, object: String {username added}
void	**run**() checks continuously for incoming server messages and processes them
void	**sendPrivateMessage**(ChatMessage chatMessage) requests that a private message be sent -status: DISPLAY_MESSAGE, object: ChatMessage {message sent}
void	**sendTeamMessage**(ChatMessage chatMessage) requests that a team message be sent -status: DISPLAY_TEAM_MESSAGE, object: ChatMessage {message sent}

Appendix D : Database JUnit Tests

```
public class DatabaseServerTest
extends java.lang.Object
```

Tests for server/database communication. The database must be cleared each time the tests are run.

Modifier and Type	Method and Description
`void`	**`addTeamMember`**`()` user1 adds user3 to team chat Expected response: SUCCESS Expected information in database: Account created in user table
`void`	**`allUsers`**`()` Obtains list of all users in database Expected response: ArrayList users contains users and is correct size Expected information in database: n/a
`void`	**`createAccount`**`()` Adds another user account to the database for testing
`void`	**`createAccountFail`**`()` Attempts to add user to database where username is already taken Expected response: ERROR Expected information in database: No information stored
`void`	**`createAccountSuccess1`**`()` Adds a user account to the database Expected response: SUCCESS Expected information in database: Account created in user table
`void`	**`createAccountSuccess2`**`()` Adds a user account to the database Expected response: SUCCESS Expected information in database: Account created in user table
`void`	**`finish`**`()`
`void`	**`getPrivateChats1`**`()` Gets list of private chats for user1 Expected response: SUCCESS & ArrayList Expected information in database: n/a
`void`	**`getPrivateChats2`**`()` Checks that the returned private chats are correct Expected response: ArrayList of Chats only contains the one chat with user2 Expected information in database: n/a
`void`	**`getTeamChats1`**`()` Obtains team chat history for user1 Expected response: SUCCESS & ArrayList Expected information in database: n/a
`void`	**`getTeamChats2`**`()` Checks that the returned team chats are correct Expected response: ArrayList of Chats only contains the one chat with user2 Expected information in database: n/a

`void`	**`getTeamHistory`**`()` Checks that team chat history is retrieved correctly and user3 can view all previous messages Expected response: ChatMessage contains the message from user1 reading "hello" Expected information in database: n/a
`void`	**`getTeamsNotInChat`**`()` user3 can no longer see the team chat they were previously part of Expected response: ArrayList of Chat has no elements Expected information in database: n/a
`void`	**`initialise`**`()`
`void`	**`loginFail1`**`()` Log in fails as user does not exist Expected response: ERROR Expected information in database: n/a
`void`	**`loginFail2`**`()` Log in fails as password in database does not match Expected response: ERROR Expected information in database: n/a
`void`	**`loginSuccess`**`()` Logs in user by querying database Expected response: SUCCESS Expected information in database: n/a
`void`	**`makeTeam`**`()` Creates a new team with user1 and user2 Expected response: SUCCESS & Chat object Expected information in database: Team is stored in team and team_member tables
`void`	**`removeUser`**`()` Adds a user account to the database Expected response: REMOVE_TEAM_MEMBER & returned message contains correct information Expected information in database: user3 is removed from team_members
`void`	**`sendPrivateMessages`**`()` Checks that the returned private chats are correct Expected response: RECEIVE_PRIVATE_MESSAGE & message text and senders username are correct Expected information in database: Message, stored correctly
`void`	**`sendTeamMessage`**`()` Checks that team messages are sent correctly Expected response: RECEIVE_TEAM_MESSAGE & message text and sender's username are correct Expected information in database: Message, stored correctly under the correct team ID
`void`	**`startPrivateChat1`**`()` Starts private chat between user1 and user2 Expected response: SUCCESS & Chat object Expected information in database: Chat logged in database with a unique chat ID
`void`	**`startPrivateChat2`**`()` Checks that a private chat between two users is given its assigned chat ID if a chat between them already exists, rather than generating a new one Expected response: ChatID equals ID in the database Expected information in database: No new chat logged in database

void	**viewAccount01**() Obtains account information from the database Expected response: SUCCESS & Account object Expected information in database: n/a
void	**viewAccount02**() Checks account information obtain from the database is correct Expected response: correct username & email Expected information in database: n/a
void	**viewPrivateChatHistory1**() Obtains chat history for user1 Expected response: SUCCESS & ArrayList Expected information in database: n/a
void	**viewPrivateChatHistory2**() Checks obtained chat history contains the message from user1 reading "hello" Expected response: ChatMessage object Expected information in database: n/a
void	**viewTeamChatHistory1**() Obtains team chat history for user1 Expected response: SUCCESS & ArrayList Expected information in database: n/a
void	**viewTeamChatHistory2**() Checks obtained chat history contains the message from user1 reading "hello" Expected response: ChatMessage object Expected information in database: n/a

Appendix E : Server JUnit Tests

```
public class ServerTest
extends java.lang.Object
```

These tests write messages to the server and check the response Expected response gives the message status and, if objects are expected, their types. The server must be running with TestDatabase to use these tests.

Modifier and Type	Method and Description
`void`	**`addUserFail`**`()` Expected response: ERROR
`void`	**`addUserSuccess`**`()` Expected response: ADD_TEAM_MEMBER
`void`	**`allUsersSuccess`**`()` Expected response: SUCCESS, ArrayList
`static boolean`	**`compareMessages`**`(Message messageA, Message messageB)` compares the status, threadID and messageID of two messages
`void`	**`createAccountFail`**`()` Expected response: ERROR
`void`	**`createAccountSuccess`**`()` Expected response: SUCCESS
`void`	**`finish`**`()` Closes socket and input/output streams
`void`	**`getPrivateChatsFail`**`()` Expected response: ERROR
`void`	**`getPrivateChatsSuccess`**`()` Expected response: SUCCESS, ArrayList
`void`	**`getTeamChatsFail`**`()` Expected response: ERROR
`void`	**`getTeamChatsSuccess`**`()` Expected response: SUCCESS, ArrayList
`void`	**`initialise`**`()` Initialises socket and input/output streams
`void`	**`loginFail`**`()` Expected response: ERROR
`void`	**`loginSuccess`**`()` Expected response: SUCCESS
`void`	**`makeTeamFail`**`()` Expected response: ERROR
`void`	**`makeTeamSuccess`**`()` Expected response: SUCCESS, Chat

`void`	**`removeUserFail`**`()` Expected response: ERROR
`void`	**`removeUserSuccess`**`()` Expected response: REMOVE_TEAM_MEMBER, String
`void`	**`sendPrivateMessageFail`**`()` Expected response: ERROR
`void`	**`sendPrivateMessageSuccess`**`()` Expected response: SUCCESS, ChatMessage
`void`	**`sendTeamMessageFail`**`()` Expected response: ERROR
`void`	**`sendTeamMessageSuccess`**`()` Expected response: SUCCESS, ChatMessage
`void`	**`startPrivateChatFail`**`()` Expected response: ERROR
`void`	**`startPrivateChatSuccess`**`()` Expected response: SUCCESS, Chat
`void`	**`viewAccountFail`**`()` Expected response: ERROR
`void`	**`viewAccountSuccess`**`()` Expected response: SUCCESS
`void`	**`viewPrivateHistoryFail`**`()` Expected response: ERROR
`void`	**`viewPrivateHistorySuccess`**`()` Expected response: SUCCESS, ArrayList
`void`	**`viewTeamHistoryFail`**`()` Expected response: ERROR
`void`	**`viewTeamHistorySuccess`**`()` Expected response: SUCCESS, ArrayList

Appendix F : Client JUnit Tests

```
public class ClientTest
extends java.lang.Object
```

These tests use a spy version of client and a dummy observer to check that when client methods are called, both field variables and the observer are updated as expected. The server must be running with ClientTestDatabase to use these tests

Modifier and Type	Method and Description
`void`	**`connect`**`()` initialises client and testObserver
`void`	**`disconnect`**`()` stops testObserver, which releases resources in client
`void`	**`test01Login`**`()` Login with valid username and password. Expected behaviour. Expected behaviour: username is initialised
`void`	**`test02Login`**`()` Login with valid username and password. Expected behaviour: allPrivateChats and allTeam-Chats are initialised
`void`	**`test03Login`**`()` Login with valid username and password. Expected behaviour: observer updated with login and chat history messages
`void`	**`test04Login`**`()` Login with valid username and password. Expected behaviour: observer updated with correct object types (String + 2x ArrayList)
`void`	**`test05Login`**`()` Login with invalid password. Expected behaviour: observer updated with error message
`void`	**`test06Login`**`()` Login with invalid password. Expected behaviour: observer with correct object type (String)
`void`	**`test07Login`**`()` Login with non-existent credentials. Expected behaviour: observer updated with error message
`void`	**`test08Login`**`()` Login with non-existent credentials. Expected behaviour: observer with correct object type (String)
`void`	**`test09CreateAccount`**`()` Create account with valid details. Expected behaviour: username initialised
`void`	**`test10CreateAccount`**`()` Create account with valid details. Expected behaviour: allPrivateChats and all TeamChats initialised
`void`	**`test11CreateAccount`**`()` Create account with valid details. Expected behaviour: observer updated with account created and chat history

void	**test12CreateAccount()** Create account with valid details. Expected behaviour: observer updated with correct object types (String + 2x ArrayList)
void	**test13CreateAccount()** Create account with invalid password. Expected behaviour: observer updated with error message
void	**test14CreateAccount()** Create account with invalid password. Expected behaviour: observer updated with String object
void	**test15CreateAccount()** Create account but database error. Expected behaviour: observer updated with error message
void	**test16CreateAccount()** Create account but database error. Expected behaviour: observer updated with String object
void	**test17StartPrivateChat()** Start private chat successfully. Expected behaviour: chat added to allPrivateChats
void	**test18StartPrivateChat()** Start private chat successfully. Expected behaviour: chat added to current chats
void	**test19StartPrivateChat()** Start private chat but database error. Expected behaviour: chat not added to current chats
void	**test20StartPrivateChat()** Start private chat successfully. Expected behaviour: observer notified of new chat
void	**test21StartPrivateChat()** Start private chat successfully. Expected behaviour: ChatWindow passed to observer
void	**test22StartPrivateChat()** Start private chat but database error. Expected behaviour: observer updated with error message
void	**test23StartPrivateChat()** Start private chat but database error. Expected behaviour: observer updated with String object
void	**test24MakeTeam()** Make team successfully. Expected behaviour: chat added to allTeamChats
void	**test25MakeTeam()** Make team successfully. Expected behaviour: chat added to current chats
void	**test26MakeTeam()** Make team successfully. Expected behaviour: observer updated with make team message
void	**test27MakeTeam()** Make team successfully. Expected behaviour: observer updated with chat window
void	**test28MakeTeam()** Make team but database error. Expected behaviour: chat not added to current chats
void	**test29MakeTeam()** Make team but database error. Expected behaviour: observer updated with error message
void	**test30MakeTeam()** Make team but database error. Expected behaviour: observer updated with String object

void	**test31ThreadCounter**() Increment thread counter by starting private and team chats. Expected behaviour: thread counter incremented by 2 to 4
void	**test32AddTeamMember**() Add team member successfully. Expected behaviour: username added to list in chat
void	**test33AddTeamMember**() Add team member but already in the team. Expected behaviour: size of username list in chat does not change
void	**test34RemoveTeamMember**() Remove team member successfully. Expected behaviour: username no longer in list of user-names in chat
void	**test35RemoveTeamMember**() Remove team member who isn't in team. Expected behaviour: size of username list is still the same
void	**test36JoinTeam**() Join team successfully. Expected behaviour: team added to allTeamChats
void	**test37JoinTeam**() Join team successfully. Expected behaviour: observer updated with make team message
void	**test38JoinTeam**() Join chat successfully. Expected behaviour: observer updated with chat window object
void	**test39LeaveTeam**() Leave team successfully. Expected behaviour: chat removed from all team chats
void	**test40LeaveTeam**() Leave team successfully. Expected behaviour: observer updated with leave team message
void	**test41LeaveTeam**() Leave team successfully. Expected behaviour: observer updated with integer
void	**test42ReceivePrivateMessage**() Receive private message for chat without window open. Expected behaviour: new chat window opened and added to current chats
void	**test43ReceivePrivateMessage**() Receive private message for chat without window open. Expected behaviour: observer updated with start private chat message
void	**test44ReceivePrivateMessage**() Receive private message for chat without window open. Expected behaviour: observer updated with chat window object
void	**test45ReceiveTeamMessage**() Receive team message for chat without window open. Expected behaviour: chat window cre-ated and added to current chats
void	**test46ReceiveTeamMessage**() Receive team message for chat without window open. Expected behaviour: observer updated with make team chat message
void	**test47ReceiveTeamMessage**() Receive team message for chat without window open. Expected behaviour: observer updated with chat window object

public class **ChatWindowPrivateChatTests**
extends java.lang.Object

Modifier and Type	Method and Description
void	**getPrivateHistory_fail_GUIMessageError**() Error gaining private message history. Expected behaviour: sends GUI message with status ERROR.
void	**getPrivateHistory_success_GUIMessageHistory**() Successfully retrieve past private message history. Expected behaviour: Expected behaviour: sends GUI message with status HISTORY.
void	**receiveMessage_success_inHasNewMessage**() Successfully sends. Expected behaviour: Message added to LinkedBlockingQueue in.
Void	**sendPrivateMessage_fail_GUIMessageError**() Error gaining private message history. Expected behaviour: sends GUI message with status ERROR.
Void	**sendPrivateMessage_success_noGUIMessageError**() Successfully sends. Expected behaviour: None expected. Checked that no error was sent.

public class **ChatWindowTeamChatTests**
extends java.lang.Object

Modifier and Type	Method and Description
Void	**addTeamMember_fail_GUIMessageError**() Error adding a team member. Expected behaviour: sends GUI message with status ERROR.
Void	**addTeamMember_success_GUIMessageAddMember**() Successfully adds team member. Expected behaviour: sends GUI message with status ADD_MEMBER
Void	**getTeamHistory_fail_GUIMessageError**() Error gaining team message history. Expected behaviour: sends GUI message with status ERROR.
Void	**getTeamHistory_success_GUIMessageHistory**() Successfully retrieves team history. Expected behaviour: sends GUI message with status HISTORY.
Void	**removeTeamMember_fail_GUIMessageError**() Fail to remove team member. Error removing member. Expected behaviour: sends GUI message with status ERROR.
Void	**removeTeamMember_success_GUIMessageRemoveMember**() Successfully removes a team member. Expected behaviour: sends GUI message with status REMOVE_MEMBER.
Void	**sendTeamMessage_fail_GUIMessageError**() Fails to send team message. Error sending team message. Expected behaviour: sends GUI message with status ERROR.
Void	**sendTeamMessage_success_noGUIMessageError**() Successfully sends. Expected behaviour: None expected. Checked that no error was sent.

Appendix G : Manual Test Plan

Action Tested	Action Taken	Expected Behaviour
Creating Account (basic)	Click create account, entered basic information	Information would store to database allowing us to login. Screen would show main gui.
Creating Account (extra)	Click create account, entered all information.	Information would store to database allowing us to login. Screen would show main gui.
Creating Account unsuccessfully	Click create account. Do not enter required info	Error
Creating Account unsuccessfully	Click create account. Do not enter password >8 characters	Error
Logging in successfully	Enter the stored username and password	Main gui shown on screen
Logging in unsuccessfully	Enter incorrect username / password	Nothing happens
Starting a private chat	Click on one person and start chat.	Private chat gui should pop up on other users screen and current users screen
Sending message on private chat	Type in a message in appropriate text field on private chat gui and click send	The message should appear on both users screens in the history area.
Receiving message on private chat	Get other user to send a message	The message should appear on both users screens in the history area.
Closing private chat window	Click close on private chat screen gui	Only closes that window.
Reopening old private chat	Open private chat again	Old messages sent should appear in history
Starting a team chat	Click on more than one person and start chat.	team chat gui should pop up on other users' screen and current users screen
Sending message on team chat	Type in a message in appropriate text field on team chat gui and click send	The message should appear on all users screens in the history area.
Receiving message on team chat	Get other user to send a message	The message should appear all users screens in the history area.
Closing team chat window	Click close on private chat screen gui	Only closes that window.
Reopening old team chat	Open team chat again	Old messages sent should appear in history
Opening profile	Click profile	Should see information excluding password and email
Closing profile	Close profile	Should only close the profile screen.
Closing Main GUI	Click close on main gui. Once closed open the program again.	Should close all the screen and log out.
More than one chat	Check that more than one chat can be fulfilled at once	All functionality should be allowed.

Appendix H : Project Diary

Diary.

24/02/15

TEAM MEETING: Full group in attendance.
ASK CORY : ARE WE ALLOWED TO USE THIRD PARTY API'S - (Update: Completed 26/02 - YES).
Specification:

- Dev chat.
- BASE GOAL:
    - Build basic messaging system.
- STRETCH GOALS:
    - Multiple people in chats
    - More than one chat open at the same time
    - Opening links
    - integrate a similar format to codshare.io.
- Need to set out guidelines clearly
    - Requirements (thinking of user etc)
    - Information / structure of databases
    - Communication methods are key.
- Not number 6 on the spec.

ORGANISATION:

- Team Structure:
    (with flexibility)
    Harry - GUI. Jess - Client. Divyjyot - Client / GUI (flexi) Sam - JDBC / Databases. Anna - Server/Sockets.
- Architecture:
    - Standard.
    - Model/Client , View/GUI
- Protocols
    - May be a starting point.
    - will determine what everything else needs.
    - will help break into teams.

26/02/15

MEETING WITH CORY: Full group in attendance.

- Most of our current stretch goals should be basic goals. Only stretch goals
  at the moment are code/ pictures (but if we can achieve the code bit we are awesome)
- WARNING: Links may have problems working across different OSs.
- Mention about work history / write work flow.
- Text, Login, Instructions. Work on protocol, what will be sent across with each message?
- Have a clear goal of how different parts are communicating/ how things are sent between elements.
    (Update - conducted meetings)
- getChatHistory()
- THERE SHOULD NEVER BE A BARE DB CALL IN PROGRAM.
- HANDLERS - these will be important for making sure when we add functionality nothing goes wrong.
- Get a skeleton uploaded for next Thursday/Friday. (For Handlers or for methods. Having communication between
  Client and Server implemented).

TEAM MEETING: Full group in attendance.

Organisation:

- Conduct separate meetings to figure out protocols/ what is needed to be communicated between different areas.
    - Weekend: Divyjyot and Anna to discuss Server/Client.
    - Monday: Anna & Sam to discuss Server/DB, Sam & Jess to discuss DB/Client

Architecture:

- How information was to be organised an sent/received:

  Sending Stuff - Sending Java objects? Object to be sent from Client to Server - Server deconstructs to send to SQL.

  This avoids the issues that Threads do not send stuff in an order.

Database:

  - Within DB side can create paired statements - SQL query with ? for bits you want to change.
  - DB Class with methods to Call.
  - Will store time stamp for messages to help order them.

Chat issues:

- Discussed how we were going to arrange the "responsibility of chats". This was important as we wanted to be able to keep DevChats open after one person left. It raised issues of ownership and of privacy (i.e. if someone was to join would they see the previous conversation etc.).
  - We are going to attempt to make two categories of chat messages - private and DevChat
  - Private chats will be between two people. Only those two people will be able to access chat history.
  - DevChats will be between 3+ people. Chat history will be open for searching.
  - For each chat, each time a person joins or goes, it makes a different chat ID.
  - A chat history will be saved an relayed between two people. If another person joins, from then on it is saved as a DevChat with a separate ID. This should mean if two people are chatting and a third is added, it will become a DevChat and the previous history of the original two will not be shown.
  - To achieve this, if a DevChat is started (more than 1 person added to a conversation) the chat will open in a new window.
  - Privacy for DevChats shouldn't be an issue as the program is designed for a professional network and is designed to work inhouse.

Possible Stretch Goals:

  - GUI: different colours for the two types of chat.
  - Being able to hide messages?

28/02/15

Anna/Divyjyot uploaded to SVN:

  - An initial Java version of the protocol
  - Text version of the protocol
  - Some classes necessary for writing protocol but currently don't do anything
  - A text file containing some design issues we need to discuss.

02/02/15

Sam/Jess Meeting:

DEVCHAT QUESTIONS:

DevChat for each team? But then what about cross team communication / access?
Chat rooms seems to be best for DevChat otherwise multiple windows will have to be opened.

GUI considerations:

- Search boxes (language, username, department(?))
- Search by date for DevChat.
- Profile - Separate screen, hover over to get box, right click to get profile??
- Chat rooms for DevChat?
- Set up a DevChat, enter team ID, team name.
  - need separate admin screen.
- Maybe an opening screen with options (I.e Start Private Chat, Start DevChat, Search For Chat, Admin Options, Edit Profile).

Client:

- Admin methods.
- Customisable - setting methods (colour etc)?

Objects server to client:

- basic information object

- Should we limit group size for DevChat?
- Need to make USER object

Connection Errors:

-If connection drops out for short period so message can't be sent - error or try again y/n?
Re-establishing connection protocol. - need something in server to check if their online.

Privacy and Login:

- user_ID never visible to user, username would be visible.
- LOGIN: create new user object with ______. See if it matches what's on database.

03/03/15

- Full group in attendance
- Jessica made permanent Secretary

DevChat:

- DevChat as Chat Room.
- Keep Private Chat and DevChat completely separate
- Set colour for private and another for DevChat.

IDs:

- Associate session ID with User ID? or Server sends session ID to Client?:
    - Each thread in Client will need an ID. (Number to ID thread and ID message)
    - Server can send those two numbers to confirm.

If someone goes offline/Offline messaging:

- When they reconnect will automatically appear as the message will be written to the database and then sent to the other client.
- Need a way for the Client to recognise there has been a connection error? Need to catch end of file exception.

- Time stamp as Date type.
- Record sending time (because multithreaded). Keep the chronology related to the Client
- As writing messages to the database, it means messages can be sent when users are "offline"
- Boolean T/F for if someone is in a chat. "Some messages haven't been seen" rather than saying which hasn't been seen.

Account Class:

- Languages
- Username
- Password

- Client needs to ensure nothing invalid gets passed to DB.

Agreed Basic:

GUI:

- Display/Receive/Send

Client:

- Connect to Server
- Send messages to Server
- Hard code: Thread ID and Chat ID.

Server to DB:

- Add Messages
- Retrieve Messages.
- Server stores messages in lookup tables.

Connections:

Client <=> GUI
Client <=> Server
Server <=> DB

Test Plan

- Each team write their manual test, then swap with other team for additional tests.

Packages:

- One for each section
- Classes which are shared -
- Messages class - comparator method which sorts by timestamp.

What does an Account have:

- AccountID ==> Client gets from DB
- User ID
- Password
- Email
- Languages
- Location
- Company
- Job title
- FName
- LName
- Profile (Stretch: to be able to update).

Set Work for next meeting:

- Skeleton for GUI, Client & Server by 05/03.

10/03/15

Agenda and discussion points for meeting (Tuesday 10th March)

Structure of Client

- Threads

  => needs to be multi-threaded.
  => Client and GUI will be observable and runnable. GUI will have the main and the client will have a run() class.
  => Main Client has a lot of field variables.
    - All threads have access too and write to the queue
    - Map of chat objects.
  => GUI needs to be threaded.

- Responses

  - Each ChatWindow has a queue for receiving messages.
  - ISSUE: Don't know when you are getting a message back. ==> REQUEST CLASS
    The Request Class has one method response.

0 - Server 1 - Main Class i - Chat window.

- Observable and Runnable

    - Client updates GUI by notifying of observable.
    - Protocol based around the GUI message class.
    - Structure is designed to be scalable.

Structure of GUI

- Define exactly the role of the GUI

    - Harry has notes on this.

- Running and communicating with the client and chat windows

    - View Class - need to run in thread and start a thread for a chat window.
        - may need runnable WindowView class for chat windows.
            - Set<Thread>
            - new Thread(new WindowView(Object));
    - Client
    - Thread Client

Issues which I need to think more about!

- Delivered/undelivered messages- retrieving undelivered messages
- Closing the system
- reconnecting to the system

Errors/Verification

- Error messages- let's programme as though everything will always work and think about this on Thursday
- Because of model view separation, the client must check that all information entered is correct and inform the GUI if it isn't- but this comes under error messages

Work to do:

- GUI:
    - sketched out design of the various frames/panels- where will the all the methods be called from? How will the transitions work?
        - Harry and Divyjyot - above.
- Client:
    - implement method stubs
        - Jess : Chat Window implementation.
        - Anna : Methods.
- Server/DB:
        - Sam.
    - update for new version of queries
    - discuss team chats
    - discuss errors
    - (start testing)

Provisional deadlines:

- Thursday 12th:
    - as much of the above as possible (we need enough time for it to all not work and to fix it!)
- Tuesday 17th:
    - most, if not all of the programming
- Thursday 19th
    - all testing

This leaves us with the weekend to write the report

12/03/15

Code not being used - put in report and in the header of methods.

Agenda for the weekend

Jess

Sort our success' and error's.
Add what we need to to GUIMessage class
Read Client - find holes
Start thinking about testing
Check protocols

Divyjyot

Updates observable.
Transitions.

Harry

Making / sorting all the frames.

Sam

Testing

Anna

Client - thread safe
Closing things

18/03/15

In attendance: Divyjyot, Harry, Jess

Methods tested and results / things to do
----

1. public void createAccount(Account account) - Works. GUI could make cancel go back to login screen and make a reset button for create account.
2. public void login(String username, String password) - Success : calls main GUI
   fail: error GUI pop's up with relevant message
   TODO: check if info stores to DB
3. public void viewAccount(String username)
   TODO: needs profiles to access account object and pull other users
   information from that object to display on the profile.
   String username; String firstname
   String lastname; String languages;
   String location; String company;
   String jobTitle; String website;
   String personalInfo
4. public void allUsers() - Need to code in a timer/ timertask for refreshing contact panel (GUI)
5. public void usersOnline() - Sorted
6. public void startPrivateChat(String username) - Launches. Can close screens individually.
7. public void sendPrivateMessage(ChatMessage chatMessage) - Didn't work
8. public void makeTeam(String teamname, ArrayList<String> usernames); - Launches
   TODO: Display names of participants on both the chats
   (Currently says Enter Username / Team Name)

Checked : Can open numerous screen11s and close individually.

19/03/15

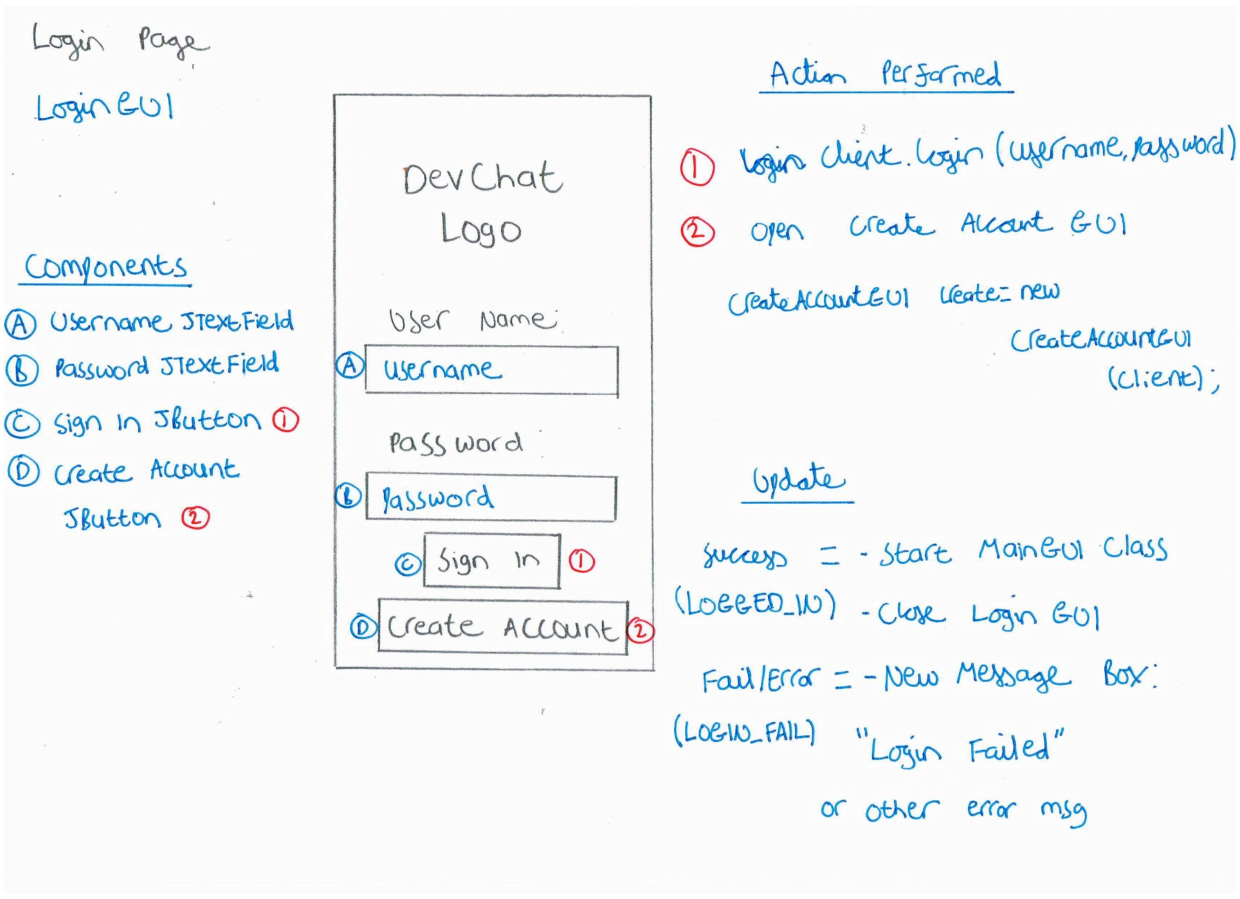
Methods tested / results of tests
----
1. public void createAccount(Account account) - Works. Details store to DB.
2. public void login(String username, String password) - Works. Details store in DB. Checks are made on login
3. public void viewAccount(String username) - Profile GUI is to be made before this is fixed.
4. public void allUsers() - FINE FROM DAY BEFORE. Not currently used.
5. public void usersOnline() - This is now called in contact panel with a times to refresh.
6. public void startPrivateChat(String username) - FINE FROM DAY BEFORE
7. public void sendPrivateMessage(ChatMessage chatMessage) - Works but messages aren't being stored to DB
8. public void makeTeam(String teamname, ArrayList<String> usernames); - can't test without team gui
9. public void sendTeamMessage(ChatMessage chatMessage) - can't test without team gui
10. public void addTeamMember(String username) - can't test without team gui
11. public void removeTeamMember(String username) - can't test without team gui
12. public void resumePrivateChat(Chat chat); - Almost. A few DB problems to fix
13. public void resumeTeamChat(Chat chat) - Almost complete. Can't test team chat yet.
14. public void logout - method is there. need logout button - this will kill all the chat window threads.

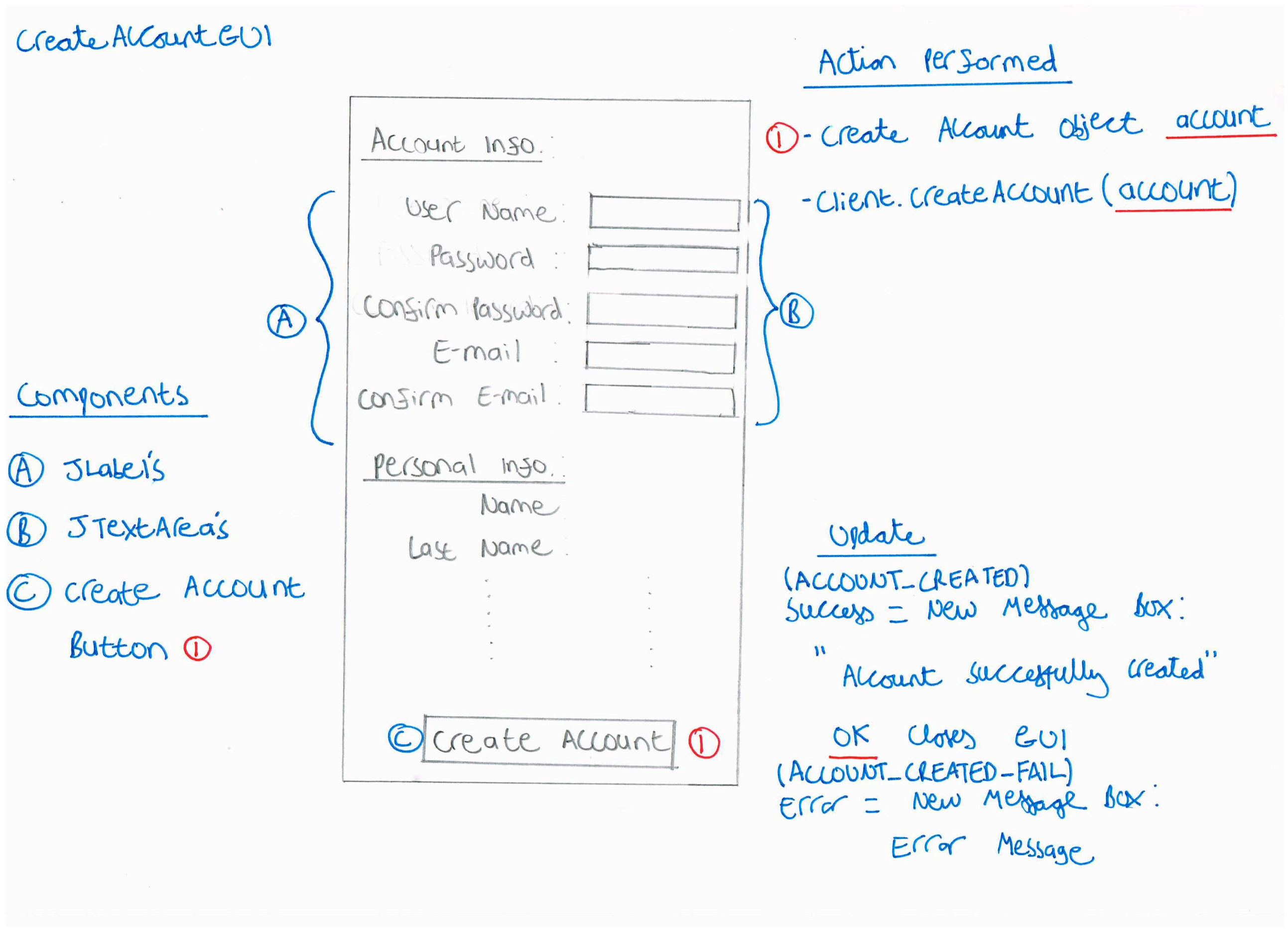
Extra Notes
----
o Private chats has had toolbar removed.

AGENDA FOR THE WEEKEND:

Jess - Report / Start presentation. look into errors.
Divyjyot - Invite remove GUI make, new guis work (profile gui), make team gui work
Harry - Report, view someone else's profile, viewing own.
Anna - Report, tests, fixing
Sam - Report.

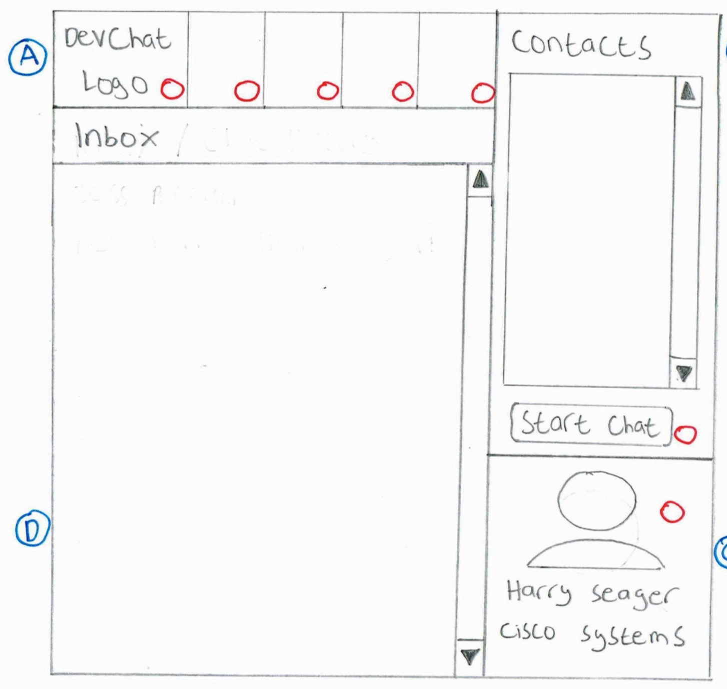
Appendix I : GUI Diagrams


Login Page
LoginGUI
Components
Ⓐ Username JTextField
Ⓑ Password JTextField
Ⓒ Sign In JButton ①
Ⓓ Create Account JButton ②
DevChat Logo
User Name
Ⓐ Username
Password
Ⓑ Password
Ⓒ Sign In ①
Ⓓ Create Account ②
Action Performed
① Login client.login(username, password)
② Open Create Account GUI
CreateAccountGUI create = new CreateAccountGUI(client);
Update
Success (LOGGED_IN) = - Start MainGUI Class - Close Login GUI
Fail/Error (LOGIN_FAIL) = - New Message Box: "Login Failed" or other error msg


CreateAccountGUI
Account Info.
User Name:
Password:
Confirm Password:
E-mail:
Confirm E-mail:
Personal Info.
Name
Last Name
Ⓒ Create Account ①
Ⓐ
Ⓑ
Components
Ⓐ JLabel's
Ⓑ JTextArea's
Ⓒ Create Account Button ①
Action Performed
① - Create Account object account
- Client.createAccount(account)
Update
(ACCOUNT_CREATED) Success = New Message Box: "Account successfully created"
OK closes GUI
(ACCOUNT_CREATED-FAIL) Error = New Message Box: Error Message

Main GUI

○ : Trigger Action



B Components

A Toolbar

B Contacts List

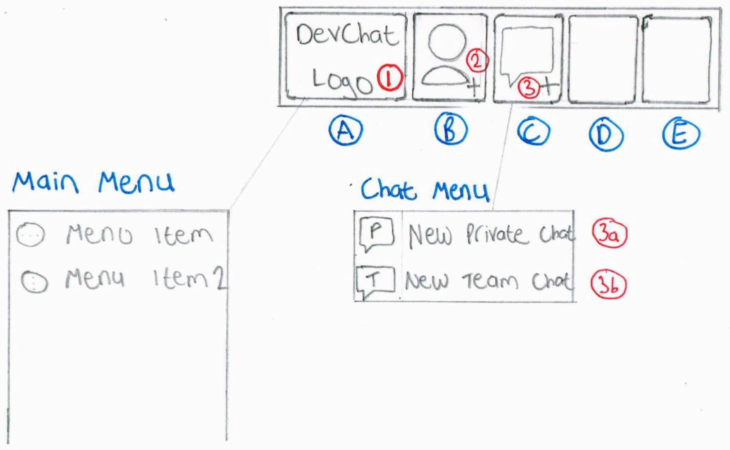
C Profile

D Inbox

All above components are individual classes

C which extend JPanel. MainGUI will then add each JPanel individually

A Toolbar



Action Performed

① Open Main Menu

② Open 'Add ContactGUI'

③ Open Menu to select chat Type

3a Open 'NewPrivateChatGUI'

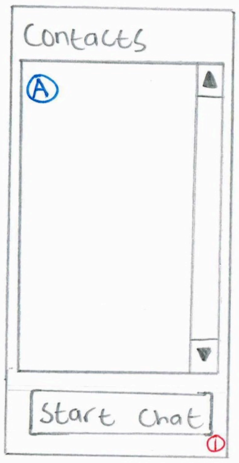
3b Open 'NewTeamChatGUI'

Ⓑ Contacts List
(Threaded)

Components

~~JList~~

A) JList with MULTIPLE_INTERVAL_SELECTION



Action performed

① If single selection:
Client.startPrivateChat (username)
If Multiple selection:
~~to~~ Open NewTeamChat GUI
(pass arrayList <usernames>)

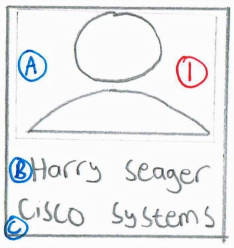
Update

Ⓐ USERS_ONLINE (At top)
ALL_USERS (If not online - at bottom)

Ⓒ Profile

Components

Ⓐ JButton
Ⓑ JLabel
Ⓒ JLabel



Action performed

① Client.viewAccount

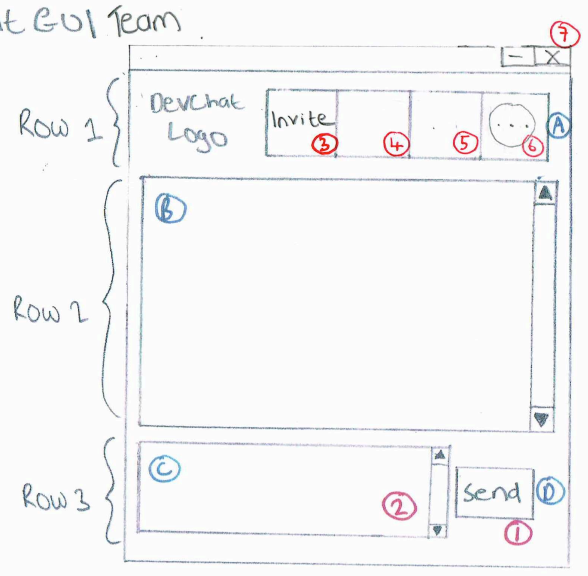
Update

Ⓐ Update Photo Icon

Ⓑ Update Name

Ⓒ Update Company Name

Chat GUI Team



Action performed

1. Chatwindow. send Team Message (message)
2. " " " " " {Called on 'Enter
3.

6. Opens more options menu bar
7. Chatwindow. Close()

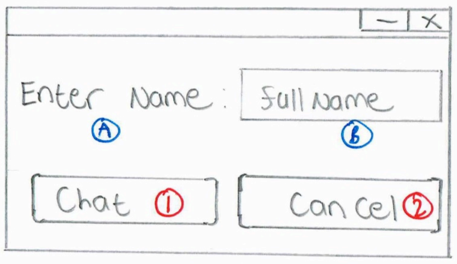
Update

B Update with HISTORY

Components

A	=	Toolbar
B	=	Message History JTextArea
C	=	Send Message JTextArea (Editable)
D	=	Send Message Button ✗

New Private Chat GUI



Action performed

1. Client. Start Private Chat (username IS (fullName))
2. Close window Cmd

Components

A JLabel

B JComboBox (For auto complete)

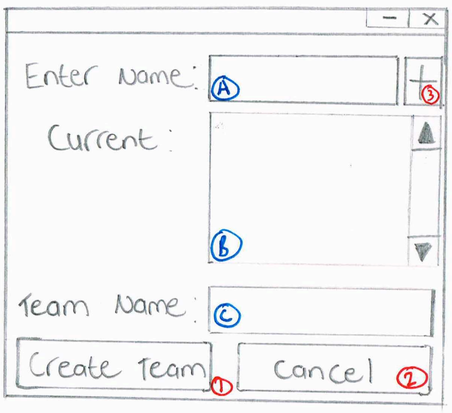
New Team Chat GUI

Components

JComboBox

) JList

) JTextBox



Action Performed

① Client.makeTeam (teamname, usernames)

② Close window cmd

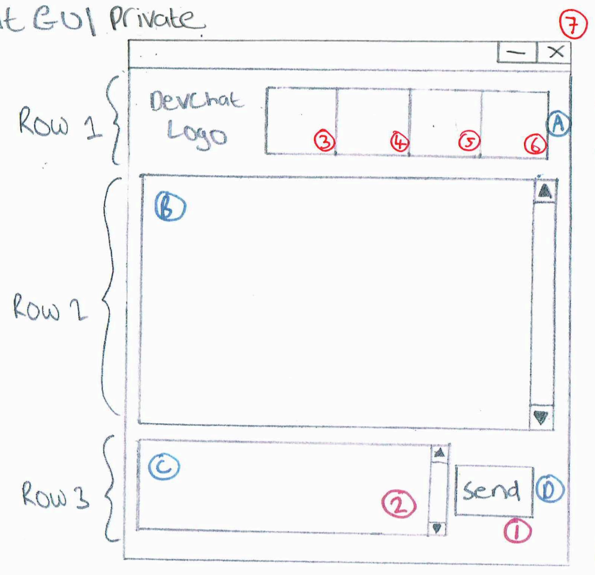
(what to do with Created Arraylist?)

③ Need method to add to current ArrayList of team members

Update

Ⓑ List of selected Team Members

Chat GUI Private



Action Performed

① ChatWindow. sendPrivateMessage (message)

② " " " " " {Called on 'Enter'

Update

Ⓑ Update with HISTORY

Components

Ⓐ = Toolbar

Ⓑ = Message History JTextArea

Ⓒ = Send Message JTextArea (Editable)

Ⓓ = Send Message Button ✗