Software Design & Implementation

Project Report

Group Members:

Hannah Ashna Jacob (N0865554)

Jarad Johnson Bailey (N0853071)

Hassaan Naveed (N0898071)

Nicholas McCaig (N0787115)

Lab Tutor:  
Pedro Baptista Machado

# Project Abstract:

The Covid-19 pandemic has significantly changed the way we communicate. Our reliance on digital applications for communication has increased, and thus, the demand for such applications has also grown. Our team was tasked with designing and implement a messages exchange platform using C++ for our Software Design and Implementation (SDI) module. This project uses MQTT to facilitate communication using a Client-Server model. We have based our application on two existing messaging services, Discord and Slack. Discord is a gaming-oriented application with a range of features that allows users to communicate with others while in-game (Discord, 2021). On the other hand, slack is a messages exchange platform service that is geared towards workplace communication (Slack, 2021).

The overall aim of our project is to allow users to log in to our platform using their credentials, connect to the service and begin exchanging messages amongst each other. Users can create and delete chat rooms. Within each room, a user can have multiple discussion channels wherein multiple conversations can run concurrently. The user can switch between these channels depending on which conversation they wish to participate in. Moreover, the application also has admin and moderator roles so that users within a room can control who is added or removed from a room. In addition to this, our team also implemented password-encryption capabilities to ensure that user accounts were secure since they had to be stored locally.

Through our team’s evaluation of the project upon completion, we found that we were able to effectively design and implement the application within the given time frame due to our detailed research process and diligent adherence to our project plan. We did conclude that for future iterations of this project it would be more effective to use a database instead of text files as this would improve the overall efficiency of the project and prevent data from being stored in numerous locations.

# Plagiarism Declaration:

This report and the software it documents is the result of my own work. Any contributions to the work by third parties, other than tutors, are stated clearly below this declaration. Should this statement prove to be untrue I recognise the right and duty of the Board of Examiners to take appropriate action in line with the university’s regulations on assessment.

**Name:** Hannah Ashna Jacob (ID – N0865554)

**Name:** Jarad Johnson-Bailey (ID - N0853071)

**Name:** Hassaan Naveed (ID – N0898071)

**Name:** Nicholas McCaig (ID – N0787115)

# Revision History:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Issue Date** | **Stage** | **Changes** | **Author** |
| 1.0.0 |  | Alpha | Created and structured report template per guidelines | Hannah |
| 1.0.1 |  | Alpha | Added Risk Analysis, Requirements List, Contribution guide and Coding Standards guide | Hannah |
| 1.0.2 |  | Alpha | Added Gantt Chart | Hannah |
| 2.0.0 |  | Alpha | Added Diagrams | Hannah |

# Table of Contents:

# List of Tables:

# List of Images:

# Introduction:

# Background Research:

An overview of external tools and libraries that were used to implement the application:

* QT
* Boost
* Crypto Library
* MQTT Mosquito

# List of Requirements & Tasks:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No** | **Requirement** | **Priority** | **Implications** | **Tasks** |
| 1 | Users **must** be able to send, receive and view messages through the application | MUST | Without this feature, the application’s primary function would not be possible as a message exchange platform relies on users being able to send and receive messages | **T2.1.0:** Setup and Configure Client/Server using MQTT  **T4.0.0:** Implement User Interface |
| 2 | Users **must** be able to create chat rooms (rooms with more than two contacts) | MUST | A key feature in a messaging platform is the ability to chat with more than one person simultaneously, hence, the need for chat rooms | **T2.1.1:** Setup and Configure Client/Server using MQTT  **T3.2.0:** Create separate user classes |
| 3 | The user that creates the chat room **must** be classified as Admin | MUST | This is to prevent other users from making modifications to a chat room when they did not create it | **T3.0.0:** Create separate user classes |
| 4 | The moderator **must** be able to invite and remove users from a chatroom | MUST | This allows for a chat room to expand its user base numbers or remove certain members if necessary | **T3.1.1:** Create separate user classes |
| 5 | Moderators **must** inherit all the admin permissions; however, Moderators **cannot** demote the Admin | MUST | The functionality of the Moderator and Admin is essentially the same, however, the Admin is the owner of the chat room, hence, moderators should not be allowed to demote or remove them as owner | **T3.1.0:** Create separate user classes |
| 6 | Application **must** provide a friendly User Interface (UI) | MUST | A welcoming and friendly UI ensures user retention and ease of use. The UI should not drive users away due to its complexity | **T1.0.0:** Design User Interface |
| 7 | Users **must** be able to see the active users in the chat room | MUST | Allowing a user to view other active users allows them to know who is available to chat | **T4.0.2:** Implement User Interface |
| 8 | Users **must** be notified when a new notification is received | MUST | A notification system ensures that users are always up to date with new conversations happening in their chats and chat rooms | **T5.0.0:** Implement event listeners |
| 9 | Clients **must not** connect directly to other clients without a server or a broker | MUST NOT | This is necessary as it would otherwise present itself as a security risk | **T2.0.0:** Setup and Configure Client/Server using MQTT |
| 10 | A server or a broker **must** allow multiple authorised clients to connect to it | MUST | A message exchange platform is going to have several users on it; hence, the broker must be capable of connecting with multiple clients | **T2.0.1:** Setup and Configure Client/Server using MQTT |
| 11 | Users **must** only access their space after the login | MUST | Having users access another user’s space would breach security and privacy protocols | **T6.0.1:** Implement security features and protocols |
| 12 | Passwords **must** be saved securely locally | MUST | This is necessary as it would otherwise present itself as a security risk | **T6.0.0:** Implement security features and protocols  **T9.0.0:** Setup a local text-file |
| 13 | Application **must** adhere to all local (and international) privacy laws | MUST | With laws like the General Data Protection Regulation (GDPR), modern-day applications must respect and handle user data appropriately | **T6.0.2:** Implement security features and protocols |
| 14 | Application **must** list all the personal contacts in the contacts pane | MUST | Users want to be able to view their contacts list and start a chat easily, this method provides them with the necessary tools in an intuitive manner | **T4.0.1:** Implement User Interface |
| 15 | The admin **should** be able to promote and demote users to moderators in chat rooms | SHOULD | This allows the Admin to provide chosen users with privileges to make modifications to the specific chat room; especially useful in cases where it is a large chat room, and the Admin cannot manage this on their own | **T3.0.1:** Create separate user classes |
| 16 | The moderator **should** be able to create and delete channels in the chatroom | SHOULD | Creating discussion threads within a chat room prevents the primary discussion thread from being flooded with several concurrent conversations | **T3.1.2:** Create separate user classes |
| 17 | The moderator **should** be able to delete a user’s messages in the chatroom | SHOULD | In the case a user sends an inappropriate message to a chat room, the moderator can then remove it immediately to avoid the message distressing other chat room members | **T3.1.3:** Create separate user classes |
| 18 | Users **should** be able to change their status | SHOULD | Users may be preoccupied with other tasks or do not wish to be disturbed, hence, would like the ability to modify their availability status within the application | **T7.0.0:** Develop user profile and control panel |
| 19 | Messages **should** be sent and received within 5-10 seconds | SHOULD | A performance requirement so that users do not spend too long waiting on a response from a contact | **T2.2.0:** Setup and Configure Client/Server using MQTT |
| 20 | Users **should** be logged off automatically after a specific amount of time | SHOULD | Prevents the system from being burdened by a user that is not active. This also doubles as a security feature to prevent an unsupervised account being compromised | **T6.1.0:** Implement security features and protocols |
| 21 | User **could** be able to change their details including their picture | COULD | A customization feature that allows a user to share their details with their contacts (i.e., profile photo, name, email) | **T7.1.0:** Develop user profile and control panel |
| 22 | User pictures **could** be displayed in the channels | COULD | An alternative identification method to just using a user’s name | **T4.1.0:** Implement User Interface  **T7.1.1:** Develop user profile and control panel |
| 23 | Application **could** allow the exchange of files with contacts | COULD | Users may wish to share images or other files with another user, hence the need for file transfer capabilities between contacts | **T8.0.0:** Implement file transfer feature |
| 24 | Users **could** be able to send emoji’s | COULD | Implements an interactivity feature to the application, allowing users to express themselves better | **T10.1.0:** Consider the implementation of additional features |
| 25 | Messages **could** come with sent and read receipts | COULD | Allows a user to know who has read their message and who has not. This is especially useful for urgent or messages of high importance | **T10.0.0:** Consider the implementation of additional features |
| 26 | Application **could** display the full history of the conversation when a specific contact is selected | COULD | Conversation history allows a user to rely on the system for information they may have forgotten about, hence, allowing them to refer to their older conversations | **T2.3.0:** Setup and Configure Client/Server using MQTT |
| 27 | Offline messages **could** be stored in the client-side and transmitted to the target user(s) once they are online | COULD | A user may not have internet access but would still like to send a message, this method ensures that as soon as they are connected to the internet, they can send those pending messages | **T9.1.0:** Setup a local text-file |
| 28 | Application **could** run on both macOS and Windows-based desktops and laptops | COULD | Not all users use windows devices, hence, the need for an application that runs on other operating systems | **T10.2.0:** Consider the implementation of additional features |
| 29 | Application **could** have a light and dark mode | COULD | Users with visual impairments may require alternative application colour schemes to make the application usable | **T4.2.0:** Implement User Interface  **T7.2.0:** Develop user profile and control panel |
| 30 | Application **could** have text-size customisation | COULD | Users with visual impairments may require alternative text sizes to be able to read their chat messages | **T4.2.1:** Implement User Interface  **T7.2.1:** Develop user profile and control panel |
| 31 | The application **could** have several language options | COULD | International users may require an application with features written in a language they are familiar with to use it | **T4.3.0:** Implement User Interface  **T7.3.0:** Develop user profile and control panel |

# Risk Analysis:

The risk analysis aspect of the project design stage involves reviewing and planning out solutions for potentials issues that could put the project’s success at risk. Highlighted below at several risks ranging from low to high probability and impact; they are each accompanied by a mitigation plan.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Risk Number** | **Description of Risk** | **Probability** | **Impact** | **Mitigation Plan** |
| 1 | Unclear or unrealistic requirements and scope | 2 | 5 | Ensure that all requirements are reviewed by all members of the team and are thoroughly discussed before being confirmed. In addition to this, actively seek out support from experts (i.e., lecturers and industry professionals) to ensure that the project scope is attainable within the allocated development window |
| 2 | Insufficient knowledge and background research of messaging applications | 3 | 4 | Carry out an intensive research process before beginning the development process to ensure that all team members are well informed |
| 3 | Security breach due to passwords being compromised | 2 | 5 | Add password encryption and (if feasible) two-factor authentication. |
| 4 | Team member falls ill due to ongoing pandemic or is otherwise unable to support the team due to extenuating circumstances | 3 | 4 | Promote the Software Tester to the role available. |
| 5 | Lost data due to technical failure | 2 | 4 | Make regular backups to GitHub and other cloud storage options used by the team. |
| 6 | Tasks go over the allotted time | 2 | 3 | Give buffer for extra time at end of the project - work to a week before the actual deadline |
| 7 | Team member overwrites an existing file's contents on accident | 3 | 1 | Regularly use version control software (i.e., Git - GitHub) so that the file contents can be easily reverted to an older version |
| 8 | Users struggle to use the application due to the unintuitive user interface (UI) | 2 | 2 | Ensure that during the testing stages user feedback is gathered with regards to the usability of the UI |
| 9 | Major bug found in the testing stage | 2 | 2 | Agile development allows for regular testing to prevent large scale bugs at the end of the project. |
| 10 | Team member struggles to engage with the group or is not actively communicating with the rest of the team | 3 | 4 | The team must have frequent check-ins to ensure how all team members are handling their workload and if anyone requires assistance with managing their tasks, they are encouraged to seek help from the rest of the team. |
| 11 | Member conflict occurs due to differing opinions | 3 | 5 | The team makes use of the quality vote to make the final decision to resolve the conflict |
| 12 | Team experiences issues with computational resources | 1 | 3 | Contact the team's assigned tutor for support in gaining access to university resources. |
| 13 | Member experiences issues with handling the workload | 3 | 4 | The team reviews the assigned task to break it down amongst other members to help support the struggling member |

|  |  |  |
| --- | --- | --- |
| **Probability** | **Impact** | **Description** |
| 5 | 5 | A risk event that if it were to occur, will have a **serious** impact on the project achieving its desired result. To the extent that one or more stated outcome objectives will not be achieved. |
| 4 | 4 | A risk event that if it were to occur, will have a **significant** impact on the project achieving its desired result. To the extent that one or more stated outcome objectives will fall below acceptable levels. |
| 3 | 3 | A risk event that if it were to occur, will have a **moderate** impact on the project achieving its desired result. To the extent that one or more stated outcome objectives will fall below goals but above minimum acceptable levels. |
| 2 | 2 | A risk event that if it were to occur, will have a **minor** impact on the project achieving its desired result. To the extent that one or more stated outcome objectives will fall below goals but well above minimum acceptable levels. |
| 1 | 1 | A risk event that if it were to occur, will have a **minimal** impact or **no** impact on achieving outcome objectives. |

# Gantt Chart:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tasks** | **December** | | **January** | | | | **February** | | | | **March** | | | | **April** | | | |
| W1 | W2 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 |
| Formulate Requirements List Table |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Formulate Risk Analysis Table |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Formulate Gantt Chart Table |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Compile and Review Documentation for Project Plan |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Milestone:** Creation and completion of Requirements List, Risk Analysis and Gantt Chart tables |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Deliverable 1:** Project Plan |  |  |  | 17 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Design Use Case, Activity and Class Diagrams |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Design Sequence, Component and FSM Diagrams |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Design Communication and Deployment Diagrams |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Milestone:** Completion project's design phase (includes several types of diagrams) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Deliverable 2:** Project Design Document |  |  |  |  |  |  | 7 |  |  |  |  |  |  |  |  |  |  |  |
| T1.0.0: Design User Interface - Finalise User Friendly UI for Application |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T2.0.0: Setup and Configure Client/Server using MQTT - Must include a server or a broker |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T2.0.1: Setup and Configure Client/Server using MQTT - Multiple Authorised Clients |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T2.1.0: Setup and Configure Client/Server using MQTT - Communications Setup |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T4.0.0: Implement User Interface - Communications Setup |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T2.1.1: Setup and Configure Client/Server using MQTT - Multiple Rooms |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.0.0: Create separate user classes - Admin Role |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.1.0: Create separate user classes - Role Permissions |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.1.1: Create separate user classes - Moderator (Invite and remove users) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.2.0: Create separate user classes - Multiple Rooms |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Milestone:** Completion of at least 5 MUST have requirements |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Deliverable 3:** Reference Manual |  |  |  |  |  |  |  |  |  | 28 |  |  |  |  |  |  |  |  |
| **Test Phase 1.0.1:** Create Test Plan |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T4.0.2: Implement User Interface - Display Active Room Users |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T5.0.0: Implement event listeners - Message Notifications |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T6.0.0: Implement security features and protocols - Password Storage |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T9.0.0: Setup a local text-file - Password Storage |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Test** **Phase 1.0.2:** Test all currently implemented requirements |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T6.0.1: Implement security features and protocols - User Access |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T6.0.2: Implement security features and protocols - Privacy Laws Review |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T4.0.1: Implement User Interface - Contacts Pane |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T2.2.0: Setup and Configure Client/Server using MQTT - Receive Messages Quickly |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.0.1: Create separate user classes - Promote and Demote Moderators |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T7.0.0: Develop user profile and control panel - Change Status |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Milestone:** Completion of at all MUST have AND at least 3 SHOULD have requirements |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Deliverable 4:** Test Plan |  |  |  |  |  |  |  |  |  |  |  |  |  | 28 |  |  |  |  |
| **Test Phase 2.0.1:** Review Test Plan |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T3.1.3: Create separate user classes - Message Deletion |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T6.1.0: Implement security features and protocols - Automatic Timeout |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T10.0.0: Consider the implementation of additional features - Read Receipts |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T10.1.0: Consider the implementation of additional features - Emojis |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| T10.2.0: Consider the implementation of additional features - macOS Version |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Test** **Phase 2.0.2:** Test all implemented requirements |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Review and Finalise Documentation |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Milestone:** All project requirements are complete and ready for submission |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Deliverable 5:** Final Report and Submission |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 25 |  |

# Monitoring Tools:

Our group aims to employ the use of several key monitoring tools to ensure that our team continues to develop and complete our project according to our outlined Gantt Chart and goals.

Our Tools:

* **Trello:** We decided to use Trello due to its incredibly comprehensive design and capabilities for teamwork and collaboration. We intend to use the in-built task tracking system to help hold all team members accountable throughout the entire process.
* **Discord:** For our communications platform, our team has decided to go with Discord as it is both a platform all of us are extremely familiar with and it is a platform that we most often frequent. Hence, ensuring that all updates sent to the platform’s channels will be reviewed within a 24-hour window. Thus, ensuring that all team members are up to date with any ongoing changes.
* **GitHub:** To ensure that all our work is frequently backed up and that we can review the contribution of each member we have decided to use GitHub as our version control solution. The platform will allow for us to collaborate while being unable to meet in person due to the ongoing pandemic while also acting as a safeguard in case one of us experiences a corruption of files or accidentally overwrites one of our files.

# Diagrams:

## Use Case Diagrams:

### Introduction:

**Figure 1.1 (Login)**

This diagram shows the login process. It assumes that the user is valid and will use their credentials to log in. The credentials introduced by the user are then validated by the text file.

**Figure 1.2 (Send Message)**

A user attempts to send a message, the system will first need to validate their connection to ensure that the target user is online and connected to the broker.

If the user is online:

* The source message is sent, and the source chat history is updated
* The target user is notified, and the target user’s chat history is also updated
* The chat histories on both ends are updated within the text file

If the user is offline:

* The source message is stored within the text file temporarily
* The source user’s connection is then repeatedly checked until they are confirmed to be online
* Once the source user is online, their message is then sent to the target user, and the following stages mentioned above occur

**Figure 1.3 (Make Room)**

The diagram shows the process that occurs when a user makes a new room. The user is assumed to be Admin as by making a room the user becomes that Room’s Admin by default. The new room is made, and the text file is updated. The Admin then has the option to add users to room and make these users moderators. The Admin can also make channels in the room. Whenever any changes are made the text file is updated.



Figure 1.1 LOGIN USE CASE

The login use case includes the input login credentials. Input login credentials includes authorise credentials. Authorise credentials extends to valid password. Authorise credentials also extends to invalid password.

Figure 1.2 SEND MESSAGE USE CASE

Source user actor is associated with the send message use case. Send Message includes validate connection. Validate connection extends to user offline. User offline includes store message. Store message includes update text file. It also includes validate connection. Validate connection also extends to user online. User online includes update logs. Update logs includes update text file. Update text file is associated with the Text File actor. User online includes notify contact. User online also includes update chat. Receive message includes update chat. Receive message also includes notify contact. Target User actor is associated with the receive message use case.

Admin actor is associated with make room use case. Make room use case includes update text file Update text file is associated with the text file actor. Make room extends to create channel. Create channel includes update text file

. Make room also extends to add user. Add user is associated with the user actor. Add user also extends to make moderator. Make moderator is associated with the moderator actor. The admin is a child of the moderator.

Figure 1.3 MAKE ROOM USE CASE

## Activity Diagrams:

### Introduction:

**Figure 2.1 (Login)**

In this scenario, the User attempts to login into the application by inputting the User credentials (i.e., Username and Password). The credentials are then authenticated against the existing credentials within the text file.

* If the password is valid, the user gets access to their user space
* If the password is invalid, the user is notified of their credentials being incorrect and are asked to try again.

**Figure 2.2 (Send Message)**

The activity diagram covers the flow of events required to send a message. The user triggers an event when sending a new message using the application. The source user’s status is validated to ensure they are connected to the broker. If the source user is offline, the message is temporarily stored in the text file. The application keeps checking the status of the users every 10 seconds. Once the source user's connection is established as being online, the system retrieves any messages that are stored in the text file. A queue of messages to be sent is created and messages are ordered accordingly in a first in, first out fashion. A fork then occurs for several activities to run concurrently, such as notifying the target user that a new message has been received, updating chat logs, and updating the chat history itself. Once these are all completed, they merge back to end the 'send message' activity.

**Figure 2.3 (Make Room)**

The activity diagram shows the process that occurs when a user makes a new room. First, the user is promoted to room Admin. The Admin is then given the option to add a user to the room. If they do, they are then given the option to make the user a moderator. Regardless of if they make the new user moderator or not, the Admin is given the option of adding a user again, looping back. Once the Admin has finished adding users, they are then given the option to add a channel to the room. If they choose yes, they can add the channel and then the option is given again, the same as when adding users. The text file is then updated, and process concludes.

While the add user and add channel options should not be unique to this process, they are still part of the process for making a new room and should be given immediately.

The login activity is initiated. User inputs their username and password. Control flow points to a decision node to check credentials combination against text file. If credentials match the control flow logs the user into their own space. Control flow points to activity final node and stops all control flows. If credentials do not match control flow notifies user of incorrect credentials. Control flow then asks the user to re-enter their credentials. The control flow then loops back to the input username and password activity.

Figure 2.1 LOGIN ACTIVITY DIAGRAM

The send message activity is initiated. The control flow points to the send a message through the chat activity. Control flow moves then moves onto a decision node to check user’s connection. If the user is offline, the control flow moves to the store message in text file activity. Control flow then loops back to check user connection. If the user is online, the control flow moves to the retrieve any stored messages activity. It then flows to create a queue of messages to be sent. Control flow forks at this point to run three activities concurrently, update the chat, update the chat log file, and notify the receiver. The control flow merges back and points to the activity final node and stops all control flows.

Figure 2.2 SEND MESSAGE ACTIVITY DIAGRAM

The make room activity is initiated by User. User is made Admin of room. Control flow moves to a decision node to ask whether to add a user to the room. If yes, the user is added to the room. Then, another decision node asks whether to make that user a moderator. If yes, user is made a moderator. Control flow loops back to add user decision node. If no to moderator option, user control flow loops back to add decision node. If no more users to add, control flow moves to another decision node asking to add a channel. If yes, the channel is added. Control flow then loops back to the add channel decision node. If no, text file is updated. Control flow moves to activity final node.

Figure 2.3 MAKE ROOM ACTIVITY DIAGRAM

## Class Diagram:

### Introduction:

The class diagram will demonstrate how data is stored and related to each other, this includes the functions and variables and how they will work together to create a unified OOP design.

Figure 3.1 CLASS DIAGRAM

**User**

The User class contains each unique person’s information, they have key functions to obtain and change the information. Note that the UserID can never be changed however the other information can.

**Admin/Moderator**

The Moderator class contains the functions needed to add and remove users as well has obtain the details of the moderator. The admin class is the same as the moderator, but they can add and remove Moderators as well as change who the Admin is.

**Room**

The room class contains channels and the information about the purpose of the room and its current members.

**Channel**

The Channel class is the main functionality of the program it contains the functions to send messages and subscribe and unsubscribe to the channels as well as meta data about the channel such as its name.

**Message**

The Message class contains the text and files sent by users into channels, this class contains the time it was sent where it was sent to and the contents. They will inherit Media class if a user chooses to send a file

**Relationships:**

**Admin (1) Administers Room (1)**

The admin class contains the functions used to administer the Room, there can be only one admin for each room.

**Moderator (\*) Moderates Room (1..\*)**

The Moderator class has all the functions to add/remove users and moderate said users there can be no moderators or many and a moderator may have several rooms under their control. The power to be a moderator is controlled by the room administrator.

**USER (1) IS A MEMBER OF ROOM(1…\*)**

A user is able to be a member of none or many rooms however a room must have one member who must also be the administrator the administrator is able to remove other and add others but not themselves unless they decide to delete the room.

**Room(1) Contains Channel(1..\*)**

One room must have at least one channel (the default channel is named ‘General’) The first channel created can change its name but it cannot be deleted. More rooms can be created but must have different names.

**Message(\*) Has Channel(1)**

There can be an infinite number of Messages however these messages can only be assigned to one channel. E.g you can send the same message content to many channels but these must be unique classes.

**User(1) Sends Message(\*)**

One user can create/send an infinite number of messages, however the message sender can only be one unique use

## Sequence Diagrams:

### Introduction:

**Figure 4.1 (Login)**

This diagram shows the login process. It highlights the interactions between the user, the application, and the text file.

**Figure 4.2 (Send Message)**

A user attempts to send a message, the system will first need to validate their connection to ensure that the target user is online and connected to the broker.

If the user is online:

* The message is sent by the sender
* The recipient requests new messages stored in the text file
* The text file returns a queue of all the messages
* The chat and log and updated and the recipient is notified

If the user is offline:

* The message is stored within the text file temporarily.
* The user’s connection is then repeatedly checked until they are confirmed to be online.
* Once the user is online, their message is then sent to the recipient, and the following stages mentioned above occur.

**Figure 4.3 (Make Room)**

The diagram shows the process that occurs when a user makes a new room. The user is assumed to be Admin as by making a room the user becomes that Room’s Admin by default. Users are added to the new room, and the admin decides whether they are moderators or not. Then channels are created in the room, and finally the text file is updated.

While the login credentials are invalid, the User lifeline sends a call message to input their username and password, which is sent to the application lifeline with a call message. The Application sends these to the text file lifeline for verification. Within an alt operator, the text file lifeline returns whether they are valid or not. The application lifeline sends a return message to the user lifeline to allow the user to login if the credentials are valid. In the guard of the alt operator, the text file lifeline sends a return message with login credentials invalid to the application, and a return message is sent to the user lifeline to inform them of incorrect credentials. This all occurs within a loop operator, that exits when the login credentials are valid.

Figure 4.1 LOGIN SEQUENCE DIAGRAM

The sender lifeline sends a call message to send a message to the application. In a loop operator while the user is offline, the application lifeline sends a call message to the recipient lifeline to check if they are connected.

Figure 4.2 SEND MESSAGE SEQUENCE DIAGRAM

Within an alt operator, if the user is connected a return message is sent to confirm. In the guard, a return message is sent to inform that there is not a connection, and a call message is sent from the application to the text file lifelines to store the message in the text file. After a connection has been established, the application lifeline sends a call message to the text file lifeline to request any stored messages, and a return message is sent back to return a queue of stored messages. Multiple call messages are then sent to the recipient lifeline to update the char, update the log file, and to notify the recipient.

The user lifeline sends a call message to the application lifeline to make a new room. The application sends a return message back to make the user the admin of the new room. Within a loop operator, until there are no more users to add, a calloperator is sent from the user to the application lifelines to add a new user. Within an alt operator, if the new user should be a mod, a call message is sent to make the user a moderator, and a return message is sent back from the application lifeline to confirm that a moderator has been added. In the guard, a return message is sent back to confirm a user has been added. Then, in another loop operator, until there are no more channels to be added, a call message is sent from the user to the application lifeline to add a new channel to the room. A return message is sent back to confirm a channel has been created. Finally, the application lifeline sends a call message to the text file lifeline to update the text file.

Figure 4.3 MAKE ROOM SEQUENCE DIAGRAM

## Component Diagram:

Figure 5.1 COMPONENT DIAGRAM

The Component Diagram shows the structure of the systems components. There are 3 main components; Messaging Application, text file and Broker. The Messaging Application contains the GUI, User Account, Chatroom and Channel components. The GUI component is linked to the user account component using the User View interface, which is provided by the User Account component. The GUI Component is also linked to the Chatroom component using the Room View interface, which is provided by User Account. Finally, the GUI also links to the Channel componenet using the Channel View interface.

The text file component contains the User text file, Chatroom text file, Channel text file and Message Backup components. The Chatroom text file component provides the roomID interface, which is used by the User and Channel text file components. The Channel text file also provides the channelID interface which is used by the Message Backup component. The User Account component, which is contained within Messaging Application, provides the Get User interface, which is outside Messaging Application, via a port. User text file uses Get User from within text file via a port. The Chatroom component, which is contained within Messaging Application, provides the Get Chatroom interface, which is outside Messaging Application, via a port. Chatroom text file uses Get Chatroom from within text file via a port. The Channel component provides, via a port from Messaging Application, the Send/Receive Message interface, which is used by the Broker component through a port. The interface is also used by the Channel text file component via a port into text file.

## FSM Diagrams:

### Introduction:

**Figure 6.1 (Login):**

This FSM Diagram considers the shift in states from when a user attempts to login to when they are successful with their login process. The diagram also considers a potential issue of the text file being unavailable. Our team’s solution is to immediately notify the text file admin and inform the user of the temporary unavailability. The application then ensures that the user is unavailable to login instead of allowing them to take advantage of a temporary system vulnerability.

**Figure 6.2 and Figure 6.3 (Publisher & Subscriber):**

Our team decided to split this FSM diagram into two parts that runs concurrently as we believe it is important to consider both the perspectives of a publisher and subscriber when it comes to message exchanges. From the publisher’s perspective, once a message is created it begins to process of transmitting this data to the necessary topics. The subscriber then picks up from this point by having the broker distribute the message and its data to the relevant topic subscribers. The application then handles the process of displaying the message to the target recipient by modifying the user interface to reflect the updates made.

The login state begins with the transition to the login form on the application. The current state being that the user is not logged in. This transitions to getting credentials state where the user enters their credentials. The next transition is to the check credentials text file status. If the text file is unavailable, it transitions to notifying the admin and the user of the error and unavailability. It then loops back to the not logged in state. If the text file is available, it transitions to the authorise state where the credentials are checked against the text file. If rejected, it transitions back to the not logged in state and the user must try again. If accepted, it transitions to a logged in state and loads the user's space before ending with the final state.

Figure 6.1 LOGIN FSM DIAGRAM

The publisher FSM diagram begins with the message being created. This then transitions to the connect to broker state to establish a connection between the publisher and broker. If it fails, it transitions to the store message in logfile state and then to the store logfile name in text file state to ensure the message contents are not lost while waiting on a successful connection. This then loops back to the connect to broker state to try again. If it is a success, it transitions to the retrieve message state where all stored message contents are retrieved before moving to the publish to topic state. Message contents are published to the relevant topic using the broker before transitioning to the wait for new message state. It then waits till a message is received to begin the process of looping back through the states again.

Figure 6.2 PUBLISHER FSM DIAGRAM

The diagram begins with the message being sent (assuming the concurrent states on the Publisher’s side have already occurred). This state then transitions to the state where the message and its data are published to the specific topic. This then transitions to the retrieve message contents state. Once the necessary data is acquired, it transitions to the distribute to topic subscriber’s state. The locate message destination state is then triggered to identify the target recipient of the message. Once identified, the transition is made to the update user interface state and then to the display message state so that the target recipient can view the message they have been sent. Once the message is displayed on the recipient’s side, the transition moves to the wait for new message state. It then waits till a message is sent via the publisher and received to begin the process of looping back through the states again.

Figure 6.3 SUBSCRIBER FSM DIAGRAM

## Communication Diagram:

Figure 8.1 SEND MESSAGE COMMUNICATION DIAGRAM

The communication diagram demonstrates how the system will send and receive messages. First the sender will send a message to the application server which will then store that information in the text file. Then the message will be sent to update the chat on the recipient device. If the recipient is not online, then they will request the information from the text file which will respond with the same message which was sent by the sender.

## Deployment Diagram:

The Deployment diagram demonstrates how the system will be implemented in the real world. The Client Pc will contain the application which will only directly communicate with the server via the MQTT client. The application will depend on the MQTT client to function and send messages to the server and receive messages.

Figure 9.1 DEPLOYMENT DIAGRAM

The Deployment diagram demonstrates how the system will be implemented in the real world. The Client Pc will contain the application which will only directly communicate with the server via the MQTT client. The application will depend on the MQTT client to function and send messages to the server and receive messages.

The Server will have 2 separate executables the Chat server will depend on the MQTT broker to send and receive messages while communicating with the client devices. The Chat server will use a text File to store information about users and their messages on the client PC.

## GUI Mock-Up:

### Introduction:

The GUI mock-up outlines the essentials of the application’s basic functionality and user interface. It displays how various elements of the GUI would be laid out in the final version of the app and provides an idea of the application may function.



Figure 10.1 GUI MOCK-UP

This GUI Mock-up shows how the basic user interface of the application should look. Inspiration was taken from research into other popular messaging applications, such as Slack and Discord.

1. All the users’ available chatrooms are shown on the leftmost column on the screen.
2. All the available text channels for the currently selected server are displayed next to that, the channels can be named and grouped.
3. At the bottom of that column, the user can access their profile and application settings, as well as change their status.
4. The main, middle section of the screen displays the contents of the currently selected text channel. The user can send via the input bar at the bottom, and messages that are sent and received are displayed in the main area. The user can scroll through messages using the scroll bar and can search through message history using the search function at the top.
5. On the rightmost column, all the users in the currently selected chatroom are displayed, grouped by their status.

# Design Pattern:

We have looked at 3 main options for the design patterns Builder, Factory Method and Abstract Factory.

|  |  |
| --- | --- |
| Pros | Cons |
| Factory Method | |
| * Implementation and creation are separate * Object creation can be kept in one place * New types can be added without breaking existing code | * Subclasses may be needed to implement the pattern * The design pattern works well in particular circumstances but can be very complex if not appropriate. |
| Abstract Factory Method | |
| * New variants of objects can be added without modifying existing code * Object creation can be kept in one place | * Makes code more complex as lots of interfaces and classes are introduced when using the pattern. * We are unlikely to add new functionality to existing code. |
| Builder Method | |
| * Objects can be made step-by-step depending on the required parts * The components can be reused. | * Code can become much more complex as each new pattern will require multiple classes. * The builder is not necessary for the messages as they commonly inherit one type like a message that contains a file or a user who is also an administrator. |

As a result of the analysis above of our potential options, our team decided on the **Factory Method** as our design pattern (Refactoring guru, 2021). We intend to use creator classes to declare the new objects such as the User or a Message wherein each object is unique but has the same functions. Moreover, different users can inherit different functions such as a user which has the admin role has more functionality such as adding or removing other users.

We chose this method as at this early stage of design, we do not know exactly what types and dependencies the objects may rely on as they depend on the kind of message the user may send which allows for inheritance depending on what message the user sends or what level of permissions they may have in a selected channel. We also know that most will use the same foundation and we want to use existing objects rather then rebuild them each time as most functionality remains constant for each object, this helps avoid duplicated code within the codebase.

# Test Plans:

# Conclusions & Future Work:

# References:

# Individual Contributions:

# Overall Reflection:

# Appendix:

## Coding Contribution Guide:

Before contributing to this repository, please first discuss the changes you intend to make via a meeting, text thread or issue

with the other team members/owners of this repository.

As outlined below, please note we have a code of conduct. Do try to ensure that you adhere to it during all your interactions with this project.

### Pull Request Process:

You may merge the Pull Request in once you have the sign-off of all other team members, or if you do not have permission to do that or unable to get a sign-off from all three,

you may request the second reviewer/team member to merge it for you.

### Code of Conduct:

**Our Pledge**

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance,

race, religion, or sexual identity and orientation.

**Our Standards**

Examples of behaviour that contributes to creating a positive environment include:

1. Using welcoming and inclusive language
2. Being respectful of differing viewpoints and experiences
3. Gracefully accepting constructive criticism
4. Focusing on what is best for the team
5. Showing empathy towards other team members

Examples of unacceptable behaviour by participants include:

1. The use of sexualized language or imagery and unwelcome sexual attention or advances
2. Trolling, insulting/derogatory comments, and personal or political attacks
3. Public or private harassment
4. Publishing other's private information, such as a physical or electronic address, without explicit permission
5. Other conduct which could reasonably be considered inappropriate in a professional setting

**Our Responsibilities**

The Project Manager is primarily responsible for clarifying the standards of acceptable behaviour and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behaviour.

The Software Developer has the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct.

**Scope**

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its team. Examples of representing a project or team include presenting this project to university staff members, showcasing the project to external parties and receiving feedback on the project.

Representation of a project may be further defined and clarified by project maintainers.

**Enforcement**

Instances of abusive, harassing, or otherwise unacceptable behaviour may be reported by contacting the Project Manager. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project manager is obligated to maintain confidentiality concerning the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Team members who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

**Attribution**

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at <http://contributor-covenant.org/version/1/4>

## Coding Contribution Standards:

### C++ Version:

Please ensure that you use **C++17** throughout your contributions to this project. With regards to IDE preference, for development, our team will use Eclipse 2020-12 as it is currently the most stable for development.

### Header Files:

Almost every .cc file should have an associated .h file, except for unit tests and small .cc files containing just a main() function.

Correct use of header files can help improve the readability, size and performance of the code. The following advises on how best to implement header files.

**Self-contained Headers**

All header files should be self-contained (compile on their own), they should not require specific conditions to be included, and should have header guards and include all other headers it needs. Header files should usually end in .h, with exception of .inc files used for inclusion. .inc files should only be used where a file designed to be included is not self-contained, for example, it may be in an unusual location. They may not use header guards or include their prerequisites. They should see limited use, however, and in all situations, a self-contained header should be prioritised.

**Define Guards**

Within all header files ensure the use of #Define guards to prevent multiple inclusions and to avoid unnecessary code recursion. Conflicts or recursive errors could result in code failing to build.

Guards should be named uniquely. The standard naming convention is <FILENAME>\_H

**Include What You Use**

The header file should only include all the header files needed for that source and header file, where either uses a symbol defined elsewhere. Transitive inclusions, inclusions where a header is included in one file but both use symbols from each other, should be avoided at all cost. This allows includes to be simply removed without issues being caused elsewhere.

**Forward Declarations**

Forward declarations, declaration of an entity without an associated definition, should be avoided. While they do improve compile-time and reduce the need for recompilation, they are likely to cause more mistakes and use more lines than just including the header.

**Inline Functions**

Functions should not be defined inline, with exception of short, performance-critical functions. While inlining of small individual functions may cause them to generate more efficient object code, overuse may cause an overall decline in program speed as the cost is increased. As a rule, functions should not be inline if they are longer than 10 lines.

**Names and Order of Includes**

To ensure that missing includes are caught early, include headers should be grouped in the following order:

* The Related header e.g. #include "main.h"
* The C system headers e.g. #include <stddef.h>
* The C++ standard library headers e.g. #include <string>
* The other libraries' headers e.g. #include "basictypes.h"
* The project's headers e.g. #include "other.h"

Each group should be separated by one blank line. The exception to this rule is system-specific code which needs conditional includes, these may be put after other includes. System-specific should be small and localized.

### Scoping:

**Namespaces**

Code should be placed under a namespace, which divides the global scope into individual named scopes to prevent name collisions. Namespace names must be unique and should be all lowercase and underscored between words. Avoid using abbreviations unless they follow the rules laid out in the naming section.

**Internal Linkage**

Sections of code can be given internal linkage using an unnamed namespace (formatted the same as regular namespaces), and individual functions and variables can also be given internal linkage by declaring them as static. Internal linkage prevents whatever has been declared from being accessed from another file. Therefore, internal linkage should be used when the code doesn’t need to be referenced elsewhere in .cpp files. It should not be used in `.h` files.

**Non-member, Static Member and Global Functions**

Always place non-member functions in a namespace and only use global functions when absolutely necessary. Static members should not be grouped. Non-member functions should not depend on external variables and should instead exist in a namespace.

**Local Variables**

A function variable should be in the narrowest scope possible. Variables should be declared as close as possible to the use. Variables should be initialised when declared, for example; int i = 0. If variables are needed in an object such as an if-statement or for-loop they should be declared just above the constructor.

**Static and Global Variables**

Objects with static storage duration are disallowed unless there are guaranteed trivial destructors. Global/static variable initialization will depend on the initializer, as a general rule one should always allow for a constant expression. An example of one that is allowed is: int id = getid(); is allowed. Dynamic initialisation of static local variables is permitted.

### Classes:

Constructors must not call virtual functions. Do not define an implicit conversion, one should use the explicit keyword for conversion operators and single argument constructors. Type conversion operators should be marked explicit in the class definition. Every class's public interface should say which copy and move operations the class supports which should be done in the public section.

**Structs:**

Structs should only be used for objects that are passive and carry data otherwise use a class.

**Inheritance:**

Prioritise composition over inheritance. All inheritance should be public if done privately they try adding it as a member of the base class instead.

**Access**

Classes' data members should be made private unless they are constants. This helps protect data.

**Structure**

In general, group similar kinds of declarations together and do not put large method definitions inline in the class definition. See the formatting guide below for more info.

### Functions:

Functions should be written using the old-style function definitions for example: string funct(int y); this helps readers and coders who work with other languages understand the code. Default arguments must not be used on virtual functions but they can be used elsewhere.

**Short Functions**

Functions should always be short and focused; keeping code short helps isolate bugs and testing. Look to break up large functions into smaller ones unless it adds unneeded complexity.

**Inputs/Outputs**

Use return values when possible over output parameters, this improves readability. Avoid returning pointers unless it is possible for them to be null, preferably return by value, failing that, return by reference. When having non-optional input parameters they should be constant references or values, while output or input/output ones should generally be references.

**Style**

Cpplint should be used to detect style errors, it is preinstalled on QT creator but can be run separately if needed.

### Naming Conventions:

**Variables / Functions**

Variables will always be named using camel-case. Type names and function names should start with a capital letter for each new word. **DO NOT** use underscores. Class data members should end in an underscore like std::string myvar\_;.

**Constants / Enumerators**

When declaring constants, always capitalise each new word and begin the constant with the letter k, this helps keep code clear. Enumerators should also be named like constants.

Some abbreviations are ok as long as they are common or clear, for example, i/j for iteration or CPU for a central processing unit. When writing a variable think about whether it would hinder the code by using the abbreviation such that it would make it less understandable.

**File Names**

File names should always be lowercase we will use an underscore ‘\_ ‘between words. C++ files should always end in .cpp while header files should end in `.h`. File names should be as specific as possible. Use clear names like:

* main\_menu\_gui.cpp
* admin\_class.cpp
* admin\_class\_test.cpp

Note: Always end a test file in `\_test` for the sake of simplicity.

### Comments:

Comments improve readability and make code more accessible to team members and future developers, improving maintainability as well. While commenting is important, good code should be readable without comments; variables and types with good naming should not need a comment to explain them.

**Comment Style**

The commenting style should be consistent. Comments are to done using the Qt style provided by Doxygen, as seen below:

*/\*! \brief Brief Description*

*Brief Description continued.*

*The detailed description starts here.*

*More detailed description.*

*\*/*

By using Doxygen, the documentation process can be automated, by developing a Doxyfile.

**File Comments**

File comments describe the contents of a file. Due to the nature of the project, being non-commercial, the file comments should not need to include an author line or copyright notice. However, if a .h file declares multiple abstractions, a file-level comment should be used to describe the file contents and how the abstractions are related. The comment should be short, as the detailed documentation of individual abstractions should be in the region of said abstractions. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, then file comments are not required. All other files must have file comments. File comments should not be duplicated in the .cc as well as the .h, as they would likely end up diverging.

**Class Comments**

Every class or struct declaration where their use is not immediately evident should have a comment that describes its usage. The description should be as clear and concise as possible while covering all relevant information on how to correctly use the class. All information included would extend to how (such as an example use of the class) and when to use the class, and any additional considerations such as synchronisation or threading. Specifically, comments describing the use of a class would be included with its declaration, and comments about the class operation and implementation should accompany the implementation of the class's methods.

**Function Comments**

Similar to class comments, comments describing the use of a function should be at the function declaration and comments to describe the implementation of the function should be included with the function definition.

Function comments should not be unnecessarily verbose and should only cover what is not immediately obvious. At the function declaration, a comment may be an example of how to use the function. At the function definition, the comments should cover areas where the implementation is overly complex or not clear and not repeat comments in the function declaration. There should be very few comments on the function definition as good code should be clear on what it does without the need for any comments.

**Variable Comments**

Variables should be named clearly enough to not need comments, however, comments may be needed in certain cases. For class data members, comments may be used for Sentinel Values, those used as flags or pointers, where their use is not immediately clear. Another circumstance is Global Variables, which should cover their use and why they are global.

**Implementation Comments**

While good code should need few comments, where the implementation may confuse, comments should be used. Complicated blocks of conde may have a proceeding comment to explain the purpose of the entire block. Individual lines of code where their use is not clear, or specific choices may not be clear, may also require commenting to prevent future problems.

Where the arguments of a function are not obvious, comments should be a last resort. Alternatives may include using named constants over literals, avoiding nesting functions in functions over using variables, replacing bool arguments with enum arguments.

**Punctuation, Spelling and Grammar**

Comments should be easy to read themselves, so they should have good punctuation, spelling, and grammar, including proper capitalisation, punctuation and complete sentences. Shorter, single line, comments may be written in a less formal shorthand as long as they are still readable, however, whole sentences should be the priority.

**TO-DO Comments**

TODO comments may be used for short-term tasks, such as something you may do the next day. They should be written in the format: // TODO (<NAME>): <TASK>. By using TODO as a standard keyword, TODO messages can be easily searched for using a refactoring tool. TODO messages must be removed when the task is complete. The name attached to the task should almost always be the name of the task writer, to make sure that everyone is aware of who the task belongs to. If the task is addressed to another person, that person should also be informed directly of the task having been added.

### Formatting:

Code formatting is arbitrary but following the same guide helps keep consistency and make code easier to understand.

**Line length**

A line of code should always remain under 80 characters, this helps keep the code readable and understandable. It will be common that function declarations and returns will exceed the 80-character limit in this case it should be broken up onto separate lines.

Parameter names should be short but clear if possible, to assist in keeping code short.

The use of tabs is preferred in this project and should be used in favour of spaces.

Boolean expressions should also be broken up while the logical operators should always be at the end of the line.

**Class format**

Sections should be placed in order, public, protected and private.

**Whitespace**

Try to minimise vertical white space, they should be considered the ending to a paragraph and used sparingly to help separate ideas. Overuse of whitespace will make code more difficult to follow. Simply use them where they are thought to be appropriate, such as separating comments and functions.

### Other C++ Features:

**Boost**

Boost Guidelines: <https://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/utf/usage-recommendations/generic.html>

You should use tests that are specific and precise, try to avoid a complex test in favour of smaller tests. You should prefer `BOOST\_CHECK\_EQUAL` over `BOOST\_CHECK`. This is because you will see the incorrect value when the code failure.

**Group tests together**

You should aim to group similar tests and name each test so you know what it is checking to help with the debugging later on.