# SYSTEMS AND SOFTWARE REQUIREMENTS SPECIFICATION (SSRS) FOR

# sQuire Collaborative IDE


**Version 1.3**
**April 11, 2016**

**Prepared for:**
**CS383-01**

**Prepared by:**
**Domn Werner (wern0096) | Robert Carlson (carl7595)**
**Brian Cartwright (cart1189) | Max Welch (welc2150)**
**Matthew Daniel (dani2918) | Brandon Ratcliff (ratc8795)**
**Joel Doumit (doum6708) | Eric Gentile-Quant (gent7104)**
**Team 4 - It Compiled Yesterday (ICY)**
**University of Idaho**
**Moscow, ID 83844-1010**

**sQuire SSRS**

**RECORD OF CHANGES**

| Change number | Date completed | Location of change (e.g., page or figure #) | **A M D** | Brief description of change | Approved by (initials) | Date Approved |
|---|---|---|---|---|---|---|
| 1 | 02/11/16 | Page 1 | M | Updated Team Name | DW | 02/11/16 |
| 2 | 02/11/16 | Pages 8-10 | M | Updated Functional Reqs | DW | 02/11/16 |
| 3 | 02/18/16 | Section 3.3 | A | Added Use Cases | DW | 02/18/16 |
| 4 | 02/18/16 | Section 3.4 | A | Added Class Diagram Section | DW | 02/18/16 |
| 5 | 03/02/16 | Section 3.3 | M | Included Sequence Diagrams | DW | 03/02/16 |
| 6 | 04/11/16 | All Sections | M | Reorganized for Instructor Feedback | RC | 04/11/16 |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

*__A__ - ADDED __M__ - MODIFIED __D__ - DELETED

**SQUIRE SSRS**
**TABLE OF CONTENTS**

**Section Page**

# 1 REQUIREMENTS

## 1.1 INTRODUCTION

The sQuire Collaborative IDE is a collaborative IDE software project for CS383-01. The intended audience for this project is Java programmers looking for a more social collaborative experience. A large focus of the program is also to help programmers connect with others who may interested in their projects.

### 1.1.1 IDENTIFICATION

The software system being considered for development is referred to as sQuire. The customer providing specifications for the system is Dr. Jeffery and the CS383-01 class. The ultimate customer, or end-user, of the system will be Java programmers. This is a new project effort, so the version under development is version 1.0.

### 1.1.2 PURPOSE

The purpose of the system under development is to provide Java programmers with a more social collaborative experience. Instead of individual methods of source control, sQuire will provide an environment where programmers can work together in the same environment and instantly see the effect of others' code. While the system will be used by Java programmers, this document is intended to be read and understood by UI CS software designers and coders. The document will also be vetted or approved by Team 4.

### 1.1.3 SCOPE

This project is sponsored by the CS383-01 class and is being worked on by Team ICY (4) from scratch. The goal is to have a working prototype by the end of the Spring 2016 semester. We plan to operate individually for the most part by programming from our own machines at home for about 10 hours per week. We also plan on using a Windows Server running in a VM at Domn Werner's house.

### 1.1.4 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

| Term or Acronym | Definition |
|---|---|
| Alpha test | Limited release(s) to selected, outside testers |
| Beta test | Limited release(s) to cooperating customers wanting early access to developing systems |
| Final test | aka, Acceptance test, release of full functionality to customer for approval |
| DFD | Data Flow Diagram |
| SDD | Software Design Document, aka SDS, Software Design Specification |
| SRS | Software Requirements Specification |
| SSRS | System and Software Requirements Specification |
| IDE | Integrated Development Environment |
| | |
| | |

| | |
|---|---|

### 1.1.5 OVERALL DESCRIPTION

The sQuire project is an answer to the lack of real-time collaborative programming experiences. By bringing programmers together in a more social environment, this program aims to improve collaboration between programmers in a much more fast paced and agile methodology. Furthermore, for programmers who seek others to help with their projects, sQuire aims to provide a simple social platform for engaging with other programmers and start working on a project together.

### 1.1.6 PRODUCT PERSPECTIVE

This program will be a standalone executable, connecting to a central project server.

## 1.2 SYSTEM LEVEL (NON-FUNCTIONAL REQUIREMENTS)

### 1.2.1 Site dependencies

1. Central SQL Server
2. Host-side Project Server
3. Collaborator-side Client

The Central server stores user credentials, project descriptions, and user profile and achievement data.
Requirements for Central SQL Server

1. Host with high uptime percentage
2. SQL capable
3. E-mail capable for password resets
4. Fast enough connection to prevent login timeout, even while handling multiple requests
5. Prefer host with multiple backups

The Host-side server stores the project files, project access list, hosts the editing environment, runs chat channels, and serves files to collaborators for compiling.
Requirements for Host-side Server

1. Java Capable (http://java.com/en/download/help/sysreq.xml)
2. SQL Capable (WAMP/LAMP)
3. 4 GB RAM
4. Hard drive space for server + project files

The client side application connects to the host server, renders GUI elements, stores connection profiles, stores server files, and compiles the project.
Requirements for Collaborator-side Client

1. Java Capable (http://java.com/en/download/help/sysreq.xml)
2. Compatible Java version installed
3. 2 GB RAM
4. Hard drive space for project files

## 1.3 Safety, security and privacy requirements

The collaborative nature of sQuire includes several concerns for security and privacy. The program will include in the license agreement the following stipulations:

1. sQuire is a free development environment, and may be used for commercial purposes
2. No guarantee of code confidentiality is implied by use of sQuire
3. Clients assume the risk of downloading, compiling, and running project files
4. Email addresses are visible as part of a user profile

5. Host assume the risk of allowing peers to connect to their server

However, the program will provide the following minimum features to address security and privacy concerns:

1. All SQL servers will include input sanitization and appropriate anti-injection safeguards
2. Project hosts may turn off guest access to their project
3. Uploads for assets will be limited to folders within the project directory
4. Visibility to host file structure will be limited to project folders only

### 1.3.1 Performance requirements

1. Up to 33 concurrent connections will be supported
2. Edits will be visible to all connected collaborators within 10 seconds
3. Login and server connections will report success or failure within 45 seconds

### 1.3.2 System and software quality

Adaptability

1. The program will allow selection of different compiling programs and command line arguments.
2. The program will allow importing of files of key words to allow other development languages to be used.

### 1.3.3 Packaging and delivery requirements

The executable system and all associated documentation (i.e., SSRS, SDD, code listing, test plan (data and results), and user manual) will be delivered to the customer on CD's and/or via email, as specified by the customer at time of delivery. Although document "drops" will occur throughout the system development process, the final, edited version of the above documents will accompany the final, accepted version of the executable system.

### 1.3.4 Personnel-related requirements

The system under development has no special personnel-related characteristics.

### 1.3.5 Training-related requirements

No training materials or expectations are tied to this project other than the limited help screens built into the software and the accompanying user manual.

## 1.4 FUNCTIONAL REQUIREMENTS

### 1.4.1 Project Browsing

These requirements involve the ability for users to find and learn about projects that they may wish to contribute to. Users shall be able to:

1. See a list of open projects.
   Use Case Description: 2.1.7
   Sequence Diagram: 2.4.36
2. Filter or search projects.
   Use Case Description: **??**
   Sequence Diagram: **??**
3. View more information about a specific project.
   Use Case Description: **??**
   Sequence Diagram: **??**
4. Upvote and downvote projects.
   Use Case Description: **??**
   Sequence Diagram: **??**
5. Comment on projects and interact with its contributors.
   Use Case Description: **??**
   Sequence Diagram: **??**
6. Request to join a specific project.
   Use Case Description: **??**
   Sequence Diagram: **??**

### 1.4.2 Authentication

These requirements involve the ability for users to have individual accounts and the security that comes from that. Users shall be able to:
Use case diagram: 2.2.1

1. Sign up for a sQuire user account.
   Use Case Description: 2.1.1
   Sequence Diagram: 2.4.1
2. Log in to the program using their user account.
   Use Case Description: 2.1.2
   Sequence Diagram: 2.4.2
3. Log out of the program using their user account.
   Use Case Description: 2.1.3
   Sequence Diagram: 2.4.3
4. Change their password.
   Use Case Description: 2.1.4
   Sequence Diagram: 2.4.4

5. Change their email.
   Use Case Description: 2.1.5
   Sequence Diagram: 2.4.5
6. Change their username.
   Use Case Description: 2.1.6
   Sequence Diagram: 2.4.6

#### 1.4.3 Communication

These requirements involve the ability for users to be able to communicate with other users. Users shall be able to:

1. Open and close project chat.
2. Write to project chat.
3. Read from project chat.
4. Message a user by their name.
5. Leave a comment in a file.

#### 1.4.4 File Management

These requirements involve the ability for users to manage the files that compose a project. Users shall be able to:

1. Open one or more files.
2. Close one or more files.
3. Delete one or more files.
4. Download one or more files.
5. Add a new file to the project.
6. Add an existing file to the project.
7. Save one or more files.

#### 1.4.5 File Editing

These requirements involve the collaborative editor part of sQuire. Users shall be able to:

1. Enable or disable line numbers.
   Use Case Description: **S3.3.53**
   Sequence Diagram: **S3.3.54**
2. Enable or disable viewing reference counts above each line.
   Use Case Description: **S3.3.55**
   Sequence Diagram: **S3.3.56**

3. Enable or disable viewing date of last edit above each line.
   Use Case Description: **S3.3.57**
   Sequence Diagram: **S3.3.58**
4. Enable or disable view author of each line.
   Use Case Description: **S3.3.59**
   Sequence Diagram: **S3.3.60**
5. Comment/Uncomment a selected section code.
   Use Case Description: **S3.3.64**
   Sequence Diagram: **S3.3.65**
6. Format the document to adhere to code style.
   Use Case Description: **S3.3.61**
7. Find/Replace specified text.
   Use Case Description: **S3.3.62**
   Sequence Diagram: **S3.3.63**
8. View text highlighted by other users.
9. Type text and have the system apply syntax coloring for Java files and display errors.
   Use Case Description: **S3.3.68**
   Use Case Description: **S3.3.70**
   Sequence Diagram: **S3.3.69**
   Sequence Diagram: **S3.3.71**
10. View other users' carets as they type.
    Use Case Description: **S3.3.66**
    Sequence Diagram: **S3.3.67**

#### 1.4.6 Project Management

These requirements involve the management of entire projects. Users shall be able to:

1. Compile a project.
   Use Case Description: 2.1.16
   Sequence Diagram: 2.4.35
2. Execute a compiled project.
   Use Case Description: 2.1.16
   Sequence Diagram: 2.4.35
3. Create a new project.
   Use Case Description: 2.1.17
   Sequence Diagram: 2.4.34

4. Delete a project.
   Use Case Description: 2.1.18
   Sequence Diagram: 2.4.14
5. Invite a user to a project.
   Use Case Description: 2.1.22
   Sequence Diagram: 2.4.17
6. Join a project.
   Use Case Description: 2.1.19
   Sequence Diagram: 2.4.15
7. Leave a project.
   Use Case Description: 2.1.21
   Sequence Diagram: 2.4.16

#### 1.4.7 Project User Management

These requirements involve project admins managing their Users. Admins shall be able to:

1. Add users to a project.
   Use Case Description: **S3.3.49**
   Sequence Diagram: **S3.3.50**
2. Remove users from a project.
   Use Case Description: **S3.3.51**
   Sequence Diagram: **S3.3.52**
3. Change user permissions to a project.
   Use Case Description: **S3.3.53**
   Sequence Diagram: **S3.3.54**

#### 1.4.8 User Preferences

These requirements involve managing user preferences. Users shall be able to:

1. Update their username.
2. Update their password.
3. Update their email address.
4. Update their biography.
5. Update their display name.
6. Enable receiving email updates.
7. Enable receiving messages from any user.
8. Display their email address to selected groups.
9. Change program colors.

# 2 DESIGN

## 2.1 USE CASE DESCRIPTIONS

### 2.1.1 Authentication Feature 1: Sign Up Use Case Description

---

**Actors:** User

**Summary:** The user signs up and creates an account using their email address and creates username and password in order to access the program.

**Purpose:** To register and create an account in the program

**Preconditions:** None

**Steps:**

1. User clicks Register button.
2. System prompts the user to enter email and password.
3. User enters email and password and clicks Submit button.
4. System sends confirmation email.
5. User verifies email by clicking a link.
6. System adds verified user to database.

**Alternative 1:** User already has an account.

**Alternative 2:** User doesn't confirm email. Delete request after timeout period.

**Relevant Classes:**

- **User** in **S3.4.5**
- **Email** in **S3.4.5**
- **Validator** in **S3.4.5**
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

---

### 2.1.2 Authentication Feature 2: Log In Use Case Description

---

**Actors:** User

**Summary:** A registered logs in to the program in order to access its features.

**Purpose:** Allow registered users access to the program.

**Preconditions:** User must already have a registered account.

**Steps:**

1. Users clicks Log In button.
2. System prompts the user for their username and password.
3. User enters their login information.
4. System verifies the login information and grants user access to their account.

**Alternatives:**

1. User enters incorrect information. System prompts for credentials again.
2. User has not clicked email confirmation. System resends email and tells user.
3. User makes 5 incorrect login attempts. System prevents more login attempts for 5 minutes.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

---

### 2.1.3 Authentication Feature 3: Log Out Use Case Description

---

**Actors:** User

**Summary:** A user logs out of the program.

**Purpose:** Allows logged in users to log out in order to protect their account.

**Preconditions:**

1. User must have a registered account.
2. User must be logged in.

**Steps:**

1. Users clicks log out button.
2. System logs user out.
3. Browser cookies are updated to reflect user being logged out.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

---

#### 2.1.4 Authentication Feature 4: Change Password Use Case Description

---

**Actors:** User

**Summary:** A user changes their password while logged in.

**Purpose:** Allows logged in users to change their passwords.

**Preconditions:**

1. User must have a registered account.
2. User must be logged in.

**Steps:**

1. Users clicks Change Password button.
2. System prompts user to enter their password twice.
3. User enters their password twice.
4. System hashes both passwords.

**Alternatives:**

1. If passwords match, System updates user password and sends email to registered email account.
2. If passwords don't match, system notifies user.

**Related Use Cases:**

1. Change Email
2. Change Username

**Relevant Classes:**

- **User** in **S3.4.5**.
- **Email** in **S3.4.5**.
- **Validator** in **S3.4.5**.
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

---

#### 2.1.5 Authentication Feature 5: Change Email Use Case Description

---

**Actors:** User

**Summary:** A user changes their email while logged in.

**Purpose:** Allows logged in users to change their email.

**Preconditions:**

1. User must have a registered account.
2. User must be logged in.

**Steps:**

1. User clicks Change Email button.
2. System prompts user to enter their email.
3. User enters their email.
4. System sends a confirmation link to the email entered.

**Alternatives:**

1. If User clicks confirmation link, System updates the user's email and sends an email to the new email stating so.
2. If user doesn't click confirmation link in an hour, the link becomes invalid.

**Related Use Cases:**

1. Change Password
2. Change Username

**Relevant Classes:**

- **User** in **S3.4.5**.
- **Email** in **S3.4.5**.
- **Validator** in **S3.4.5**.
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

---

### 2.1.6 Authentication Feature 6: Change Username Use Case Description

**Actors:** User

**Summary:** A user changes their username while logged in.

**Purpose:** Allows logged in users to change their username.

**Preconditions:**

1. User must have a registered account.
2. User must be logged in.

**Steps:**

1. Users clicks Change Username button.
2. System prompts user to enter a new username.
3. User enters a username and clicks Change Username.

**Alternatives:**

1. If username doesn't exist, System changes the user's username and notifies the user in the UI and through an email.
2. If username exists, System asks user to enter a different username.

**Related Use Cases:**

1. Change Password
2. Change Email

**Relevant Classes:**

- **User** in **S3.4.5**.
- **Email** in **S3.4.5**.
- **UserController** to be added.
- **ServerController** to be added.
- **Database** to be added.

#### 2.1.7 Project Browsing Feature 1: Project Browsing Use Case Description

---

**Actors:** User

**Summary:** User looks through posted project ideas to find projects to work on and/or discuss.

**Purpose:** To find and view projects relevant to the user's search parameters

**Preconditions:** User is signed in

**Steps:**

1. Actor selects Browse Project Ideas button
2. Actor refines search by selecting from list of project categories as desired
3. Actor enters terms into search field as desired and views a list of top projects
4. Actor selects desired project
5. System displays detailed project information

**Alternative 1:** None

**Relevant Classes:**

- **User** in **S3.4.5**
- **Project Browser** in **S3.4.5**
- **Project** in **S3.4.5**

---

#### 2.1.8 Project Browsing Feature 2: Project Creation Use Case Description

---

**Actors:** User

**Summary:** User will create a project.

**Purpose:** To allow users to create projects and make them accessible to other users

**Preconditions:** User is signed in

**Steps:**

1. User selects Create Project button
2. User will enter the information on the project, including its name, goals, and identifying tags.
3. If project name does not match any existing project, the system will create a project with the specified parameters and set user as an admin for the project.

**Alternative 1:** Project name matches the name of an existing project and will ask the user to rename it.

**Relevant Classes:**

- **User** in **S3.4.5**
- **Project Browser** in **S3.4.5**
- **Project** in **S3.4.5**

---

#### 2.1.9 Project Browsing Feature 3: Project Commenting Use Case Description

---

**Actors:** User

**Summary:** Provide detailed feedback on project ideas

**Purpose:** To allow users to write longform feedback on projects as necessary.

**Preconditions:** User is signed in

**Steps:**

1. User selects Comment button
2. User types feedback into field
3. User clicks Submit button
4. System shows confirmation that feedback was received

**Alternative 1:** If the project requires comments to be made by project members only and the user is not a project member, the user will be shown an error message.

**Relevant Classes:**

- **User** in **S3.4.5**
- **Project** in **S3.4.5**

---

#### 2.1.10 Project Browsing Feature 4: Project Voting Use Case Description

---

**Actors:** User

**Summary:** Support promising project ideas or offer criticism to unfavorable ones

**Purpose:** Allow for feedback and help users search for well received projects

**Preconditions:** User is signed in

**Steps:**

1. User selects Browse Project Ideas button
2. User refines search by selecting from list of project categories as desired
3. User enters terms into search field as desired and views a list of top projects
4. User selects desired project
5. System displays detailed project information
6. User selects Up Vote or Down Vote button
7. Project receives the vote and updates its total score

**Alternative 1:** None

**Relevant Classes:**

- **User** in **S3.4.5**
- **Project Browser** in **S3.4.5**
- **Project** in **S3.4.5**

---

#### 2.1.11 Communication Feature 1: Read Project Chat Use Case Description

---

**Actors:** User

**Summary:** Open a window to view the conversation in a project

**Purpose:** Allow users to communicate quickly without permanently taking up screen space in a project

**Preconditions:** User is signed in and viewing a project

**Steps:**

1. User selects the "Open Chat" button
2. System opens the chat window, and notifies the other users in the project of the new arrival
3. System displays any messages from other users in the project in that window until it is closed/left.

**Alternative 1:** None

**Relevant Classes:**

- **User**
- **TextChat**
- **ChatDisplay**
- **Message**

---

#### 2.1.12 Communication Feature 2: Write to Project Chat Use Case Description

---

**Actors:** at least one User

**Summary:** Contribute to the conversation in a project

**Purpose:** Allow users to communicate quickly without permanently taking up screen space in a project

**Preconditions:** User is signed in and has joined the project chat

**Steps:**

1. User types a message in the project chat window and presses Send
2. System sends that message to the project server, which delivers it to the other users in the project chat
3. Other users may read and/or respond to the message at their leisure

**Alternative 1:** None

**Relevant Classes:**

- **User**
- **TextChat**
- **ChatDisplay**
- **Message**

---

#### 2.1.13 Communication Feature 3: Message User by Name Use Case Description

---

**Actors:** User

**Summary:** Start talking an individual user

**Purpose:** Allow users to communicate outside of a project, or in private

**Preconditions:** At least one of the users are signed in

**Steps:**

1. User selects "Send PM" from the chat menu and types or selects the other user's ID
2. System checks to see if the user exists and is online, and if so creates a chat channel for the two users
3. Both users then use the chat as normal

**Alternative 1:** If the second user exists but is offline, the first user is notified and the second user gets the message from the server the next time they're online.

**Relevant Classes:**

- **User**
- **TextChat**
- **ChatDisplay**
- **Message**

---

#### 2.1.14 Communication Feature 4: Comment on Project Use Case Description

---

**Actors:** User, sometimes also a Project Admin

**Summary:** Communicate more important info about a project

**Purpose:** Allow users to record and semi-permanently attach messages to be displayed alongside a project

**Preconditions:** User is signed in and has joined the project

**Steps:**

1. In a section of the window seperate from temporary chat, the user writes a comment and presses Post.
2. System sends that message to the project server, which attaches it to the project. The message is then displayed with previous comments in the post.
3. Other users may read and/or respond to the message at their leisure

**Alternative 1:** A project administrator may remove the post.

**Relevant Classes:**

- **User**
- **TextChat**
- **ChatDisplay**
- **Message**

---

#### 2.1.15 Communication Feature 4: Comment on Project Use Case Description

---

**Actors:** User or Admin

**Summary:** Clean up when a user leaves a project chat

**Purpose:** Close the window and remove the user from the list of active users in the project chat so they don't receive more messages

**Preconditions:** User is signed in and has joined the project chat

**Steps:**

1. User clicks "close" on the project chat window
2. System minimizes the window, and removes the user from the list of active users in the project chat

**Alternatives: Step one is skipped if one of the following happen:**

1. The user closes the entire project or program windows
2. The user is idle for too long
3. An administrator removes them from the project or project chat

**Relevant Classes:**

- **User**
- **TextChat**
- **ChatDisplay**
- **Message**

---

#### 2.1.16 Project Management Feature 1: Compile and Execute Project Use Case Description (dani2918)

---

**Actors:** User

**Goals:** Compile and execute active project

**Pre-conditions:** Actor is logged in, navigated to desired project.

**Summary:** User compiles a project and the project executes.

**Related use cases:**

**Steps:**

1. User clicks "Compile"
2. System displays results of compilation
3. System executes compiled project.

**Alternatives:** Compilation fails.

**Post-conditions:** None.

---

#### 2.1.17 Project Management Feature 2: Create project Use Case Description(dani2918)

---

**Actors:** User

**Goals:** Create a new project.

**Pre-conditions:** Actor is logged in.

**Summary:** User creates a new project with a description and includes any desired files.

**Related use cases:** Import file

**Steps:**

1. User clicks “New Project.”
2. User gives project a title.
3. User adds an applicable description.
4. User imports any files by clicking ”Import.”
5. User clicks “Create”.
6. System imports files and instantiates project.

**Alternatives:** None.

**Post-conditions:** None.

---

### 2.1.18 Project Management Feature 3: Delete Project Use Case Description (dani2918)

2.1.17

---

**Actors:** Project Administrator, sQuire Administrator.

**Goals:** Remove project from sQuire sever.

**Pre-conditions:** Actor is logged in, viewing desired project.

**Summary:** Actor chooses to delete or remove an irrelevant or inappropriate project.

**Related use cases:** Create project
**Steps:**

1. Actor clicks "Delete" icon on active project.
2. Delete dialog opens.
3. Actor presses "Delete".
4. Confirmation window is displayed.
5. Actor confirms or disregards deletion.
6. System notifies collaborators that project was deleted.

**Alternatives:** Actor clicks "Cancel."

**Post-conditions:** None.

---

#### 2.1.19 Project Management Feature 4: Request to Join Project Use Case Description (dani2918)

---

**Actors:** User

**Goals:** Join an existing project.

**Pre-conditions:** Actor is logged in, viewing desired project.

**Summary:** Actor sends a request to join as a collaborator on a project

**Related use cases:** Manage request to join project

**Steps:**

1. Actor clicks "Join project."
2. Notification is sent to project administrator for review.

**Alternatives:** None.

**Post-conditions:** None.

---

### 2.1.20 Project Management Feature 5: Manage Request to Join Project Use Case Description (dani2918)

---

**Actors:** Project Administrator

**Goals:** Approve .

**Pre-conditions:** Actor is logged in, viewing desired project.

**Summary:** Actor approves/rejects a user who has requested to join as a collaborator on a project.

**Related use cases:** Request to join project

**Steps:**

1. Actor clicks "Review join requests."
2. Actor reviews information about potential collaborator.
3. Actor clicks "Approve User" to approve a collaborator or "Reject user" to reject a collaborator.
4. System notifies user that they have been approved/rejected.

**Alternatives:** None.

**Post-conditions:** None.

---

#### 2.1.21 Project Management Feature 6: Leave Project Use Case Description (dani2918)

---

**Actors:** User

**Goals:** Remove actor as a collaborator from project.

**Pre-conditions:** logged in, viewing desired project, collaborator on desired project, not project owner.

**Summary:** A member of a project leaves said project, leaving the project intact.

**Related use cases:** Delete project

**Steps:**

1. User clicks “Leave Project”.
2. System prompts user to confirm decision.
3. User clicks "Confirm".
4. User is removed from project member list.

**Alternatives:** User clicks "Cancel" at step 4.

**Post-conditions:** None.

---

### 2.1.22 Project Management Feature 7: Invite to Project Use Case Description (dani2918)

---

**Actors:** Project Administrator, Authorized Project Collaborator (User).

**Goals:** Invite user(s) to collaborate on project

**Pre-conditions:** Actor is viewing project which he or she created, is logged in

**Summary:** A project administrator requests help from a user on a project. The sQuire system facilitates the invitation process

**Related use cases:** Respond to project invite

**Steps:**

1. Actor clicks "Invite User."
2. Actor enters the username(s) of the user(s) to be invited to the project.
3. Actor enters any message to the user(s) in a text box.
4. Actor clicks "Send invite."
5. System sends notification of invite to user(s).

**Alternatives:** Actor clicks "Cancel." **Post-conditions:** None.

---

### 2.1.23 Project Management Feature 8: Respond to Project Invite Use Case Diagram (dani2918)

---

**Actors:** User

**Goals:** Actor responds to an invitation to a project.

**Pre-conditions:** Actor is signed in, viewing invitation.

**Summary:** Actor receives notification that he or she has been invited to a project and either accepts the invitation or declines it.

**Related use cases:** Invite to project

**Steps:**

1. Actor clicks "Respond to Invitation."
2. Actor clicks "Accept" or "Reject."
3. Actor types any message to invitation-sender in text box.
4. Actor clicks "Confirm."

**Alternatives:** Actor clicks "Cancel."

**Post-conditions:** Actor becomes collaborator on project if invitation was accepted.

---

#### 2.1.24 File Editing Feature 1: View Line Numbers Use Case Description

---

**Name:** View Line Numbers

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to view line numbers to the left of the document.

**Purpose:** Makes it easier to communicate position in code. It is also a useful metric to have.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has view permission.
4. A file is open.

**Steps:**

1. User selects the *View* menu option.
2. System displays a drop-down with various options.
3. User selects the *View Line Numbers* option.
4. System displays line numbers to the left of the document.

**Relevant Classes:**

1. **TextOperation**

---

#### 2.1.25 File Editing Feature 2: View References Use Case Description

---

**Name:** View References

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to view the number of references to a given function.

**Purpose:** It is useful to know the number of references to a given function for optimization and debugging purposes.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has view permission.
4. A **code** file is open.

**Steps:**

1. User selects the *View* menu option.
2. System displays a drop-down with various options.
3. User selects the *View References* option.
4. System displays the number of references above each method declaration.

**Relevant Classes:**

1. **ColabFile**
2. **LineHistory**
3. **Project**

---

#### 2.1.26 File Editing Feature 3: View Dates Use Case Description

---

**Name:** View Dates

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to view the last date that each line of a document was edited.

**Purpose:** This provides a useful metric for how up-to-date parts of the document are.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has view permission.
4. A file is open.

**Steps:**

1. User selects the *View* menu option.
2. System displays a drop-down with various options.
3. User selects the *View Dates* option.
4. System displays the last date that each line of a document was edited.

**Relevant Classes:**

1. **LineHistory**
2. **FileLineHistory**

---

#### 2.1.27 File Editing Feature 4: View Authors Use Case Description

---

**Name:** View Authors

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to view the last author that edited each of line of the document.

**Purpose:** This is an accountability tool allowing other users to identify who is responsible for a change to a document.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read permission.
4. A file is open.

**Steps:**

1. User selects the *View* menu option.
2. System displays a drop-down with various options.
3. User selects the *View Authors* option.
4. System displays the name of the last editor of each line of the document.

**Relevant Classes:**

1. **FileLineHistory**
2. **LineHistory**

---

#### 2.1.28 File Editing Feature 5: Format Document Use Case Description

---

**Name:** Format Document

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to format the document to a specified format

**Purpose:** An easy tool for making sweeping changes to a large part of a document.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read/write permission.
4. A file is open.
5. The document has formatting options set.

**Steps:**

1. User selects the *Edit* menu option.
2. System displays a drop-down with various options.
3. User selects the *Format Document* option.
4. System formats the current document to the formatting settings currently set.

**Alternatives:**

1. If no formatting settings are currently set, display a dialog box after step 3 and give the option for the user to do so now.

**Relevant Classes:**

1. **TextOperation**

---

#### 2.1.29 File Editing Feature 6: Find/Replace Use Case Description

---

**Name:** Find/Replace

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to find and/or replace phrases.

**Purpose:** This is a powerful tool that allows a user to make safer, quicker, and more efficient changes to a document.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read/write permission.
4. A file is open.

**Steps:**

1. User selects the *Edit* menu option.
2. System displays a drop-down with various options.
3. User selects the *Find/Replace* option.
4. System displays a small form in an unobtrusive location.
5. User enter the phrase to find and selects find.
6. System highlights and focuses on the first occurrence of the phrase and all highlights all other occurrences.

**Alternatives:**

1. User selects option to replace in step 5 and enters a phrase with which to replace the found occurrences of the searched phrase. The system replaces each occurrence.

**Relevant Classes:**

1. **Project**

---

#### 2.1.30 File Editing Feature 7: Comment Section Use Case Description

---

**Name:** Comment Section

**Category:** File Editing

**Actor:** User

**Summary:** Allows the user to comment out a part of a document.

**Purpose:** A useful and quick way to disable a large part of a document.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. A file is open.
4. User has read/write permission.
5. Current open document supports commenting.

**Steps:**

1. User selects the *Edit* menu option.
2. System displays a drop-down with various options.
3. User selects the *Comment Section* option.
4. System comments the selected area.

**Alternatives:**

1. If document does not support commenting, display a dialog box telling the user.

**Relevant Classes:**

1. **TextOperation**

---

### 2.1.31 File Editing Feature 8: Display Typing User Use Case Description

---

**Name:** Display Typing User

**Category:** File Editing

**Actors:**

1. User
2. Other Users

**Summary:** As the user types, the system displays their name, their typing, and their caret, in a different color, to other users.

**Purpose:** Differentiate who is typing what.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read/write permission.
4. A file is open.
5. Other users have the same document open.

**Steps:**

1. User begins typing.
2. System displays the user's typing, the user's name, and the user's caret, in a different color, to Other Users.
3. Other Users see User typing, his username, and his caret, in a different color.

**Relevant Classes:**

1. **ColabFile**
2. **Cursor**
3. **Project**

---

#### 2.1.32 File Editing Feature 9: Display Syntax Errors Use Case Description

---

**Name:** Display Syntax Errors

**Category:** File Editing

**Actor:** User

**Summary:** As the user types code, the editor will underline syntax errors with a red line.

**Purpose:** Aids the user is writing correct code.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read/write permission.
4. A supported code file is open.

**Steps:**

1. User begins typing.
2. System displays any syntax errors as a red underline under the incorrect section.

**Relevant Classes:**

1. **ColabFile**
2. **Project**

---

#### 2.1.33 File Editing Feature 10: Display Syntax Highlighting Use Case Description

---

**Name:** Display Syntax Highlighting

**Category:** File Editing

**Actor:** User

**Summary:** As the user types code, the editor will change font color for different code structures and keywords.

**Purpose:** Aids the user is writing code and identifying key code parts.

**Preconditions:**

1. Must be registered.
2. Must be logged in.
3. User has read/write permission.
4. A supported code file is open.

**Steps:**

1. User begins typing.
2. System automatically colors special code structures and keywords.

**Relevant Classes:**

1. **Project**

---

#### 2.1.34 File Management Feature 1: Open File Use Case Description

---

**Actors:** User

**Summary:** The user selects a file to open based on a filename, which is then opened by the software.

**Purpose:** To open a file in the program.

**Preconditions:** The desired file must already exist.

**Steps:**

1. User clicks Open File button.
2. System prompts the user to select file to open from a list of existing files.
3. User selects desired file and clicks Submit button.
4. System opens selected file and displays it.

**Alternative 1:** The User decides they don't want to open a file and presses Cancel at step 3.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UI** to be added.
- **Project** to be added.
- **File** to be added.

---

#### 2.1.35 File Management Feature 2: Close File Use Case Description

---

**Actors:** User

**Summary:** The user chooses to close the file they are working on.

**Purpose:** To close a file in the program.

**Preconditions:** The desired file must already exist, and be already opened by the User.

**Steps:**

1. User clicks Close File button.
2. System closes the file and updates the Controller's status on the file being open.

**Alternative 1:** The file cannot be closed for some reason.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UI** to be added.
- **Project** to be added.
- **File** to be added.

---

#### 2.1.36 File Management Feature 3: Save File Use Case Description

---

**Actors:** User

**Summary:** The user chooses to save the file they are currently working on.

**Purpose:** To save a file in the program.

**Preconditions:** The desired file must already exist, and be opened by the user.

**Steps:**

1. User clicks Save File button.
2. System prompts the user to choose a name to save the file under.
3. User selects desired name and clicks Submit button.
4. System saves selected file and allows the user to keep working.

**Alternative 1:** The User decides they don't want to save the file and presses Cancel at step 3.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UI** to be added.
- **Project** to be added.
- **File** to be added.

---

#### 2.1.37 File Management Feature 4: Add File Use Case Description

---

**Actors:** User

**Summary:** The user chooses to add a file to the directory.

**Purpose:** To add a file to a directory.

**Preconditions:** The desired file must already exist.

**Steps:**

1. User clicks Add File button.
2. System prompts the user to choose a file to add to a directory..
3. User selects desired file and clicks Submit button.
4. System prompts the user to choose a directory to add the file to.
5. User selects directory to add the file to and click Submit button.
6. System adds the file to the selected directory and lets the user return to their work.

**Alternative 1:** The User decides they don't want to add the file and presses Cancel at step 3.
**Alternative 2:** The User decides, after they chose a file, not to add it to a directory and clicks Cancel at step 5.

**Relevant Classes:**

- **User** in **S3.4.5**
- **UI** to be added.
- **Project** to be added.
- **File** to be added.

---

#### 2.1.38 Project User Management Feature 1: Add User to Project Use Case Description

---

**Actors:** User
**Goals:** Add a user to project
**Pre-conditions:** User has admin rights to project.
**Summary:** User adds a user to a project
**Related use cases:** Kick User
**Steps:**

1. User clicks add user button.
2. System prompts user to enter the username of the user they wish to invite.
3. User enters username.
4. System adds the specified user to the project, and notifies them.

**Alternatives:**

1. User enters an invalid username, in which case an error is reported

**Post-conditions:** None.
**Relevant Classes:**

- **User**
- **Project**
- **UserManager**
- **Permissions**
- **Email**

---

#### 2.1.39 Project User Management Feature 2: Kick User Use Case Description

---

**Actors:** User
**Goals:** Kick a user from project.
**Pre-conditions:** User is an admin, and the user they wish to kick is a membor of the project.
**Summary:** User removes a selected user from the Project
**Related use cases:** Add User
**Steps:**

1. User clicks Kick User button.
2. System displays list of Members of the project.
3. User selects one or more other users from the list and presses Remove.
4. System prompts User for verification.
5. User presses Confirm.
6. System removes the selected users from the project.

**Alternatives:** None.
**Post-conditions:** None.

- **User**
- **Project**
- **UserManager**
- **Permissions**

---

#### 2.1.40 Project User Management Feature 3: Set User Permissions Use Case Description

---

**Actors:** User
**Goals:** Modify a userś permissions.
**Pre-conditions:** User has admin permissions, and the user whose permissions they wish to change is a member of the project
**Summary:** User modifies another Userś permissions to the project.
**Related use cases:** None.
**Steps:**

1. User clicks Permissions Management button.
2. System displays permissions management window.
3. User selects the user whose permissions they want to edit.
4. System displays a list of toggles for the users's permissions.
5. User makes changes to the user's permissions.
6. System modifes the target Userś permissions.

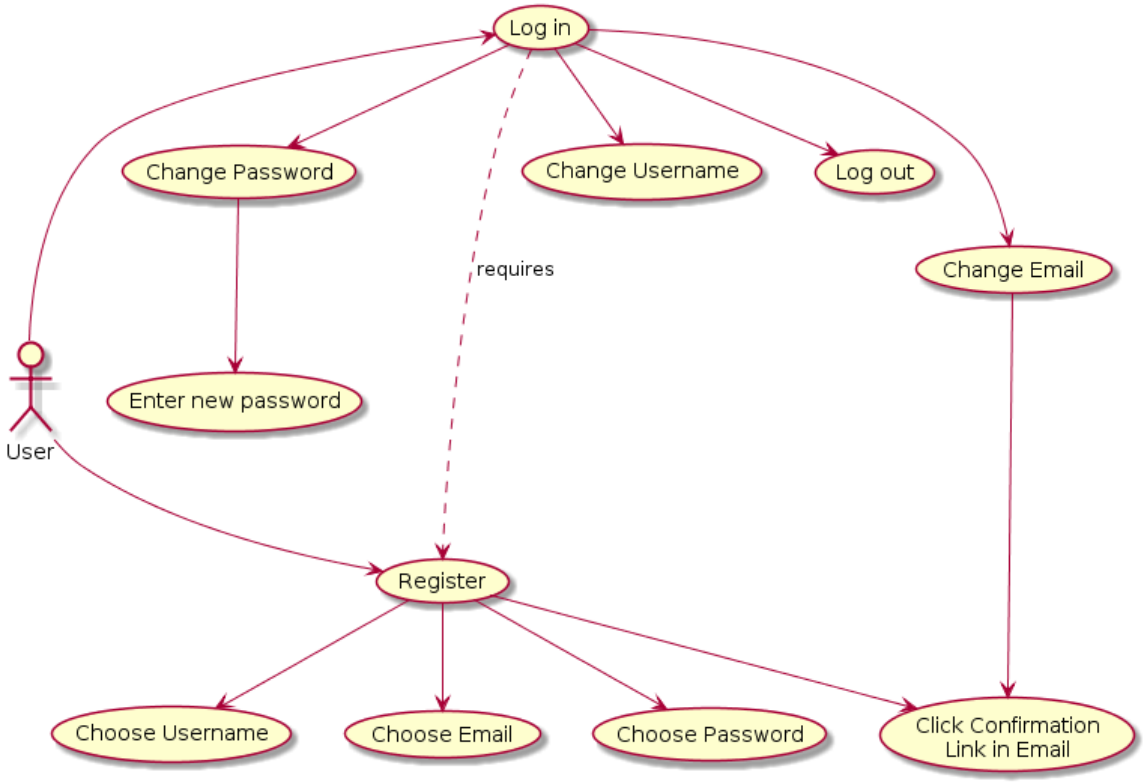
**Alternatives:** None.
**Post-conditions:** None.

- **User**
- **Project**
- **UserManager**
- **Permissions**

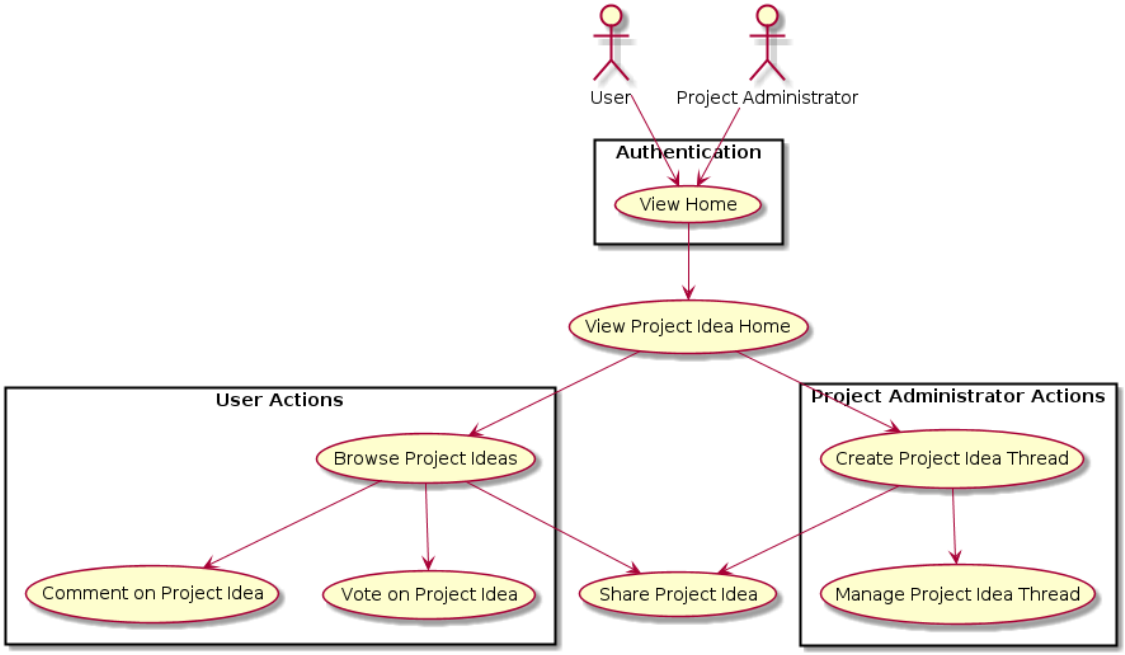
---

## 2.2 USE CASE DIAGRAMS

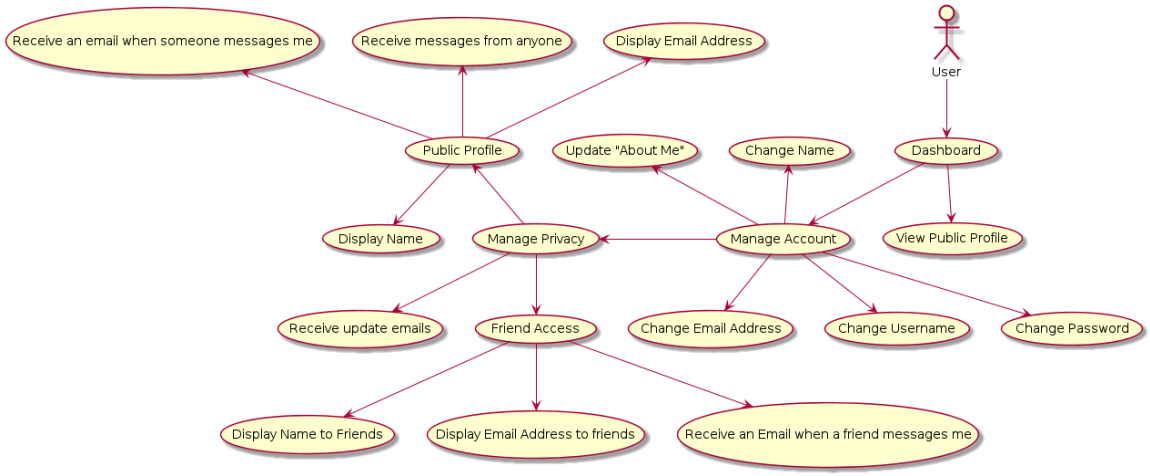
### 2.2.1 Use Case Diagram 1: Authentication



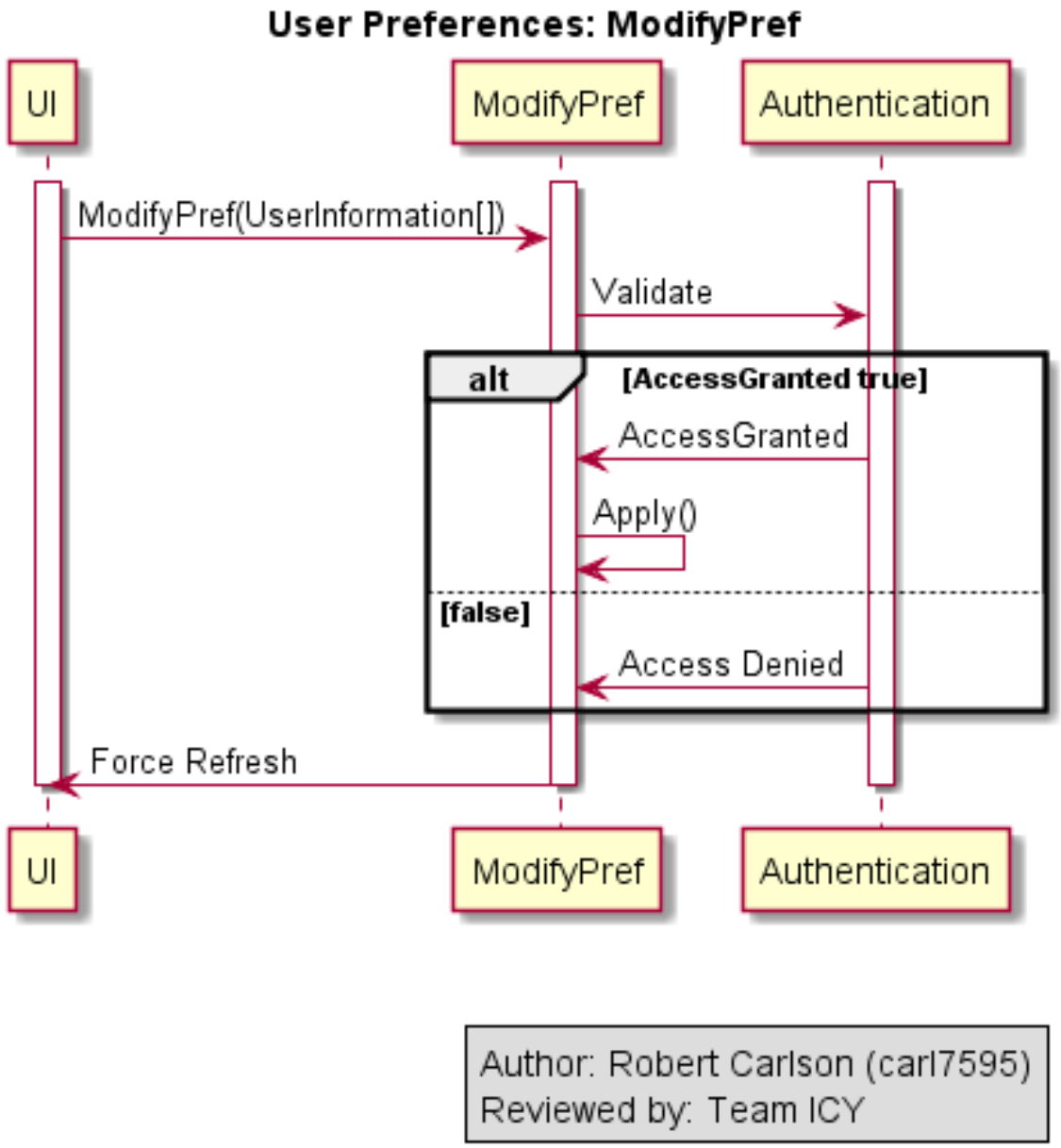
### 2.2.2 Use Case Diagram 2: Project Browsing



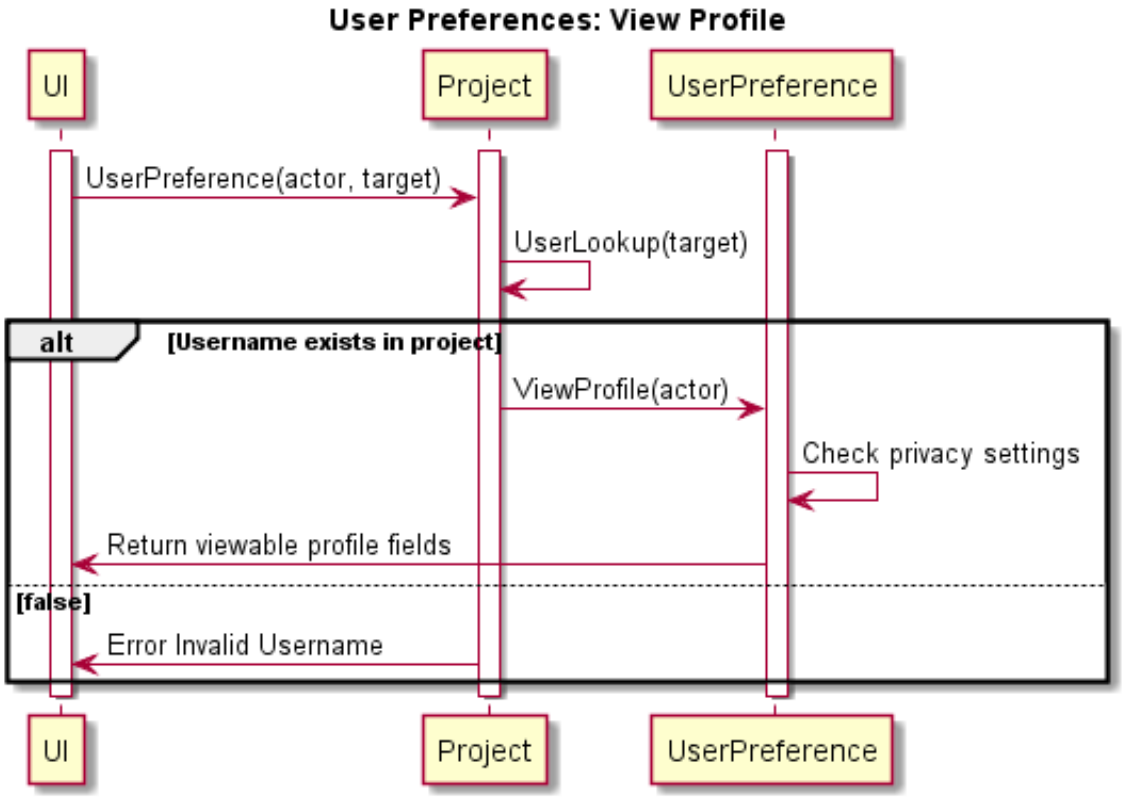
### 2.2.3 Use Case Diagram 4: User Preferences



### 2.2.4 User Preferences Feature 1: Use Case Diagram 1

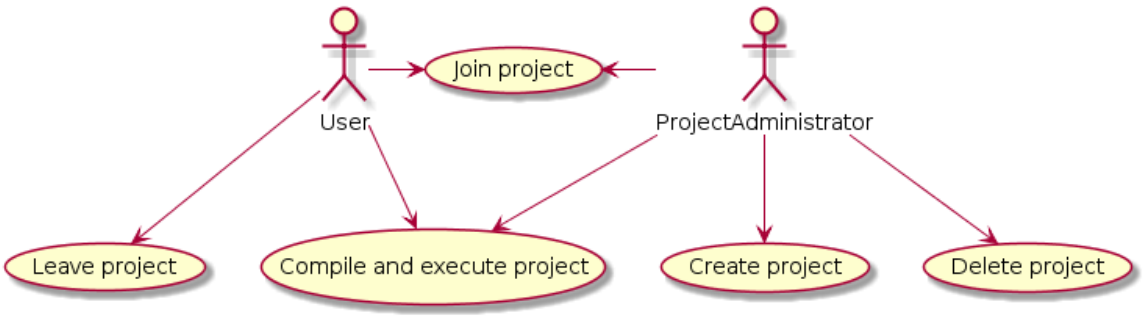


### 2.2.5 User Preferences Feature 2: Use Case Diagram 2




Author: Robert Carlson (carl7595)
Reviewed by: Team ICY

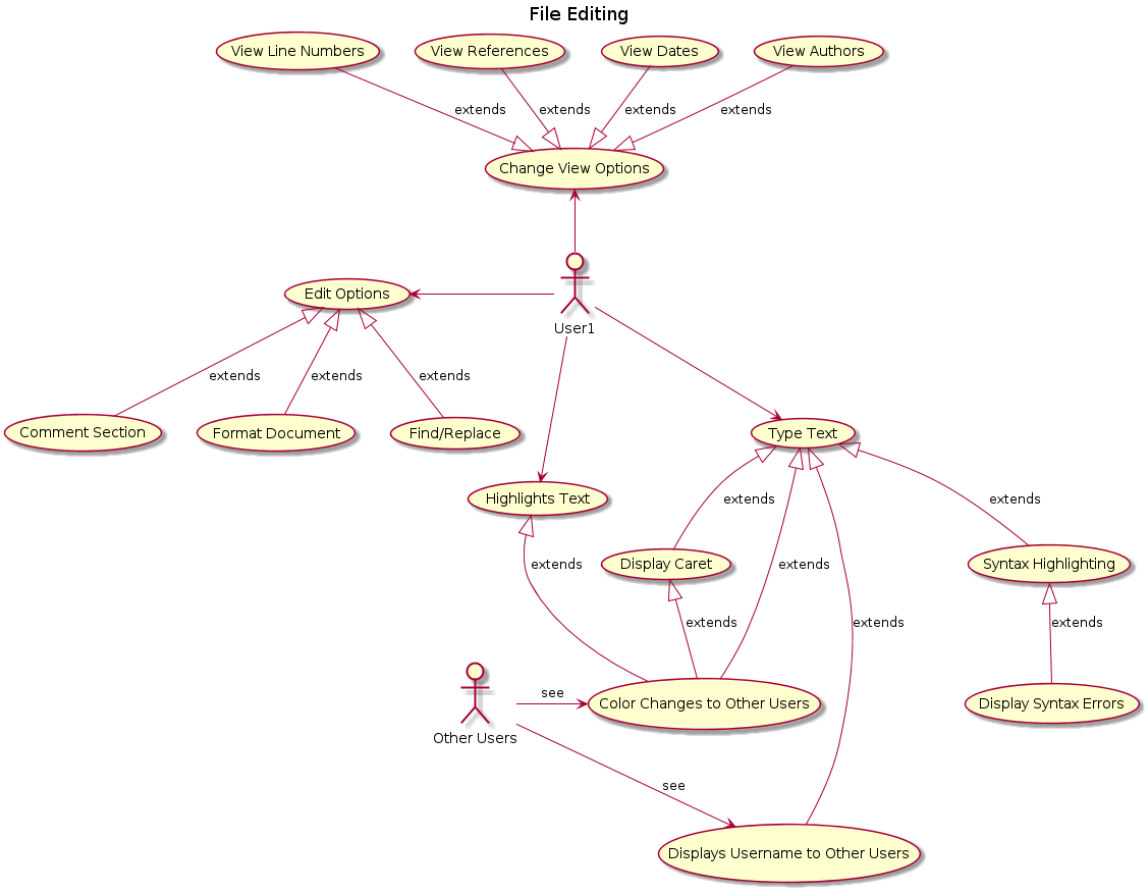
### 2.2.6 Use Case Diagram 5: Project Management



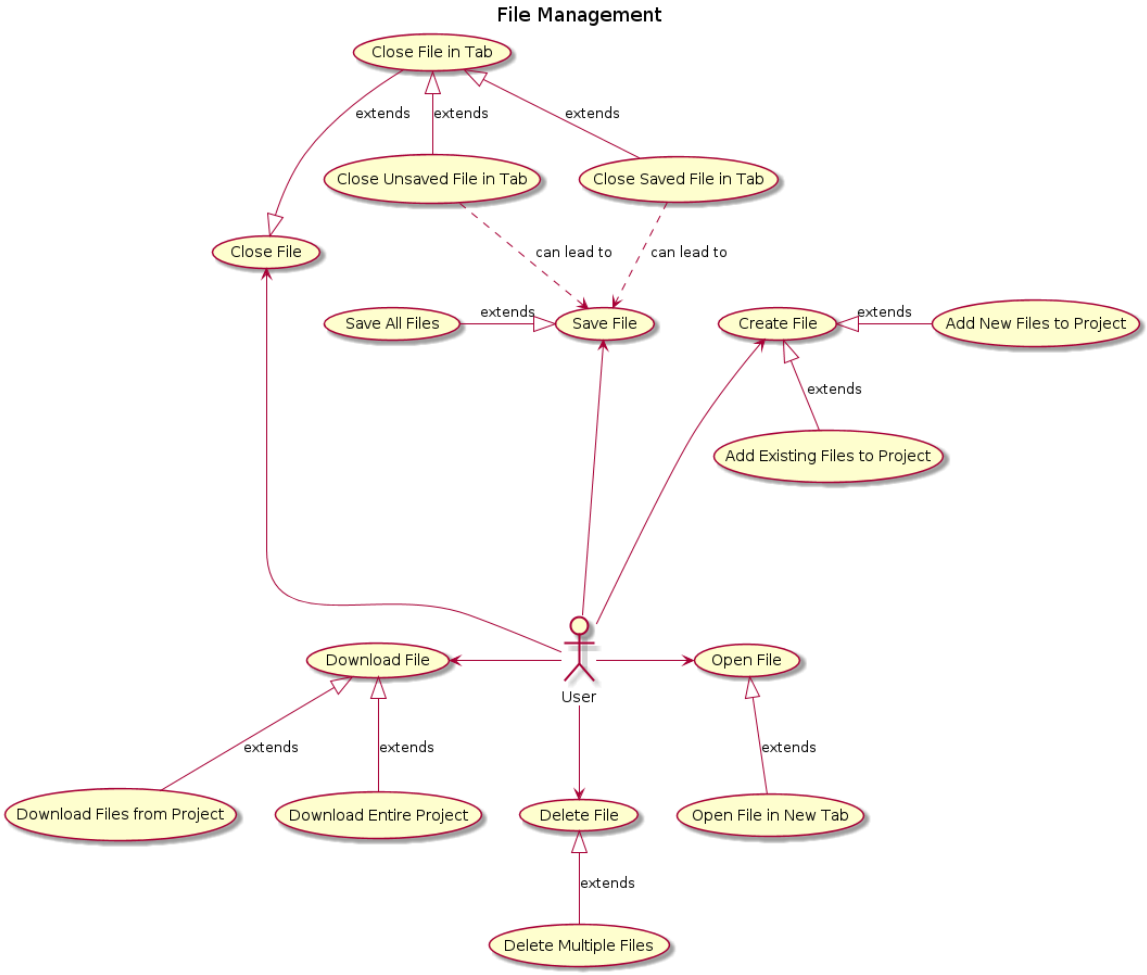

Use case descriptions were roughly based upon cases from HW2, Team 4. sass8427 worked on the original use cases in this section.

**Traceability**: Relevant classes are found in **Project Management** section and include *User* and *Project*. For the *User* class, methods and fields from all class diagrams will be used.

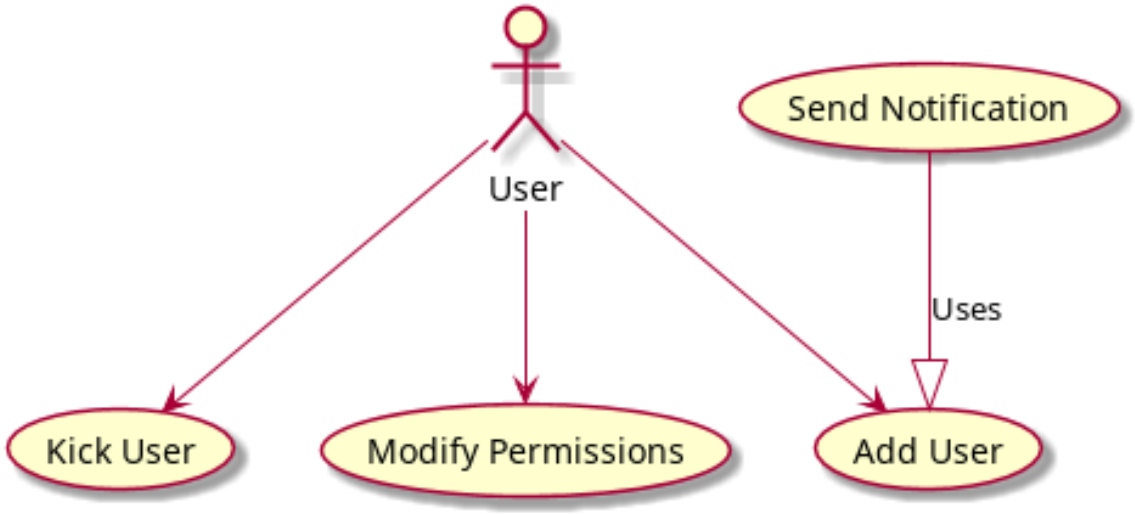
### 2.2.7 Use Case Diagram 6: File Editing



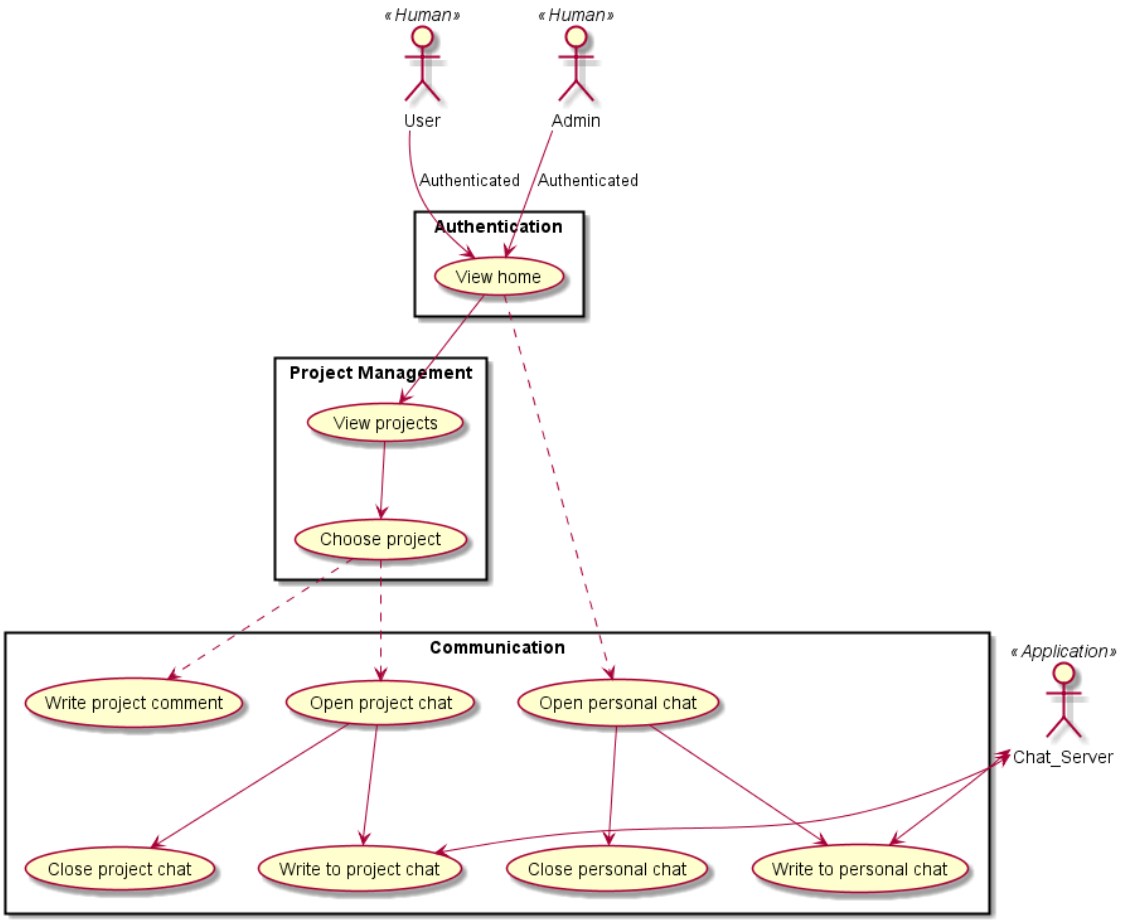
### 2.2.8 Use Case Diagram 7: File Management



### 2.2.9 Use Case Diagram 8: Project User Management

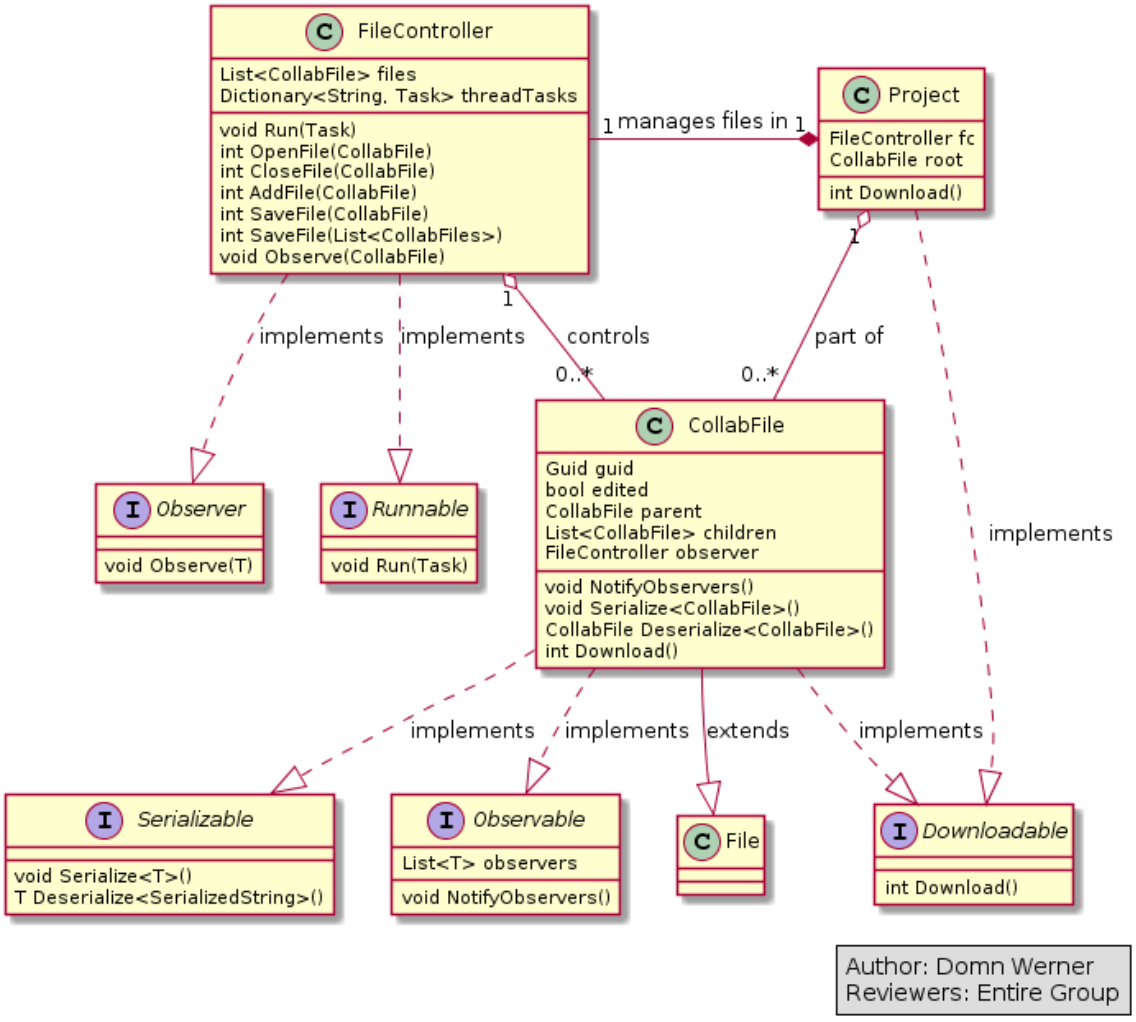


### 2.2.10 Use Case Diagram 3: Communication



## 2.3 CLASS DIAGRAMS

### 2.3.1 Class Diagram 1: File Management



### 2.3.2 Class Diagram Description 1: File Management Description

**Interfaces:**

- The *Observer* interface requires its implementers to implement a method with the following signature: **void Observe(T)**. The purpose of this method is for classes to implement ways in which to observe other classes. We foresee the **FileController** class implementing this interface in order to observe changes in **CollabFile** objects.

- The *Runnable* interface requires its implementers to implement a method with the following signature: **void Run(Task)** where **Task** is a method that can be run in a separate thread. The **FileController** class will implement this interface in order to execute its IO operations in a separate thread. This will keep the UI thread free and our program responsive.

- The *Serializable* interface requires its implementers to implement the **void Serialize<T>()** method and serializes objects of type **T**. It also requires the implementation of the **T Deserialize<T>()** method which will operate on a serialized string object and return an instantiated object of type **T**.

- The *Observable* interface requires its implementers to have a list of observers and a method to notify its observers of changes to itself. The purpose of this implementation is to communicate with the **FileController** object and notify it when a CollabFile changes.

- The *Downloadable* interface requires its implementers to implement a method with the following signature: **int Download()**. The purpose of this method is for classes to implement ways in which their objects can be downloaded. The **int** return type will be used as a status code. We foresee this interface being used with the **Project** and **CollabFile** classes, as the diagram shows, allowing users to download files or projects with the **Download()** method.

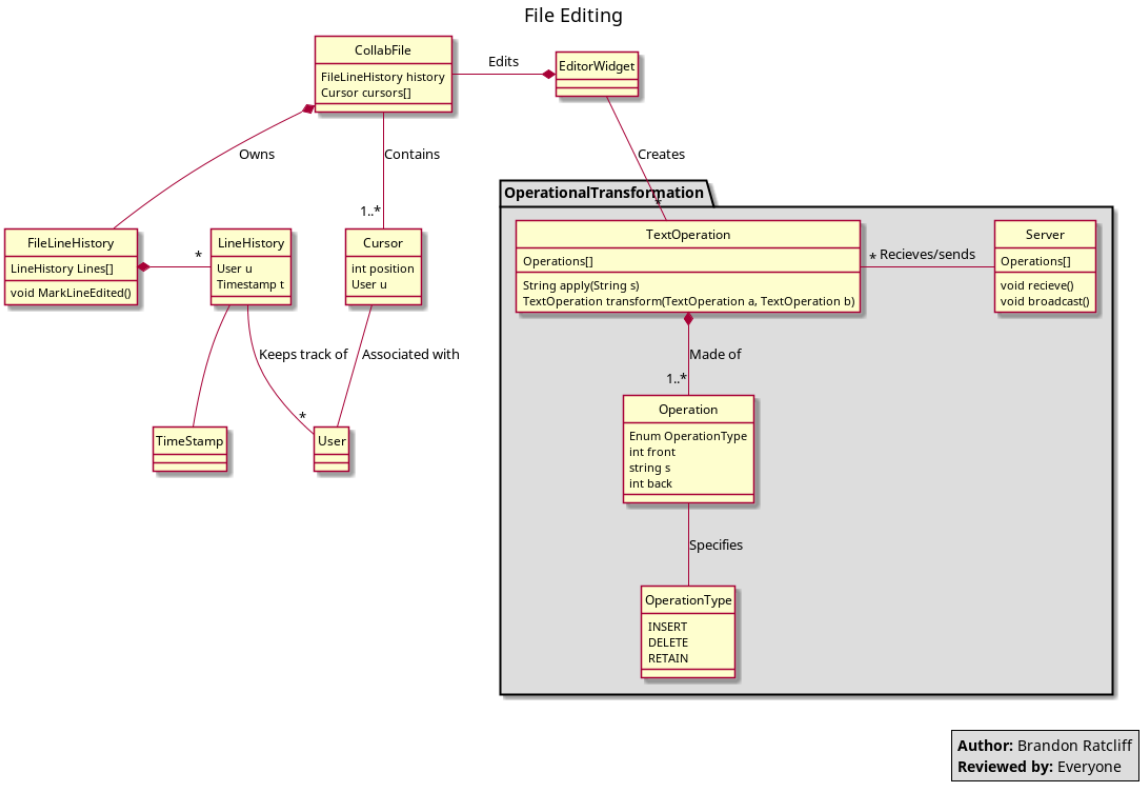
**Classes:**

- The **FileController** class will manage the **CollabFile** objects in the **Project** class. It will do so by storing a list of files and a dictionary of its methods that can be run in a separate thread. Its methods all deal with managing files. It will implement the *Observer* and *Runnable* interfaces. Refer to the interfaces list above to see the details of such implementations.

- The **Project** class represents the entire project that users work on. This includes users, files, permissions, etc. For the sake of simplicity, this diagram only lists properties and methods relating to file editing. Objects of type **Project** will have a **FileController** and a root **CollabFile** as per a file-tree structure. The **Project** class must also implement the *Downloadable* interface in order to specify how projects are downloaded.Refer to the interfaces list above to see the details of this implementation.

- the **CollabFile** class represents a file in a **Project**. It extends the **File** class for the purposes of allowing collaborative editing, among other project functions. It implements the *Serializable* interface to allow its information to be transported over the internet in the best possible format. This requires the implementation of the **void Serialize<T>()** and **T Deserialize<SerializedString>()** methods which will handle serialization and deserialization. This class also implements the *Observable* interface which will specify how it communicates with the **FileController** class in order to notify of relevant changes to **CollabFile** objects. This requires the implementation of a list of observers and a method to notify observers. Lastly, it implements the *Downlodable* interface which will specify how **CollabFile** objects are to be downloaded. Refer to the interfaces list above for more details of such implementations.

### 2.3.3 Class Diagram 2: File Editing




**Author:** Brandon Ratcliff
**Reviewed by:** Everyone

### 2.3.4 Class Diagram Description 2: File Editing Description

**Editor:**

- The *CollabFile* class is a class used in many of the other class diagrams in this project. It is the general class containing all the methods and variables for managing a file. It contains a **FileLineHistory** object. CollabFile has a list of **Cursor** objects, one for every user editing the file..
- The *User* class is a class used in many of the class diagrams. It represents a single user of the sQuire program.
- The *TimeStamp* class is used in several other places. It represents a date and time.
- The *FileLineHistory* class is a class that keeps track of who last edited every line in the file. This will be used to display the changes inside the editor. It does this by having an array (one element per line in the file) of **LineHistory** objects.
- The *LineHistory* class is contains the information used by the **FileLineHistory** class. It contains a **User** the last one to edit a particular line and a **Timestamp** (when the line was last edited). More fields can easily be added to this if it turns out there is more information we'd like to keep track of on a line-by-line basis.
- The *EditorWidget* class is something that we will (hopefully) not write ourselves. It will be the editor widget we use for providing the code editor. Preliminary research found RSyntaxtTextArea (https://github.com/bobbylight/RSyntaxTextArea). This looks like a good fit because it has syntax highlighting, auto completion, code analysis, and of course, support from java. It also has a simple plugin architecture, so it looks like it would be easy to extend to our needs. More research needs to be done to figure out the exact class structure for this.
- The *Cursor* class is made up of a **User** and a position within a file- everything that is needed to display a users cursor inside the editor.

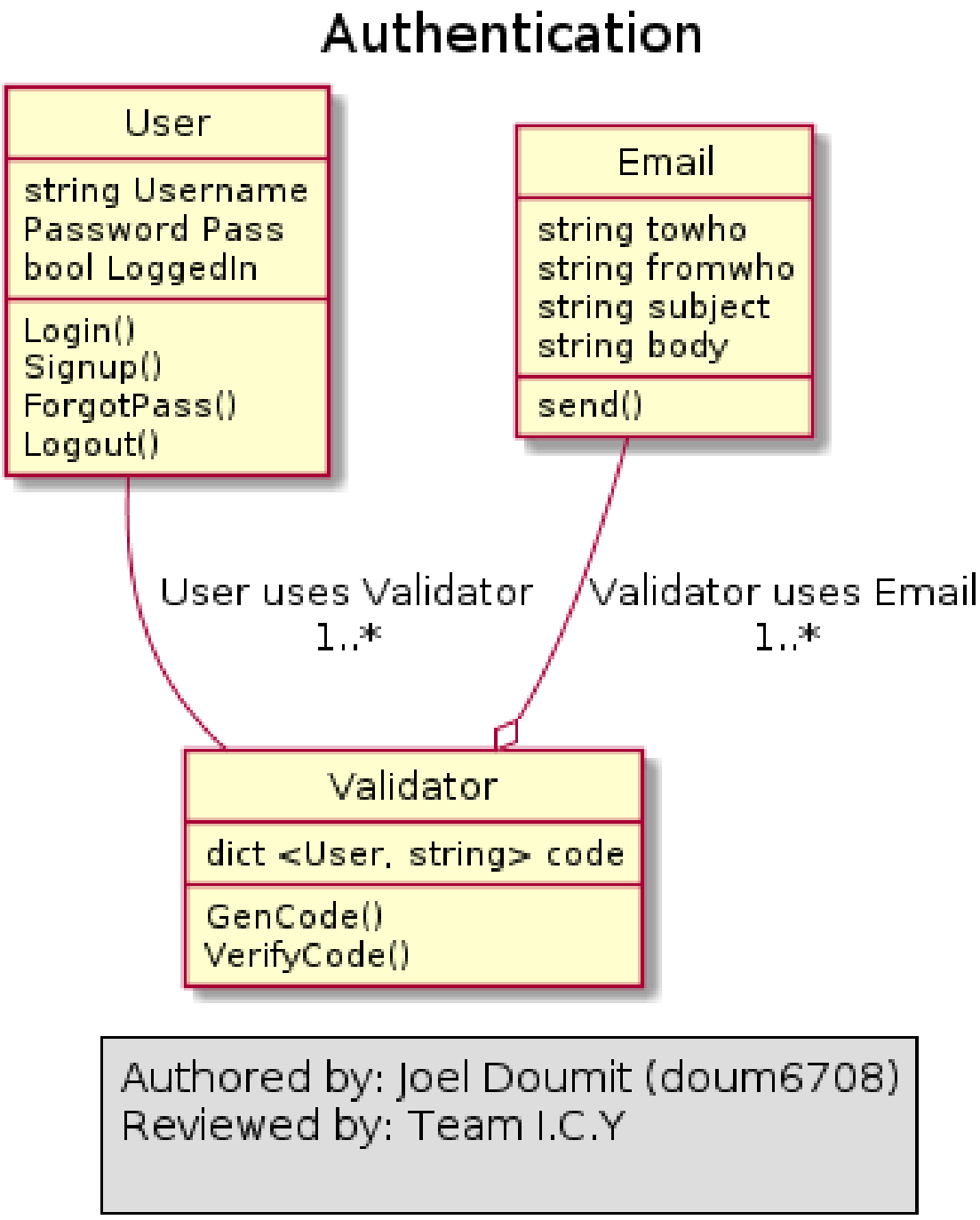
**OperationalTransform:**

I organized this one into a separate package because that's how I'm pretty sure we'll write it. The OperationalTransform algorithm is a collaborative editing algorithm used to allow multiple people to edit the same document at the same time, and keep the documents in sync. Ideally, we would use a pre-built library for this, as the algorithm is quite complex and there are lots of special cases, but I was unable to find one written in Java. Our best best will probably be to port an existing library in another language. The clearest, best documented implementation I found was OT.js (https://github.com/Operational-Transformation/ot.js/). The following classes are the main data structures implemented by this version of Operational Transformation.

- The *TextOperation* class represents a sequence of **Operation** objects, or changes. The TextOperation can then be applied to a string, or transformed with anther TextOperation (from another client) in order account for changes that occurred simultaneously. See the Operational Transformation algorithm for more details.
- The *Operation* class represents a specific operation. This contains an **OperationType**, a integer front, which specifies the number of characters before the change, a string s, which contains the actual character changed (ex, inserted or deleted). And then an integer back, which contains the number of characters until the end of the document.
- The *OperationType* Enum is used to specify what type of operation a **Operation** is. Type can be either an INSERT, when character(s) are inserted to a document, DELETE, when character(s) are deleted from a document, or RETAIN, used to shift other operations.

- the *Server* class is a class that will be running on the server to sync changes between clients. It's job it to listen for **TextOperations** from all connected clients, and when it receives one, broadcast that change to all connected clients.

#### 2.3.5 Class Diagram 3: Authentication

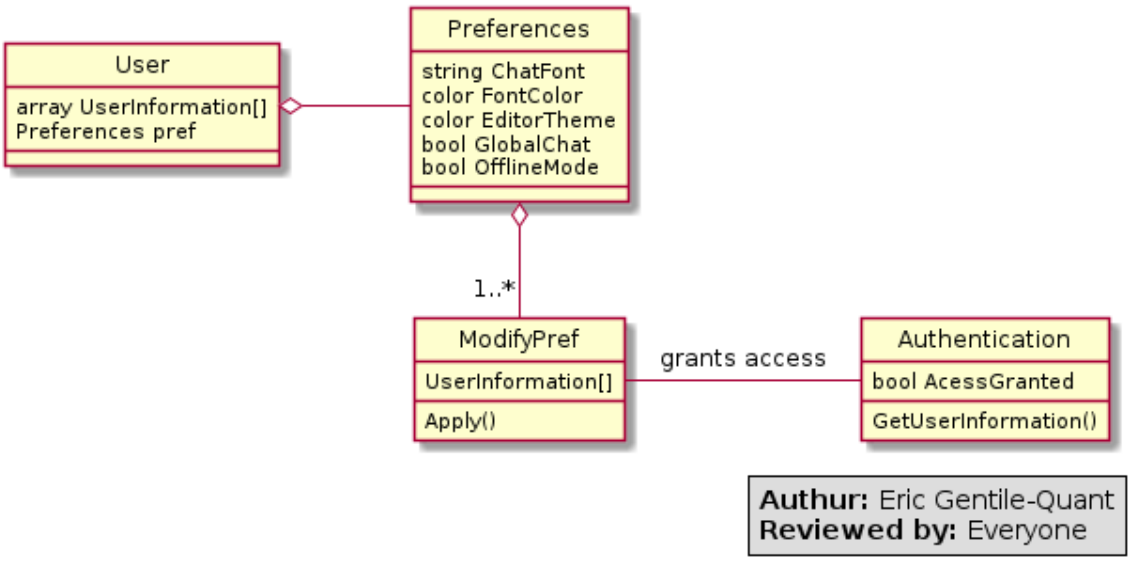


#### 2.3.6 Class Diagram Description 3: Authentication Description

**Classes:**

- The **User** class represents the main user of the entire program. It details the basic information of each individual user, and allows each user the ability to create an account, to log into an existing account, and once logged in, to log out of the user account. It also allows a user to change his/her password, which involves the other classes.
- The **Email** class allows the program to send emails to Users who have signed up, or are signing up. It stores User information as a series of strings to be used by the **send()** function, which sends validation codes to Users' email addresses.
- The **Validator** class will run validation functions when called to do so by the User. Upon a User indicating they would like to change/have forgotten their password, it generates a validation code for that particular User, which it then stores in a dictionary. This validation code is sent to Users by means of the **send()** function denoted earlier.

### 2.3.7 Class Diagram 4: User Preferences

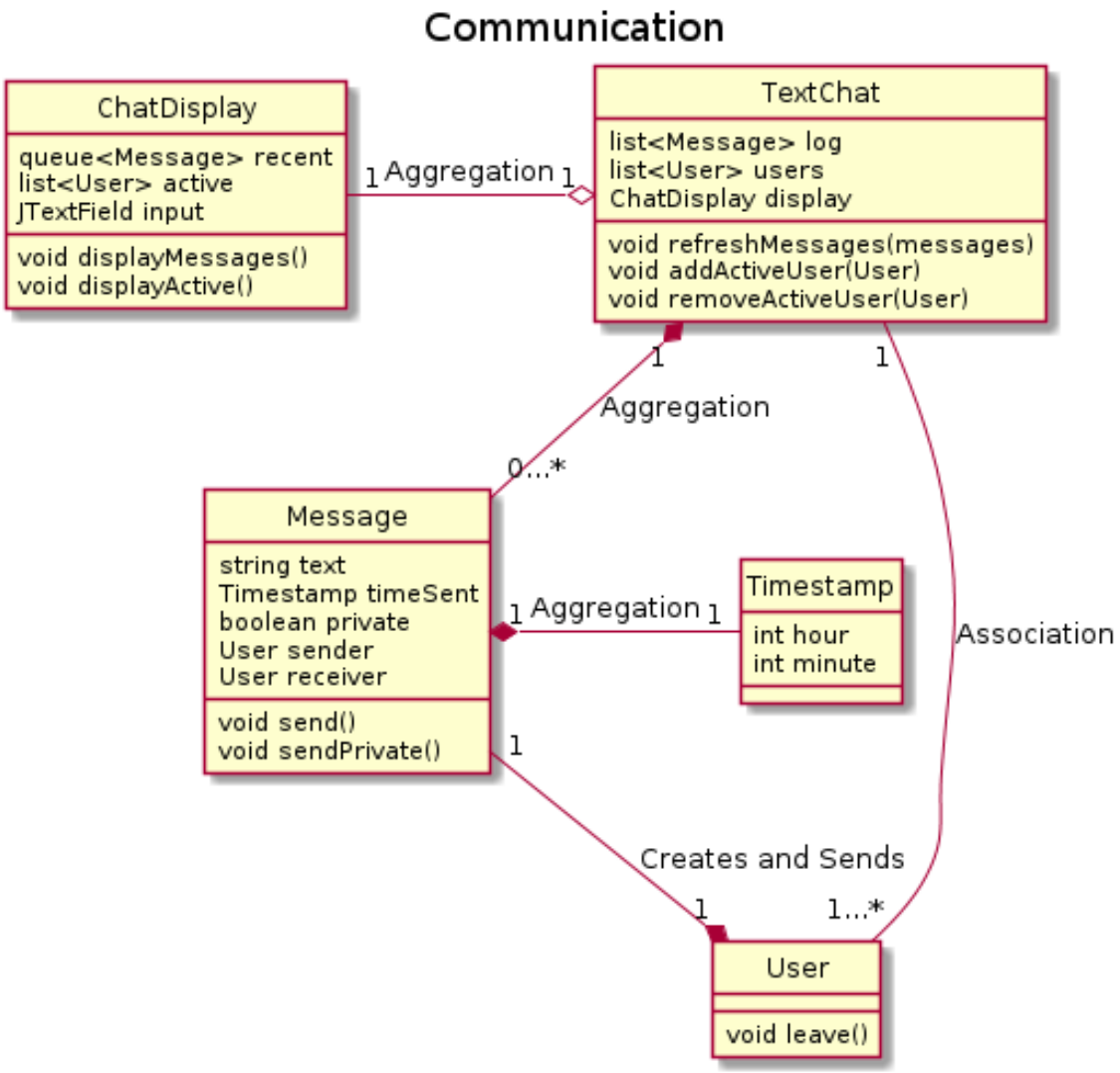


#### 2.3.8 Class Diagram Description 4: User Preferences Description

**Classes:**

- **User:** Represents the human user of the program. It will hold the user's profile information so that it can be validated later.
- **Preferences:** Holds all the user's account preferences. This includes profile picture, chat font, chat color, ect.
- **ModifyPref:** Allows the user to modify his or her's preferences.
- **Authentication:** Authenticates the user's username and password to allow access to make changes on their account.

### 2.3.9 Class Diagram 5: Communication



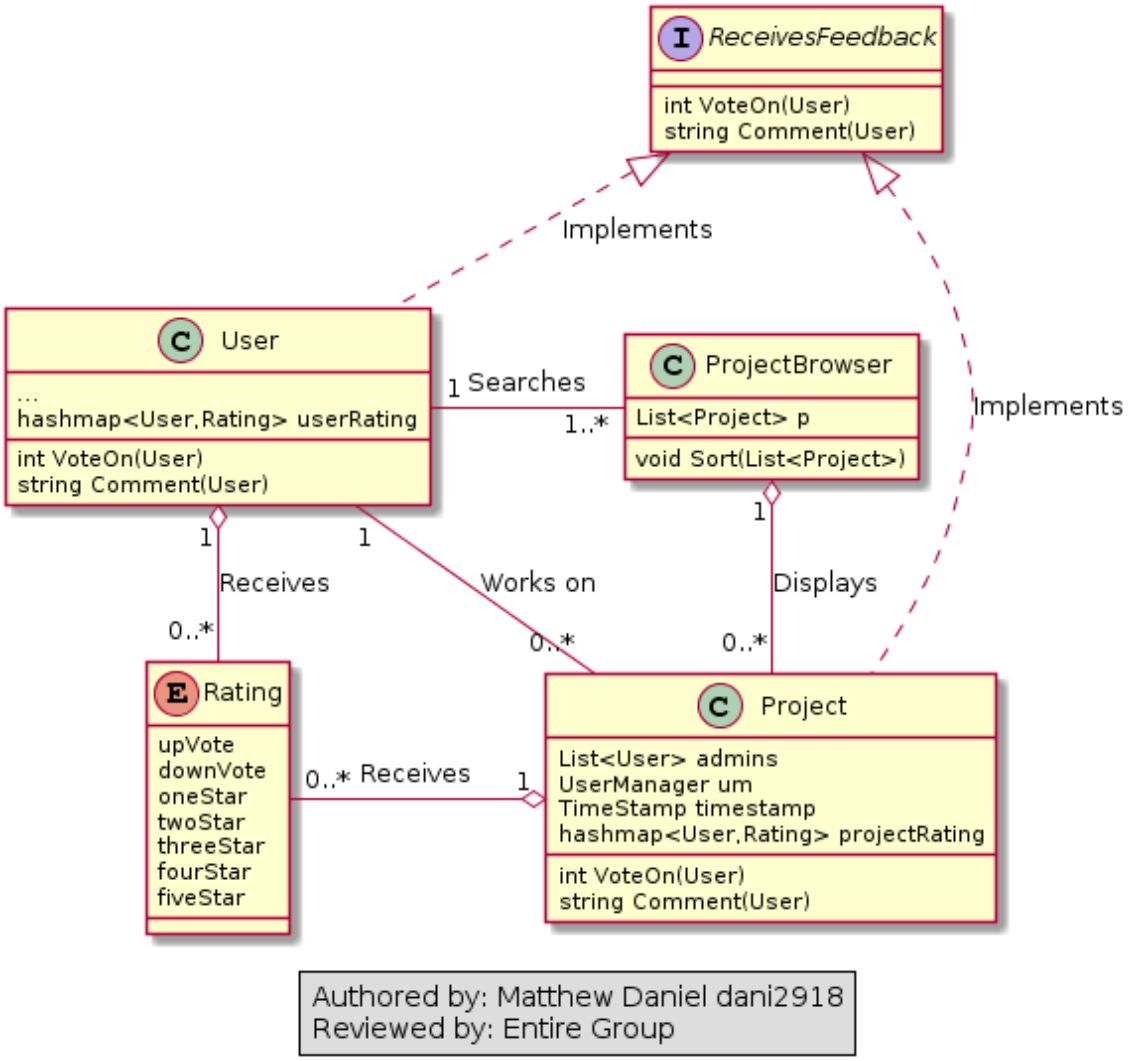

Authored By: Max Welch
Peer Reviewed By: Team I.C.Y.

#### 2.3.10 Class Diagram Description 5: Communication Description

**Classes:**

- **TextChat:** The master class to manage the messages, users, and display of the system.
- **ChatDisplay:** Displays relevant information including most recent 20 messages and active users
- **User:** Users interacting with the chat system.
- **Message:** Messages sent by users to TextChat, contain a string, timestamp, and sender/receiver data.
- **Timestamp:** Recorded time of when message was sent.

### 2.3.11 Class Diagram 6: Project Browsing



#### 2.3.12 Class Diagram Description 6: Project Browsing

**Enums:**

- The *Rating* enum produces a value based upon the user's desired rating of another user or a project.

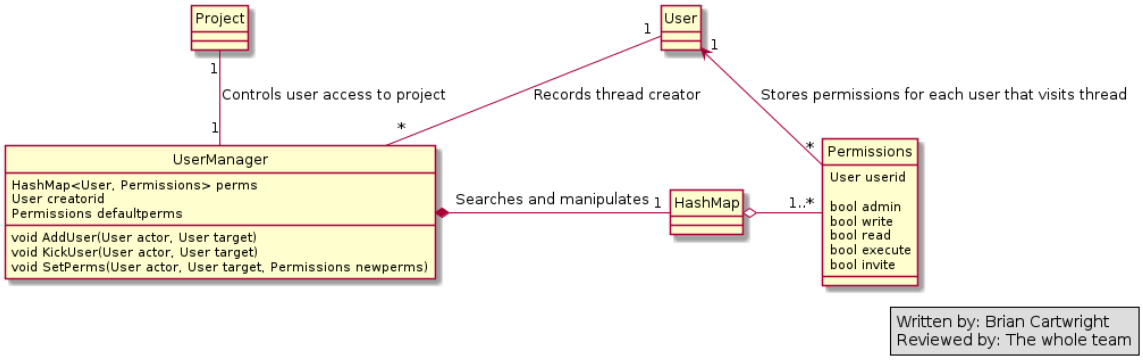
**Interfaces:**

- The *Receives Feedback* interface requires its implementers to implement two methods: **int VoteOnUser(User)**, which returns a value based on a user's rating, and **string Comment(User)**, which leaves feedback in the form of a comment.

**Classes:**

- The **User** class will be the class, shared across many of the class diagrams, that stores information about a user. The User class will have not only the methods and fields shown in this diagram, but a concatenation of the ones shown here and all other methods and fields from the other diagrams. Here, the User class implements the ReceivesFeedback interface so that other users may leave comments/reviews of a User object, and so that they may receive an accompanying rating from one to five stars. In relation to browsing projects, a User is the agent who searches a ProjectBrower object and works on a Project object.
- The **Project** class will also have the methods and fields of other class diagrams, similar to the User class above. Here, the Project class implements the ReceivesFeedback interface so that users may leave comments/reviews on projects, as well as vote up or down on projects that they come across. Projects are displayed in the ProjectBroswer class and worked on by users.
- The **ProjectBrowser** class contains a search-able list of zero or more projects tailored to a user's search. Users browse the list in order to find projects of interest.

### 2.3.13 Class Diagram 7: Project User Management

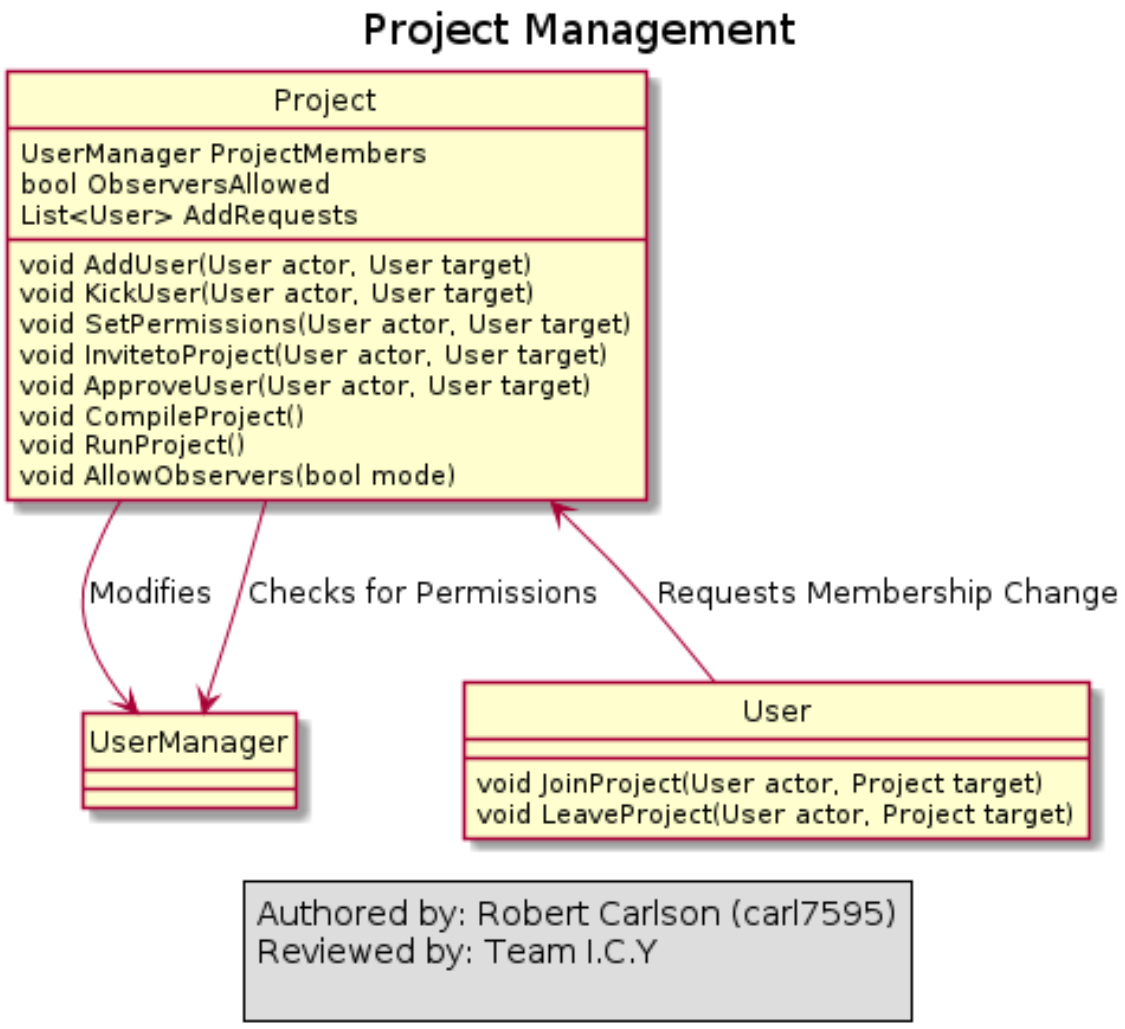


#### 2.3.14 Class Diagram Description 7: Project User Management Description

**Classes:**

- The **User** class contains the profiles of everyone who uses sQuire, and identifies anyone who tries to access a board.
- The **Project** class contains all of the information pertinent to an individual project, including one UserManager, which the client uses to control what kind of access each user has to that specific project.
- The **UserManager** class is referenced to check if a user has permission to read, modify, or run a project, or invite, ban, or change the permissions of another user. It does this by updating and checking against a HashMap of Permissions idexed by User. It records the User profile of the project creator to prevent the creator being demoted by another admin. It also contains a set of Permissions to use by default, before users are manually added to the project. The functions AddUser, KickUser, and SetPerms all modify the permissions HashMap after checking against it to make sure the active user has the authority to change the permissions of the target user.
- The **HashMap** class, in this case, functions as a permissions lookup table. It's indexed by User, and for each User in it there's one set of Permissions that it returns.
- The **Permissions** class is a set of bools that store whether each User has permission (within the instance's parent project) to read, write, execute the project, invite users, and/or modify the permissions of other users regarding the project.

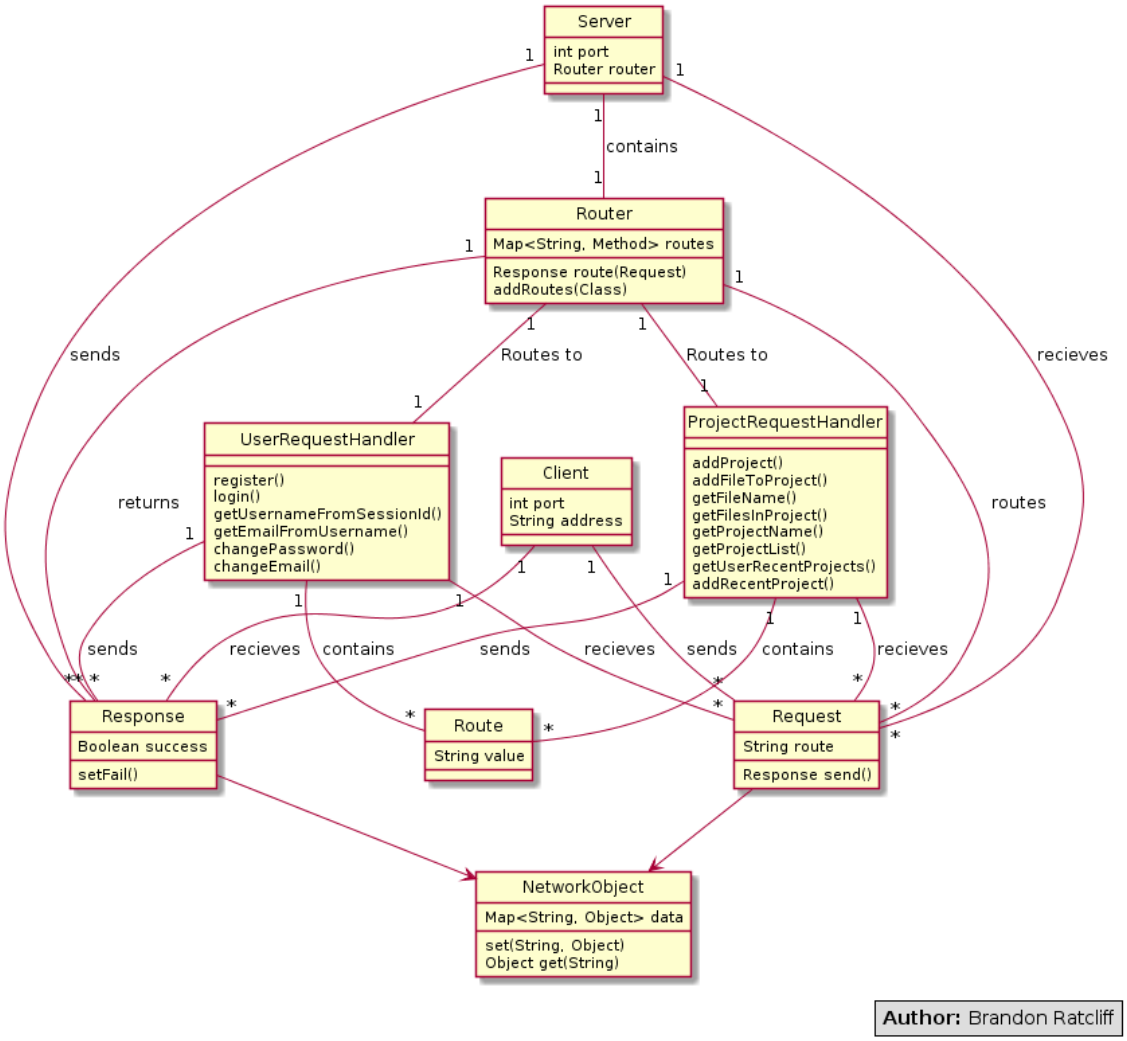
### 2.3.15 Class Diagram 8: Project Management



**Classes:**

- The **User** class contains the profiles of everyone who uses sQuire, and identifies anyone who tries to access a board.
- The **Project** class contains all of the information pertinent to an individual project, including one UserManager, which the client uses to control what kind of access each user has to that specific project.
- The **UserManager** class is referenced to check if a user has permission to read, modify, or run a project, or invite, ban, or change the permissions of another user. It does this by updating and checking against a HashMap of Permissions indexed by User. It records the User profile of the project creator to prevent the creator being demoted by another admin. It also contains a set of Permissions to use by default, before users are manually added to the project. The functions AddUser, KickUser, and SetPerms all modify the permissions HashMap after checking against it to make sure the active user has the authority to change the permissions of the target user.
- The **HashMap** class, in this case, functions as a permissions lookup table. It's indexed by User, and for each User in it there's one set of Permissions that it returns.
- The **Permissions** class is a set of bools that store whether each User has permission (within the instance's parent project) to read, write, execute the project, invite users, and/or modify the permissions of other users regarding the project.

### 2.3.16 Class Diagram 9: Networking



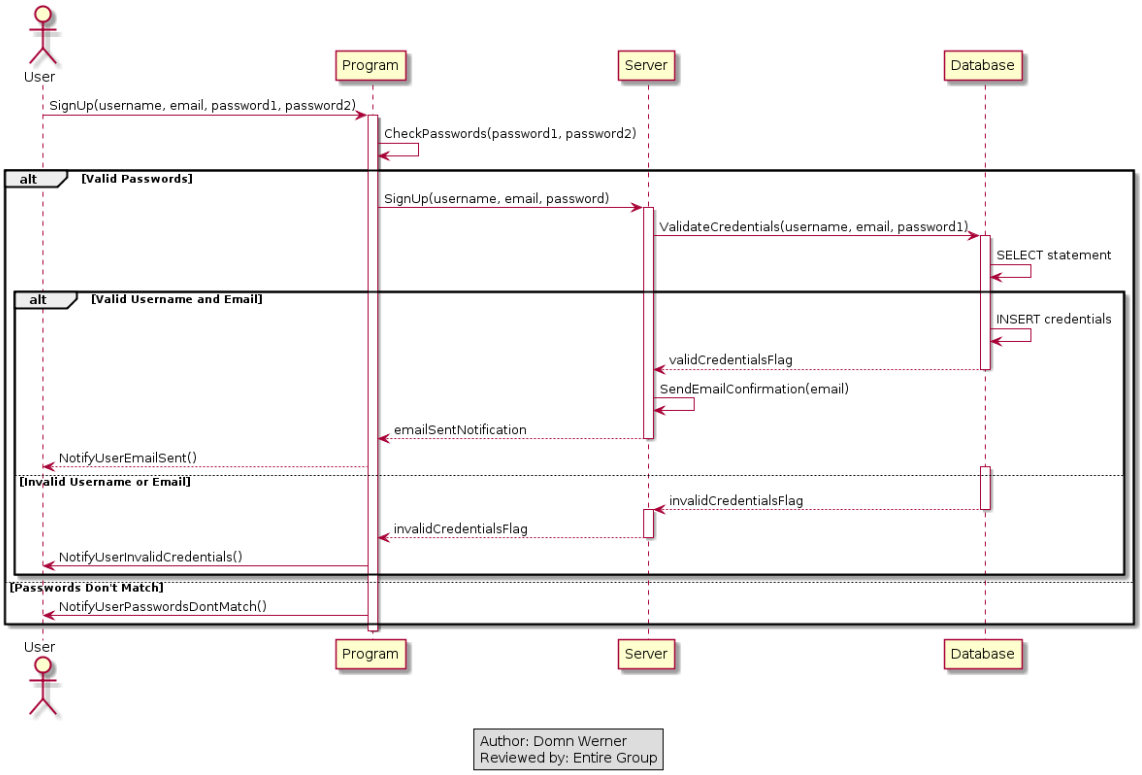

### 2.3.17 Class Diagram Description 9: Networking

- The **Server** class waits for socket connections containing a **Request** object. The server then passes this **Request** object to the **Router**, which returns **Response** object, and then the server then returns this.
- The **Client** class connects to the server, and sends **Request** objects, and waits for a **Response object to be returned**.
- The **NetworkObject** class implements a serilizable object that is designed to be passed between the client and the server. It contains a data field containing key value pairs with all the information to be either sent or returned.
- The **Request** class inherits from **NetworkObject**. It adds a string called Route. This is used by the **Router** class, and any RequestHandler classes to match up with a **Route** to determine what function to run. It also contains a send function, which is just a shortcut to send the request to the server.

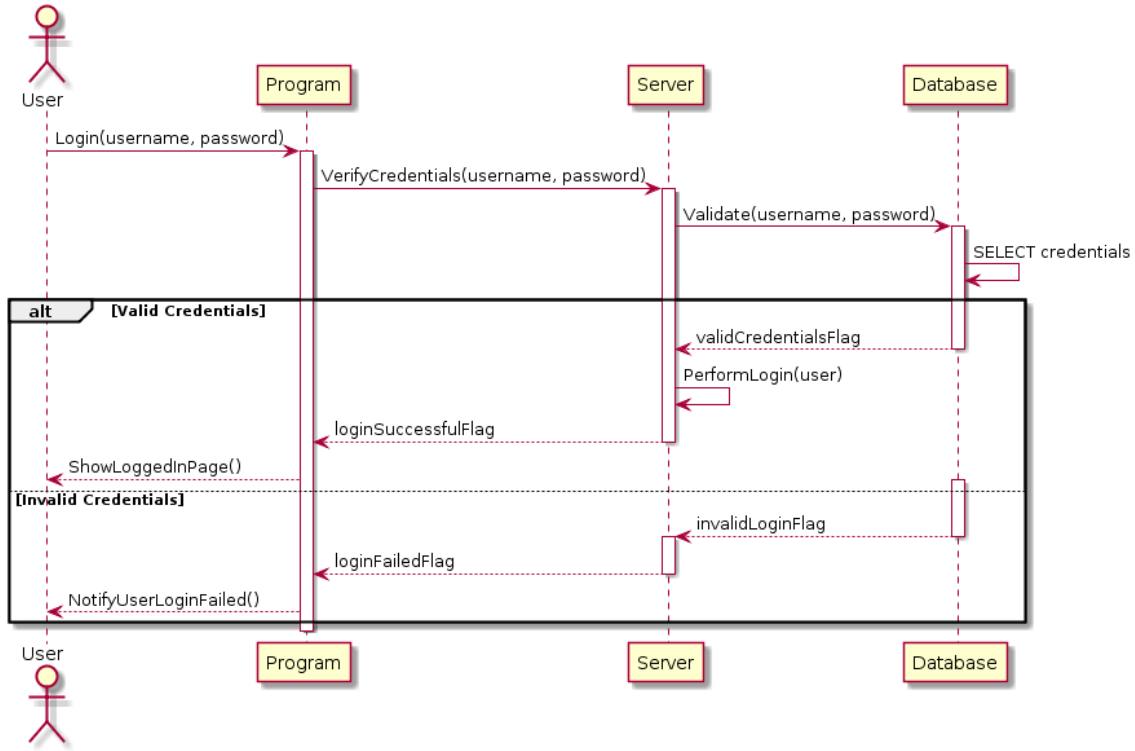
- the **Response** class inherits from the **NetworkObject** class. Additionally, it contains a success value, and a setFail() method used to signify if the particular request succeed, or failed.
- The **Router** class is initialized with RequestHandler classes. These RequestHandler classes must contain functions that each have a **Route** annotation. The Router keeps a map with all the possible routes, and the associated function. The route method takes a Request, looks up the route specified by the Request, and then passes it to the appropriate function in one of the RequestHandlers. It then returns a **Response** object generated by the RequestHandlers.
- The Route annotation is applied to either RequestHandler classes, or functions in the RequestHandler classes. If it is applied to the class, then any routes in side the function have the route of the class prefixed to them.
- The **UserRequestHandler** and the **ProjectRequestHandler** classes are RequestHandlers that deal with request related to Users and Projects respectively. They contain functions annotated with **Routes** to be used by the **Router**. Each of these functions accepts a **Request** and a **Response**. The Request is the object that is send by the client, and the Response object can be modified, and it will be returned to the Client.

## 2.4 SEQUENCE DIAGRAMS

### 2.4.1 Authentication Feature 1: Sign Up Sequence Diagram

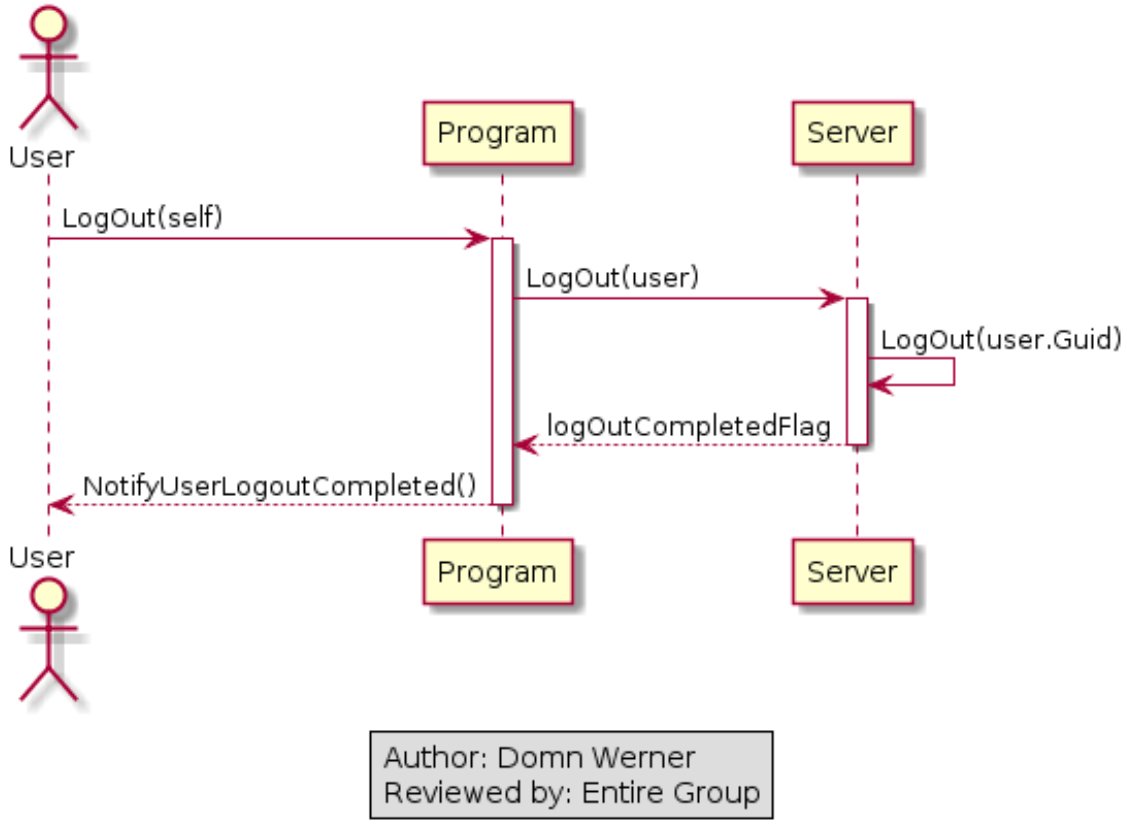


### 2.4.2 Authentication Feature 2: Log In Sequence Diagram

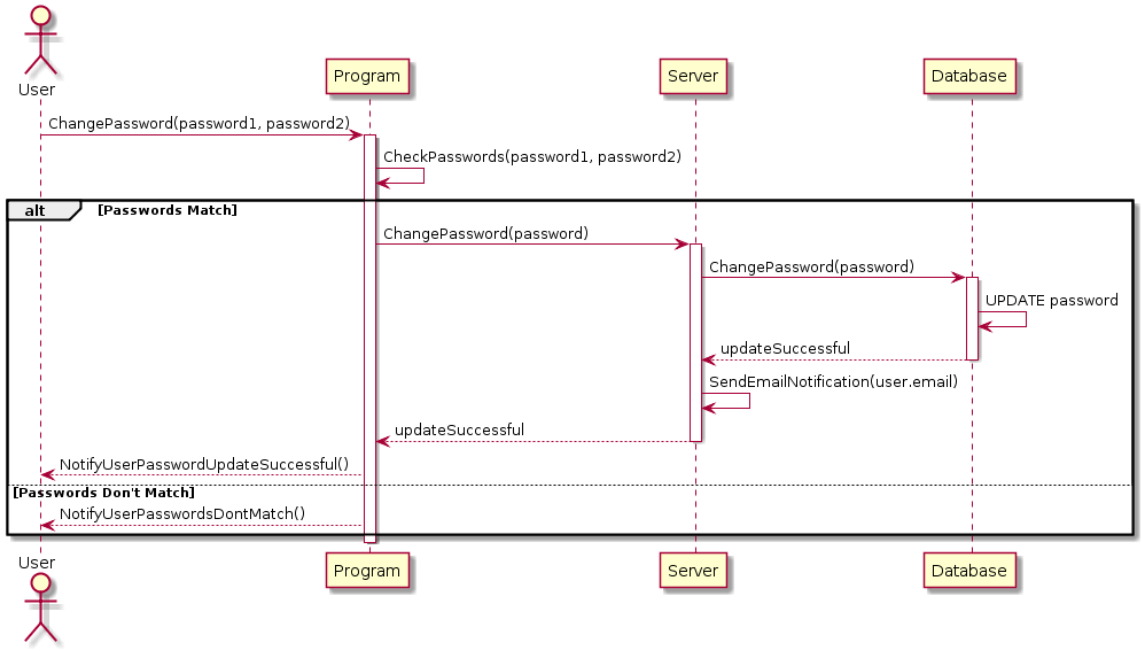



Author: Domn Werner
Reviewed by: Entire Group

### 2.4.3 Authentication Feature 3: Log Out Sequence Diagram

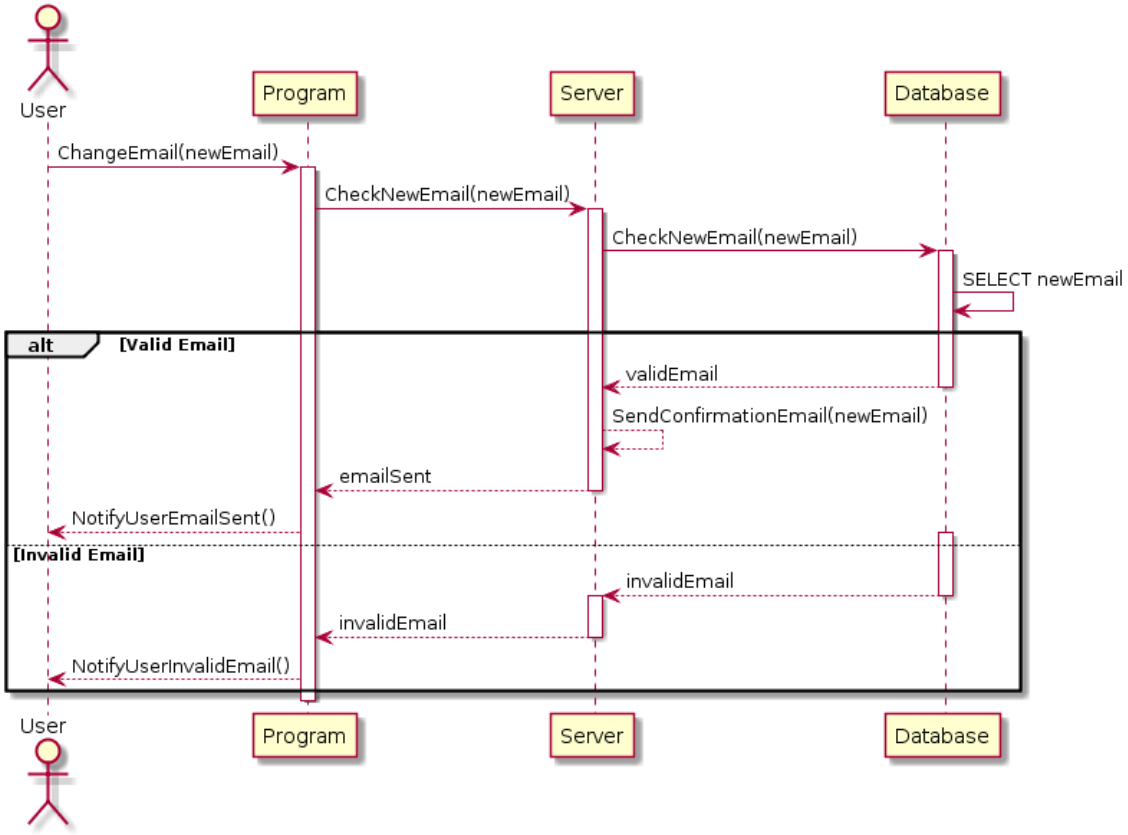


### 2.4.4 Authentication Feature 4: Change Password Sequence Diagram



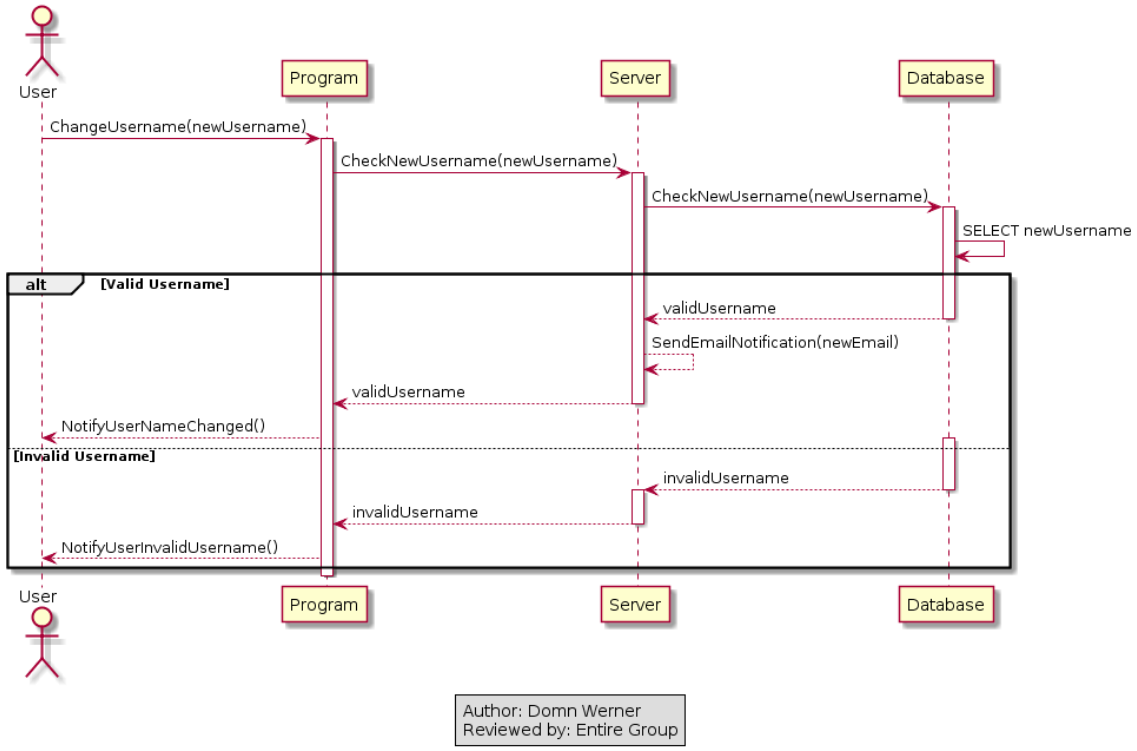

Author: Domn Werner
Reviewed by: Entire Group

### 2.4.5 Authentication Feature 5: Change Email Sequence Diagram

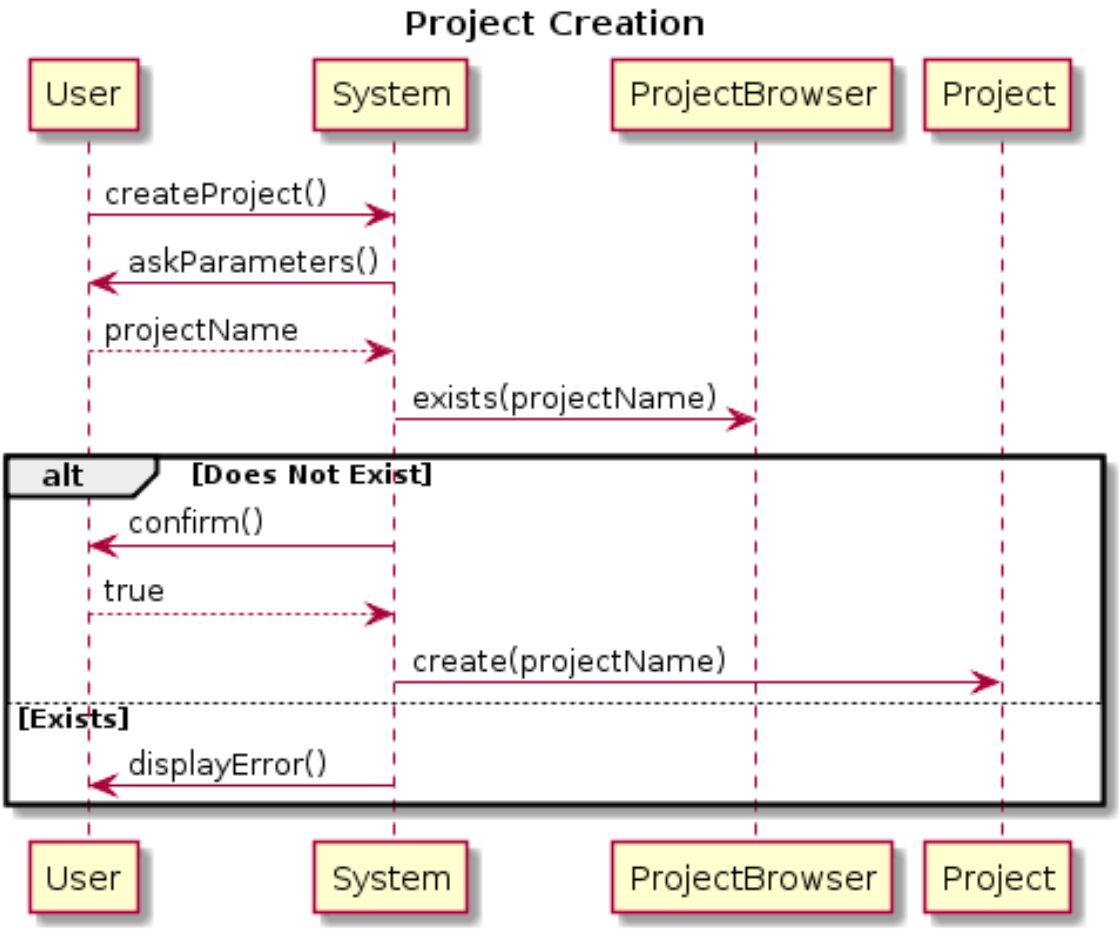



Author: Domn Werner
Reviewed by: Entire Group

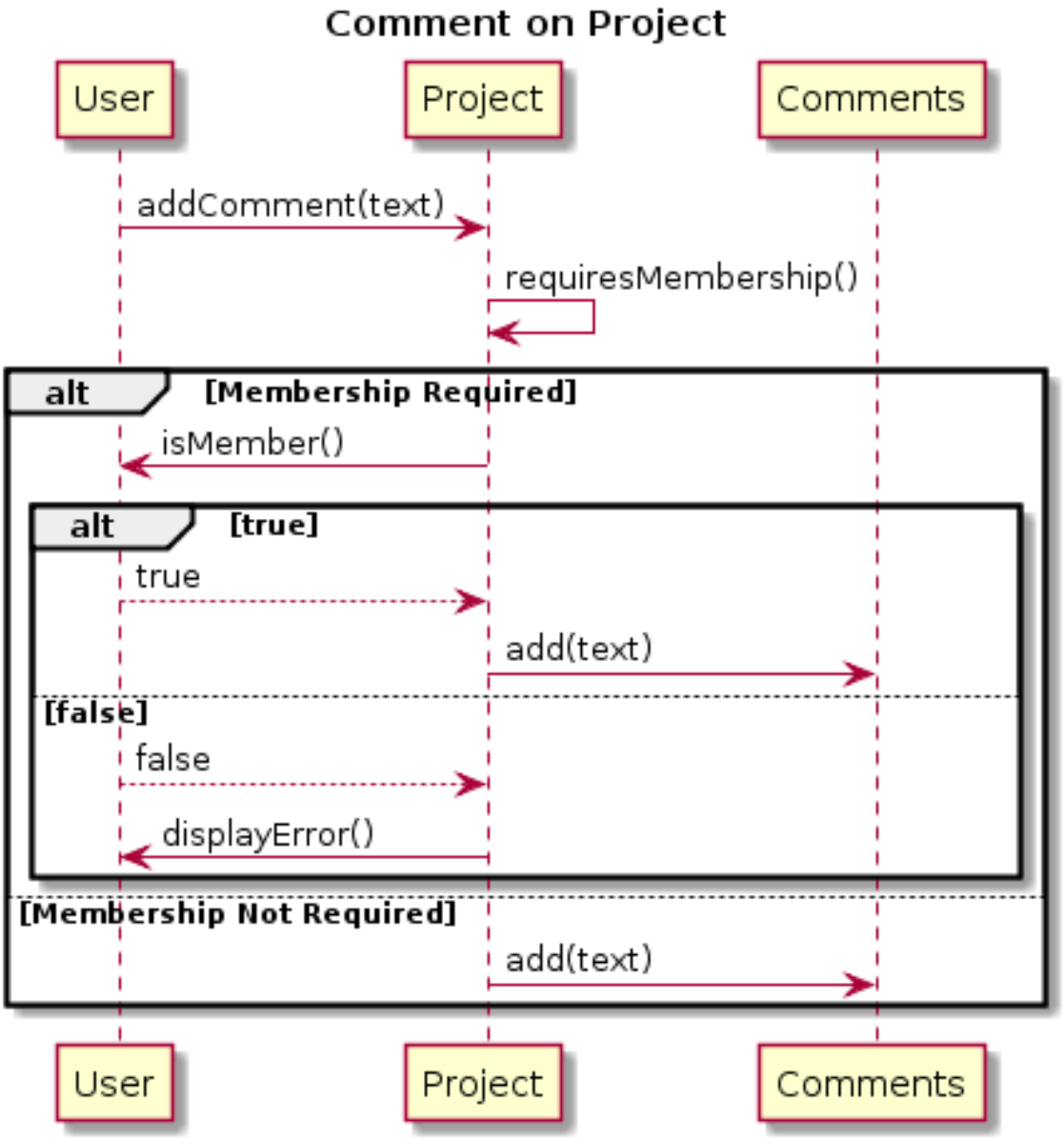
### 2.4.6 Authentication Feature 6: Change Username Sequence Diagram



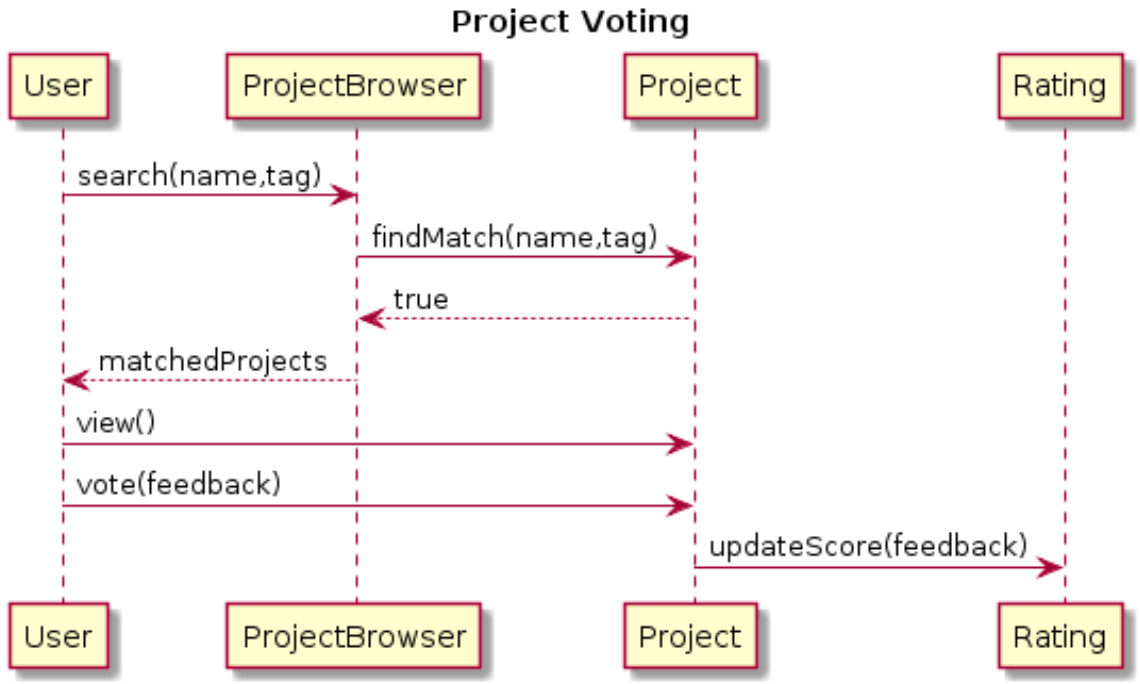
### 2.4.7 Project Browsing Feature 2: Project Creation Sequence Diagram



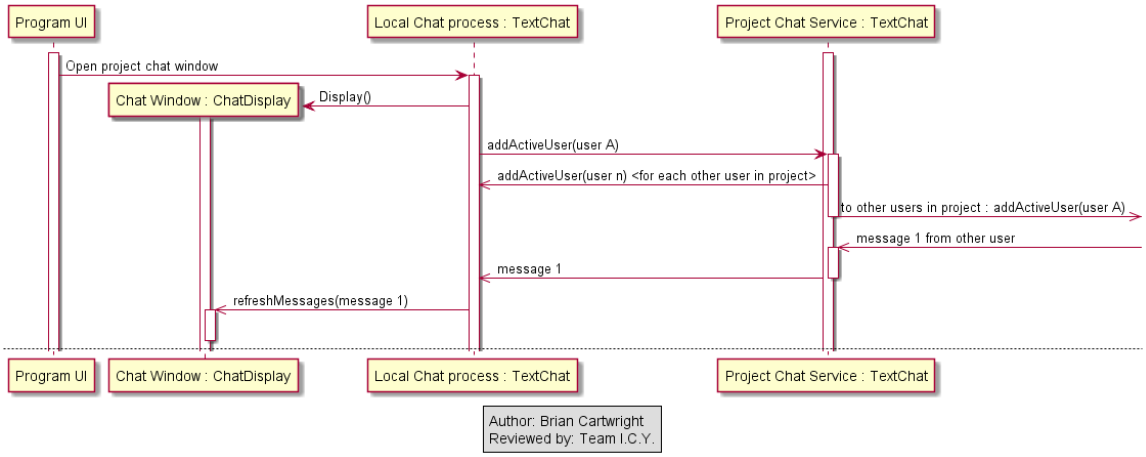
### 2.4.8 Project Browsing Feature 3: Project Commenting Sequence Diagram



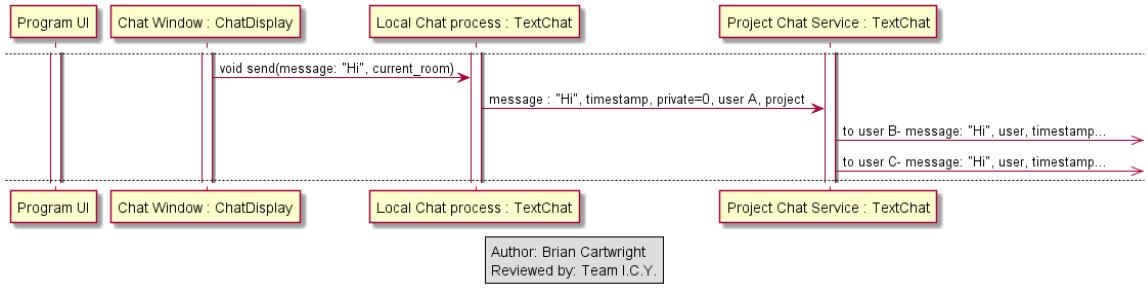
### 2.4.9 Project Browsing Feature 4: Project Voting Sequence Diagram



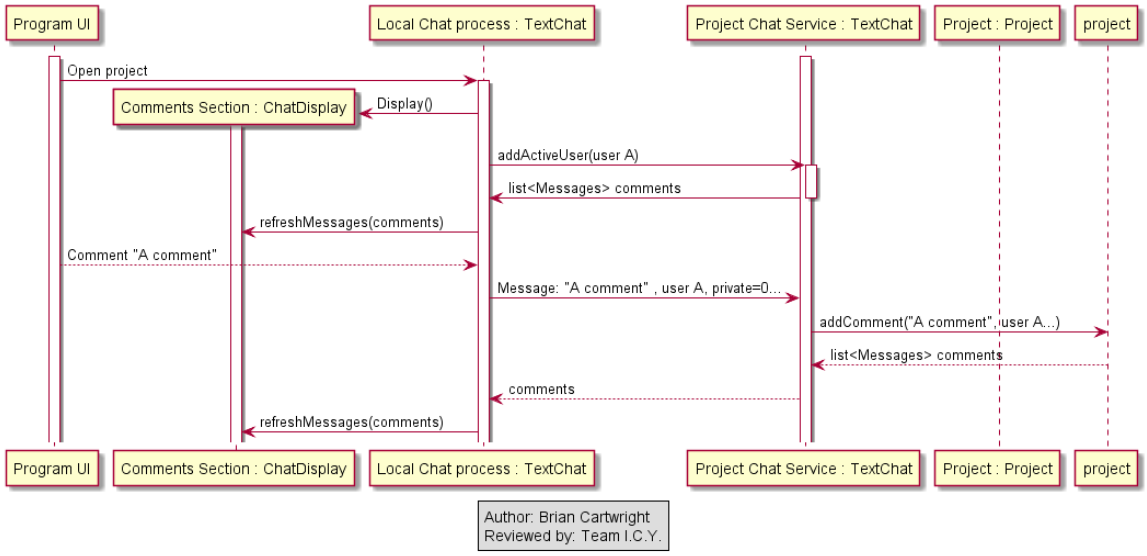
### 2.4.10 Communication Feature 1: Read Project Chat Sequence Diagram



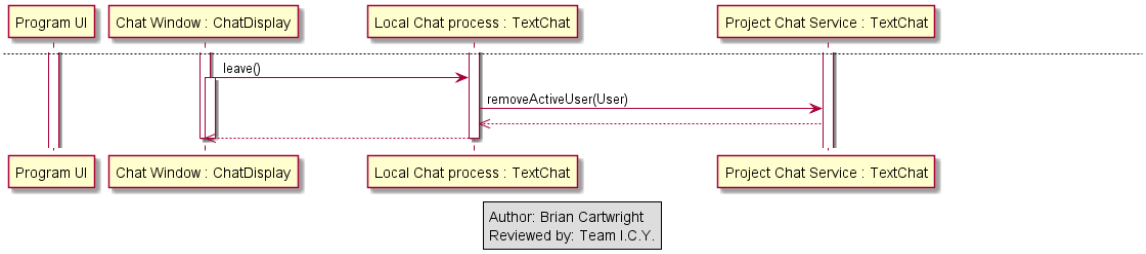
### 2.4.11 Feature 2: Write to Project Chat Sequence Diagram



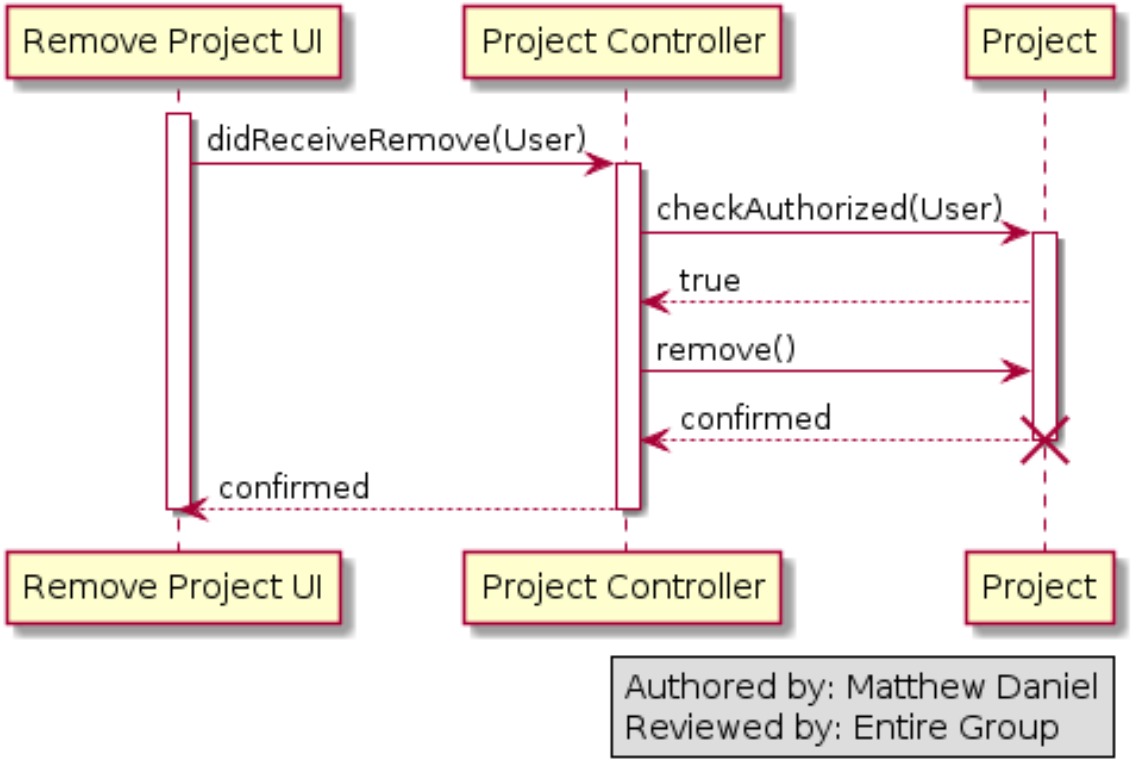
### 2.4.12 Feature 4: Comment on Project Sequence Diagram



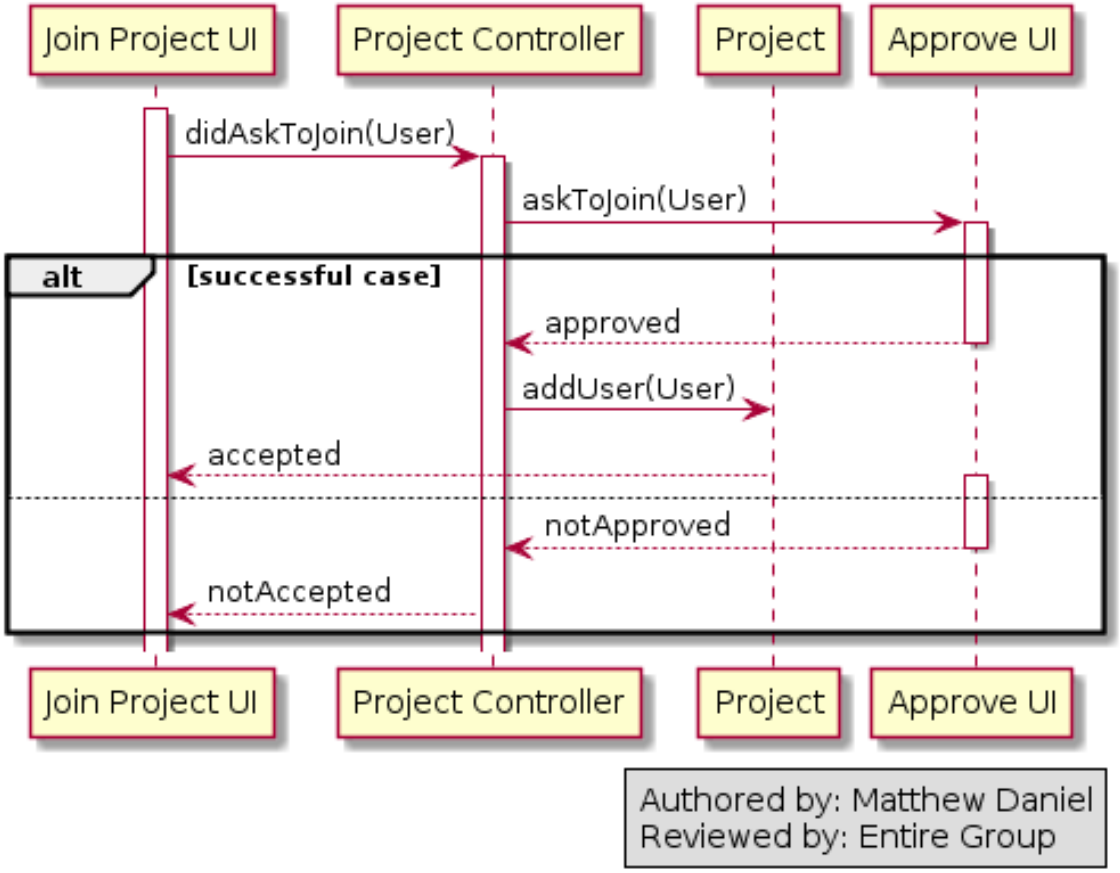
### 2.4.13 Feature 5: Close chat Sequence Diagram



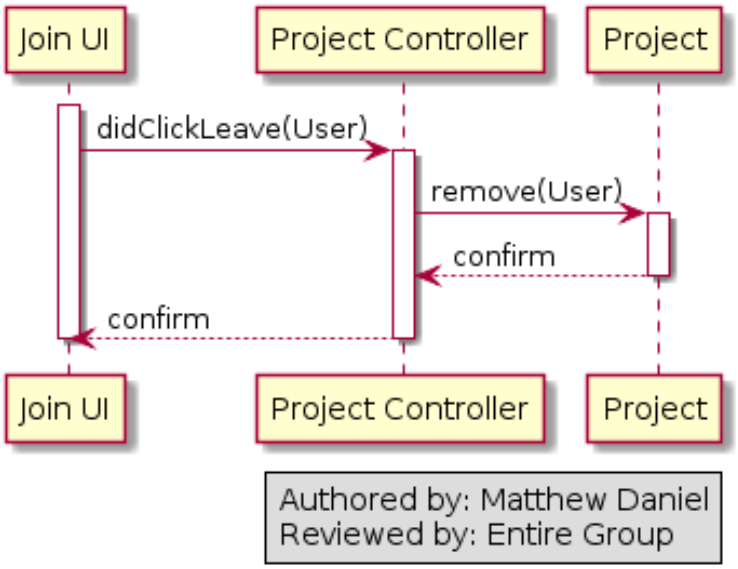
#### 2.4.14 Project Management Feature 3: Delete Project Sequence Diagram (dani2918)



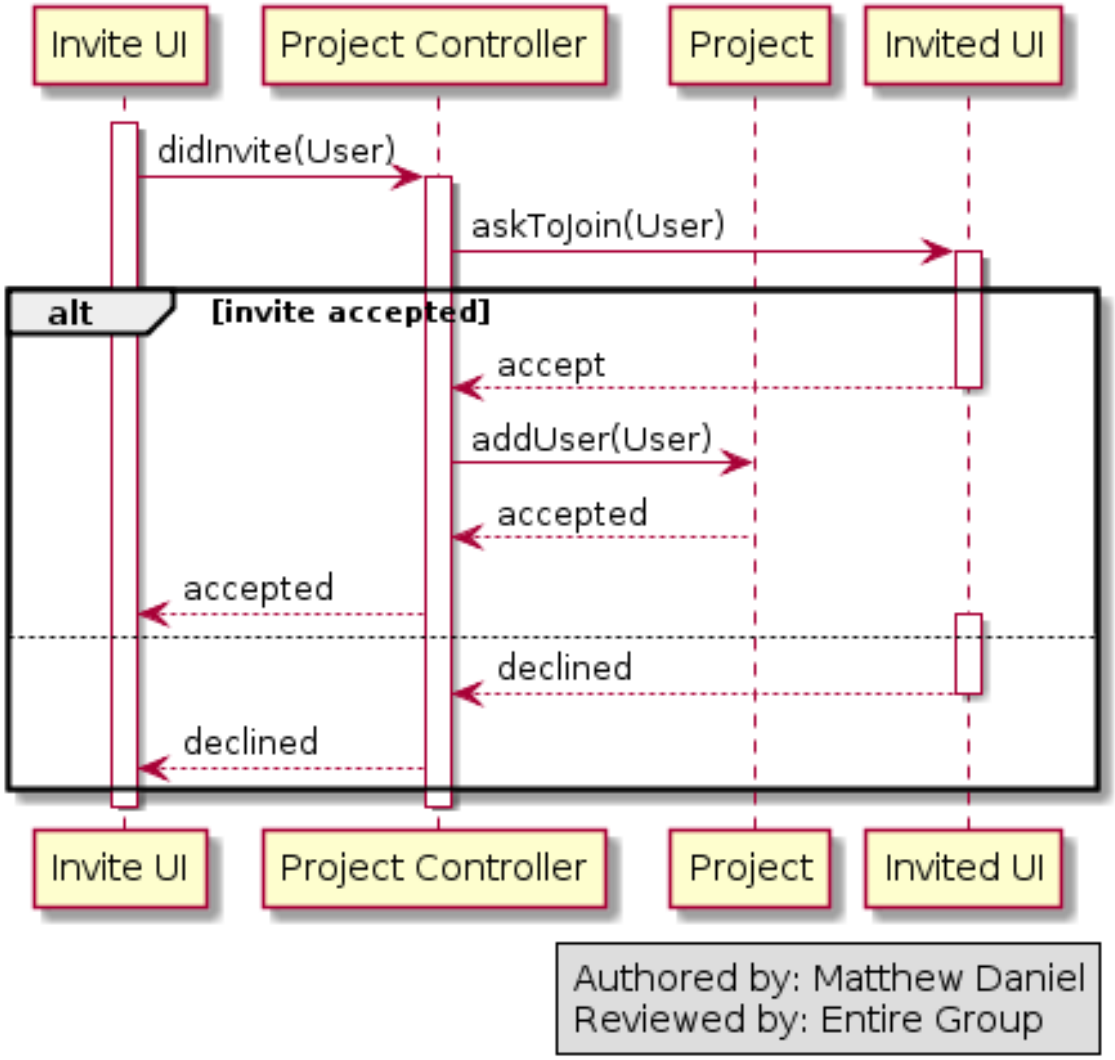
#### 2.4.15 Project Management Feature 4/5: Request/Manage Request to Join Project Sequence Diagram (dani2918)



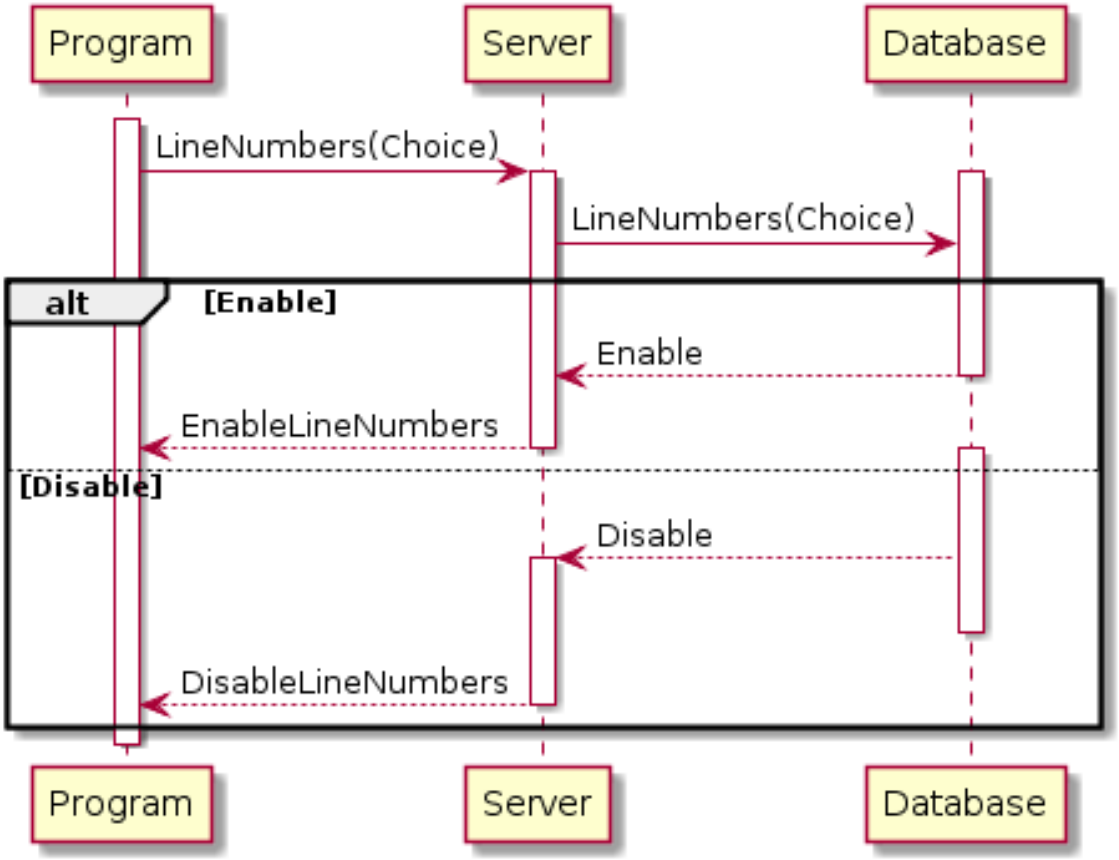
#### 2.4.16 Project Management Feature 6: Leave Project Sequence Diagram (dani2918)



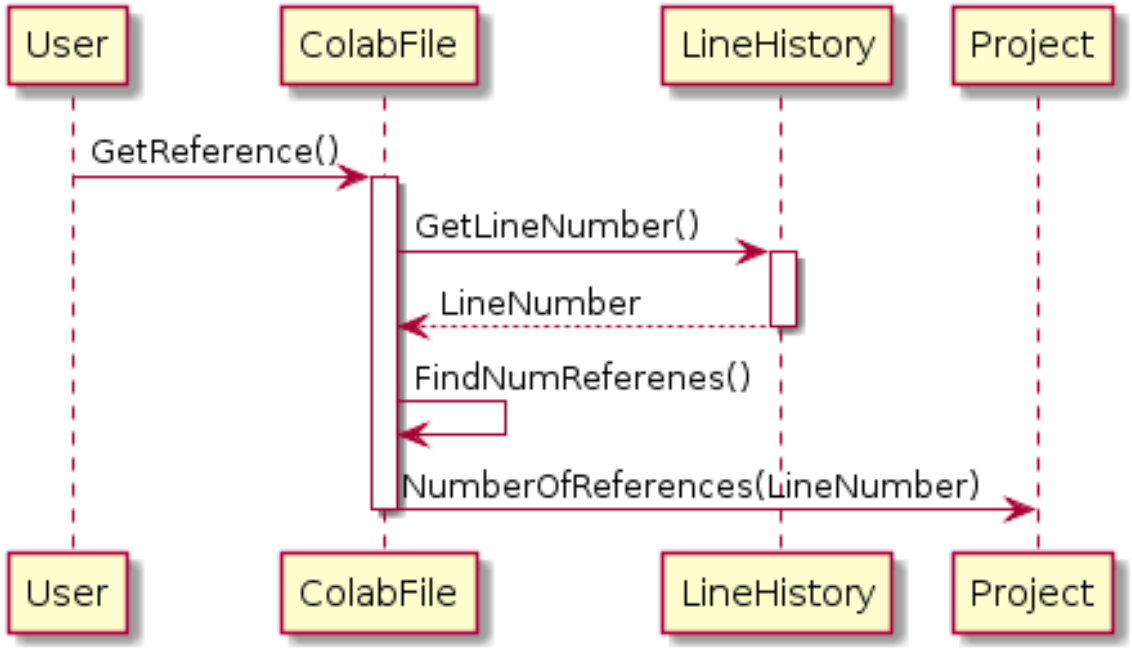
#### 2.4.17 Project Management Feature 7/8: Invite/Respond to Project Invite Sequence Diagram (dani2918)



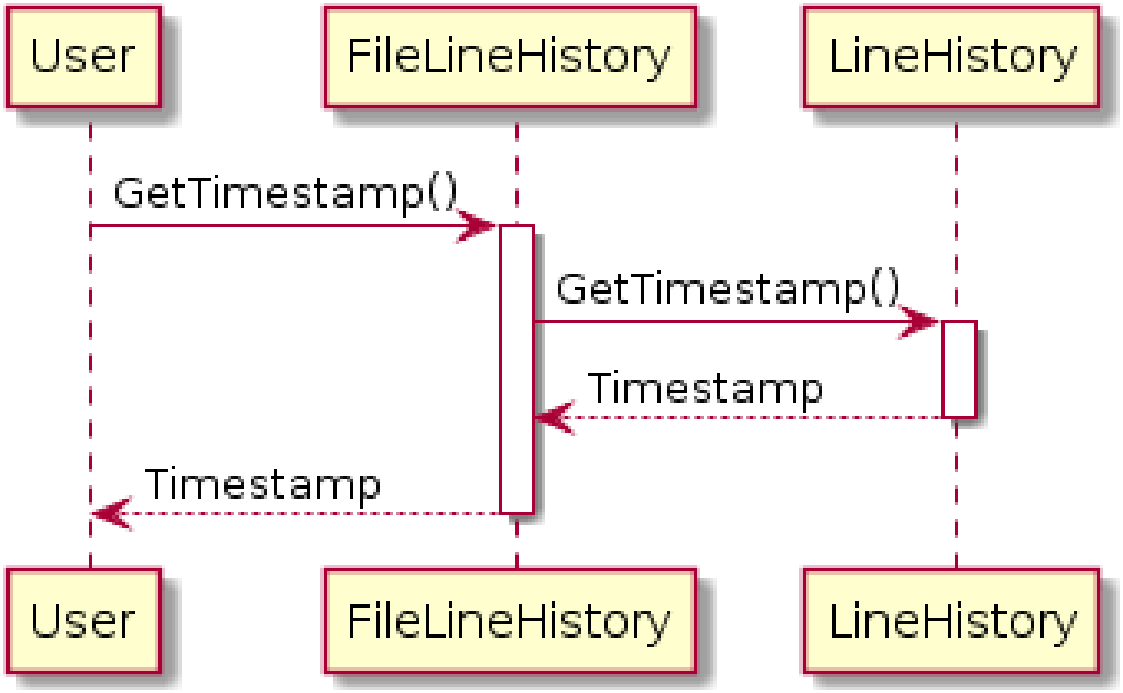
### 2.4.18 File Editing Feature 1: View Line Numbers Sequence Diagram



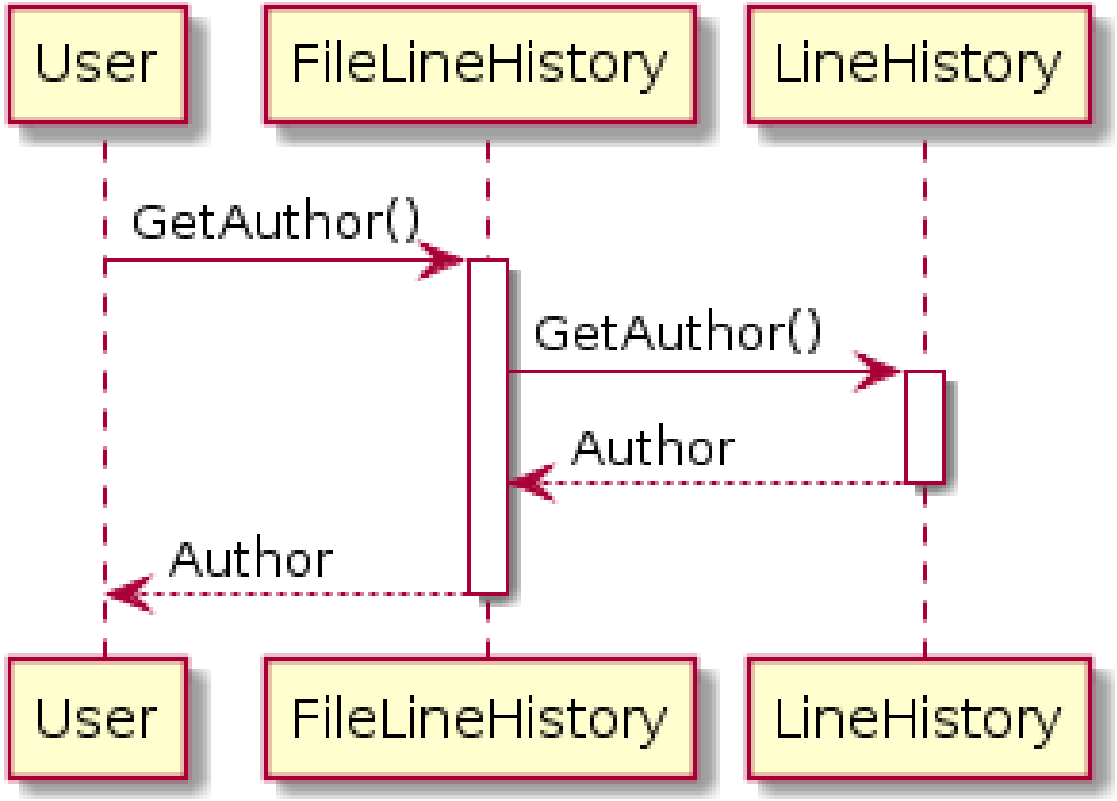
#### 2.4.19 File Editing Feature 2: View References Sequence Diagram



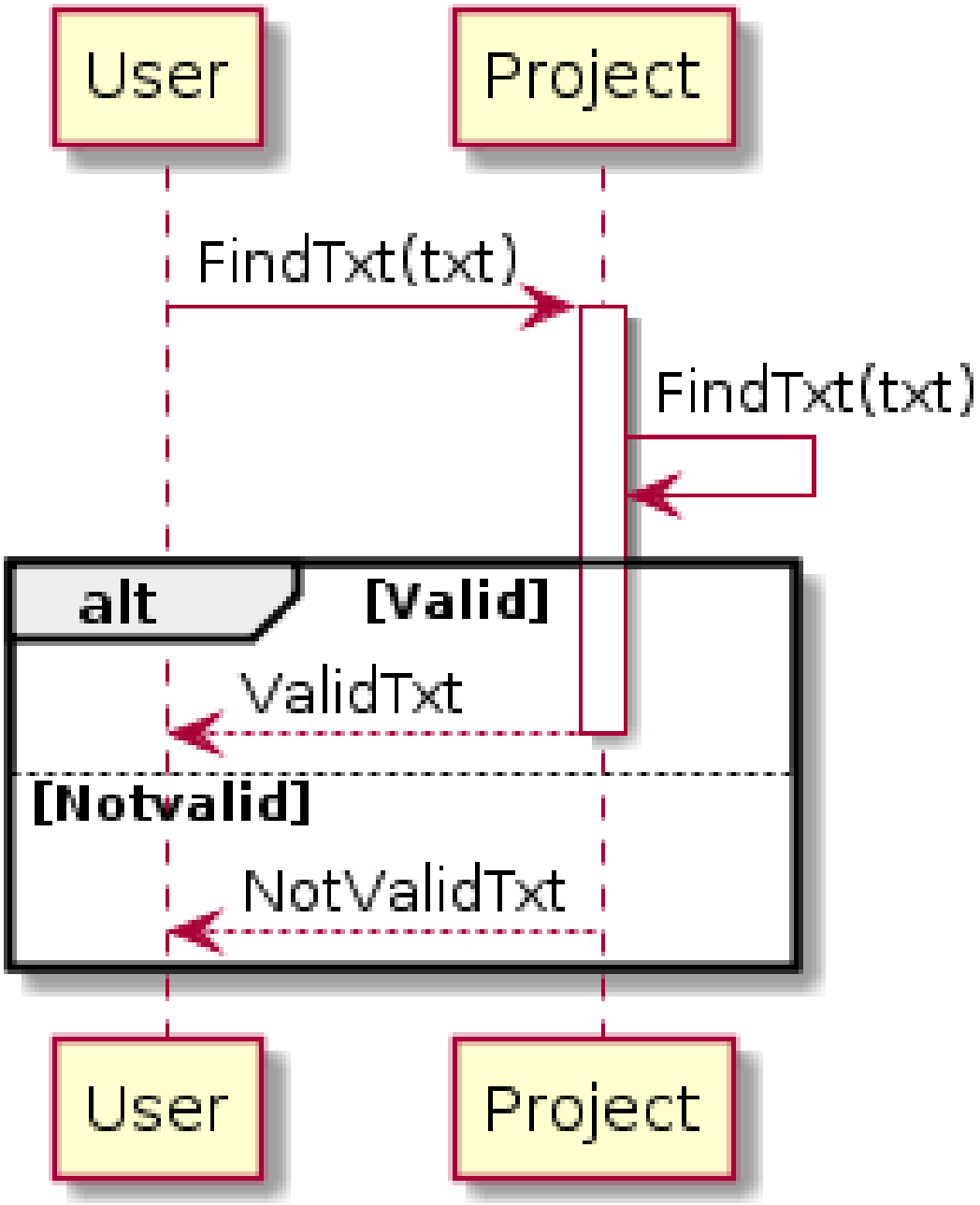
#### 2.4.20 File Editing Feature 3: View Dates Sequence Diagram



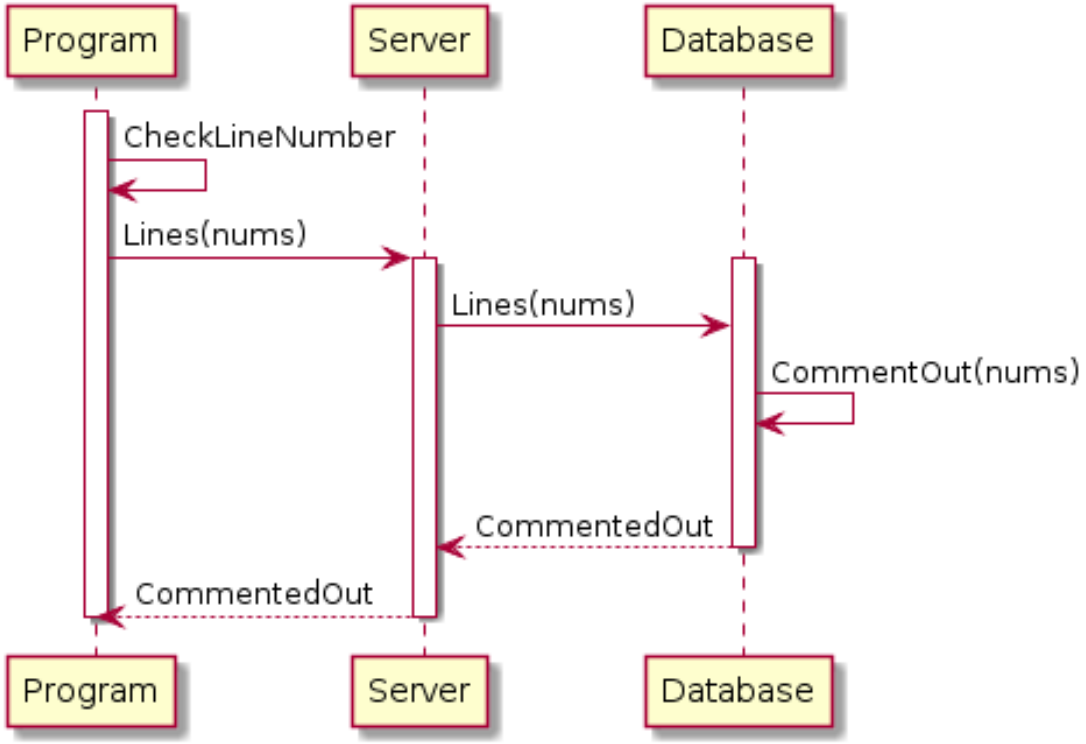
#### 2.4.21 File Editing Feature 4: View Author Sequence Diagram



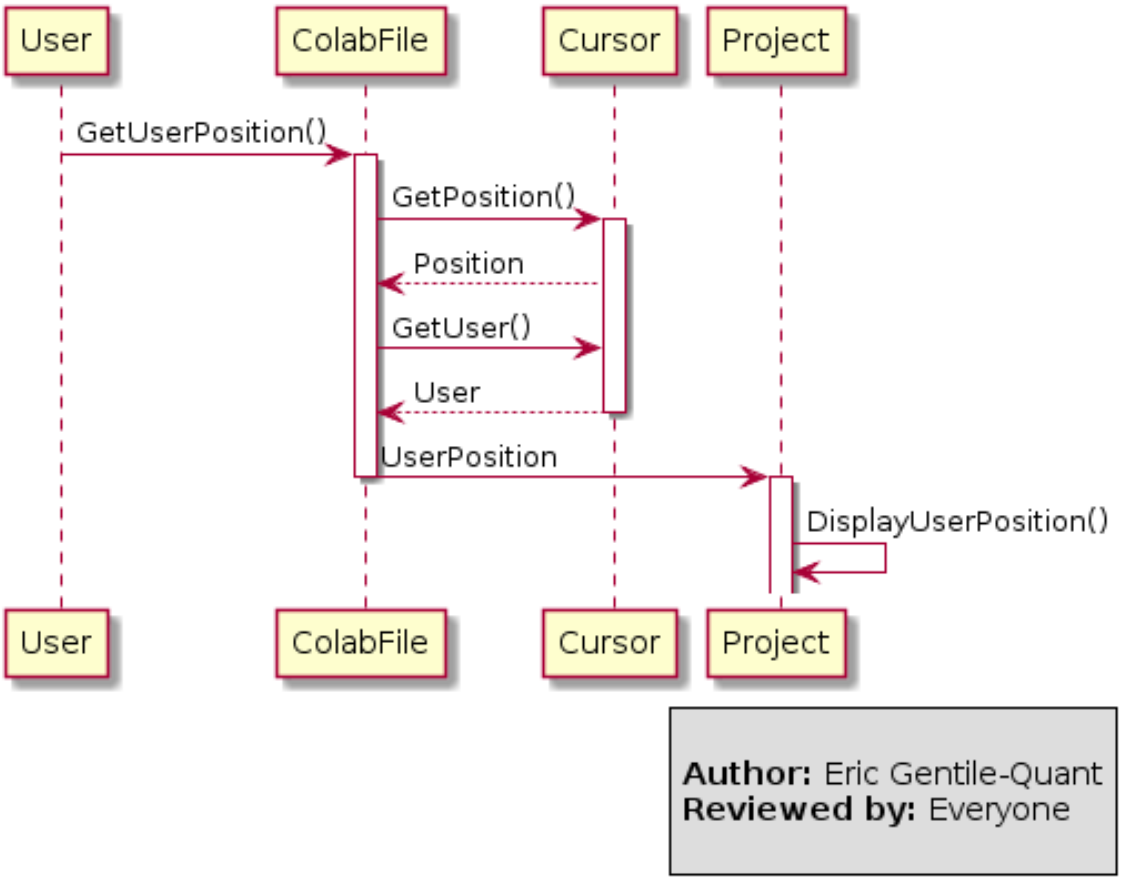
#### 2.4.22 File Editing Feature 6: Find Sequence Diagram



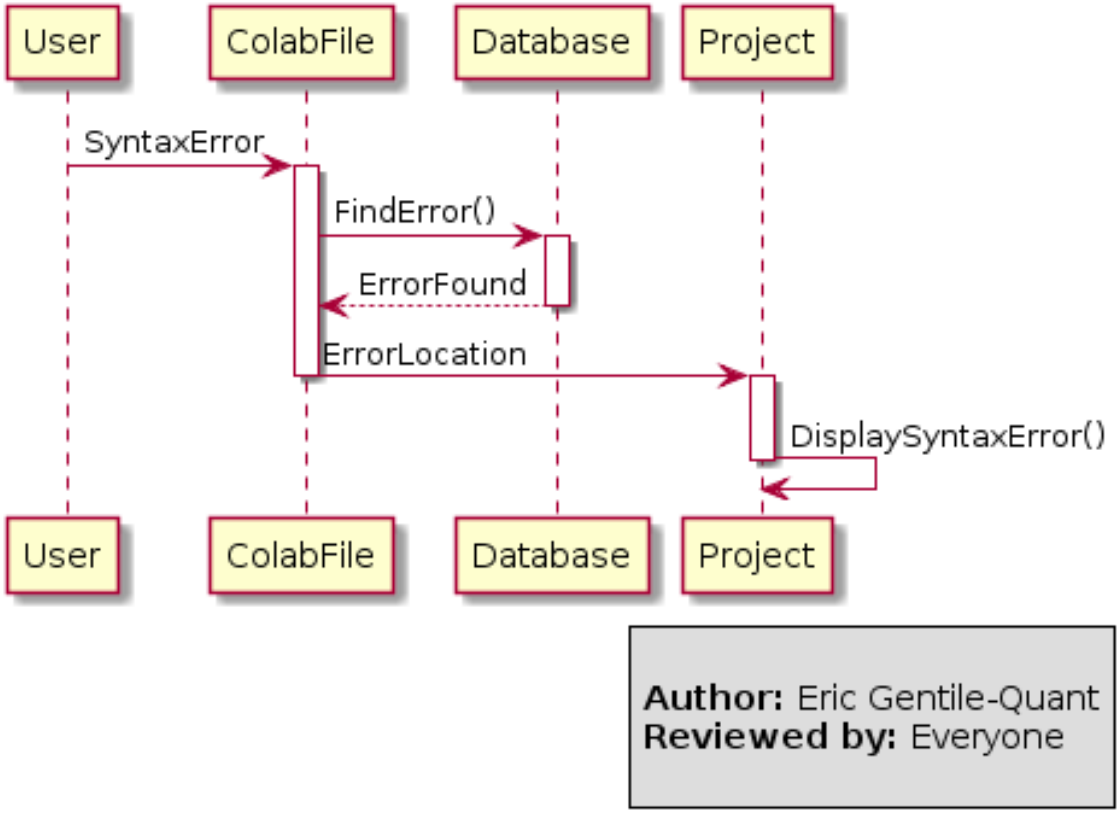
#### 2.4.23 File Editing Feature 7: Comment Section Sequence Diagram



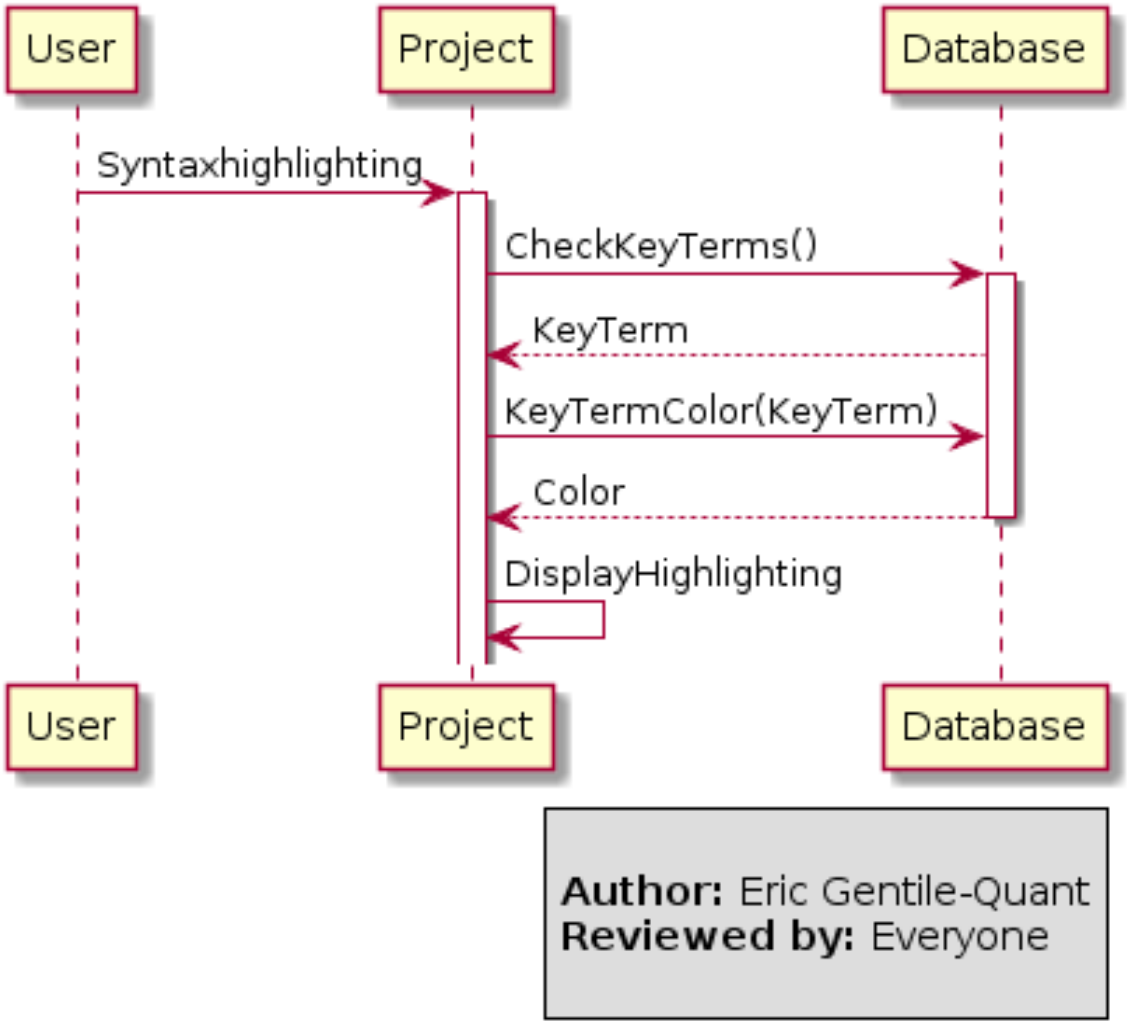
#### 2.4.24 File Editing Feature 8: Display Typing User Sequence Diagram



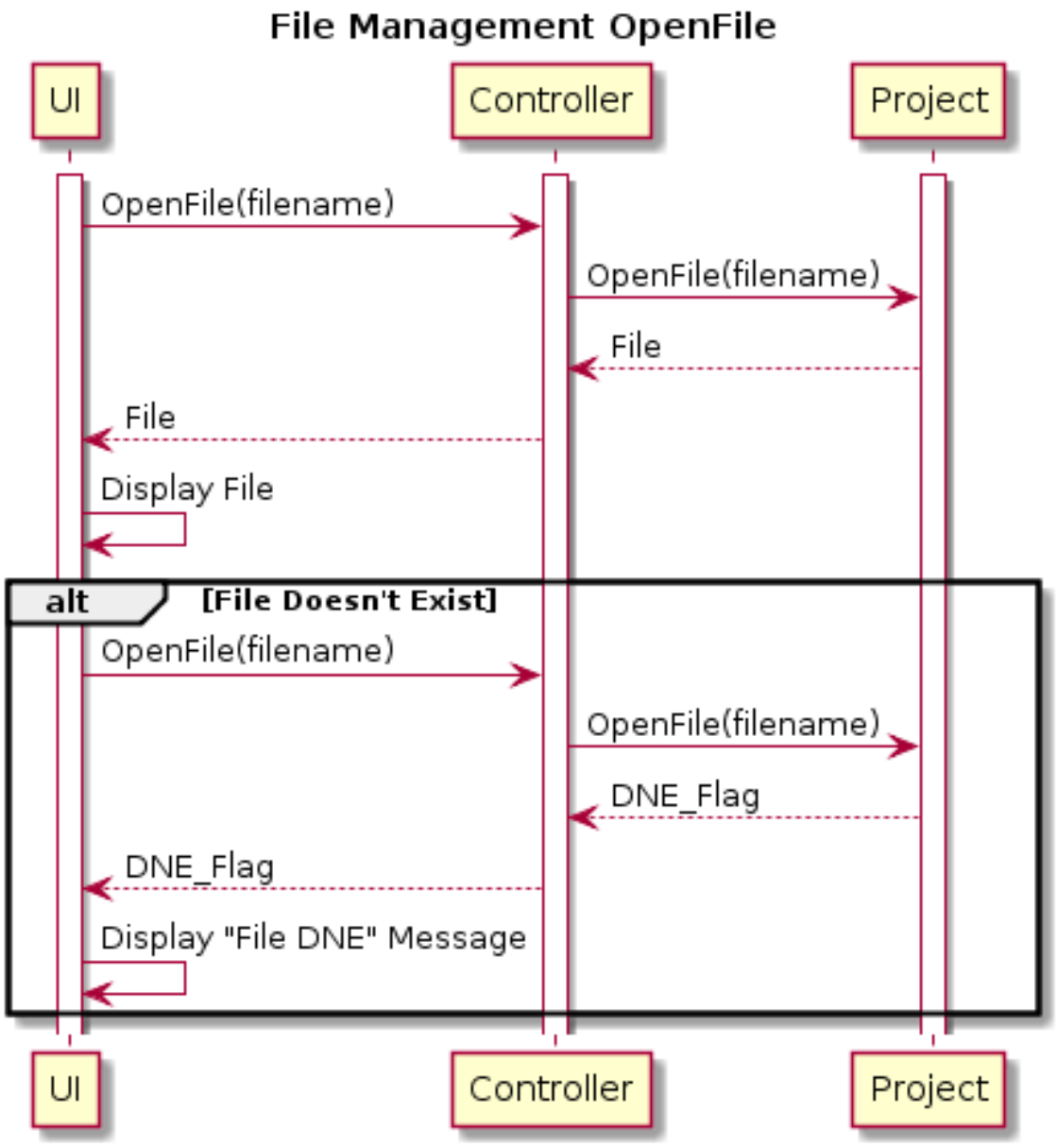
#### 2.4.25 File Editing Feature 9: Display Syntax Errors Sequence Diagram



#### 2.4.26 File Editing Feature 10: Display Syntax Highlighting Sequence Diagram

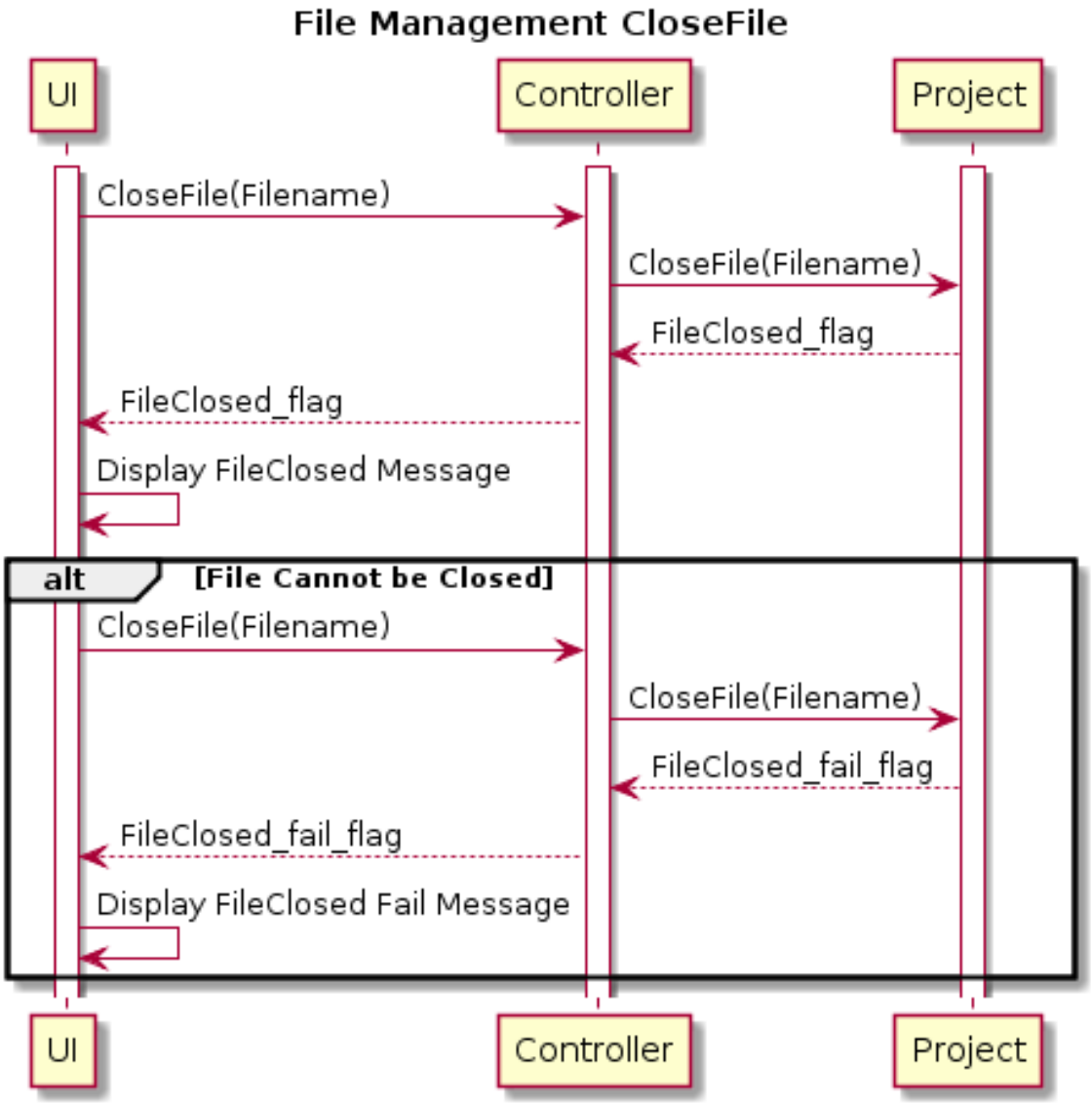


#### 2.4.27 File Management Feature 1: Open File Sequence Diagram



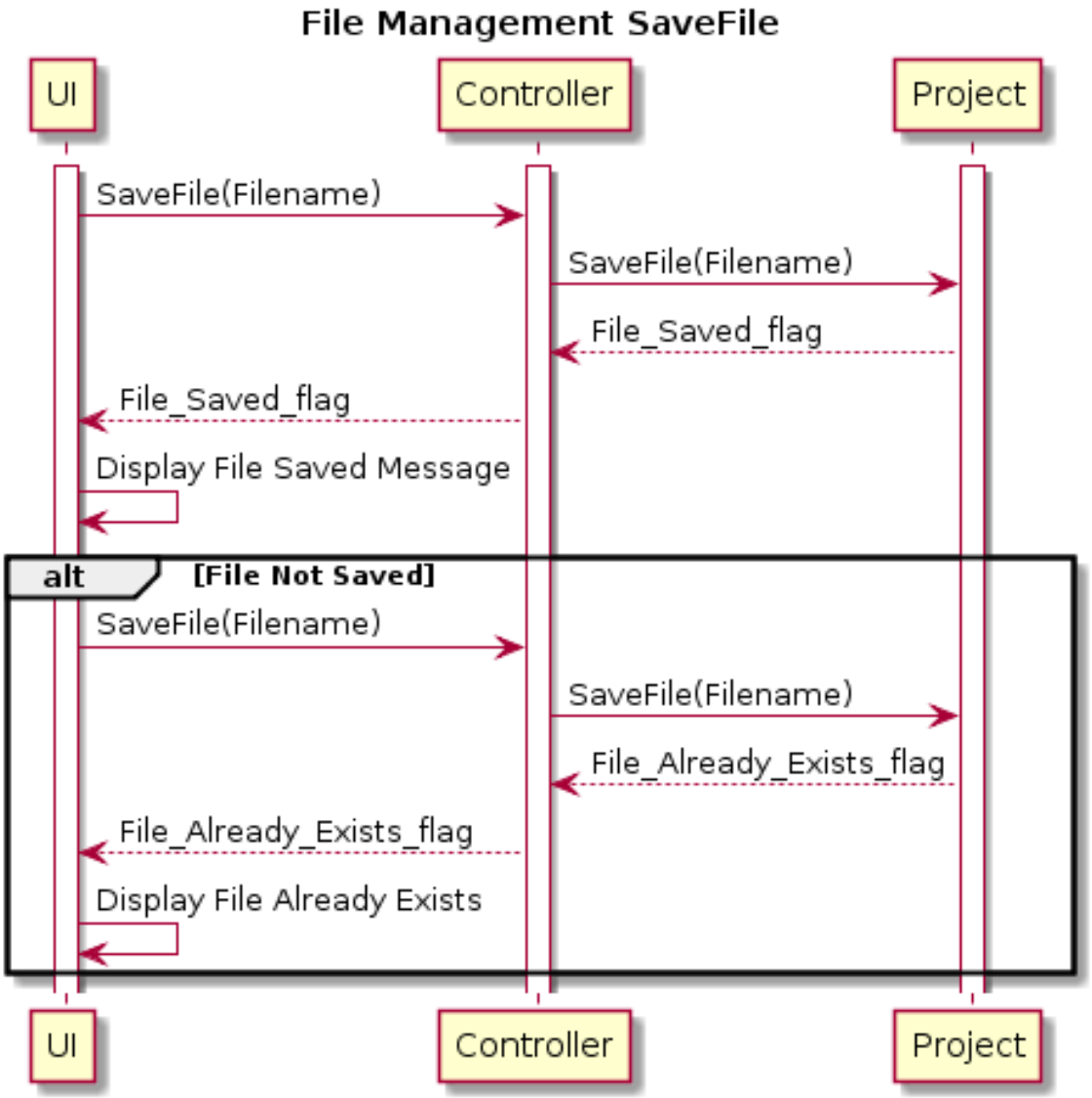

Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

#### 2.4.28 File Management Feature 2: Close File Sequence Diagram



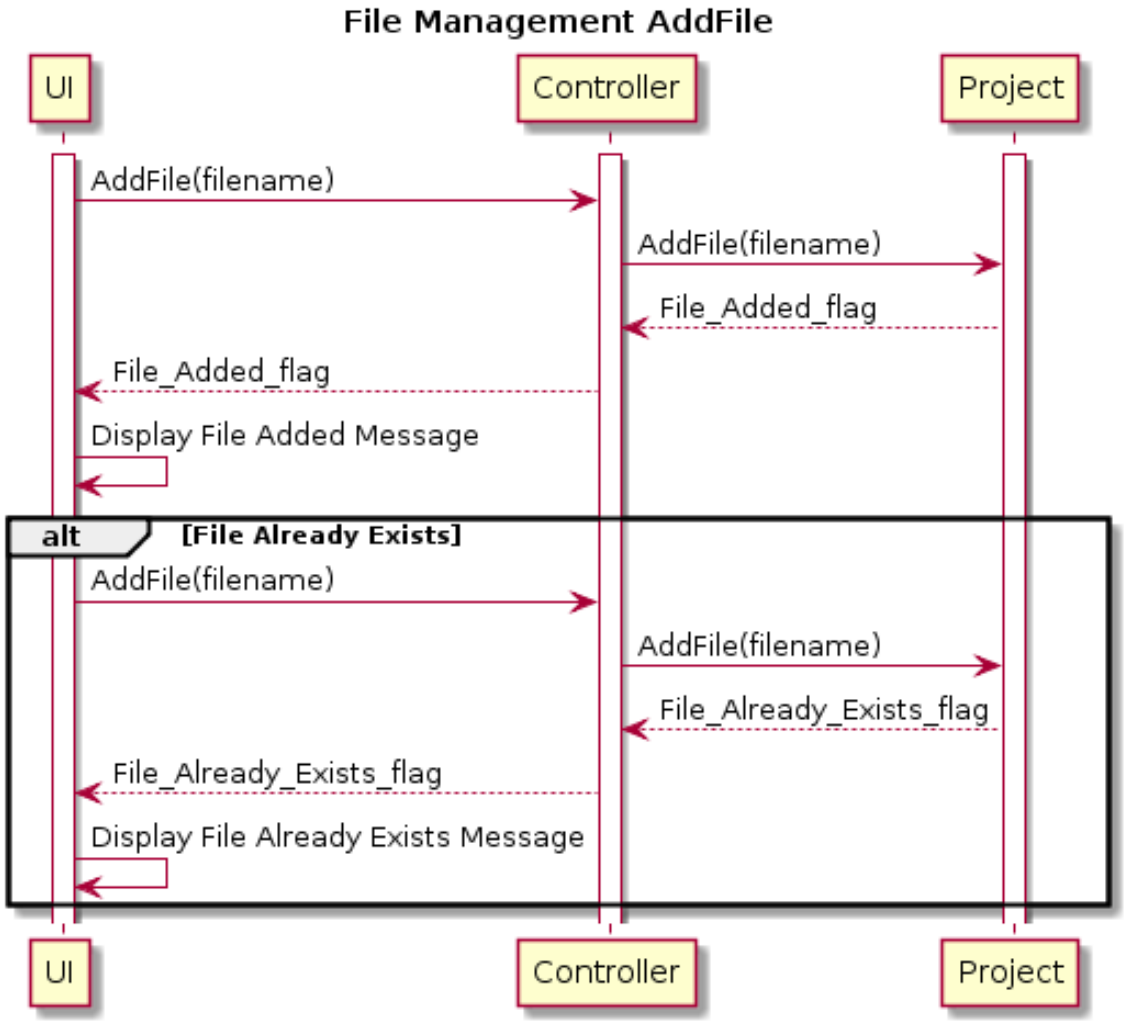

Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

#### 2.4.29 File Management Feature 3: Save File Sequence Diagram



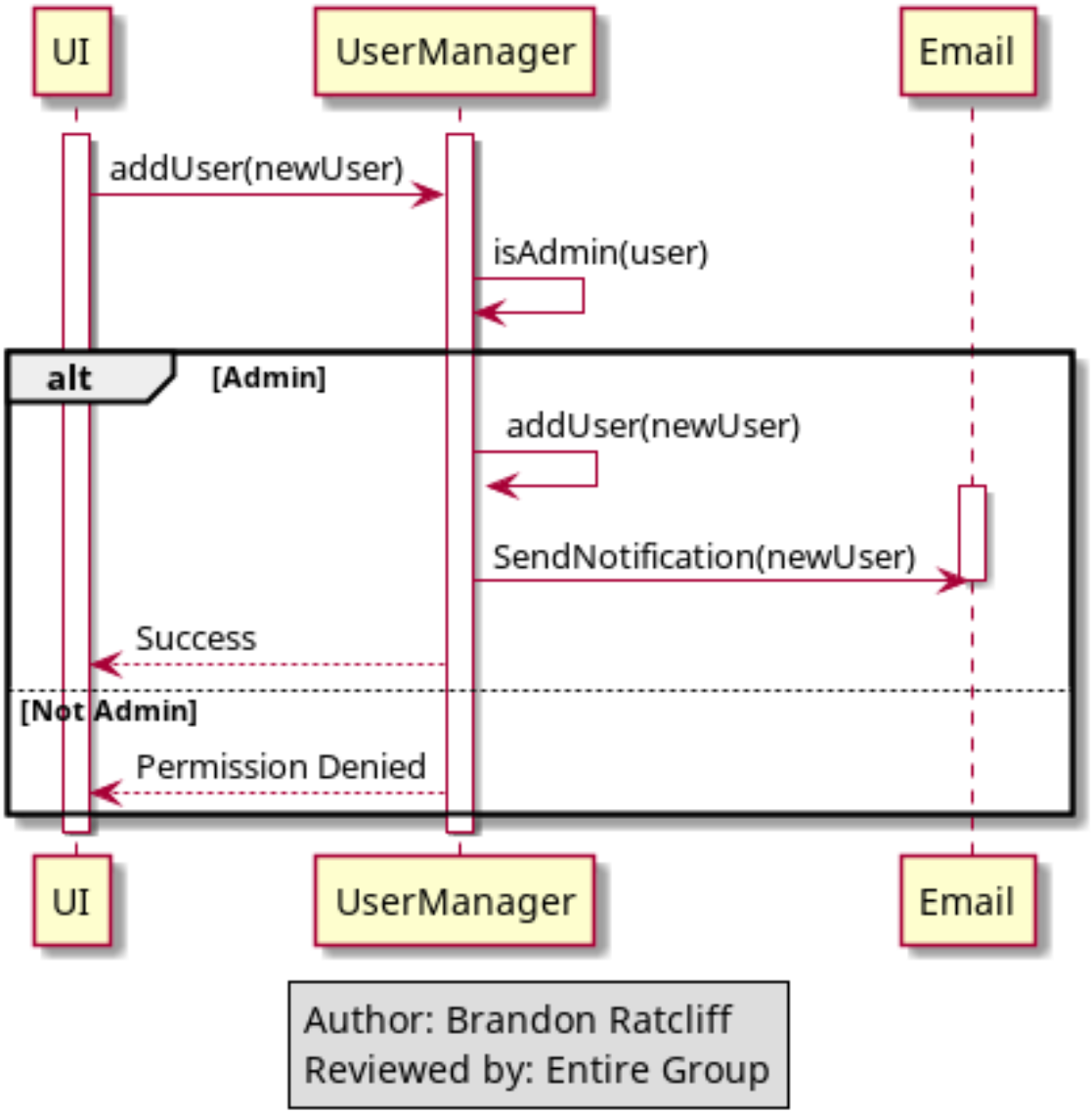

Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

### 2.4.30 File Management Feature 4: Add File Sequence Diagram

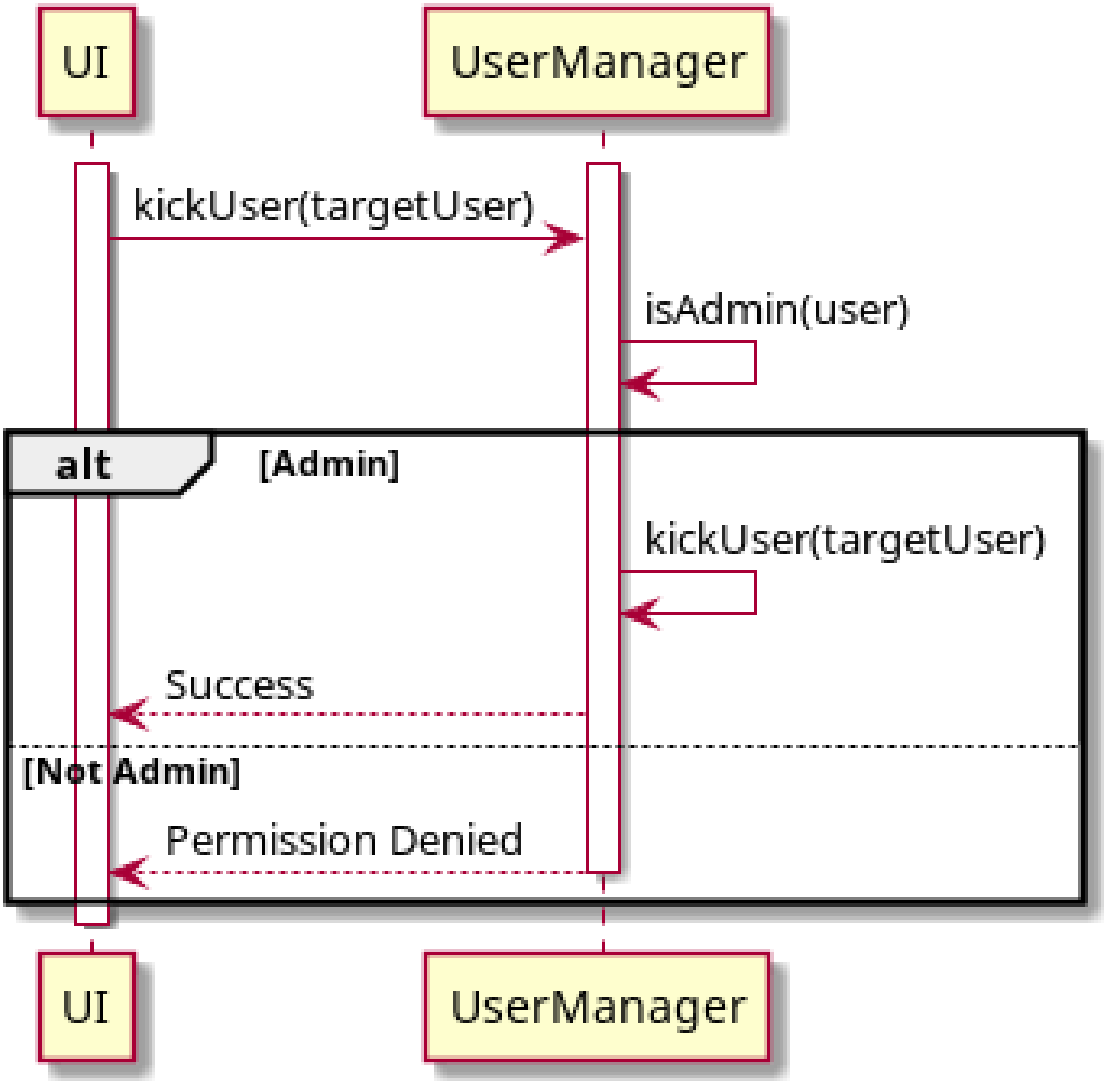



Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

#### 2.4.31 Project User Management Feature 1: Add User Sequence Diagram

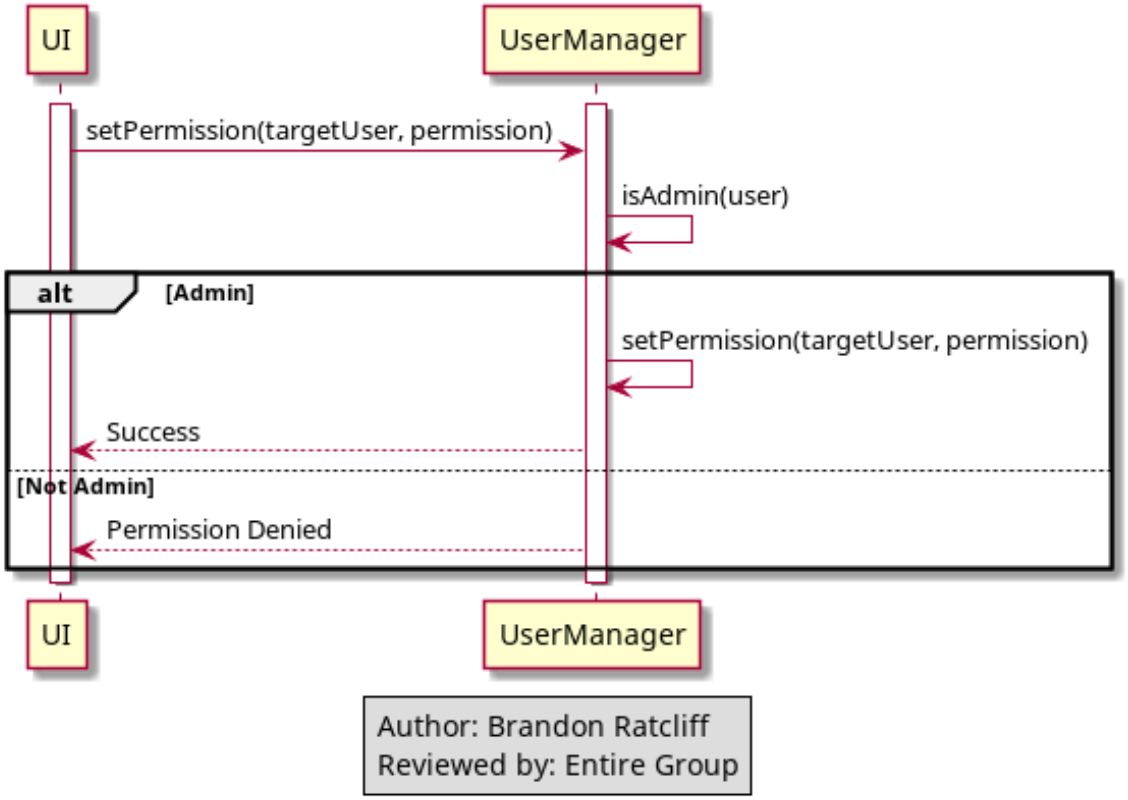


#### 2.4.32 Project User Management Feature 2: Kick User Sequence Diagram

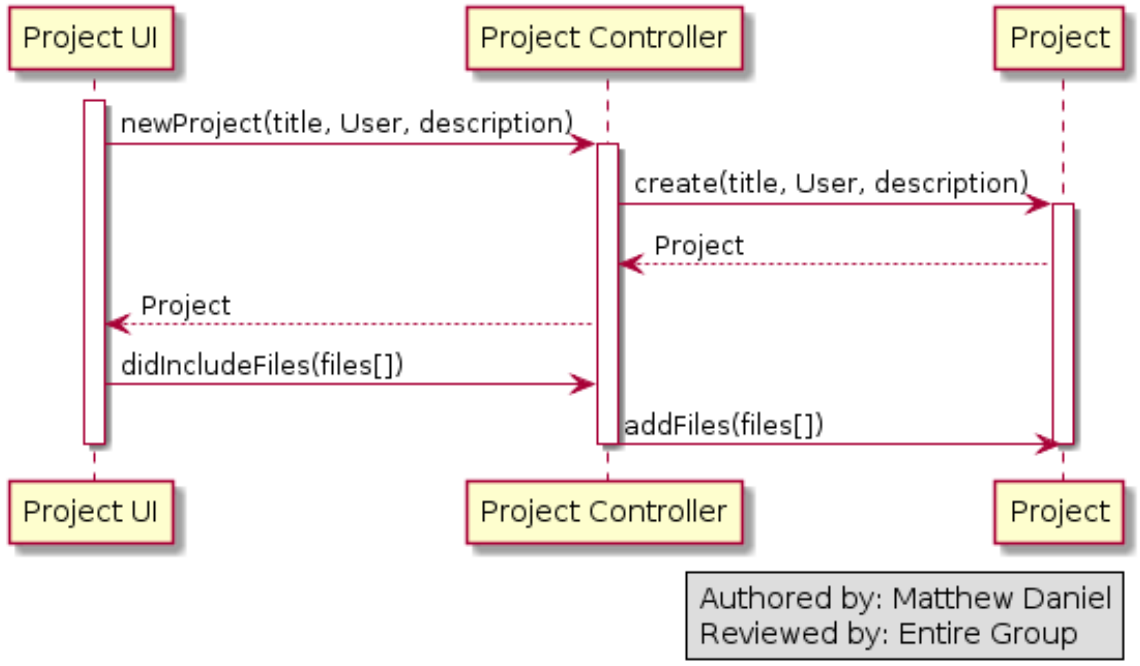



Author: Brandon Ratcliff
Reviewed by: Entire Group

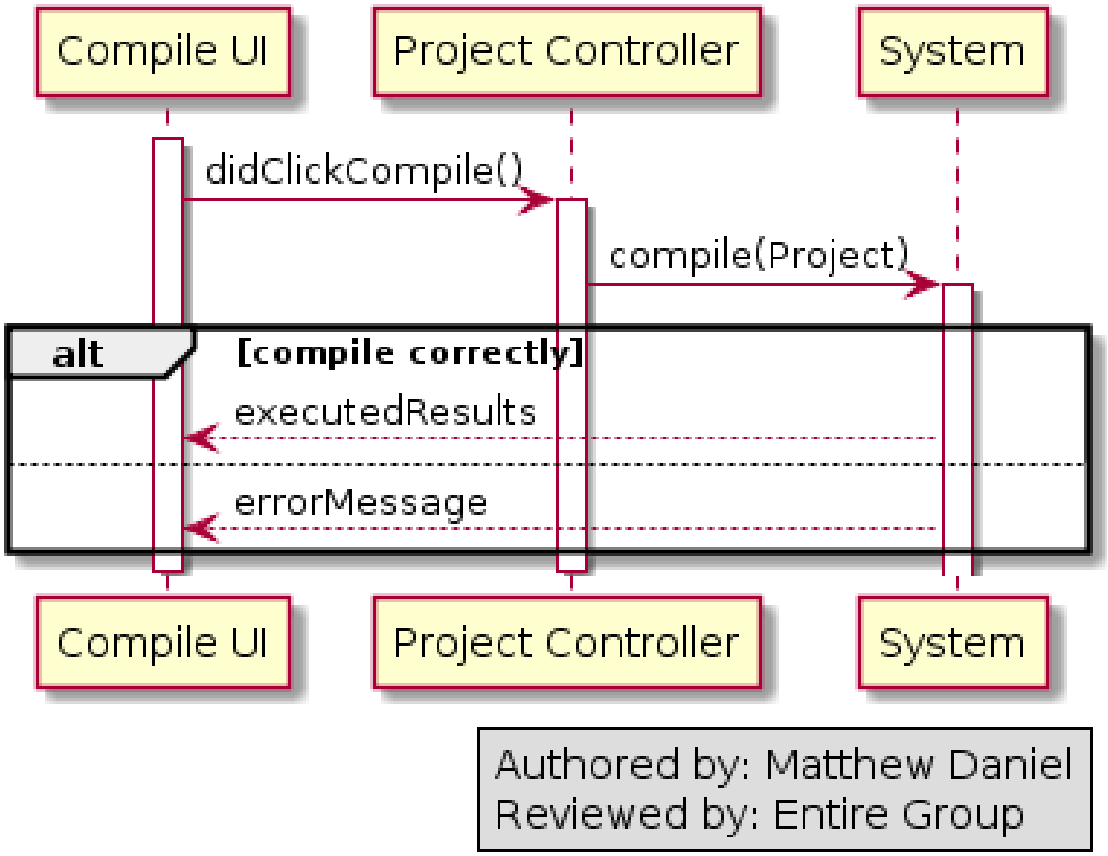
#### 2.4.33 Project User Management Feature 3: Set User Permissions Sequence Diagram



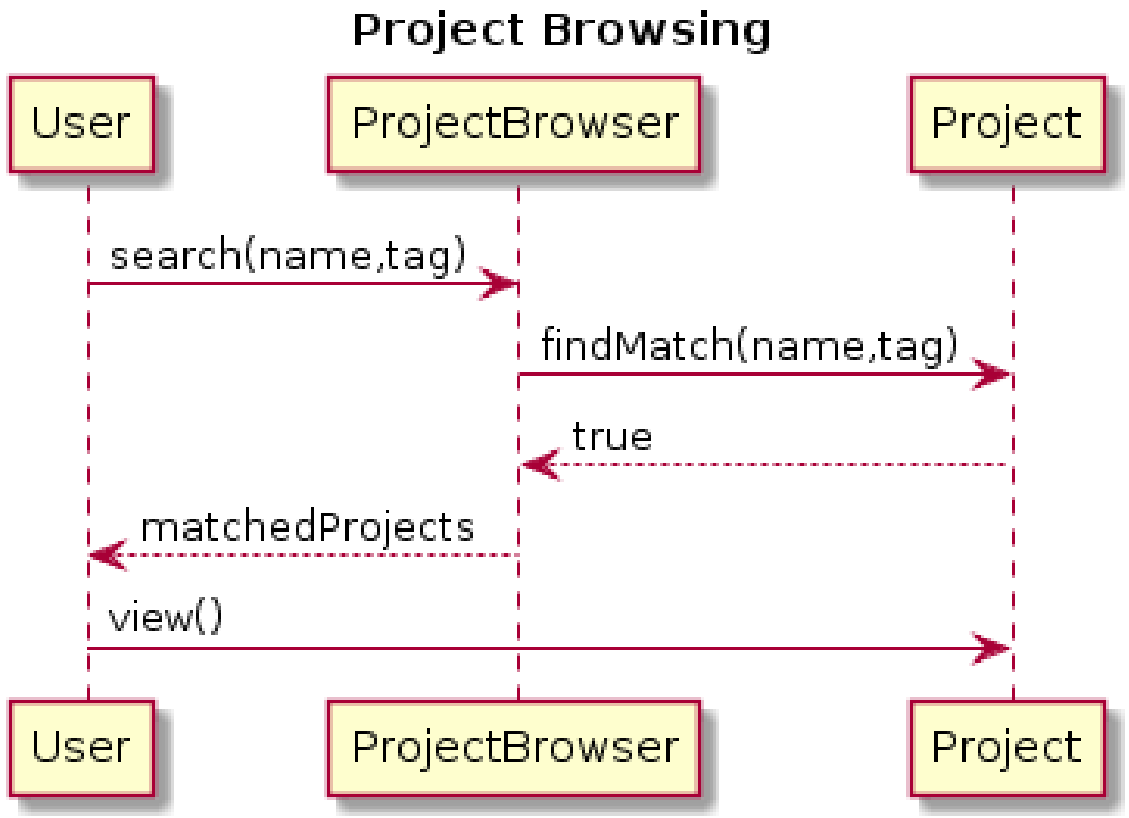
#### 2.4.34 Project Management Feature 2: Create project Sequence Diagram(dani2918)



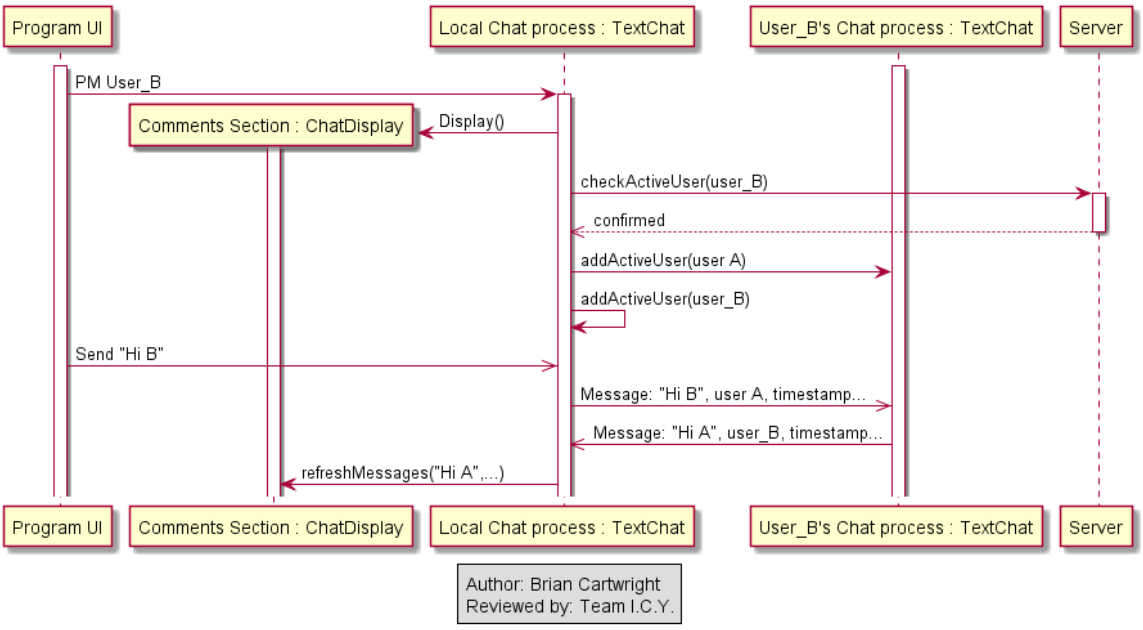
#### 2.4.35 Project Management Feature 1: Compile and Execute Project Sequence Diagram (dani2918)



#### 2.4.36 Project Browsing Feature 1: Project Browsing Sequence Diagram

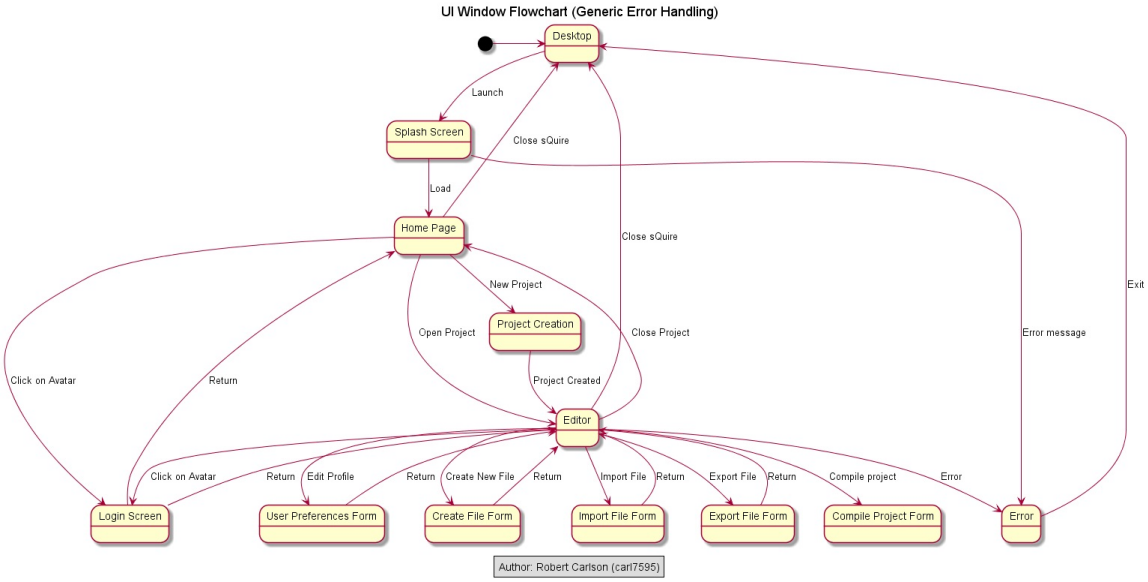


### 2.4.37 Feature 3: Message User by Name Sequence Diagram



## 2.5 USER INTERFACE DIAGRAMS

### 2.5.1 UI Window Flowchart



# 3 IMPLEMENTATION

Code organization overview. Description of major components/folders (client, server, ..., but maybe a level or two more detailed). Description of how it is built. Which IDE(s)? Number of targets in IDE. What does it look like to run outside an IDE? What does a client binary distribution look like, and how portable is it? What about a server (or client/server) binary distribution, and how portable is it?

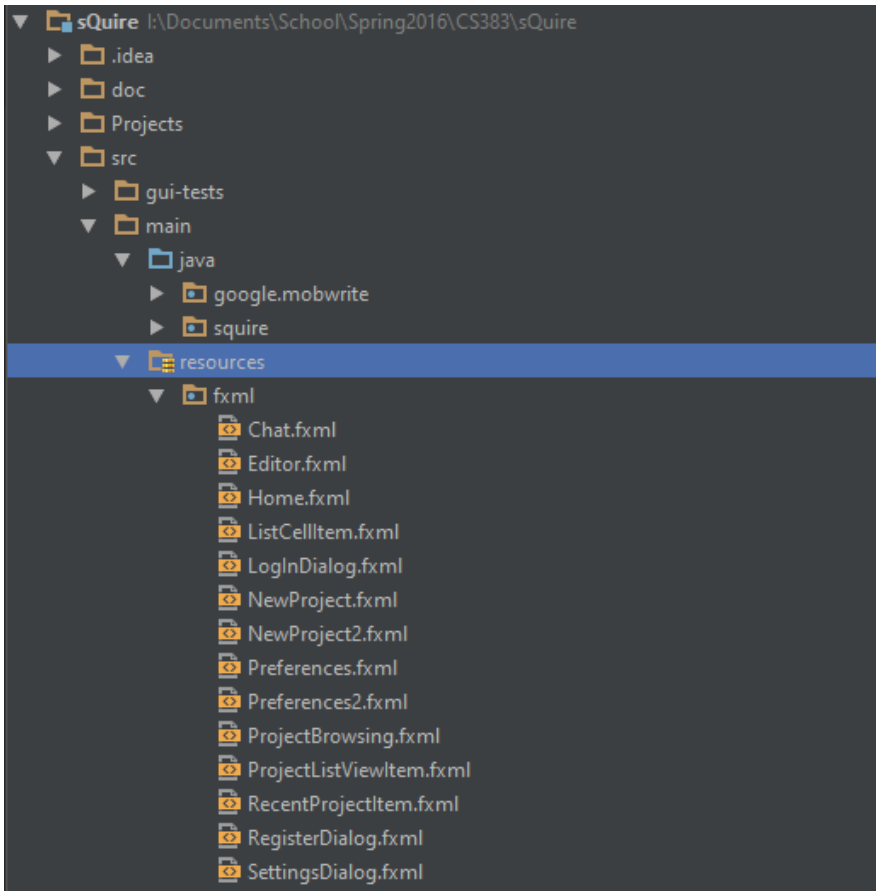
## 3.1 User Interface

### 3.1.1 JavaFX Framework (wern0096)

We used the JavaFX framework for the implementation of our user interface. It comes by default as part of the current JDKs. There are two major parts to the JavaFX framework: FXML resource files and their respective controllers.

The FXML files are XML files dictating the static structure of each user-interface "scene" - the current content being show in a window of the program. Each FXML file is associated with a controller class that has the FXML file's items, the UI elements, injected into it via Java annotations. The controller class is then able to handle any events that take place on the user interface.

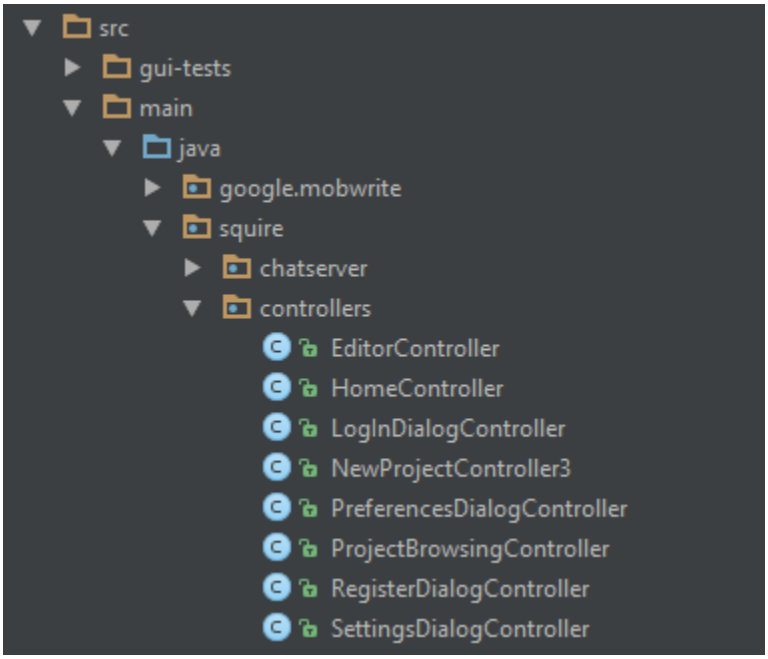
Our project's FXML files can be located in the **src/main/resources** directory:



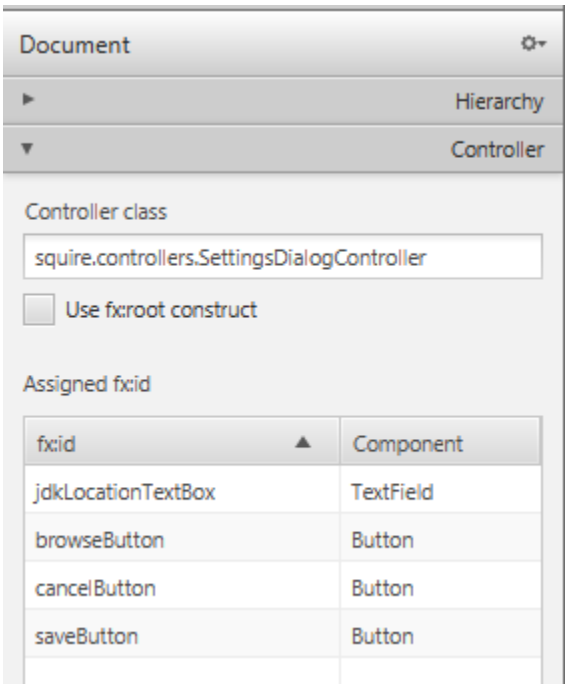
By storing the fxml files in a specified resources folder, IntelliJ copies them into the output directory and preserves their relative path. The files can then be loaded with most resource loading APIs. For example, here is a function call for loading an FXML resource file:

```
Parent root = FXMLLoader.load(getClass().getResource("/fxml/ProjectBrowsing.fxml"));
```

Since the *ProjectBrowsing.fxml* file is in that path relative to the *resources* directory, it can be loaded by the FXMLLoader's API. Once loaded into a scene – or window – the FXML file relies on a controller to handle any interactions with the scene. Our controllers can be found in the **src/main/java/squire/controllers** package:



Controllers and FXML files have a one-to-one association that can be set in the FXML file itself in two ways. One way is to set the controller via the SceneBuilder application that we used to create our user inteface scenes like this:




Whenever you set the controller in SceneBuilder, it will set it in the FXML file. Alternatively, you can set the controller in the FXML file itself with the **fx:controller** attribute in the parent node of the scene like this:

```
<AnchorPane maxHeight="-Infinity"  xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1" fx:controller="squire.controllers.SettingsDialogController">
```

Once a controller is associated with an fxml file, you can use **@FXML** Java annotations to inject the UI elements into the Controller class, allowing dynamic interaction with them. Here is an example of a some UI elements injected into a controller class via Java annotations:

```
@FXML private ImageView avatarImageView;
@FXML private Button homeButton;
@FXML private TreeView fileExplorer;
@FXML private TextArea editorTextArea;
@FXML private Button saveButton;
@FXML private TextArea chatTextArea;
@FXML private TextField chatTextField;
@FXML private Button sendButton;
@FXML private Button addButton;
```

Similarly, functions annotated with **@FXML** can be referenced in the fxml file to be used as handlers for actions on the UI.

## 3.2 Server

The server side portion of sQuire was hosted all on an Azure server. It was divided into two main sections: the MobWrite server handling the collaborative editing, and the sQuire server handling all the database requests.

### 3.2.1 MobWrite (ratc8795)

As stated earlier in this document, MobWrite is a set of libraries created by Google written on top of their Diff-Match-Patch algorithms. It consists of a server written in python, and several different clients written in several different languages. As the server worked fine without modifications, we did not include it in our repository. It can be retrieved from the MobWrite website (https://code.google.com/archive/p/google-mobwrite/).

Setting up the MobWrite server is a little interesting, as most of the documentation is gone, or never existed, it required digging into the code to see what was going on. It turns out that the MobWrite server consists of two parts. An Apache server with mod_python listens for https requests on port 80, and then passes them to a python script. This python script takes the raw http requests, formats them, and then connects to the MobWrite daemon listening on localhost via Telnet.

Each request to the MobWrite server consists of a Room ID, and patch information about that room (any additions, or deletions made). Inside the MobWrite daemon, Berkeley DB (an embedded key/value database) is used to store the contents of the room. The patch information is then used on the contents of the room on the server to update it, and to determine if any changes need to be sent back to the client.

The client portion of MobWrite is contained in the src/main/java/google/mobwrite folder. For the most part, this has been left unchanged as well. Some changes were required to get it compiling with our other code, and significant changes were required to get it to work with the CodeArea component, instead of the Swing components it was built for.

In the client, it's possible to configure the update frequency. This specifies the maximum time the client goes between contacting the server to see if there are any updates. If changes are actively being made, the client will report them to the sever as often as it can. In our demo, we used an update frequency of 500ms. We found that this was the most effective, as any less then than didn't really improve the experience too much. This speed still allows for effective collaboration while not overburdening the server too much.

This entire setup is surprisingly efficient. MobWrite comes with a couple of stress testing tools that tests 200 clients connecting to the MobWrite daemon at the same time. During this test, the CPU usage of our modest server never past 75%. It's reasonable to assume then, that in a production environment with a server of the same size as the one we had, the system could support at least that many concurrent users. This number could be increased if needed by reducing the update frequency, using a more powerful server, or splitting the load across multiple servers based on the Room ID.

### 3.2.2 Chat (ratc8795)

Our chat client was originally implemented as a stand alone client/server program. It worked by opening a socket on a separate thread between the client and the server. When messages were sent from the client, the server then broadcasted them to every other connected client.

Unfortunately, when we tried to integrate this with the JavaFX GUI, it didn't work to well due to the fact that everything in JavaFX runs in a separate thread. As the chat client used it's own thread to communicate with the server, it didn't work when plugged into the GUI. Due to time constraints, we ended up using MobWrite to power our chat.

The chat window consisted of an uneditable MobWrite Room unique to every project. Chat messages are appended to the MobWrite buffer, which is then synced to all the other clients. This approach is a little limited, because it means that all the clients have to see the same thing, so it is impossible to implement

a whisper mechanism, for example. It was however extremely easy to implement, and works without any additional sever architecture, and works very well for a simple chat client.

### 3.2.3 Networking (ratc8795)

The networking layer is the part of the application that handles storing and retrieving information in the database. It consists of two main parts: The Router and the Server. The Server accepts socket connections one at a time in a queue. Upon receiving a connection, the server accepts a Request object, and then passes this object to the Router.

The Request object contains a route field. This field tells the router what to do with this request. Upon the initialization, the Router accepts RouteHandlers. The route handles contain a bunch of functions that all register themselves to a specific route. When the Router receives a Request, it uses the route field of the request to pass it to the correct function in the correct RouteHandler. That function then returns a Response object containing all the data requested by the Request object.

The Router passes this Response back to the Server, which returns it to the client, and then moves onto the next waiting connection in the queue.

The client makes requests to the sQuire server only when doing things like creating new files or projects, browsing project, and logging in. Most of the collaborative part that requires continuous requests is handled by the MobWrite server. As such, this server won't be overly stressed, even with a large amount of clients. Several easy optimizations are available to us if needed are: splitting of a new thread for every connection so multiple can me served at once instead of only one at a time, or creating a way to send multiple requests in one go, reducing the overhead of creating and destroying sockets.

### 3.2.4 Database (ratc8795)

The database code was written using the EBean ORM (https://ebean-orm.github.io/). It has a very simple structure. Using Java's persistence annotations, we defined several models. These models live in the src/main/java/squire/Users/ directory in our repository. All of these models inherit from a Model class provided by Ebean, this provides several methods such as **save()** that make retrieving and saving models to the database work senselessly with normal getters and setters. Using the settings defined in src/main/resources/ebean.properties, every time the program is run, either a IDE plugin, or a Maven plugin generate a couple of files:

- **Typed query files** For every model, a Query class is created. This class allows querying the database using typed java code, so queries can be build by stringing together fields and method calls.
- **create-all.sql** A sql file that creates all the tables defined in the model classes.
- **drop-all.sql** A sql file that drops all the tables defined in the model classes.

EBean also preforms some Java bytecode modification upon compilation to make all the calls to model functions work with the database. In our experience, Ebean seems very reliable. The only difficult issue is getting everything building properly, but once that is set up, everything just works.

We configured EBean to use a Sqlite database. This made it really easy to develop with because the only extra thing that needs to be installed is the Sqlite Java libraries, which can be entirely installed with Maven. This means no additional work setting up a database server is needed to set up the server, and it can easily be run on a local machine during development. Since Sqlite stores everything in a single file, it's easy to share this database file with others for testing or debugging.

EBean uses a separate configuration for tests. Every time our unit tests run, EBean creates a new, clean database, and then runs our tests against it. This means that our tests are much more repeatable and reliable as they always start with the same state.

## 3.3 Client

### 3.3.1 CodeArea (ratc8795)

For the code editor, we used a CodeArea component from RichTextFX (https://github.com/TomasMikula/RichTextFX). This is a library of rich text editing components built for JavaFX. As JavaFX is a relatively new technology (compared to Swing), the libraries available for it are not as refined as some of the ones built for other GUI frameworks. RichTextFX is no exception. It includes the capability to do basic syntax highlighting, and code completion, but all of this requires extensive configuration and set up that we did not get to due to time constraints.

Some work had to be done to integrate MobWrite into the CodeArea. This work mainly consisted of replacing API calls used in MobWrite to call the ones in CodeArea instead of a JTextComponent, and then debugging it and diving into the source code of the two libraries when it didn't work. The results of this work can be seen in the src/main/java/google/mobwrite/ShareJTextComponent.java file.

Each instance of this ShareJTextComponent class contains a CodeArea (the name is a little misleading after we made our changes). A new instance of this is created every time a new editor tab is created, or the chat is created.

### 3.3.2 Networking (ratc8795)

All of the data regarding Users and Projects is stored on the server, and the client has no direct access to this. To retrieve information it has to send a request to the server. This is done by creating a Request object with the specified path (described more in the server section above). The Request object contains a HashMap of a string associated with an object. Any data the server needs to process the request is stored in this HashMap. Any object can be stored in this, just so long as it implements Serializable. Each Request object has a send function, which when called, opens up a new socket with the server, sends a serilized copy of itself to the server, and then returns the provided Response object.

## 3.4 Deployment

### 3.4.1 GitHub Repository

### 3.4.2 Dependencies

### 3.4.3 IDE and Plugins

# 4 TESTING

Stuff that Jeffery wants in bold and then followed by my comments:

- **Test plans, tests actually run, test results, test coverage.** I'll just copy our test plan document that we turned in for homework in. I think it needs some more stuff added to it though so add it here from now on and not there. Basically, we need to write tests for our major pieces of functionality. I think we should just focus on the logic instead of the UI here.
- **Write textual descriptions of test cases for everything that's not junit-integrated.** I think we only need to this for ORMTest so far.

## 4.1 Introduction

The purpose of this test plan is to provide a outline and provide reference for developers testing the complete sQuire program. This document is currently a standalone, but will be integrated with the SSRS document before the final submission. This document covers sQuire's logic tests, GUI tests, back-end tests, coverage tests, and any additional testing methods deemed necessary to ascertain that the complete sQuire software program adheres to our group's acceptable quality standard.

## 4.2 Logic Testing

The purpose of this section of tests is to list and describe logic tests in the sQuire program and the required output deemed as a "pass". These include algorithmic functions, arithmetic functions, validator functions, and other pieces of functionality that can easily be decoupled from the main project and/or reused as part of different projects.

### 4.2.1 Test Classes

Table 1: PasswordHashTest (wern0096)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| createHash() | Verifies that the hashing algorithm does not create colliding hashes. | A false assertion that all hashes created inside this function are different. |
| validatePasswor() | Verifies that the PasswordHash.validatePassword() function correctly authenticates a user based on their hashes password. | A true assertion that the a valid password and hash were validated. A false assertion that an invalid password and hash were validated. |

Table 2: EditorControllerTest (dani2918)

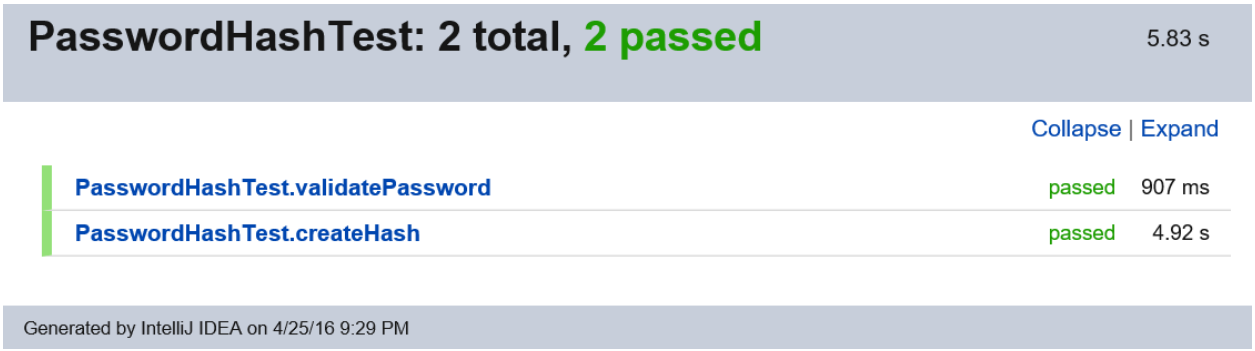
| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| testSetupMobWrite() | Verifies that we can set up mobwrite components with various names. | Successful creation of mobwrite components with various names constitutes a success. |

Table 3: NewProjectControllerTest (dani2918)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| testInitProjectFields() | Verifies that projects with various names and descriptions (in the form of strings due to the controller class's use of strings from TextFields) are properly created. | Successful creation of project directory inside a test directory constitutes a passing test. |
| testCopyMainFile() | Verifies that the initial dummy Main "Hello World" class is successfully copied into a directory. | Existence of the file at the specified location constitutes a passing test. |

### 4.2.2 Results

**PasswordHashTest (wern0096)**




This passed test indicates that we our hashing algorithm successfully validates a user's entered password with their hashed password, and properly creates non-colliding hashes. It also shows us that the createHash method takes 5 seconds to run, indicating a possible place for better optimization.

Table 4: EditorControllerTest (dani2918)

| Function Name | Result | Description |
|---|---|---|
| testSetupMobWrite() | FAILURE | Attempting to create a new CodeArea, which the setupMobWrite function requires as a parameter, was not working in the test class. Further investigation will be required to determine whether the function works properly. |

Table 5: NewProjectControllerTest (dani2918)

| Function Name | Result | Description |
|---|---|---|
| testInitProjectFields() | FAILURE | The test fails when attempting to create a project based upon the empty string as a title. We will have to implement logic to ensure that a user enters a project title in the appropriate field. This is the only case of those tested which caused a failure. |
| testCopyMainFile() | PASS | The copied file existed with multiple attempts. |

## 4.3 GUI Testing

This section governs our GUI unit tests and the required output deemed as a "pass". Since we are using the JavaFX framework, every test case requires an initialization step of loading the .fxml file for the GUI scene to be tested. Once it is loaded we perform tests on individual parts of the scene using the TestFX libraries that integrate with JUnit.

### 4.3.1 Test Classes

Table 6: HomeTest (wern0096)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| verifyUiElementsLoaded() | Checks that every UI element loaded properly. | No exceptions thrown by the verifyThat() function calls. |

Table 7: EditorTest (wern0096)

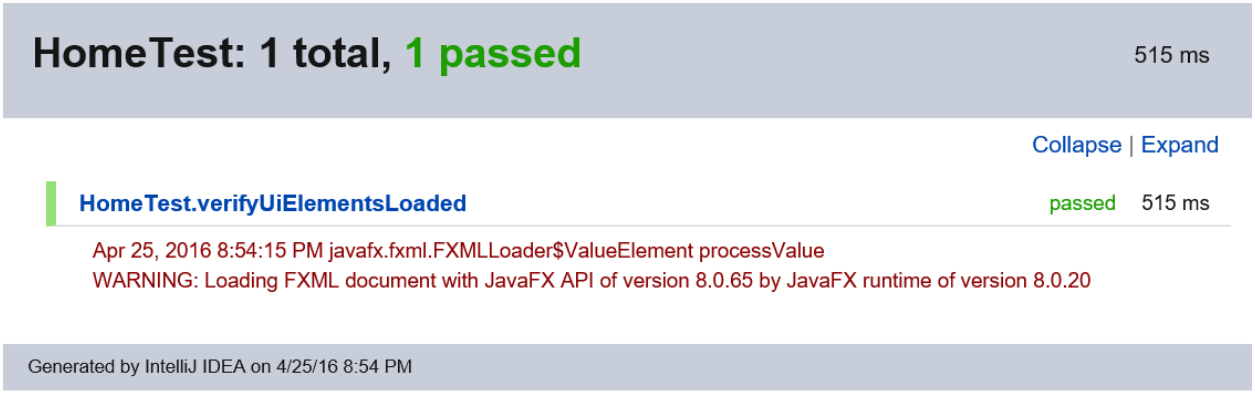
| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| verifyUiElementsLoaded() | Checks that every UI element loaded properly. | No exceptions thrown by the verifyThat() function calls. |

Table 8: NewProjectTest (dani2918)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| verifyUiElementsLoaded() | Checks that every UI element loaded properly. | No exceptions thrown by the verifyThat() function calls. |

### 4.3.2 Results

**HomeTest**




This passed test indicates that all UI elements are loading properly. It does create a warning message about a mismatch of the JavaFX API and the runtime, however, and wee will remedy that next sprint.

**EditorTest**


This test currently fails due to it requiring initialization with data from another class. This will be better tested with an automation script, but it is still useful to know what data it requires to successfully initialize. Here is the following exception it throws:

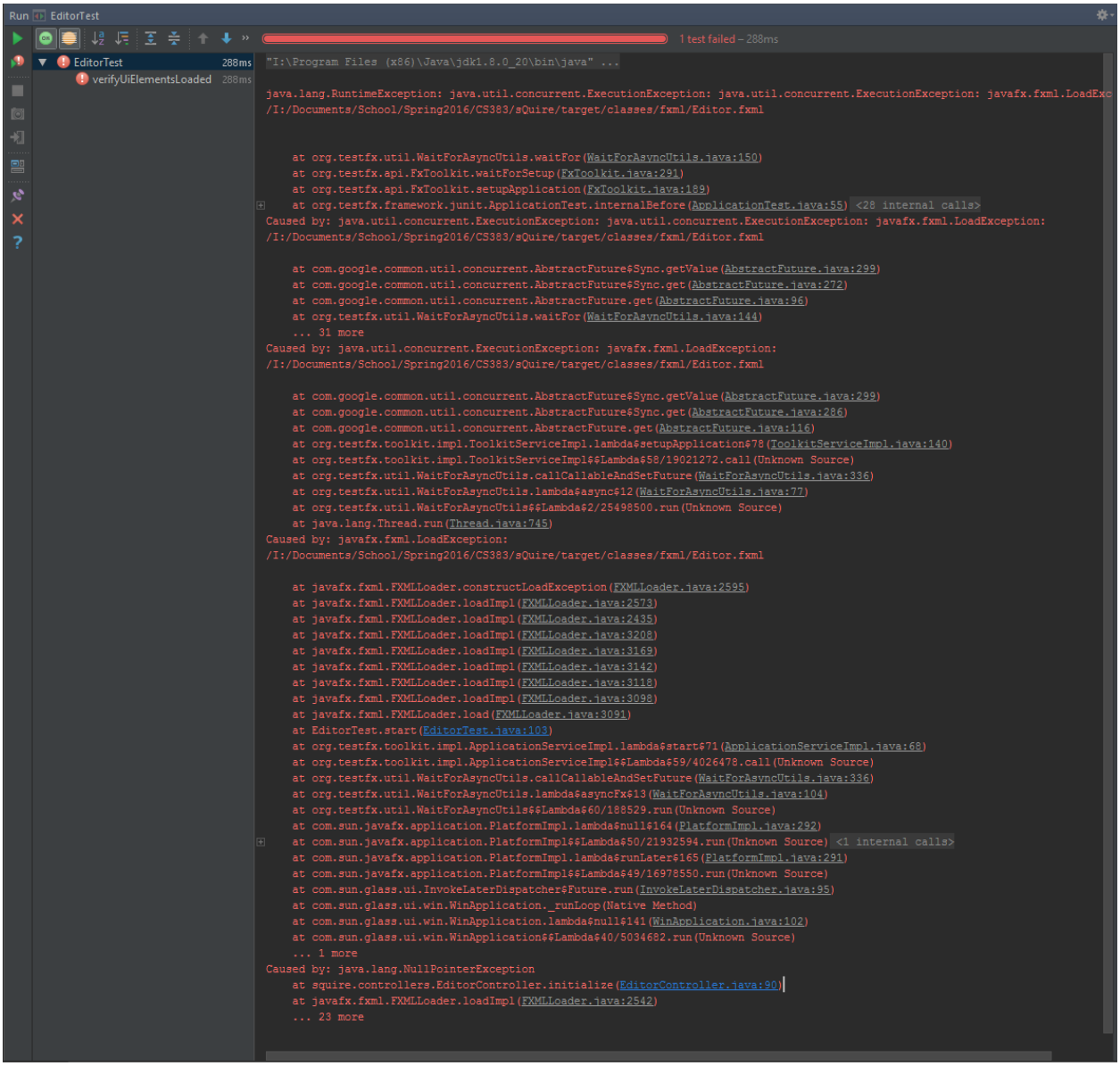



Table 9: NewProjectTest (dani2918)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| verifyUiElementsLoaded() | PASS | No exceptions were thrown after loading all elements from the FXML file. |

## 4.4 Back-end Testing

This section governs any tests aimed at our database(s) or server(s) and the required output deemed as a "pass".

One of the major libraries we use on the server (MobWrite) contains a full test suite written in JUnit, however, since we use the library as-is, without any modifications, we do not integrate these tests with our test suite. Instead, we just test to see if the client can successfully connect to the server, as that should be the only variable.

### 4.4.1 Test Classes

Table 10: MobWriteServerTest (ratc8795)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| connectToServer() | Checks to see if the server can be connected to. | No exceptions thrown by the Connection.connect() function. |

Table 11: MobWriteClientTest (ratc8795)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| testComputeSyncInterval() | Tests that the sync interval is calculated correctly. | The sync interal is calculated correctly under a variety of conditions. |
| testUniqueId() | Tests that unique IDs are created correctly. | The ID generated is 8 characters long, and two seperate IDs are unique. |

Table 12: SessionTest (ratc8795)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| isExpired() | Check that the isExpired funciton works correctly. | A not-expired session returns false, and a expired session returns true. |
| logout() | Check that logout function works correctly. | Once the logout function is called on the session, the session no longer exists. |
| activeSession() | Check the activeSession function returns an active session given a token. | The correct session is returned for a given token, a not active session won't be returned, and a non exsistant token won't return a session. |

Table 13: ChatServerTest (gent7104)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| VerifyConnection() | Verify that the a connection has been made | No exception is thrown when testconnection() is called. |

Table 14: ChatClientTest (gent7104)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| VerifySocektConnection() | Verify that the client was able to connect with the specified host and and port number | No exception is thrown when connecting to the socket. |

Table 15: ProjectDatabaseTest (cart1189)

| **Function Name** | **Description** | **Pass Criteria** |
|---|---|---|
| testAddProject() | Checks the ability to create and save new Project objects to their database table. | No exceptions thrown during Project initialization or database save(). The new Project entry is in the table during the test. |
| testRemoveProject() | Checks the ability to remove Project object entries from their database table. | No exceptions thrown during Project initialization or database save(). The Project in question is removed from the table by the the end of the test. |
| testGetSetTextFields() | Checks ability to set each string field of a Project database entry, and retrieve those fields from the database. | No exceptions thrown by the test. Each retrieved value matches the value it was set to. |
| testLoadFile() | Checks ability to upload and download raw files into the ProjectFile table | No exceptions thrown by the test. Retrieved copy of uploaded file matches the original version. |

#### 4.4.2 Results

Table 16: MobWriteServerTest (ratc8795)

| **Function Name** | **Result** | **Description** |
|---|---|---|
| connetToServer() | PASS | |

Table 17: MobWriteClientTest (ratc8795)

| **Function Name** | **Result** | **Description** |
|---|---|---|
| testComputeSyncInterval() | PASS | |
| testUniqueID() | PASS | |

Table 18: SessionTest (ratc8795)

| **Function Name** | **Result** | **Description** |
|---|---|---|
| isExpired() | PASS | |
| logout() | PASS | |
| activeSession() | PASS | |

Table 19: UserTest (ratc8795)

| **Function Name** | **Result** | **Description** |
|---|---|---|
| getSetUsername() | PASS | |
| setCheckPassword() | PASS | |
| authenticate() | PASS | |
| userQuery() | PASS | |

## 4.5 Coverage Testing (wern0096)

### 4.5.1 Methodology

The coverage test of sQuire was performed by running the program with the coverage testing option enabled in IntelliJ IDEA and a developer manually navigating through all of the functionality in the program.

### 4.5.2 Results

The following report was then generated, showing usability per package:

[ all classes ]

Overall Coverage Summary

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| all classes | 58.2% (32/ 55) | 44% (147/ 334) | 34.9% (942/ 2699) |

**Coverage Breakdown**

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| squire.controllers | 90.9% (10/ 11) | 87.7% (57/ 65) | 83.1% (375/ 451) |
| squire.chatserver | 0% (0/ 3) | 0% (0/ 9) | 0% (0/ 136) |
| squire.Users.query.assoc | 0% (0/ 4) | 0% (0/ 16) | 0% (0/ 24) |
| squire.Users.query | 25% (1/ 4) | 16.7% (2/ 12) | 20% (4/ 20) |
| squire.Users | 100% (9/ 9) | 45.5% (30/ 66) | 40.4% (61/ 151) |
| squire.Projects | 66.7% (2/ 3) | 32.1% (9/ 28) | 25% (18/ 72) |
| squire | 66.7% (2/ 3) | 51.6% (16/ 31) | 56.3% (49/ 87) |
| google.mobwrite | 44.4% (8/ 18) | 30.8% (33/ 107) | 24.7% (435/ 1758) |

generated on 2016-04-25 21:15

As indicated, most of the squire.controllers package was hit by our coverage test, indicating very little extraneous UI code. The chat server was not tested with this run, and the rest of the packages had about half of their functionalities hit, due to being heavily in development at the moment. As an example of the granularity of this report, let's examine the squire.controllers package to see what code didn't run during this test:

[ all classes ] [ squire.controllers ]

Coverage Summary for Package: squire.controllers

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| squire.controllers | 90.9% (10/ 11) | 87.7% (57/ 65) | 83.1% (375/ 451) |

| Class | Class, % | Method, % | Line, % |
|---|---|---|---|
| SettingsDialogController | 100% (1/ 1) | 100% (5/ 5) | 100% (17/ 17) |
| RegisterDialogController | 100% (1/ 1) | 100% (5/ 5) | 100% (15/ 15) |
| ProjectBrowsingController | 100% (1/ 1) | 100% (2/ 2) | 83.3% (10/ 12) |
| PreferencesDialogController | 100% (1/ 1) | 100% (4/ 4) | 100% (10/ 10) |
| NewProjectController3 | 100% (1/ 1) | 100% (10/ 10) | 92% (80/ 87) |
| LogInDialogController | 100% (1/ 1) | 80% (4/ 5) | 67.9% (19/ 28) |
| HomeController | 100% (1/ 1) | 88.9% (8/ 9) | 75% (75/ 100) |
| EditorController | 75% (3/ 4) | 76% (19/ 25) | 81.9% (149/ 182) |

generated on 2016-04-25 21:

Delving further, let's see what code in the EditorController didn't run:

[ all classes ] [ squire.controllers ]

Coverage Summary for Class: EditorController (squire.controllers)

| Class | Method, % | Line, % |
|---|---|---|
| EditorController | 100% (14/ 14) | 93.8% (121/ 129) |
| EditorController$1 | 100% (2/ 2) | 100% (17/ 17) |
| EditorController$TextFieldTreeCellImpl | 42.9% (3/ 7) | 36.7% (11/ 30) |
| EditorController$TextFieldTreeCellImpl$1 | 0% (0/ 2) | 0% (0/ 6) |
| **total** | 76% (19/ 25) | 81.9% (149/ 182) |

Lastly, we can get very granular and see exactly which lines of code didn't run. Here's an example of the updateItem() function's code and how the IntelliJ Coverage Report displays code that was hit with a green outline and code that wasn't hit with a red outline:

```
@Override
public void updateItem(String item, boolean empty)
{
    super.updateItem(item, empty);

    if (empty)
    {
        setText(null);
        setGraphic(null);
    }
    else
    {
        if (isEditing())
        {
            if (textField != null)
            {
                textField.setText(getString());
            }
            setText(null);
            setGraphic(textField);
        }
        else
        {
            setText(getString());
            setGraphic(getTreeItem().getGraphic());
        }
    }
}
```

## 4.6 Software Risk Issues (wern0096)

### 4.6.1 High Risk

The following items are deemed high risk to the security of users and to the usability and quality of the sQuire software:

- **MobWrite Server and Client** Our collaborative editing relies on Google's MobWrite software library. We are rating it high risk because it is the core of the collaborative editing functionality, and it is also a third-party tool with a high impact on the functionality of our product. This module also incorporates the diff-match-patch algorithm running on clients' machines that talks to our Azure server that broadcasts text changes from one client to the rest of the clients. Since it will be sending code over the internet to and from the server, the security implications for all machines involved are very serious. As such, we plan to have extensive testing and review of code using this module.
- **Chat Client** Our chat client is simply a Java application running indefinitely in the same Azure server as the MobWrite server. Nevertheless, it is broadcasting possibly confidential data through the internet and should have the proper encryption and security reviews as the rest of the high risk items.
- **JavaFX** The JavaFX framework is also core to our program. Since it has been used to build most of the user interface, it will require great usability and coverage testing to make sure that it is friendly and intuitive to our users. The breakdown of the user interface would essentially render the rest of the program useless. Thus, it receives a high risk rating.
- **Database Credentials** The project uses a SQLite database for storing of user credentials and project information. We rate the security of user and project credentials in the database as high risk. In order to protect the confidentiality of user credentials and integrity of user projects, we will have to adhere to database security best practices such as salting and hashing stored credentials, encrypting traffic over the internet, and ensuring SQL-injection attacks are mitigated.

### 4.6.2 Medium Risk

The following items are deemed medium risk to the security of users and to the usability and quality of the sQuire software:

- **Database Storage and Ebean ORM** The non-confidential data stored in the database is rated as medium risk. This is because project data is also stored locally, and breakdown of the database storage functionality would not severely reduce core functionality of the sQuire software. However, it would still significantly hinder collaborative functionality. To address this, we plan on extensively testing database commands with unit tests and code reviews.
- **Code Compilation** Whenever code compilation is involved, security becomes a concern. However, there is not code being executed remotely, so the security risks become much smaller, thus earning this module a medium rating. Compilation in sQuire will rely on the user reviewing their code for malicious intent before executing. We plan on implementing tools for the user to easily track changes to the code in order to aid in the review process if we have time. Also, we will have unit tests making sure that the user is properly notified of code errors and their location(s).
- **Testing Modules** We are putting testing modules themselves as a medium risk item because of the sheer complexity of testing the user interface with JavaFX and TestFX, another open source framework. Since the user interface is rated as a high-risk item, we believe that properly testing it is at least a medium risk item to the proper functionality of our program. We didn't rate it as high-risk, because there are many ways of testing the user interface that are less complex but take more time.

#### 4.6.3 Low Risk

The following items are deemed low risk to the security of users and to the usability and quality of the sQuire software:

- **Editor Features** This module features some complicated code that will come from third-party open sources such as search/replace, syntax highlighting, and auto-complete. However, failure of this module does not severely endanger users' security or the core functionality of the software. As such, it earns a low risk rating. This module will require extensive unit testing, however.
- **Local File Structure** The local file structure stores user's projects locally in the form of a folder for each project. We rate this module as a low risk module because even if the local files were corrupted or deleted, the user's projects/files can still be restored from the database, and the program has code to handle such a case. Nevertheless, we plan to have various unit tests documenting valid/invalid file structures that we can then create code to handle such cases.
- **Program Settings** This module involves settable user preferences that should persist between runs of sQuire. We plan to have the user's settings be saved locally as well as on the database, so that upon login, sQuire will update to conform to the user's preferences. Since there is redundancy to this module and its breakdown does not constitute a large hit to the functionality of sQuire, we deem it low risk.

# 5 METRICS

Many of our metrics are measured via an IntelliJ plugin called MetricsReloaded, which measures lines of code, complexity, etc. Anything under the package google.mobwrite is a modified version of Google's Mobwrite project, which we found on Github.

- **Size of our Project**
  - Number of Classes
  - Cyclomatic Complexity
    Our project had an average cyclomatic complexity of 2.85. Mobwrite introduced the greatest amount of cycomatic complexity to our project. None of our classes had a v(G) exceeding 10.
- **How much of our functionality is implemented.** Will probably have to be done by hand. Maybe make a table of use-case/functional requirement and whether it was implemented or not and a justification.
- **How much of our code is tested and how tested is the code.** Get a ratio of unit test classes vs actual classes? How much functions have a unit test associated with them? Do we have uncovered cases?
- **Who did the work?** Use GitHub for that? Add any extra info manually.
- **Problem Areas.** What was hard to implement and why?