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:

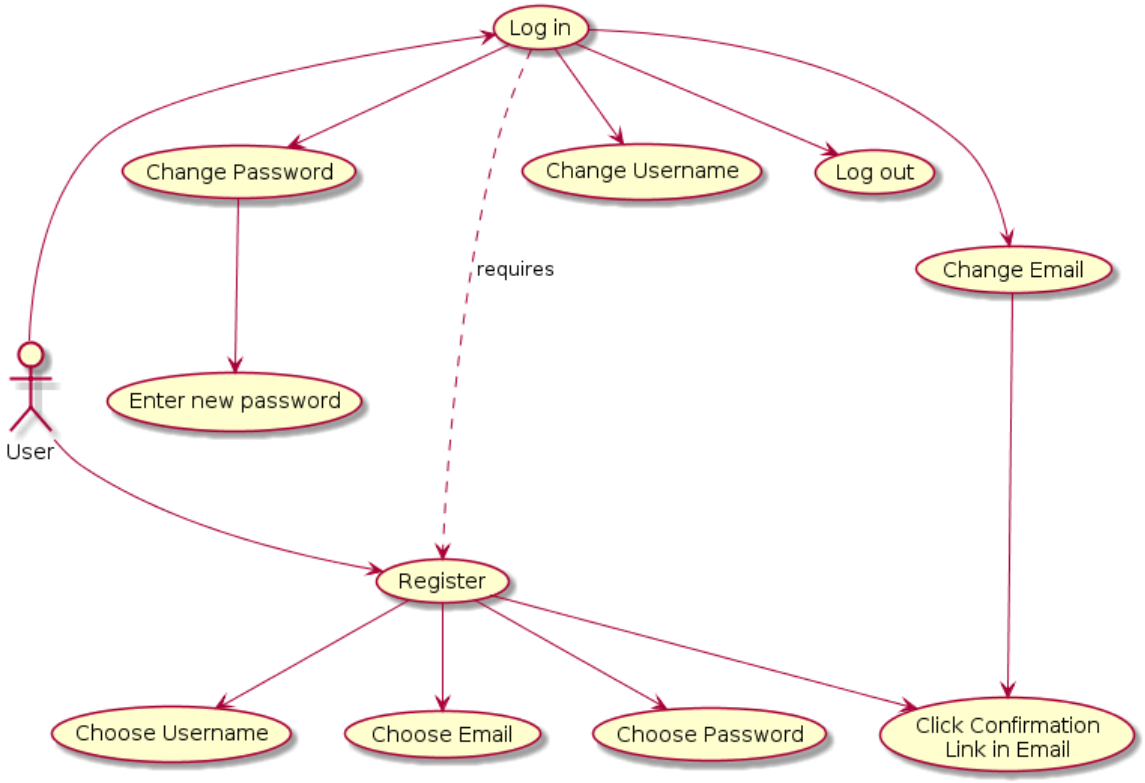
1. User clicks Permissions Management button.
2. System displays permissions management window.
3. User selects the user whose permissions they want to edit.
4. System displays a list of toggles for the users's permissions.
5. User makes changes to the user's permissions.
6. System modifes the target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

Alternatives: None.
Post-conditions: None.

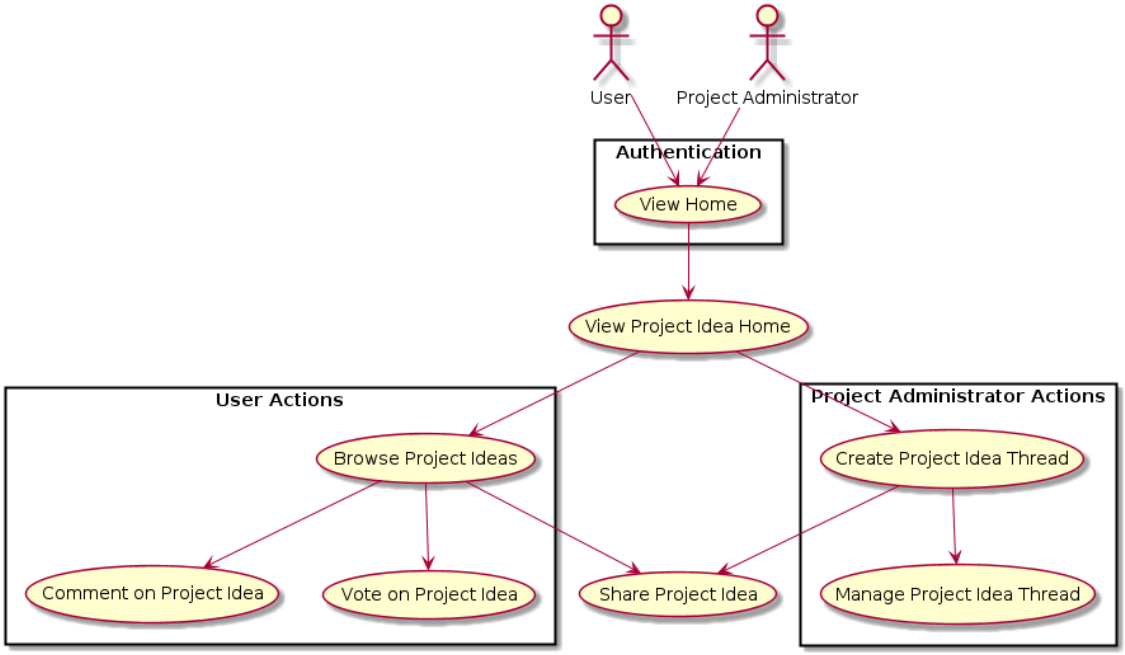
- **User**
- **Project**
- **UserManager**
- **Permissions**

2.2 USE CASE DIAGRAMS

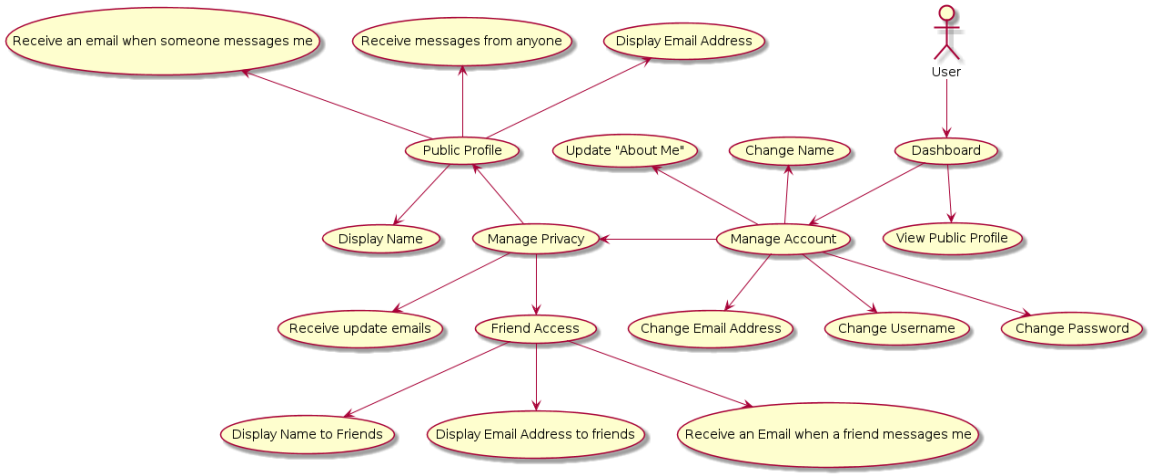
2.2.1 Use Case Diagram 1: Authentication



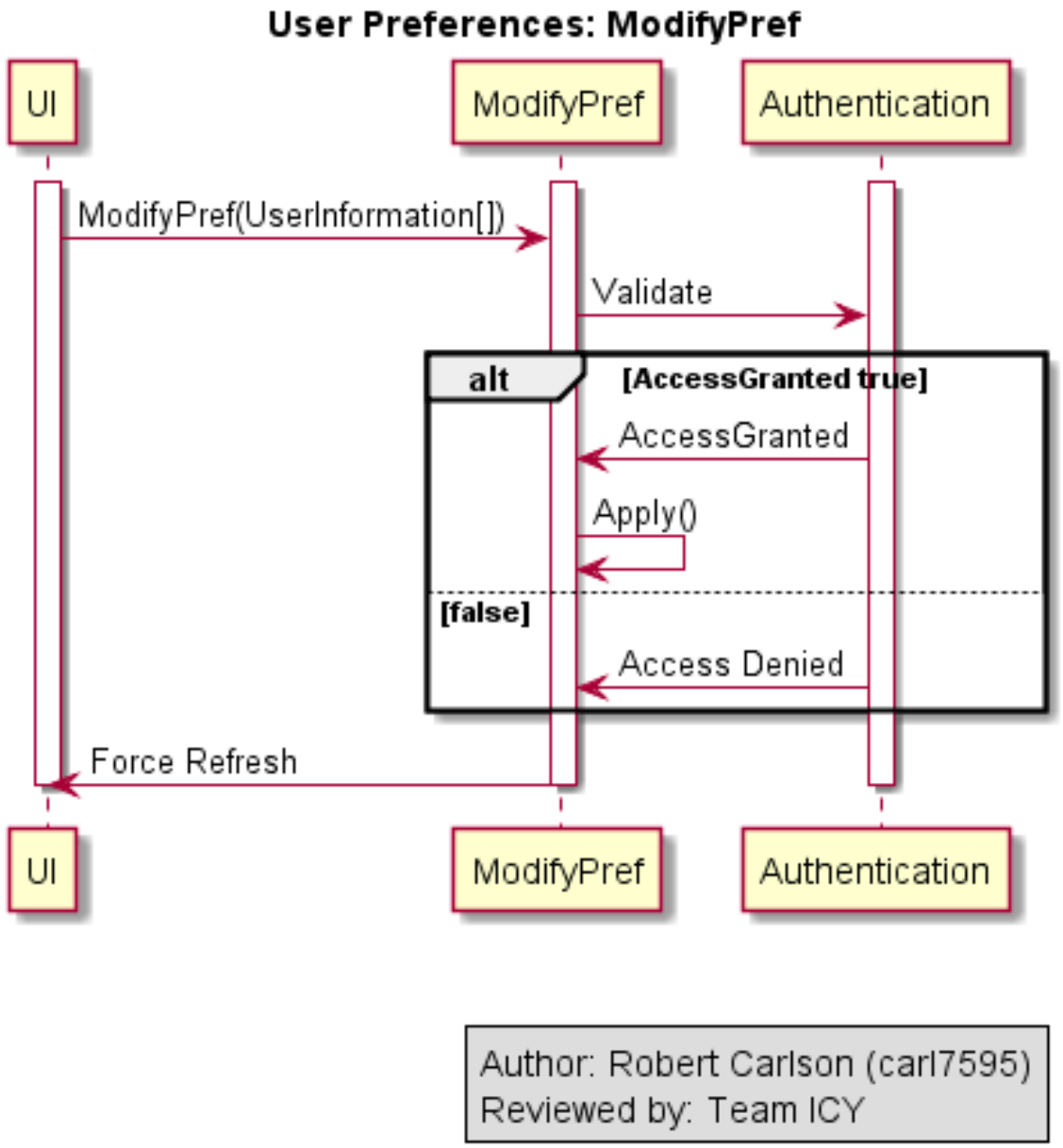
2.2.2 Use Case Diagram 2: Project Browsing



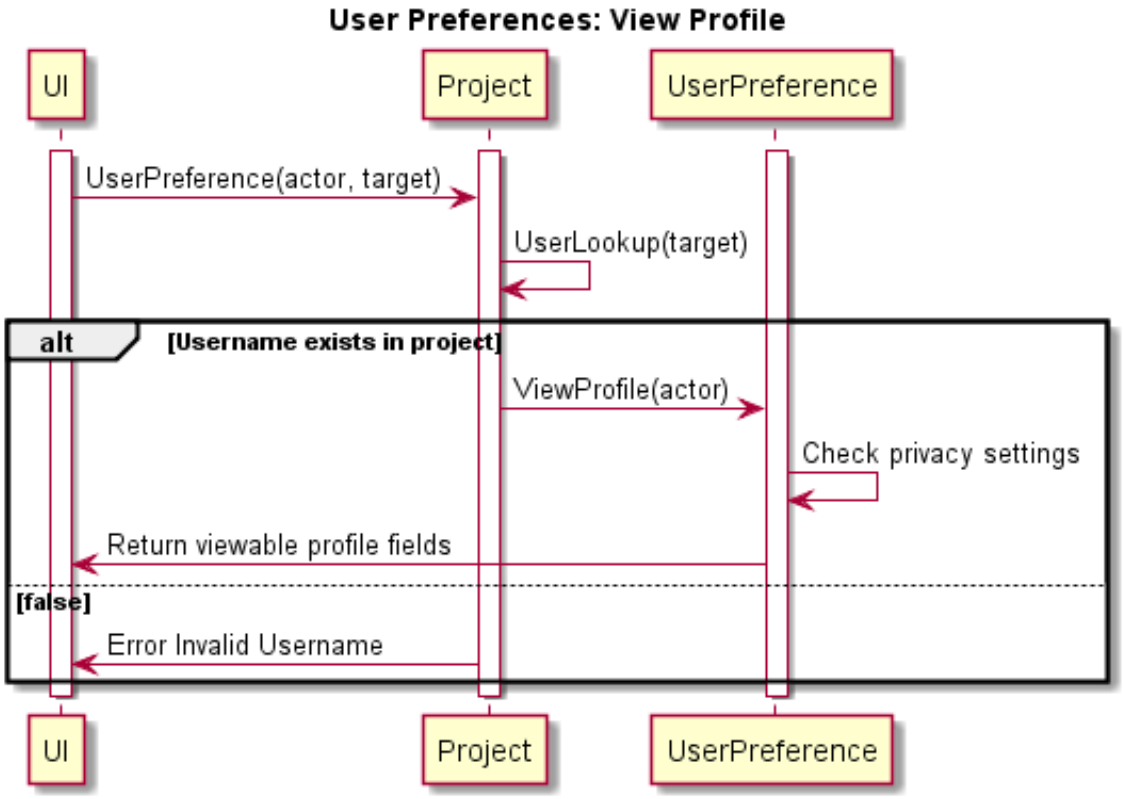
2.2.3 Use Case Diagram 4: User Preferences



2.2.4 User Preferences Feature 1: Use Case Diagram 1

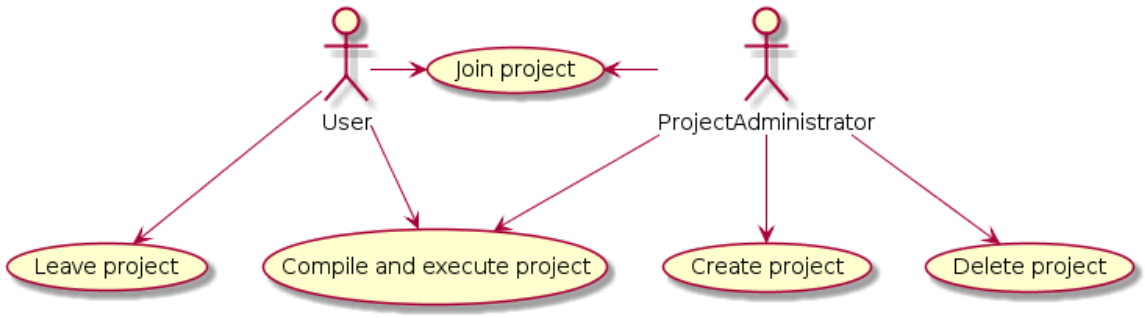


2.2.5 User Preferences Feature 2: Use Case Diagram 2



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

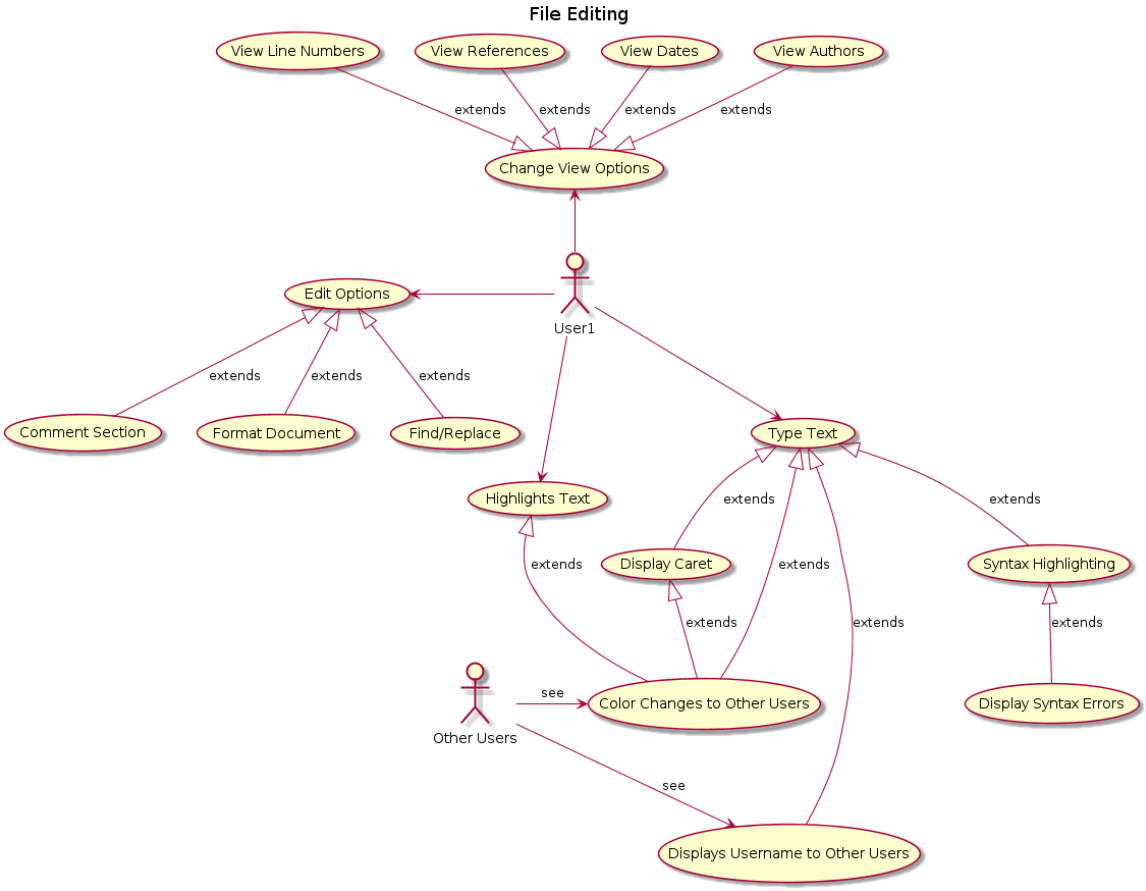
2.2.6 Use Case Diagram 5: Project Management



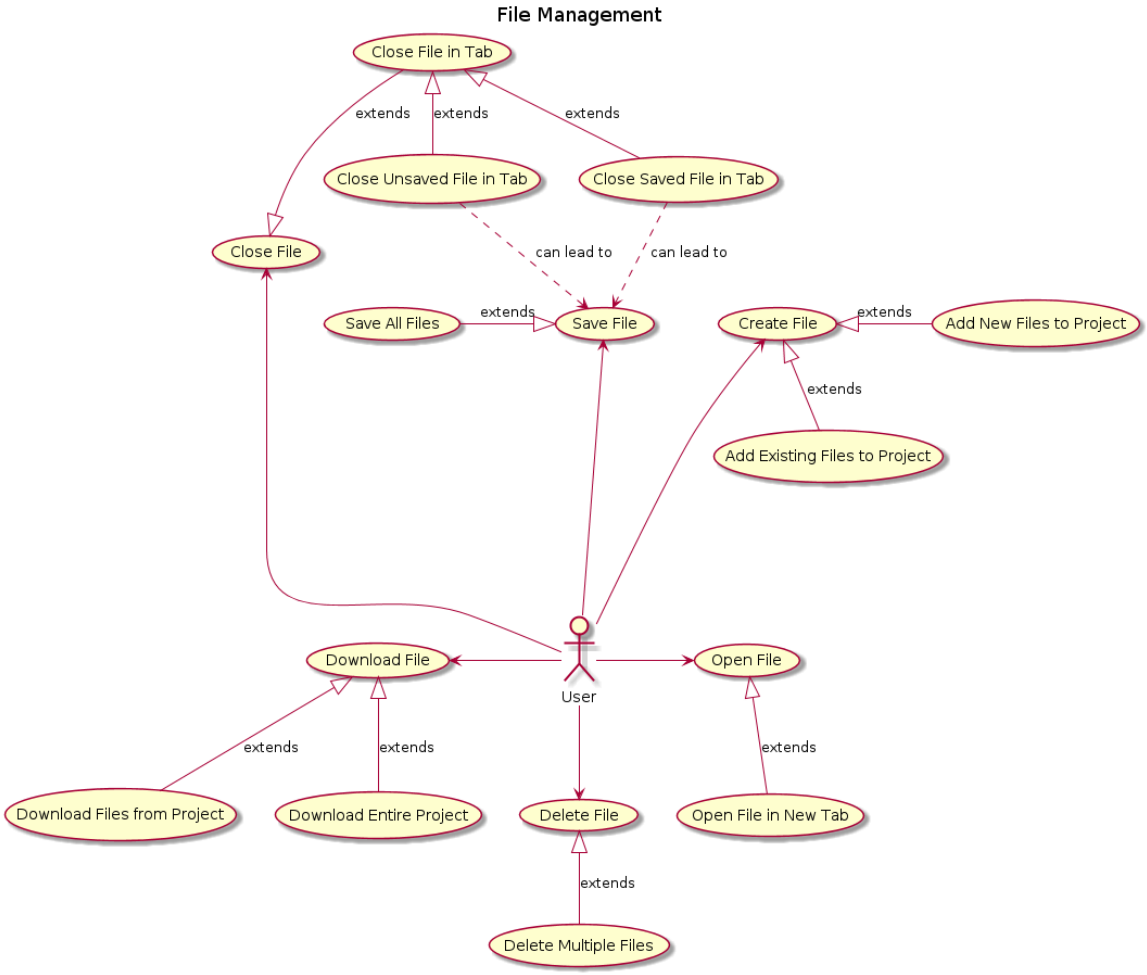
Use case descriptions were roughly based upon cases from HW2, Team 4. sass8427 worked on the original use cases in this section.

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.

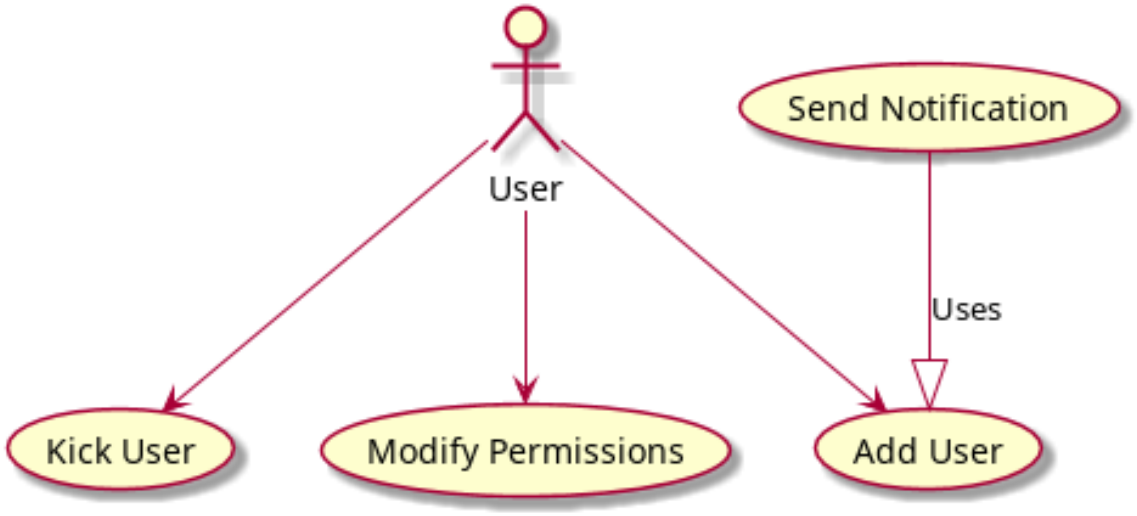
2.2.7 Use Case Diagram 6: File Editing



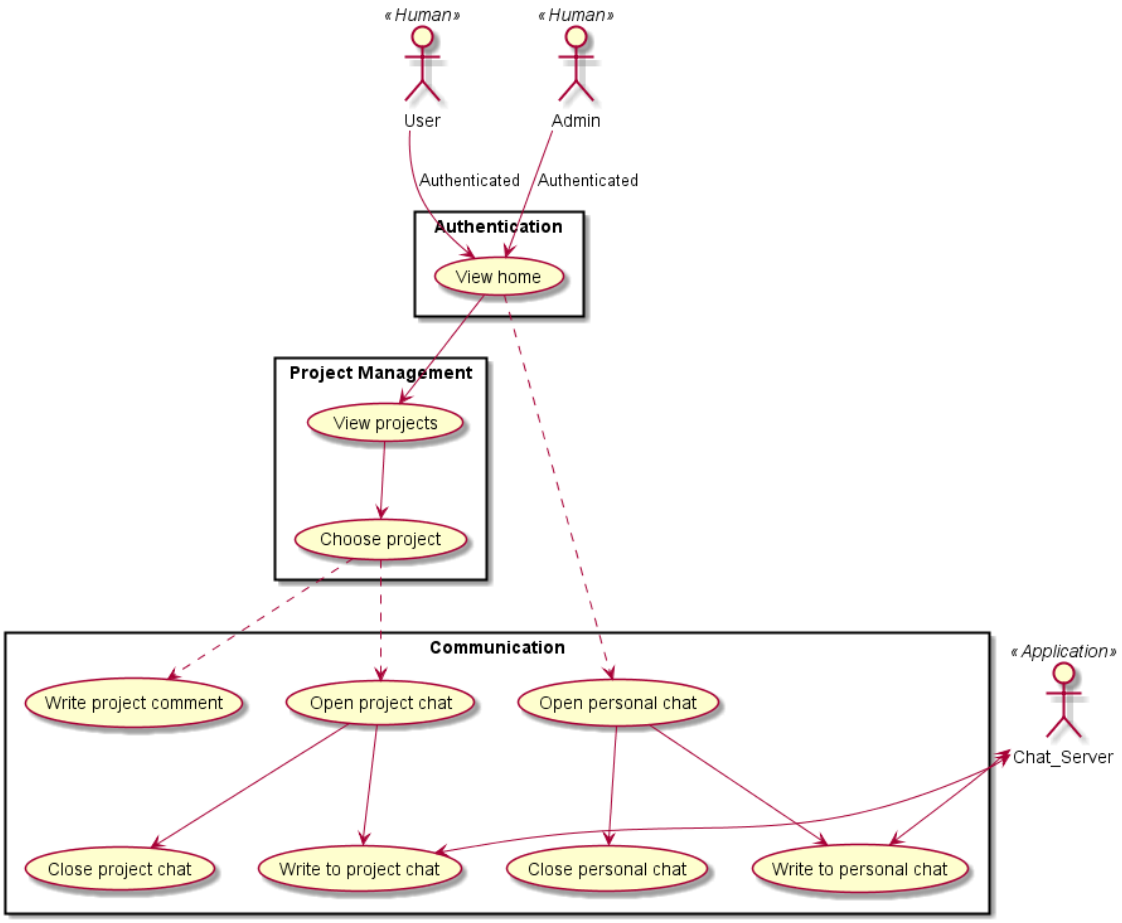
2.2.8 Use Case Diagram 7: File Management



2.2.9 Use Case Diagram 8: Project User Management

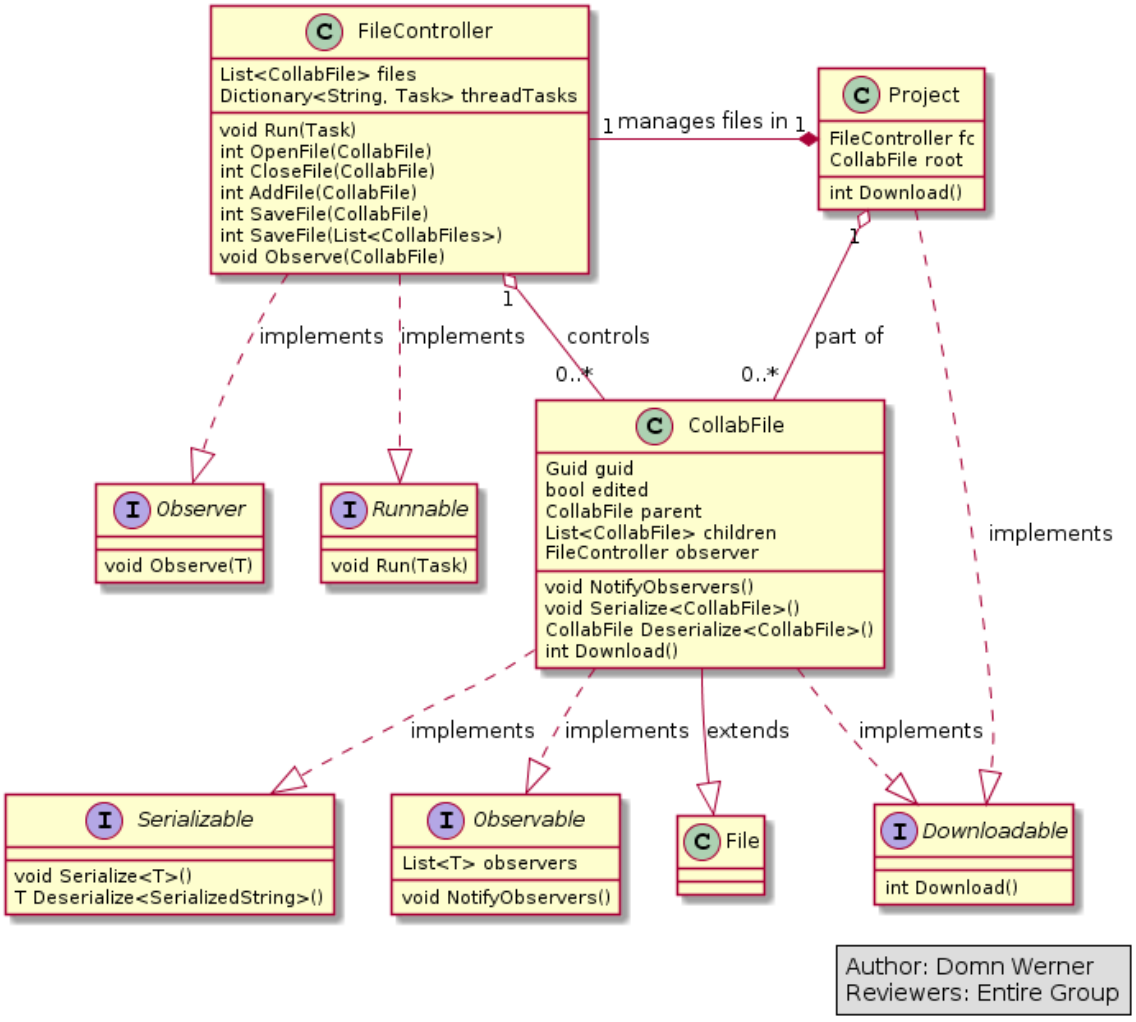


2.2.10 Use Case Diagram 3: Communication



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.

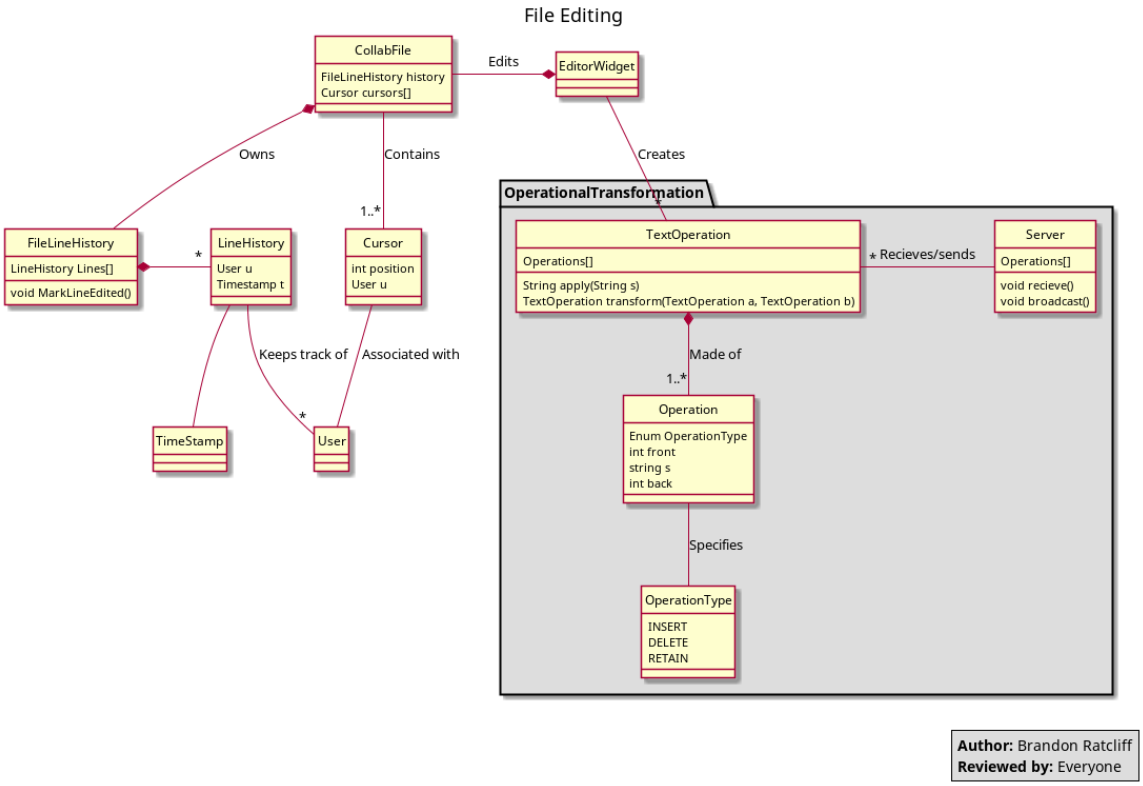
Classes:

- 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.

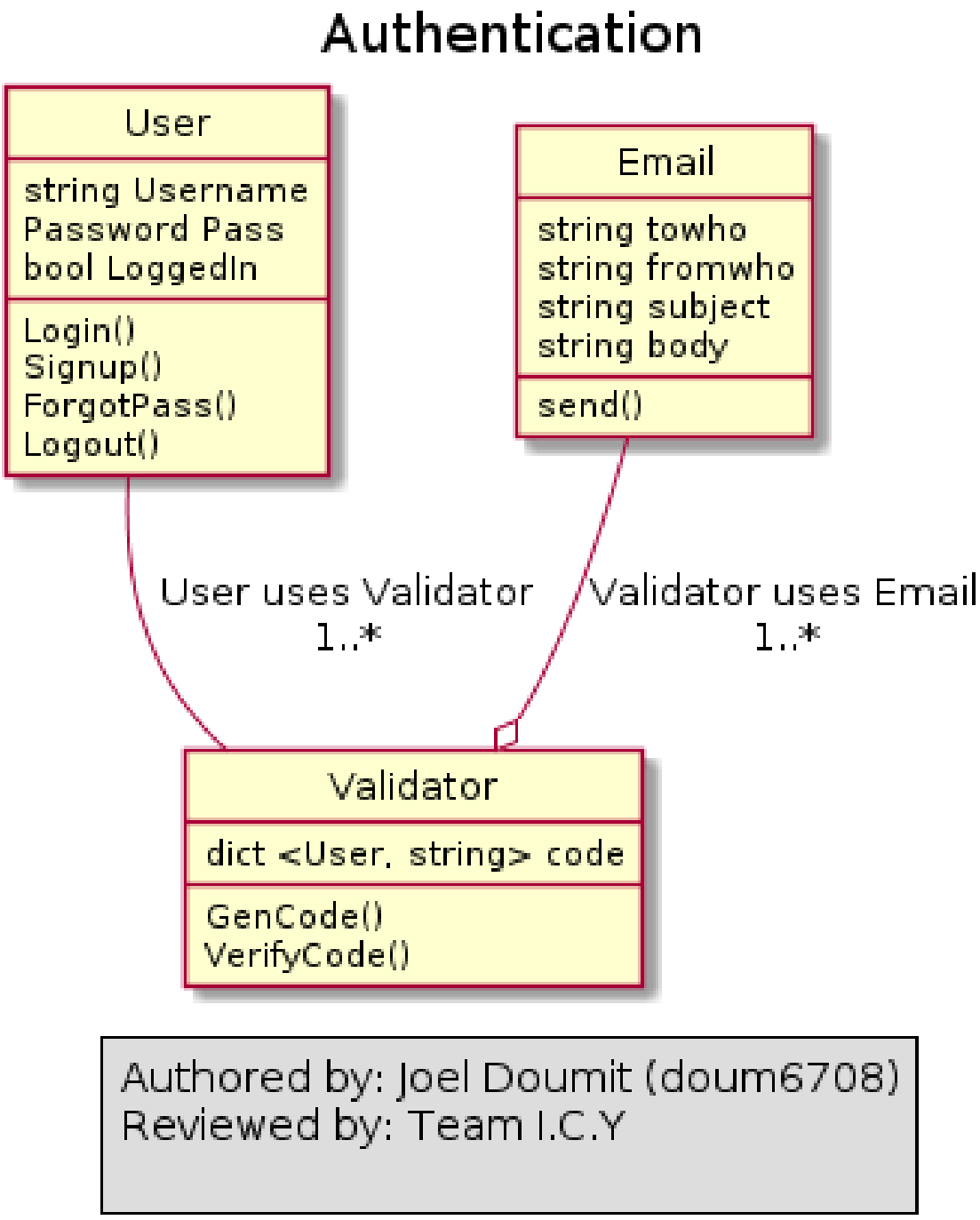
OperationalTransform:

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

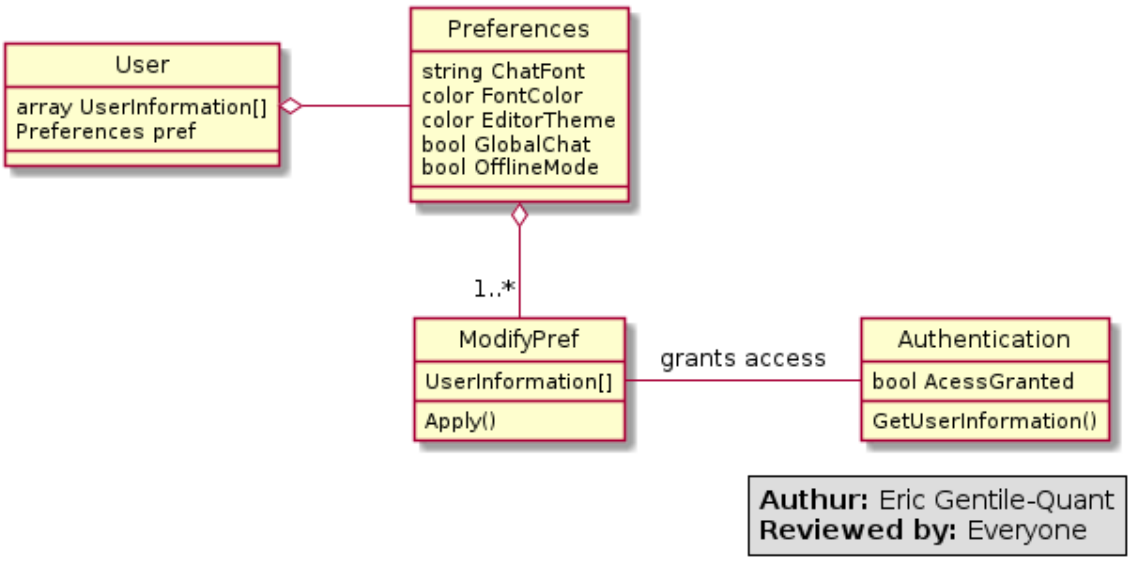


2.3.6 Class Diagram Description 3: Authentication Description

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

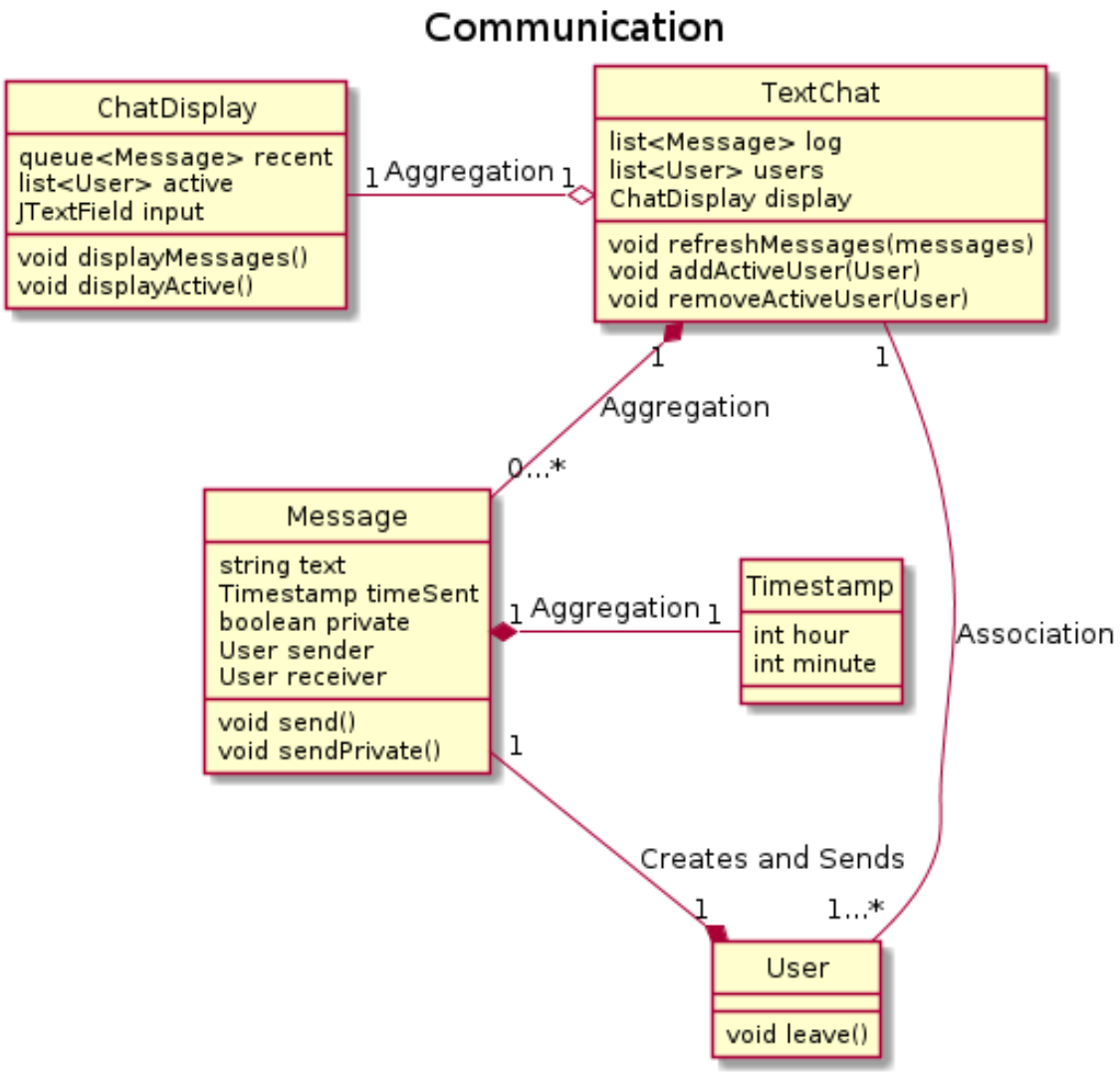


2.3.8 Class Diagram Description 4: User Preferences Description

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



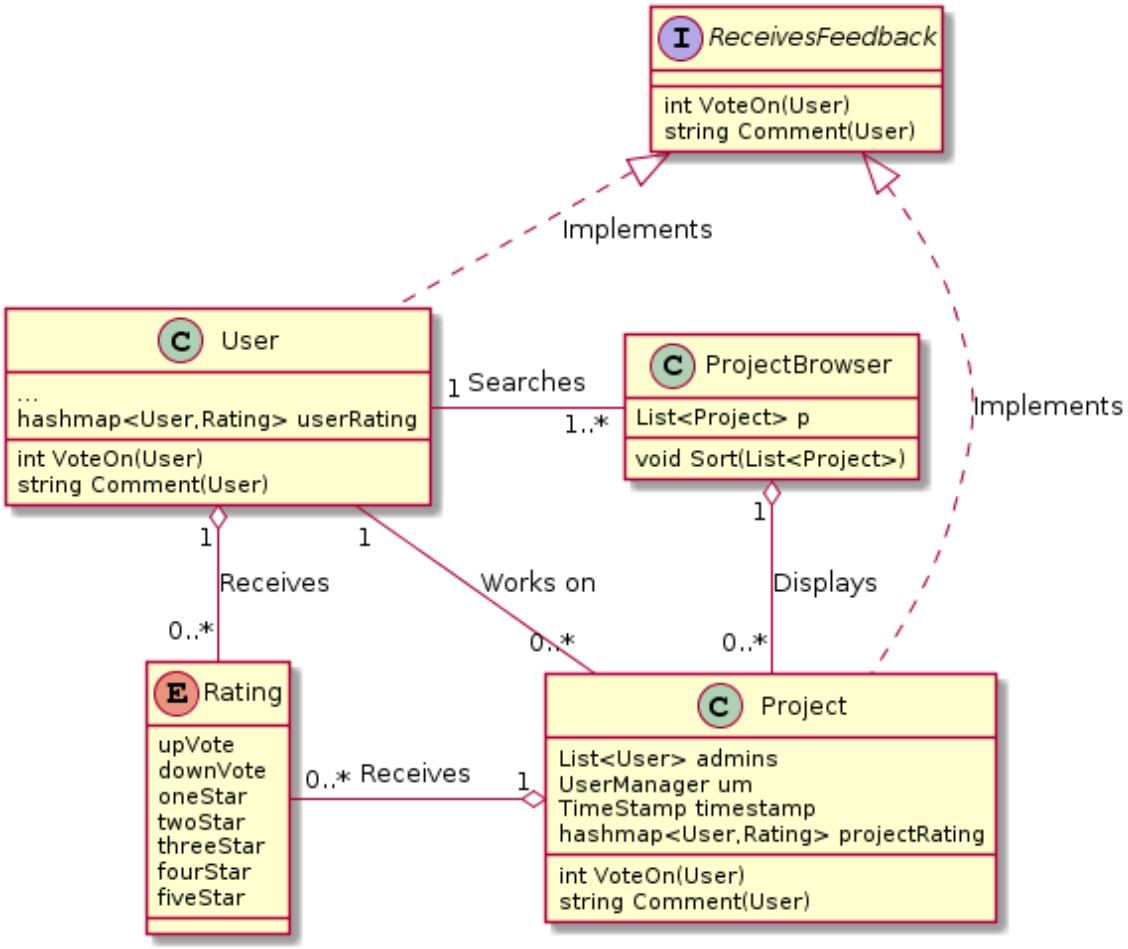
Authored By: Max Welch
Peer Reviewed By: Team I.C.Y.

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



Authored by: Matthew Daniel dani2918
Reviewed by: Entire Group

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.

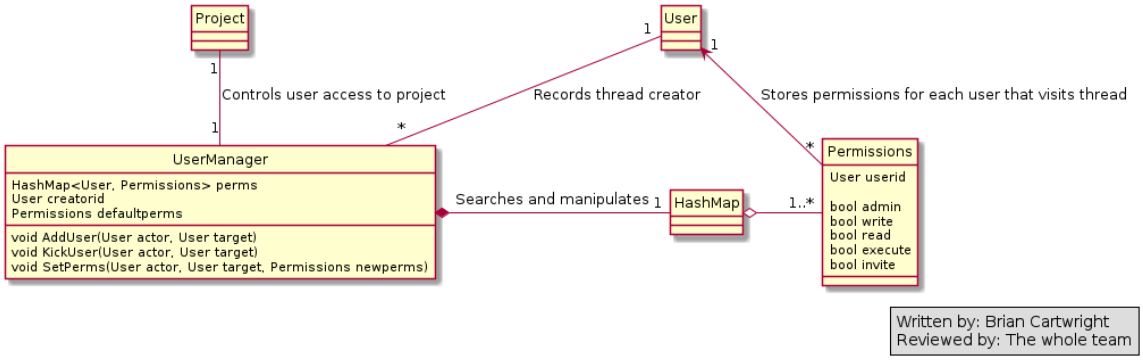
Interfaces:

- 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

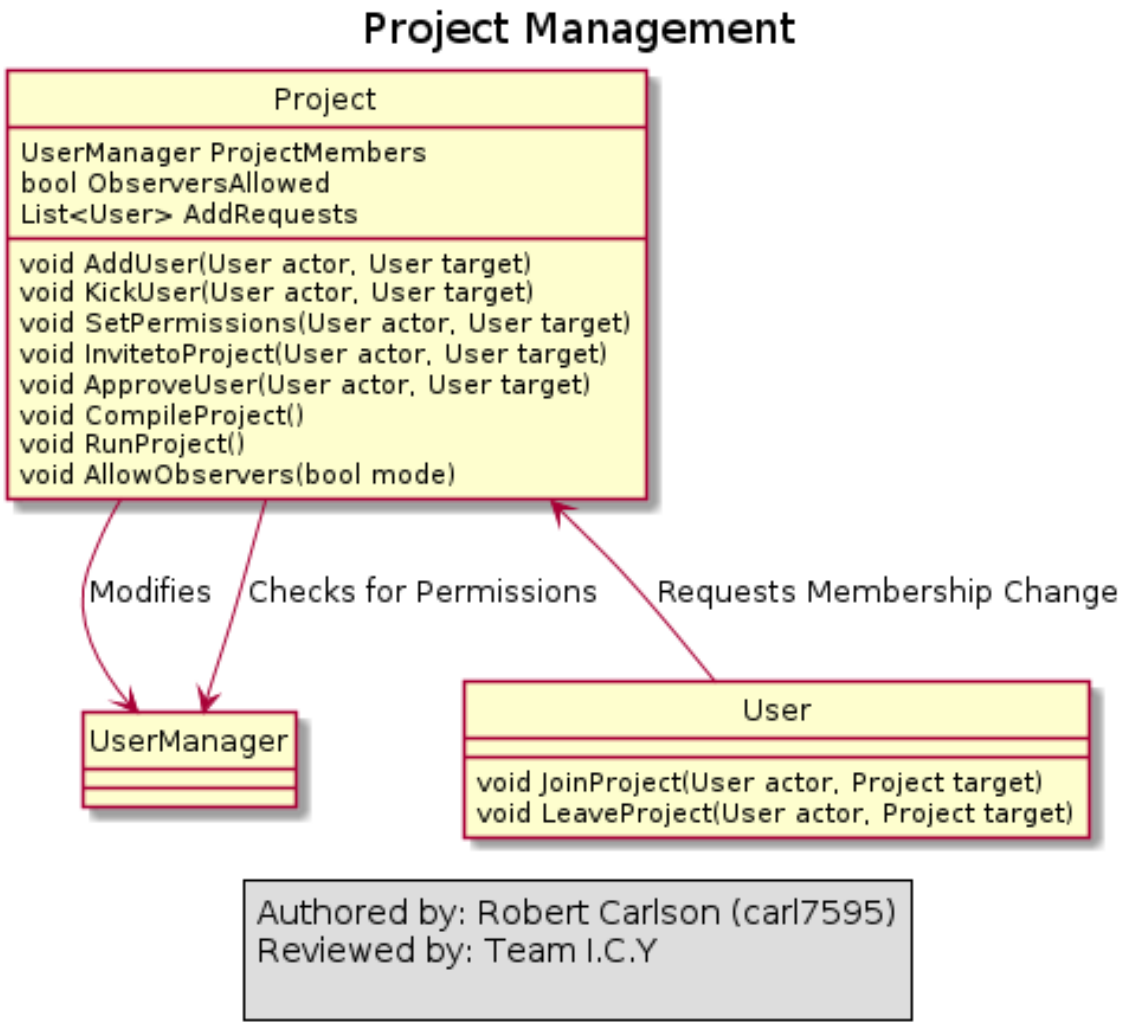


2.3.14 Class Diagram Description 7: Project User Management Description

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



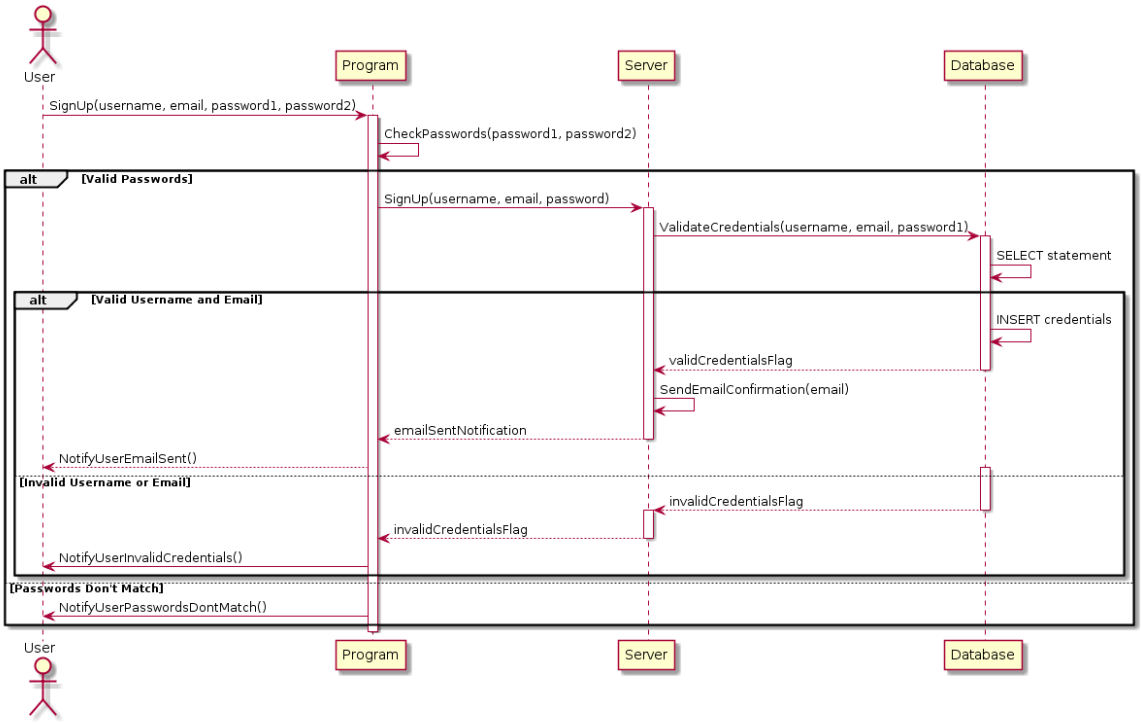
Classes:

- 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.

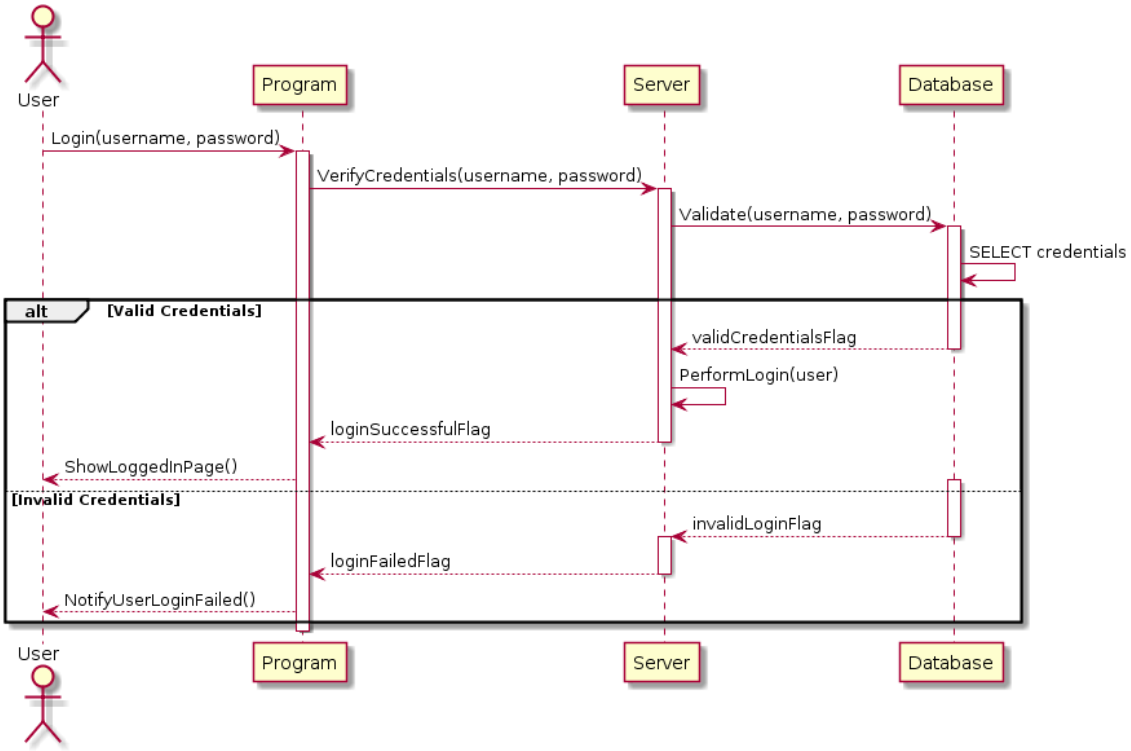
2.4 SEQUENCE DIAGRAMS

2.4.1 Authentication Feature 1: Sign Up Sequence Diagram



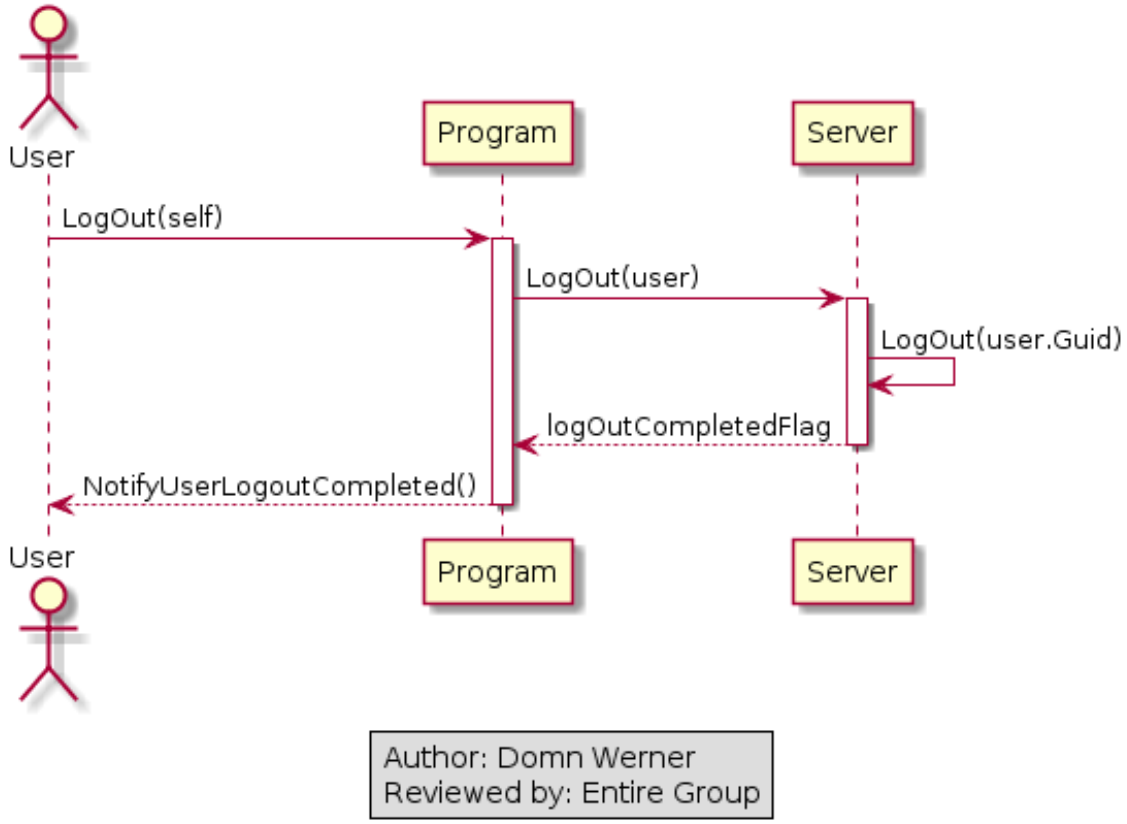
Author: Domn Werner
Reviewed by: Entire Group

2.4.2 Authentication Feature 2: Log In Sequence Diagram

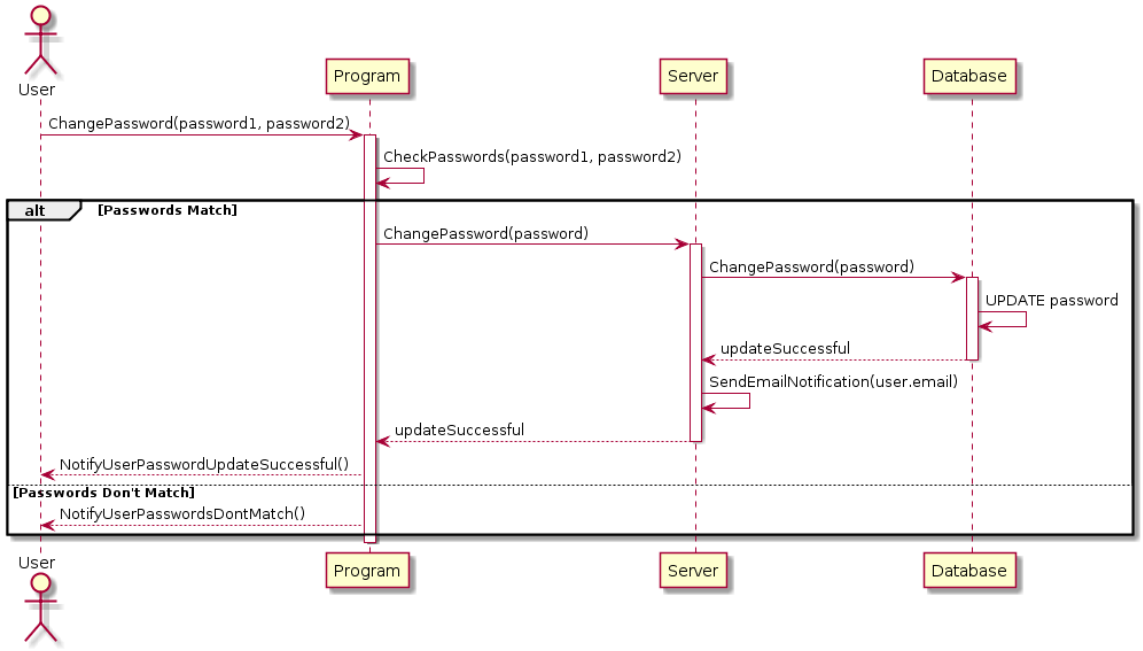


Author: Domn Werner
Reviewed by: Entire Group

2.4.3 Authentication Feature 3: Log Out Sequence Diagram

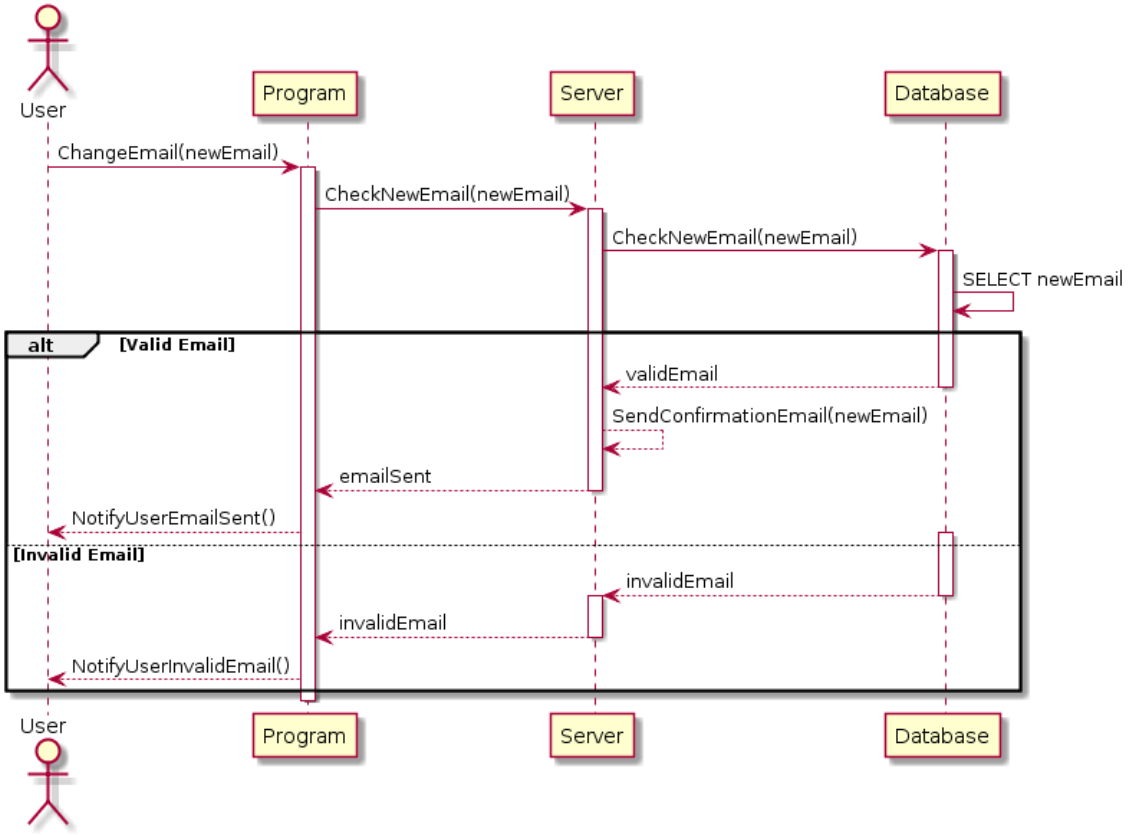


2.4.4 Authentication Feature 4: Change Password Sequence Diagram



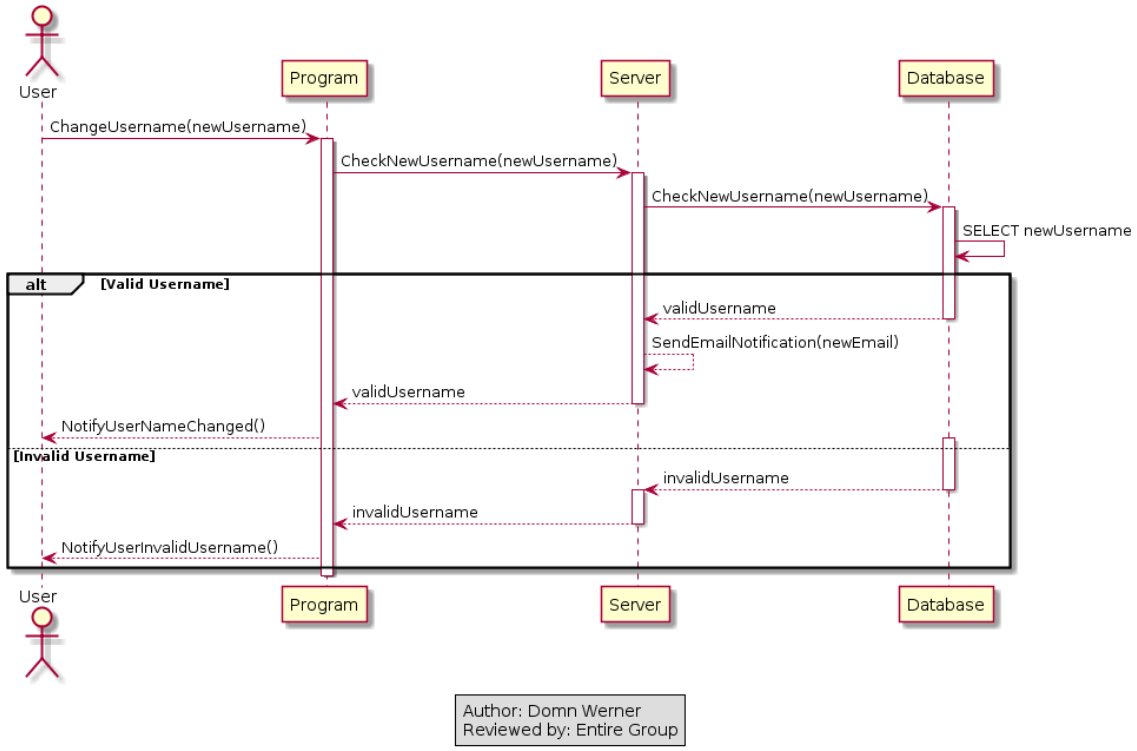
Author: Domn Werner
Reviewed by: Entire Group

2.4.5 Authentication Feature 5: Change Email Sequence Diagram

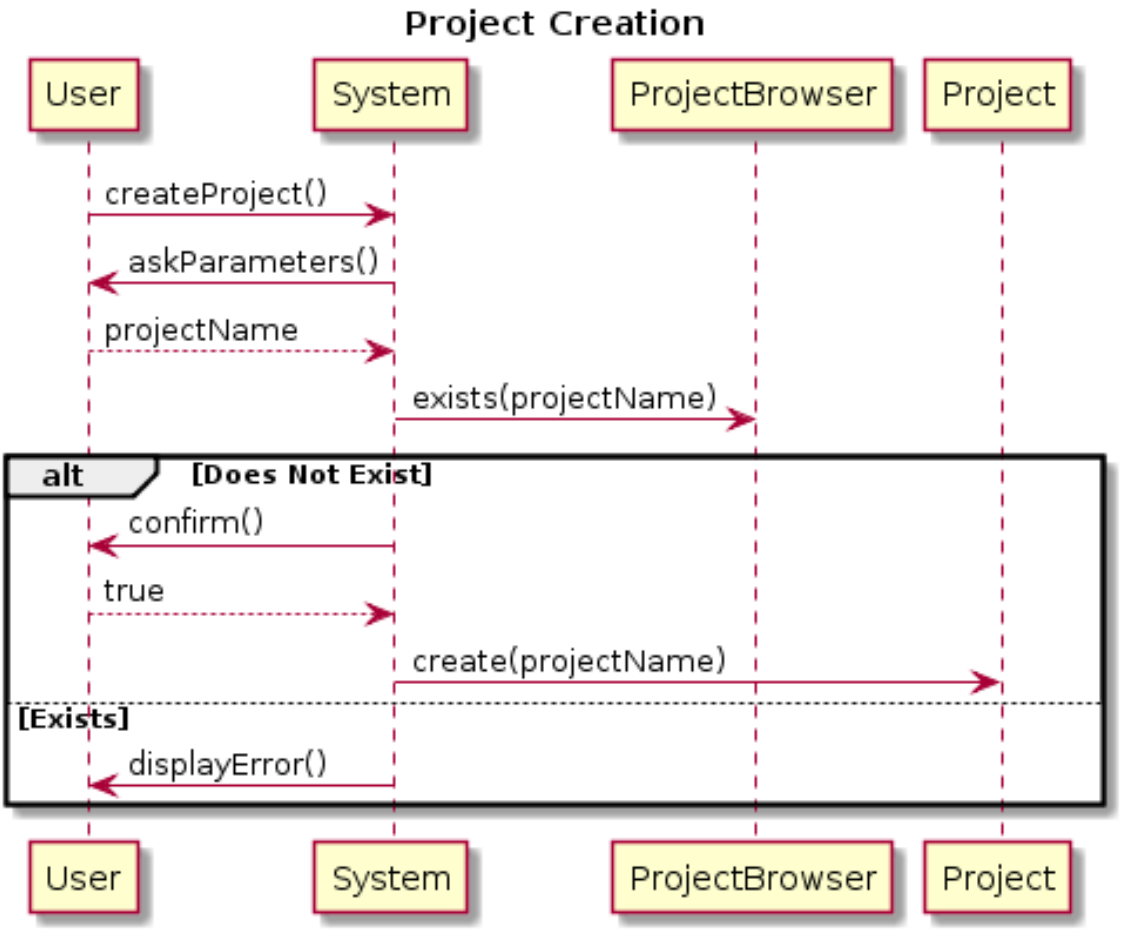


Author: Domn Werner
Reviewed by: Entire Group

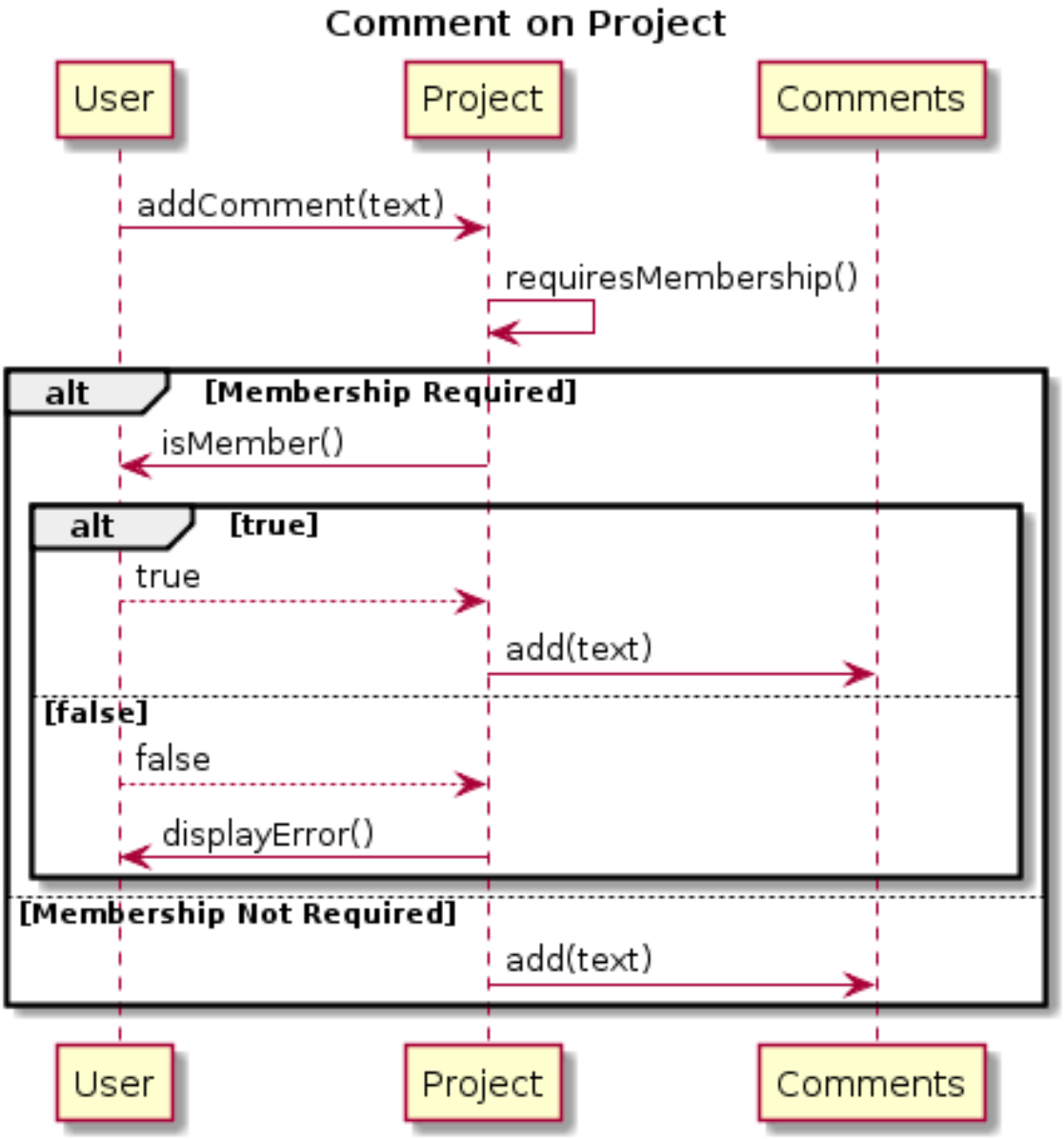
2.4.6 Authentication Feature 6: Change Username Sequence Diagram



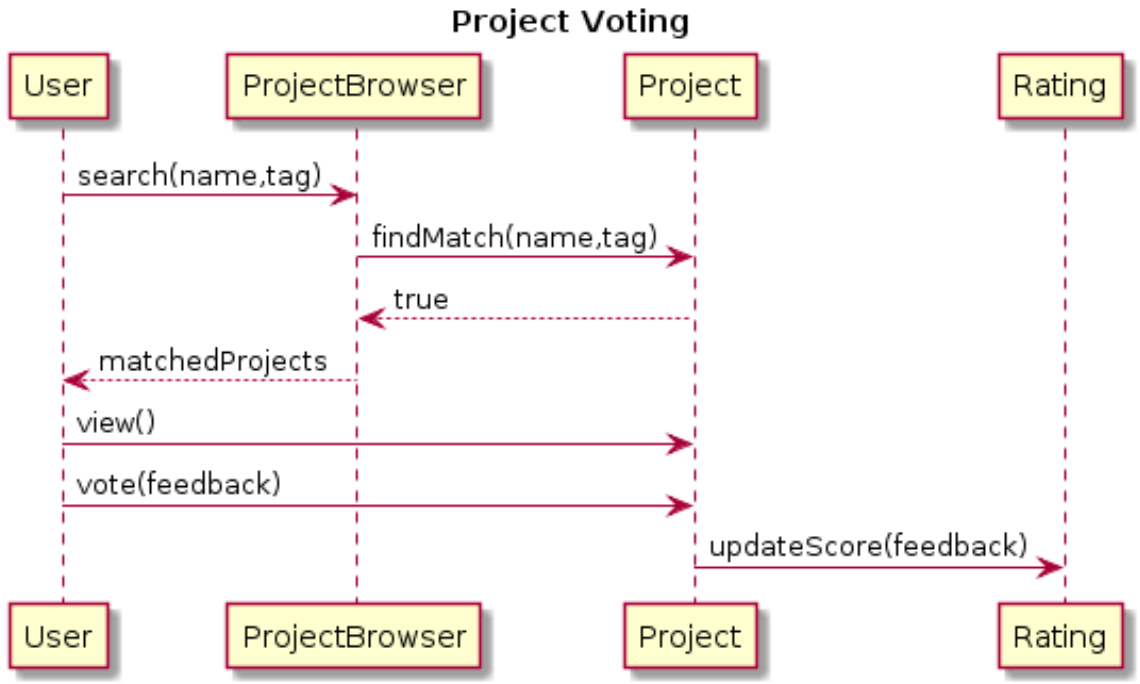
2.4.7 Project Browsing Feature 2: Project Creation Sequence Diagram



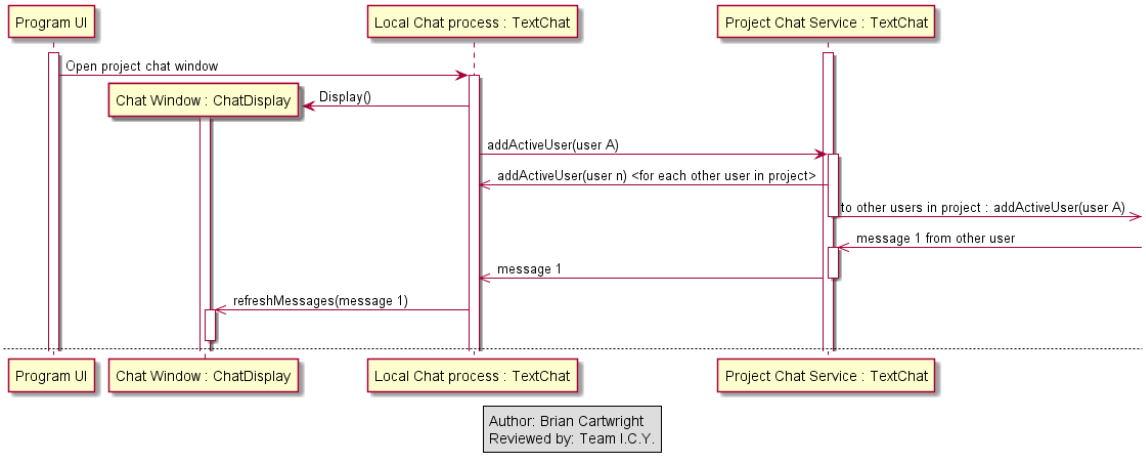
2.4.8 Project Browsing Feature 3: Project Commenting Sequence Diagram



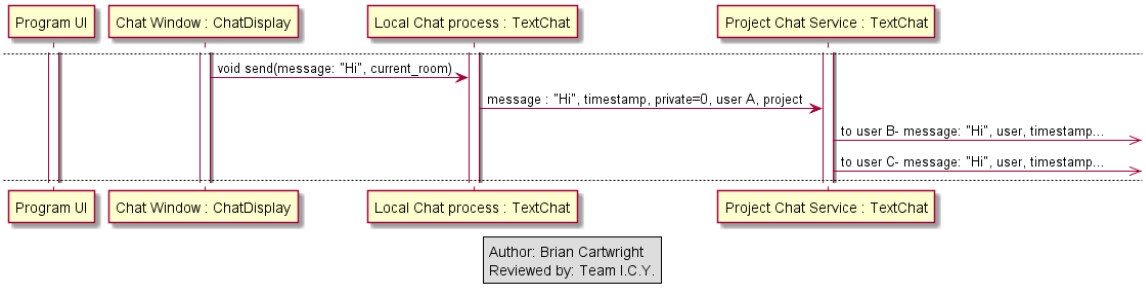
2.4.9 Project Browsing Feature 4: Project Voting Sequence Diagram



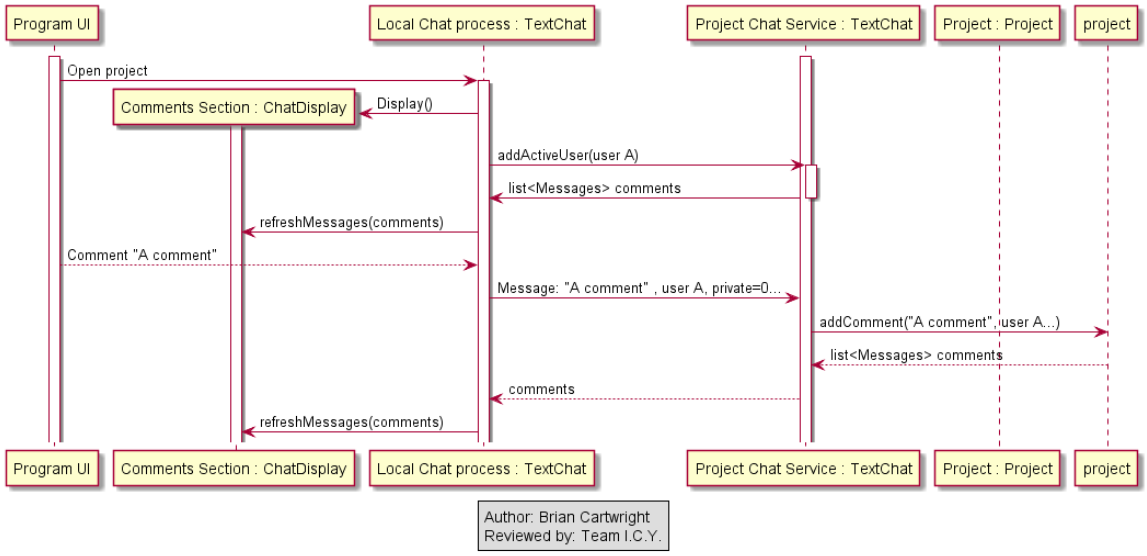
2.4.10 Communication Feature 1: Read Project Chat Sequence Diagram



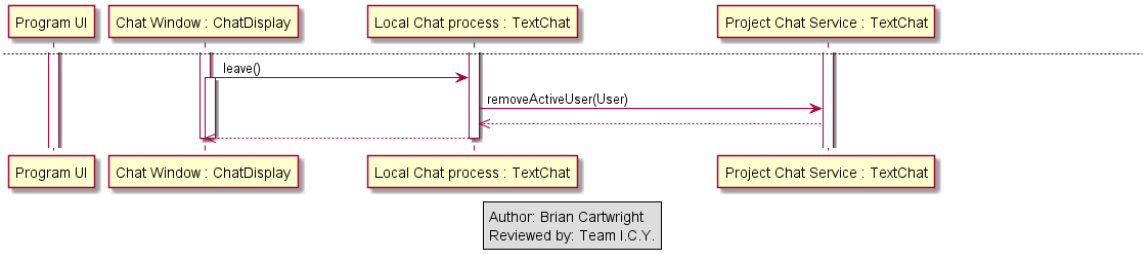
2.4.11 Feature 2: Write to Project Chat Sequence Diagram



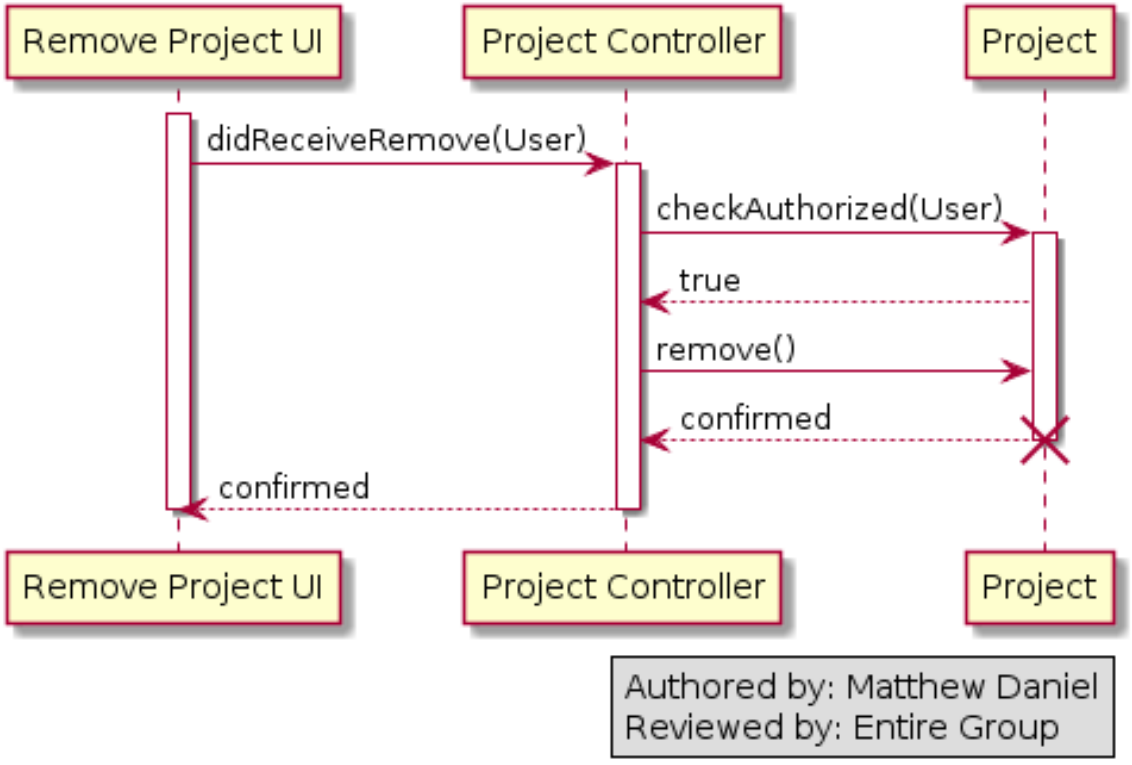
2.4.12 Feature 4: Comment on Project Sequence Diagram



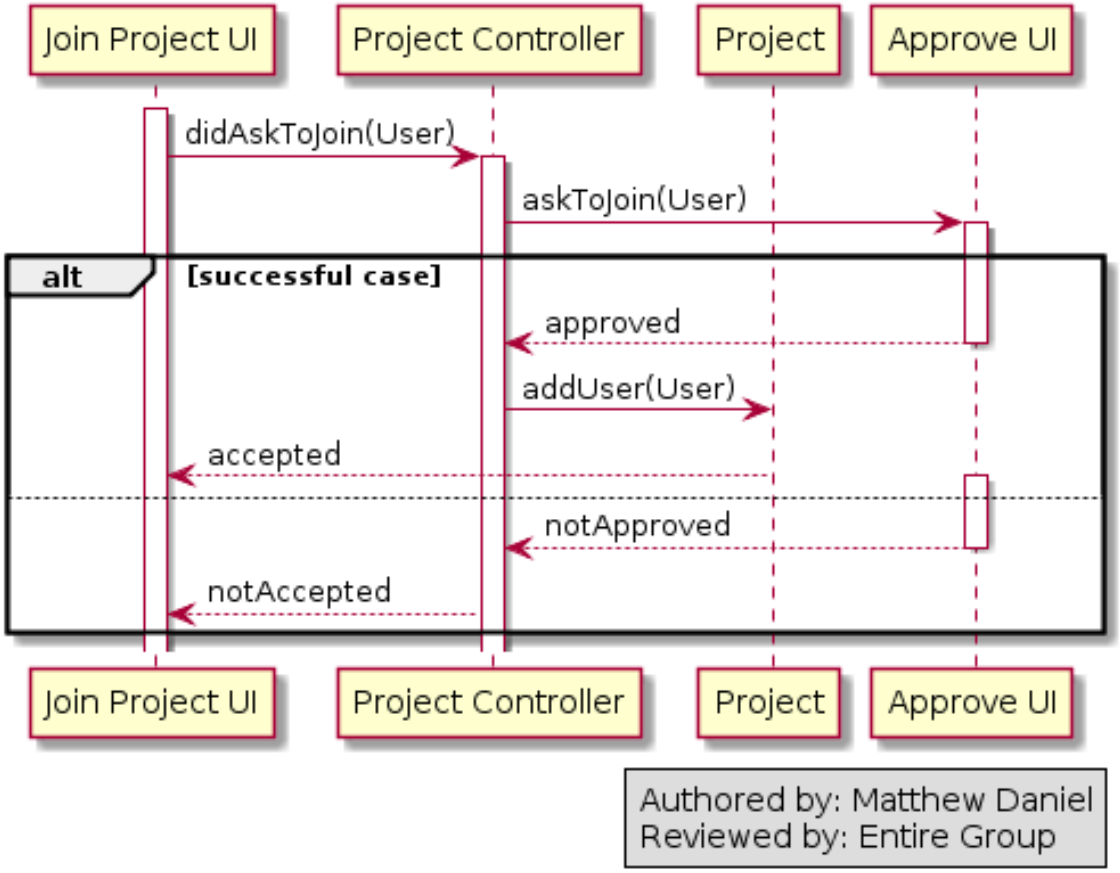
2.4.13 Feature 5: Close chat Sequence Diagram



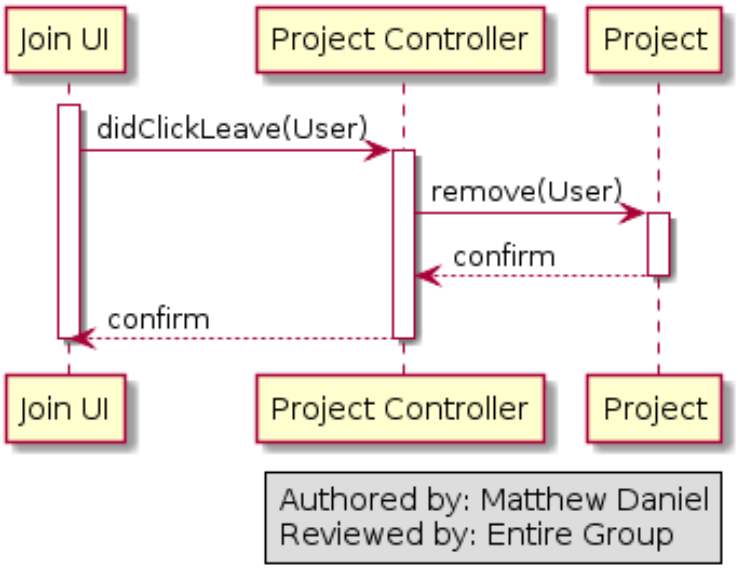
2.4.14 Project Management Feature 3: Delete Project Sequence Diagram (dani2918)



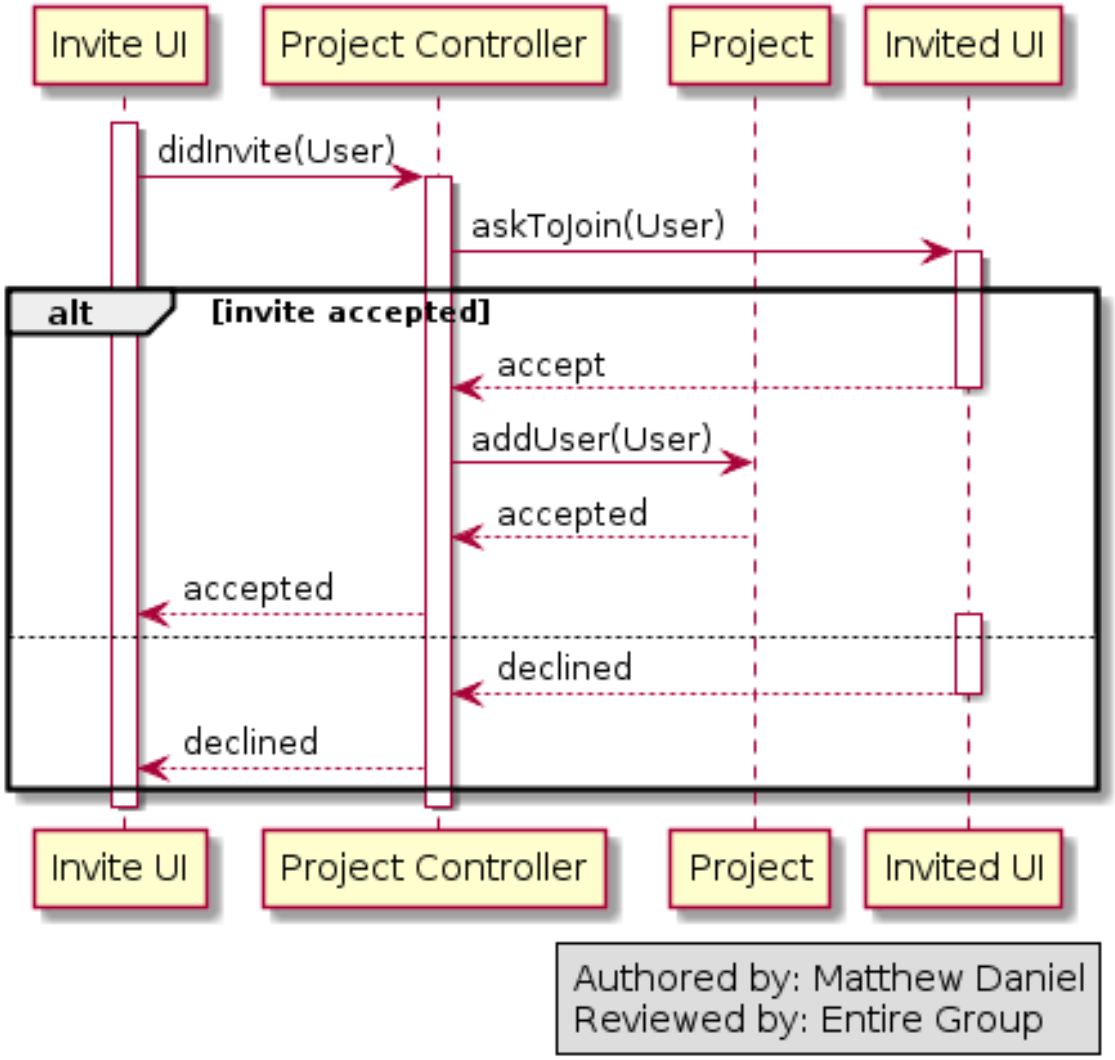
2.4.15 Project Management Feature 4/5: Request/Manage Request to Join Project Sequence Diagram (dani2918)



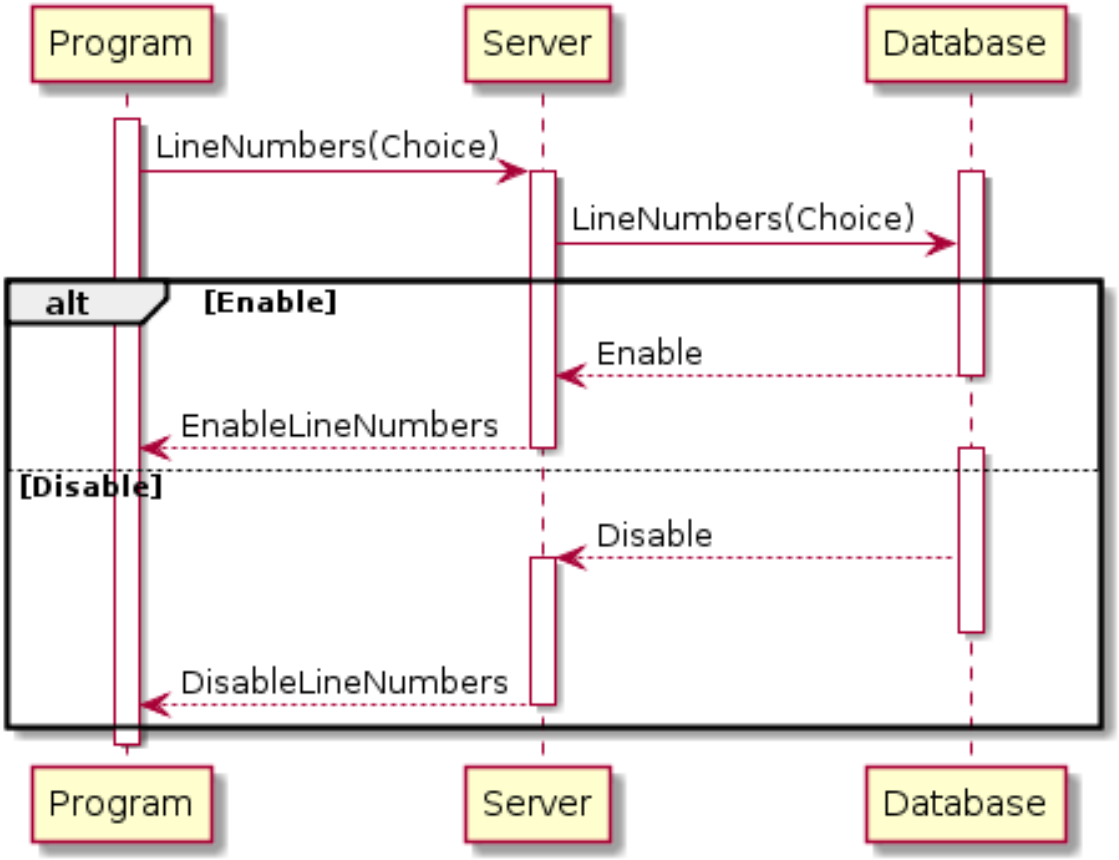
2.4.16 Project Management Feature 6: Leave Project Sequence Diagram (dani2918)



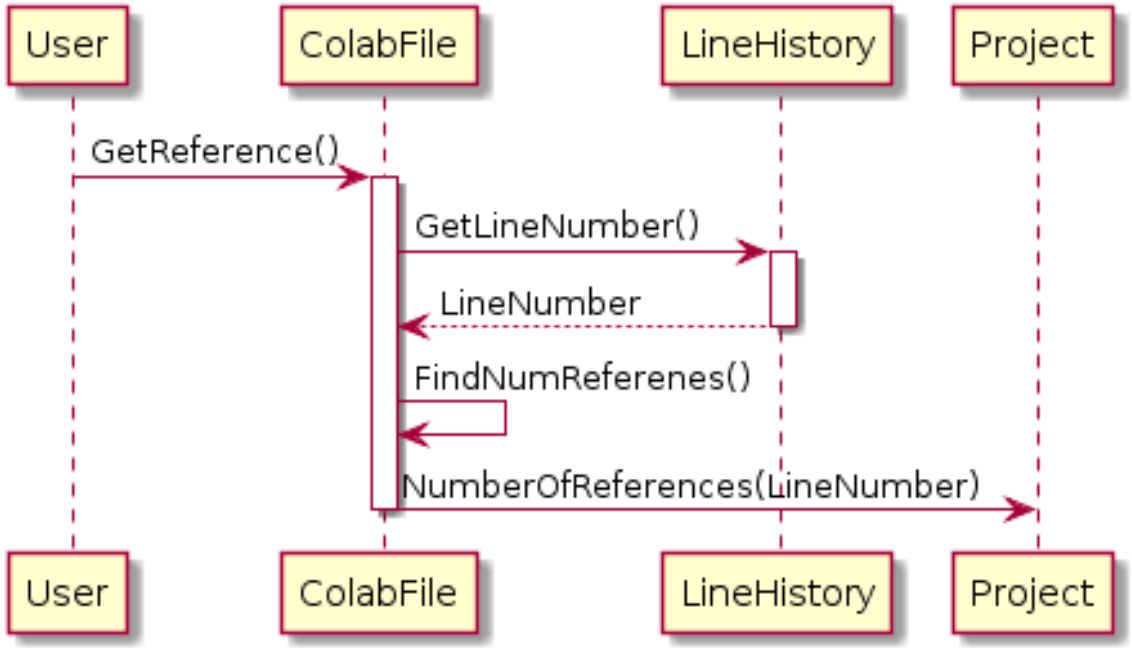
2.4.17 Project Management Feature 7/8: Invite/Respond to Project Invite Sequence Diagram (dani2918)



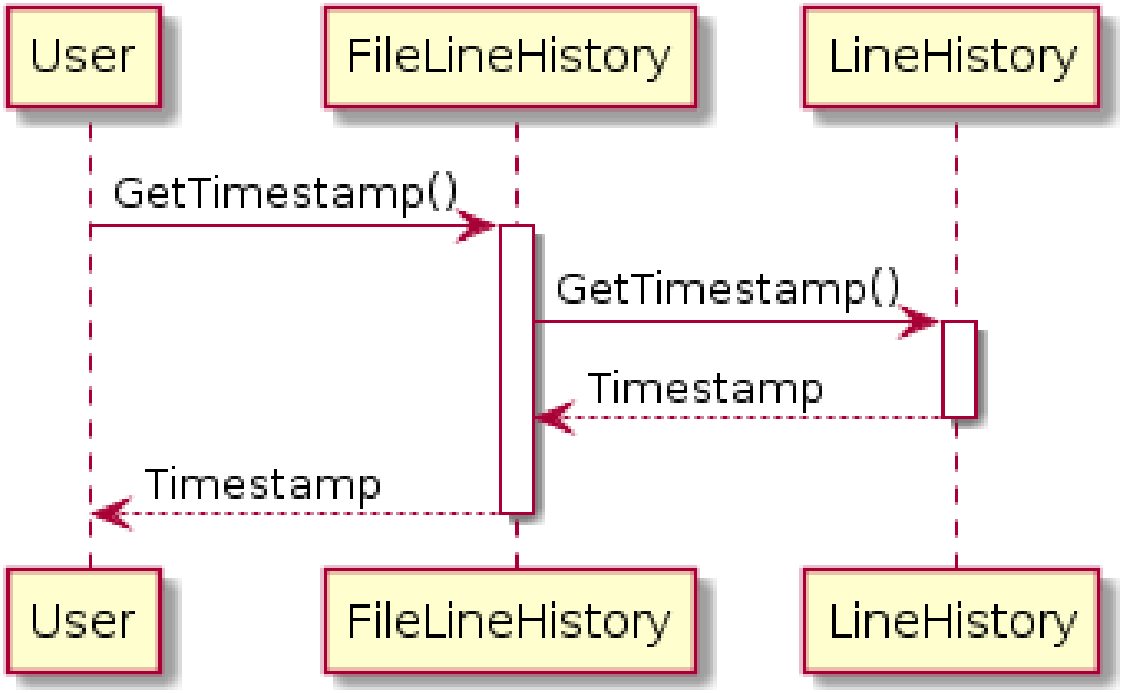
2.4.18 File Editing Feature 1: View Line Numbers Sequence Diagram



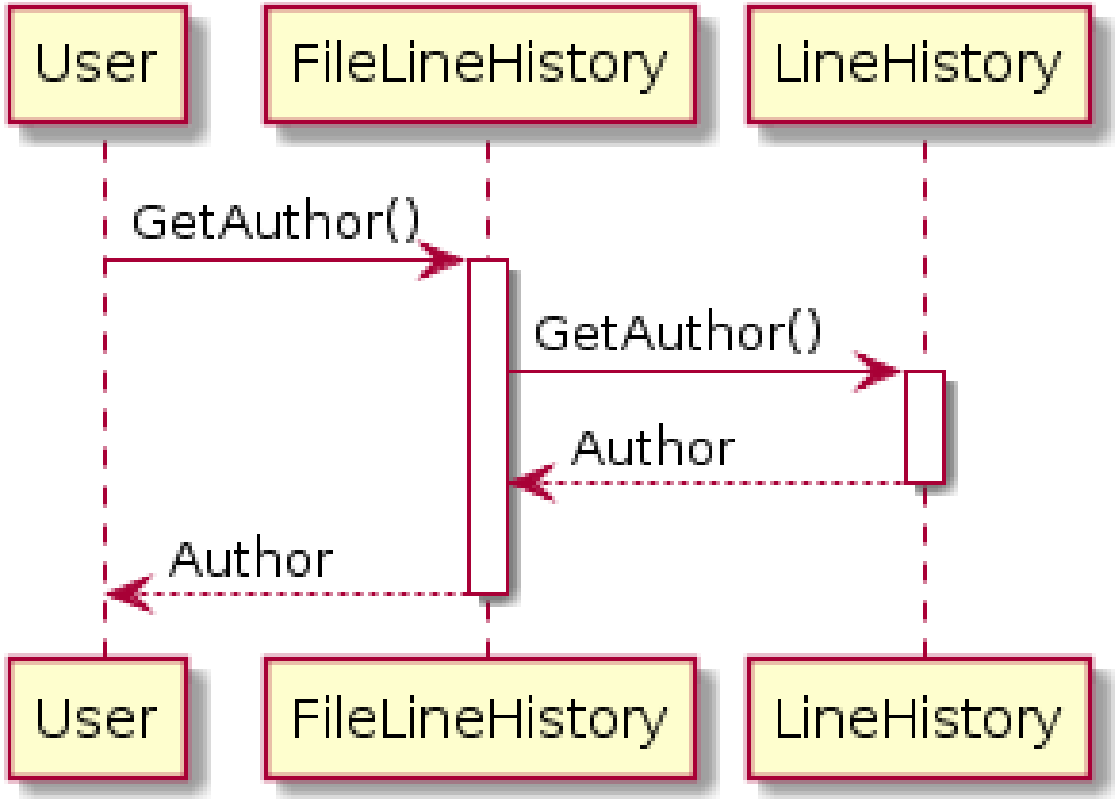
2.4.19 File Editing Feature 2: View References Sequence Diagram



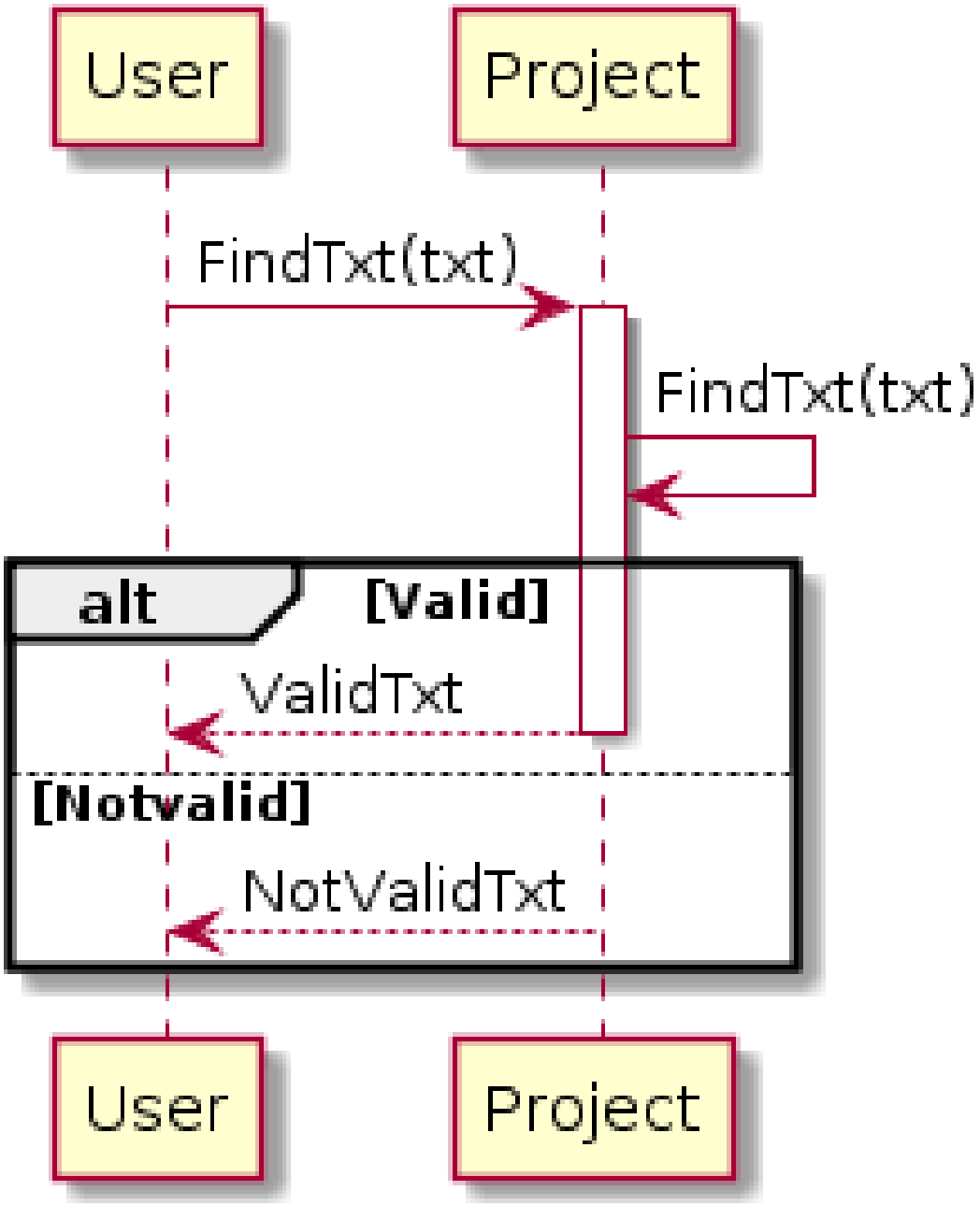
2.4.20 File Editing Feature 3: View Dates Sequence Diagram



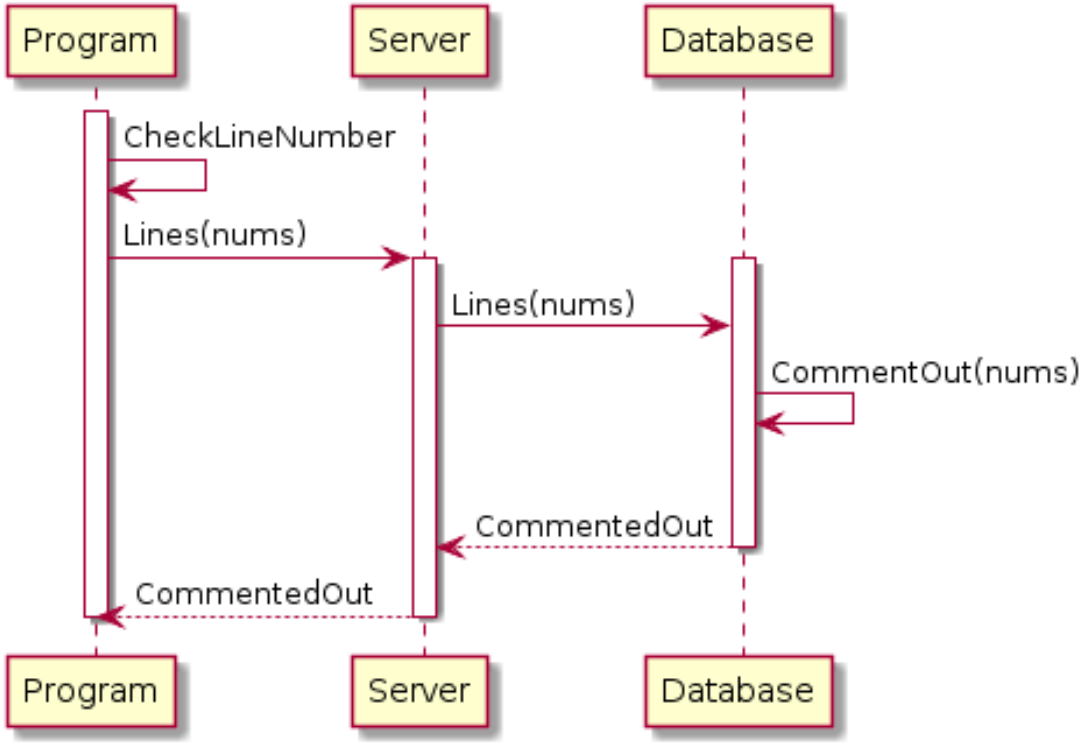
2.4.21 File Editing Feature 4: View Author Sequence Diagram



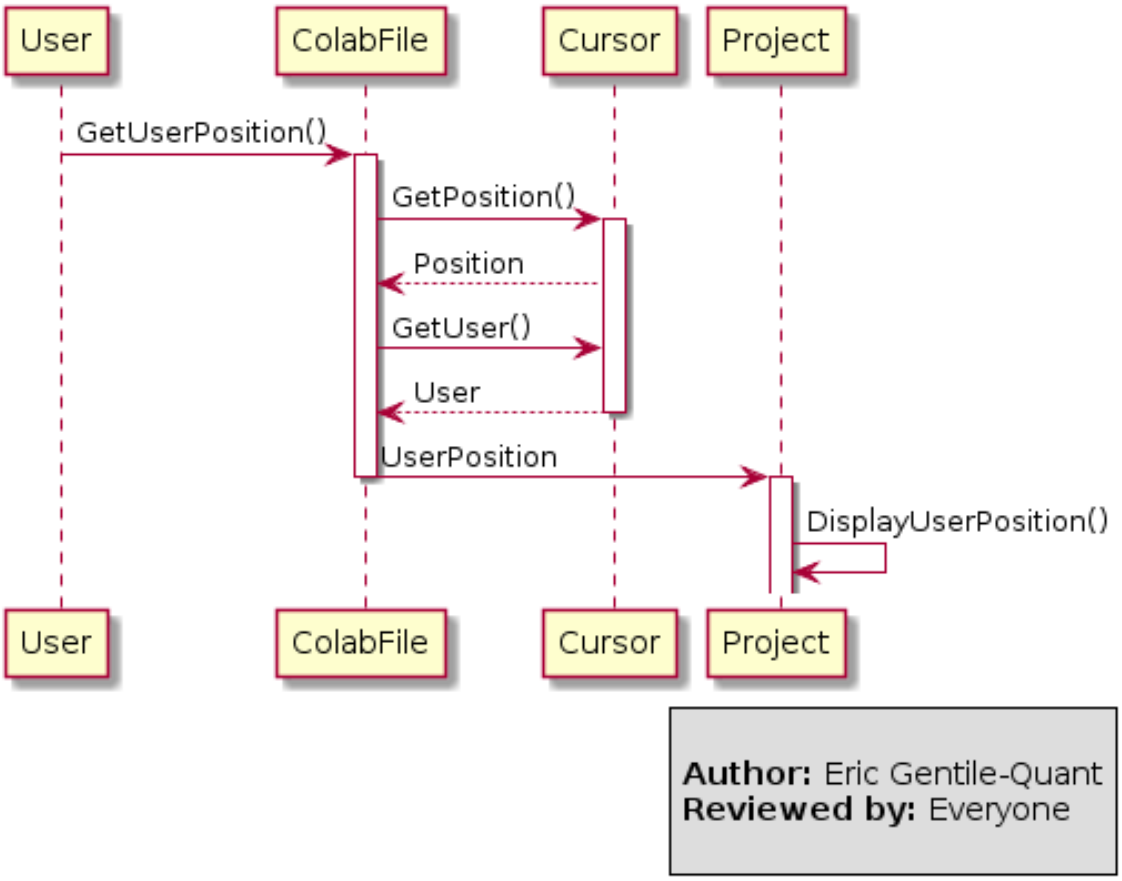
2.4.22 File Editing Feature 6: Find Sequence Diagram



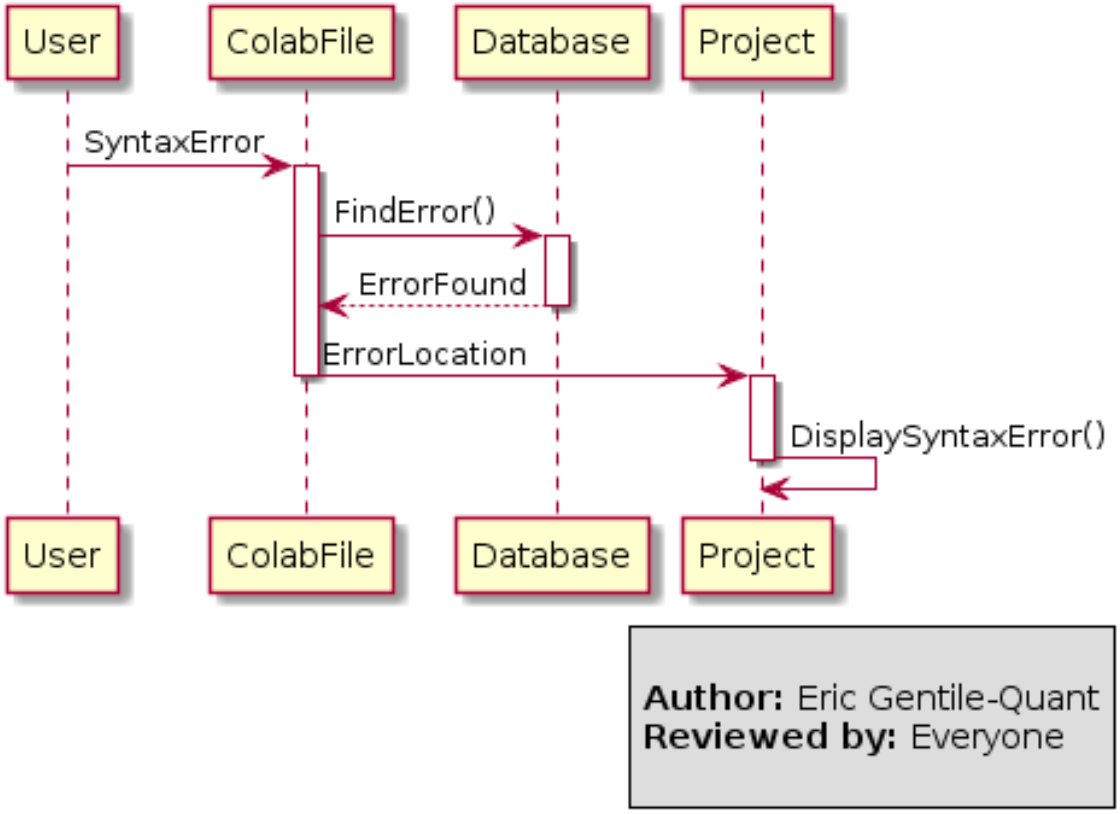
2.4.23 File Editing Feature 7: Comment Section Sequence Diagram



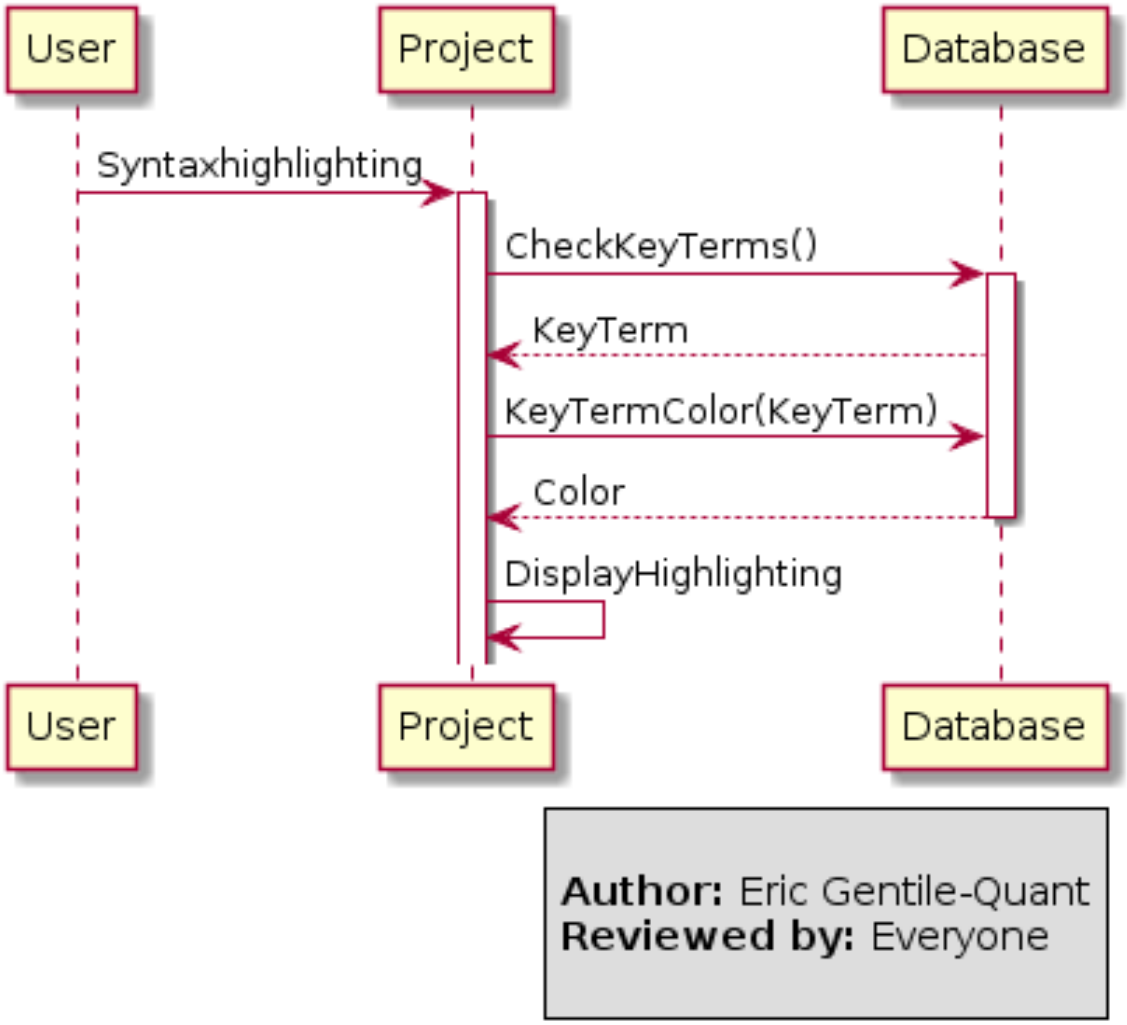
2.4.24 File Editing Feature 8: Display Typing User Sequence Diagram



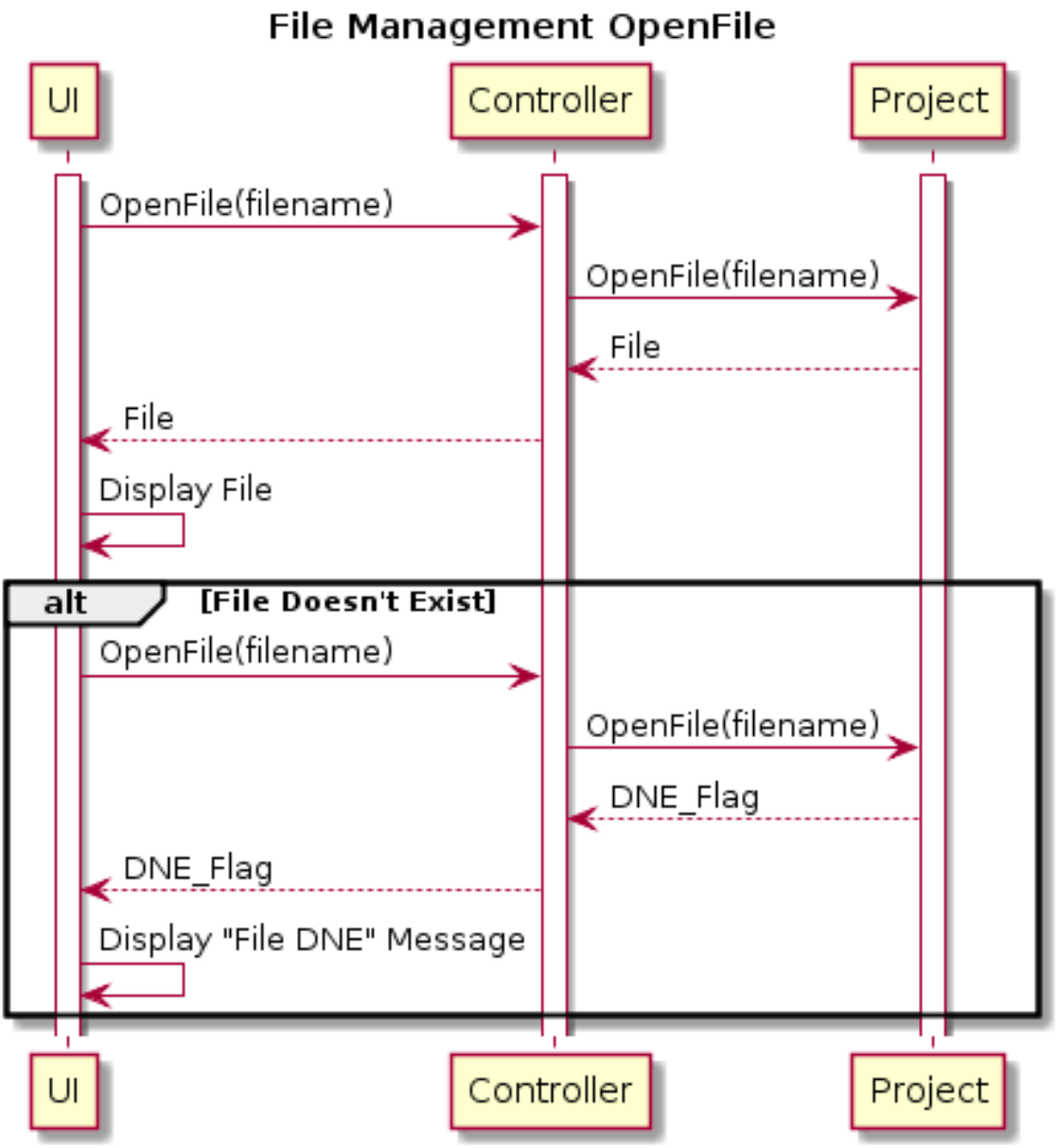
2.4.25 File Editing Feature 9: Display Syntax Errors Sequence Diagram



2.4.26 File Editing Feature 10: Display Syntax Highlighting Sequence Diagram

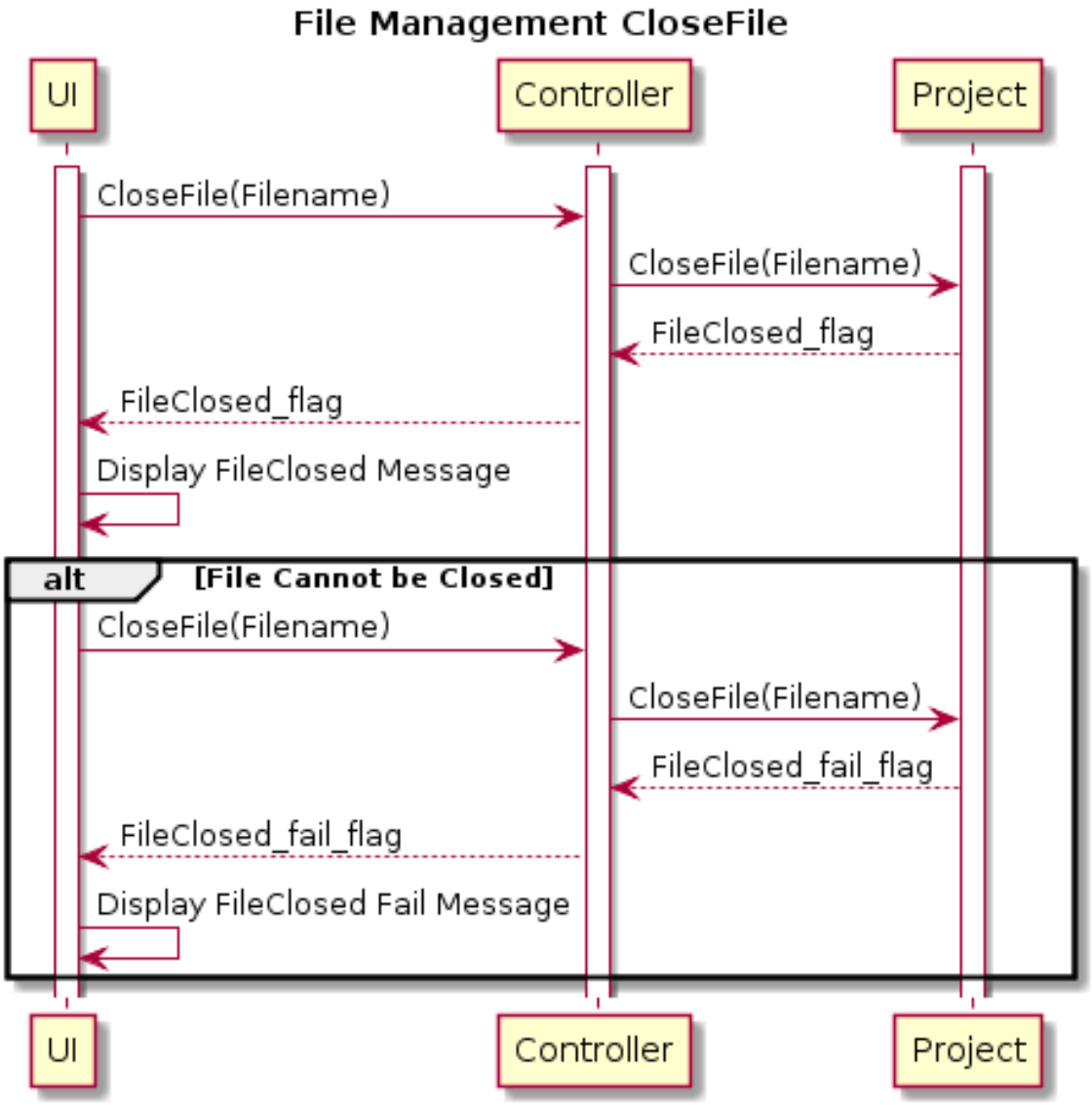


2.4.27 File Management Feature 1: Open File Sequence Diagram



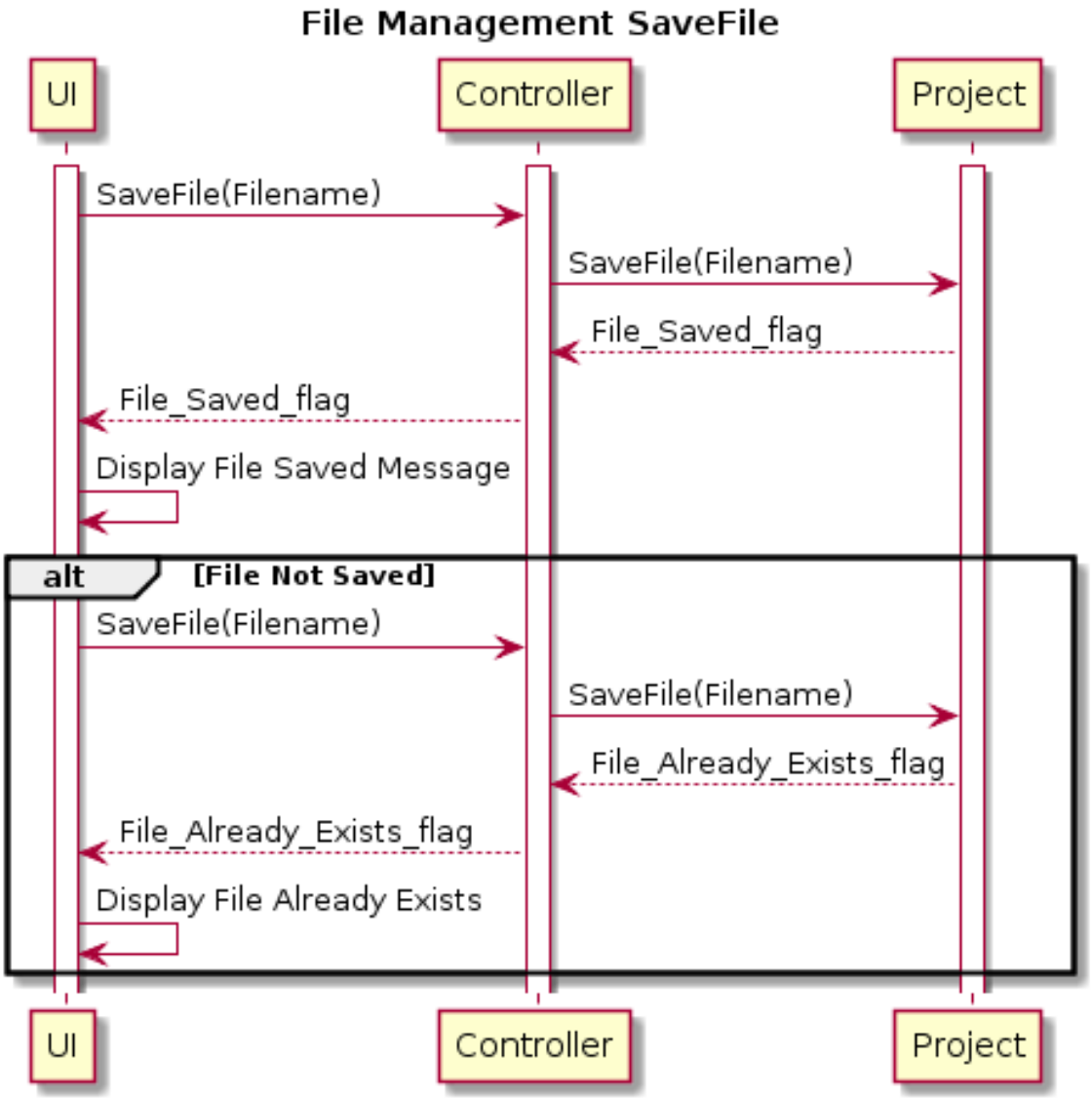
Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

2.4.28 File Management Feature 2: Close File Sequence Diagram



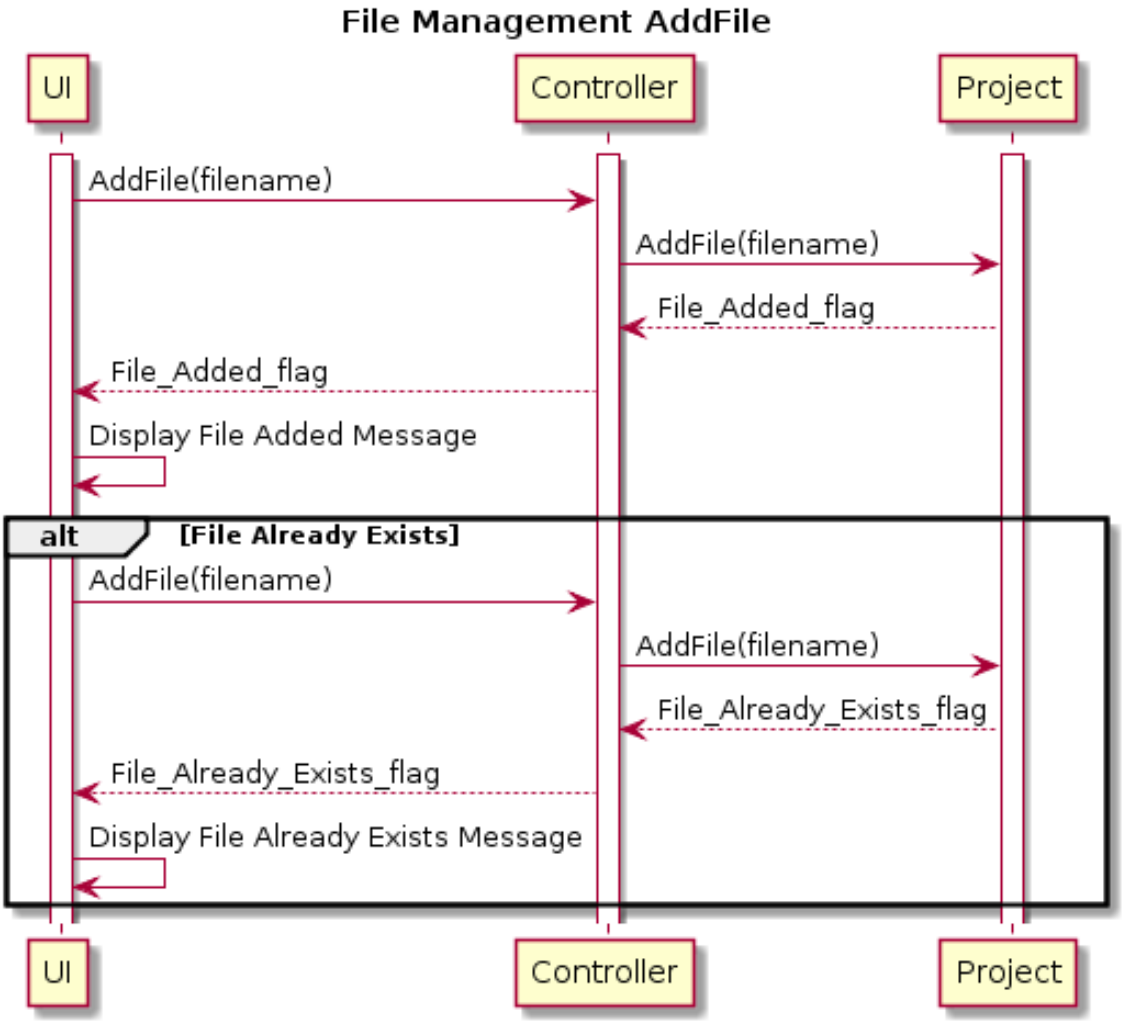
Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

2.4.29 File Management Feature 3: Save File Sequence Diagram



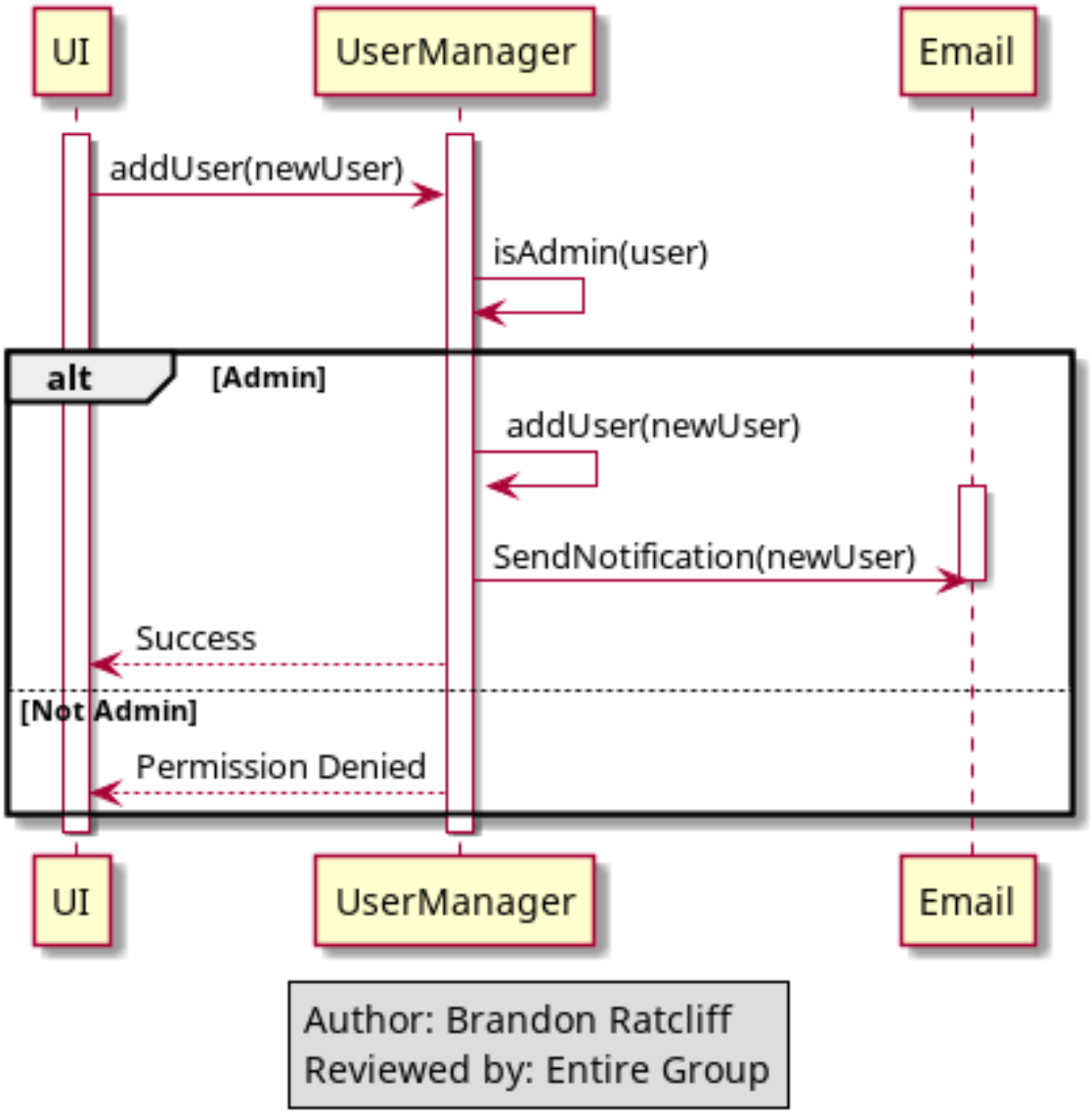
Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

2.4.30 File Management Feature 4: Add File Sequence Diagram

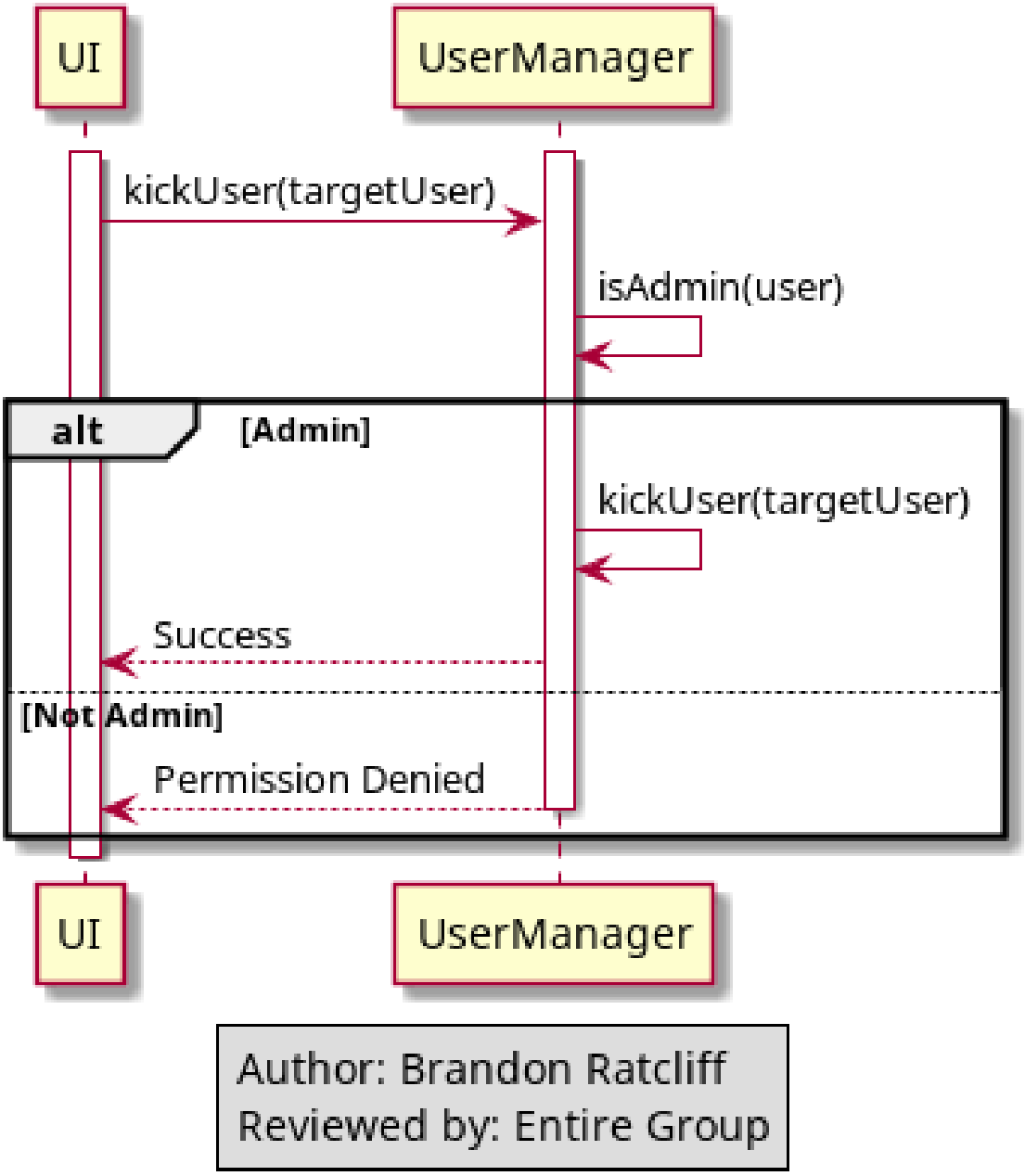


Authored by: Joel Doumit (doum6708)
Reviewed by: Team I.C.Y

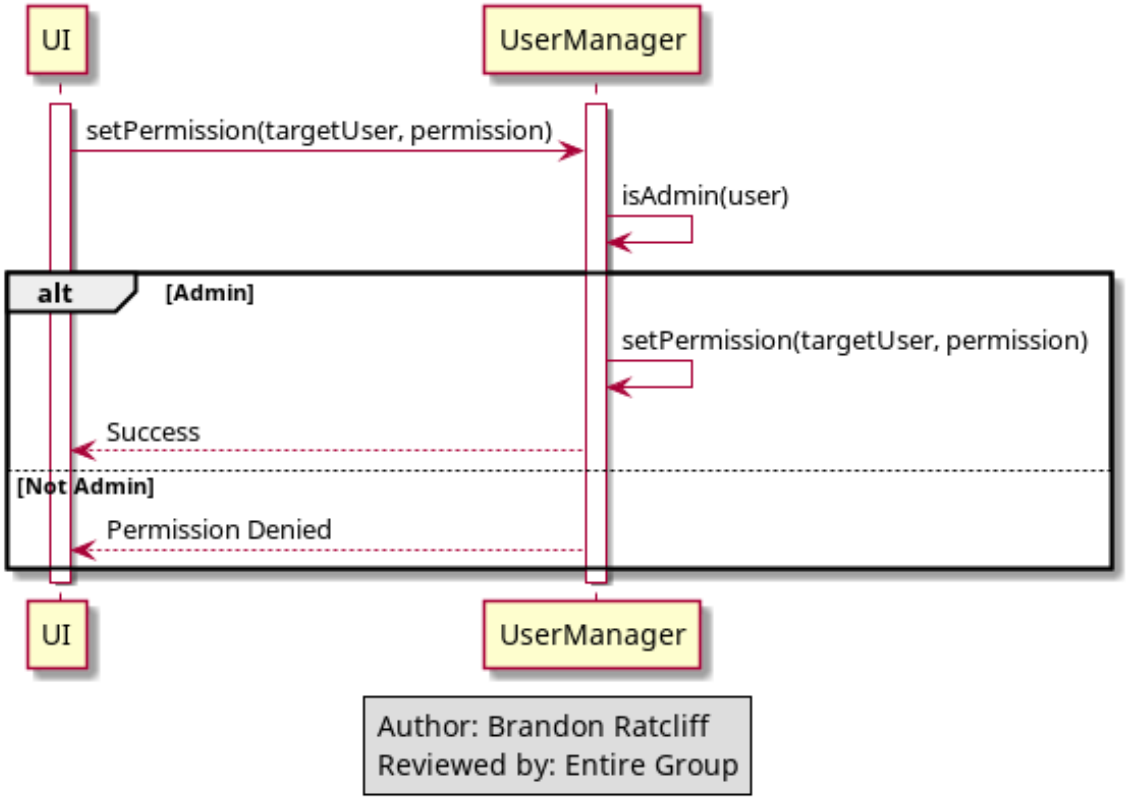
2.4.31 Project User Management Feature 1: Add User Sequence Diagram



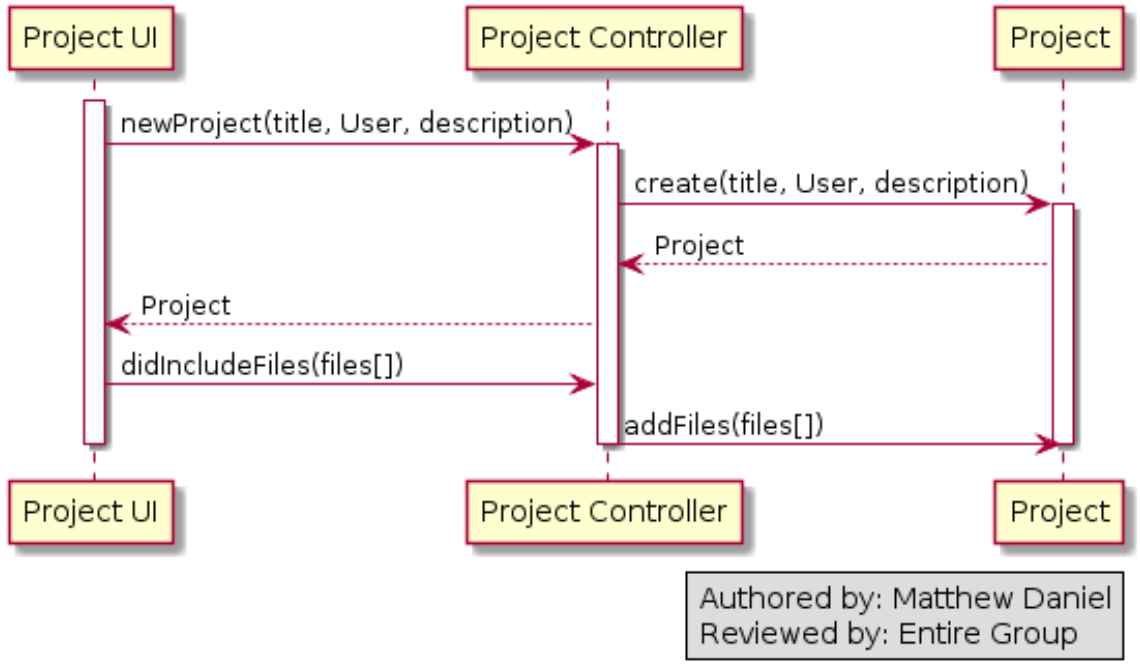
2.4.32 Project User Management Feature 2: Kick User Sequence Diagram



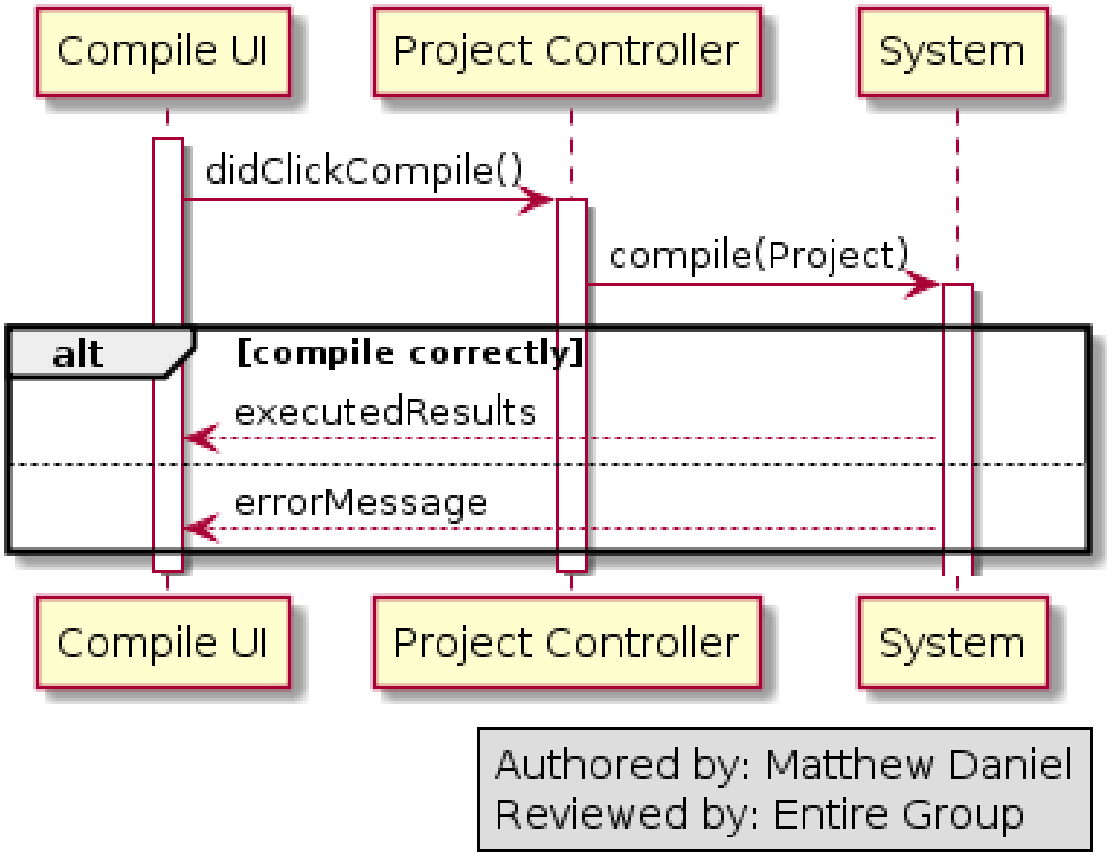
2.4.33 Project User Management Feature 3: Set User Permissions Sequence Diagram



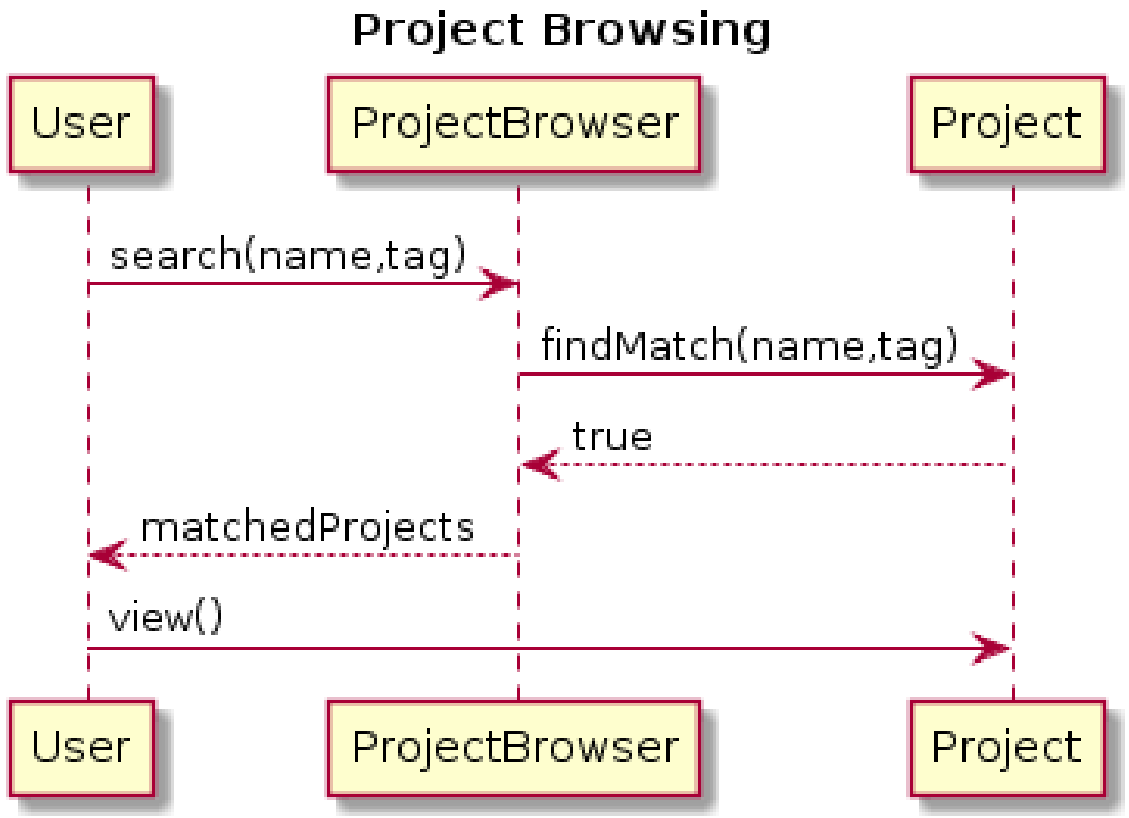
2.4.34 Project Management Feature 2: Create project Sequence Diagram(dani2918)



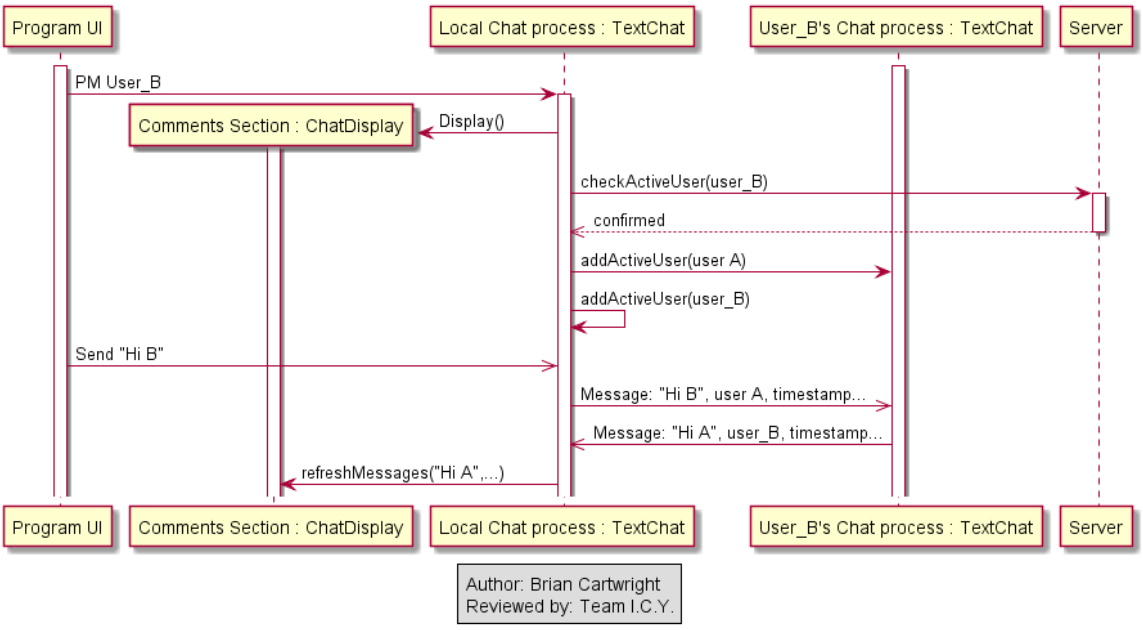
2.4.35 Project Management Feature 1: Compile and Execute Project Sequence Diagram (dani2918)



2.4.36 Project Browsing Feature 1: Project Browsing Sequence Diagram

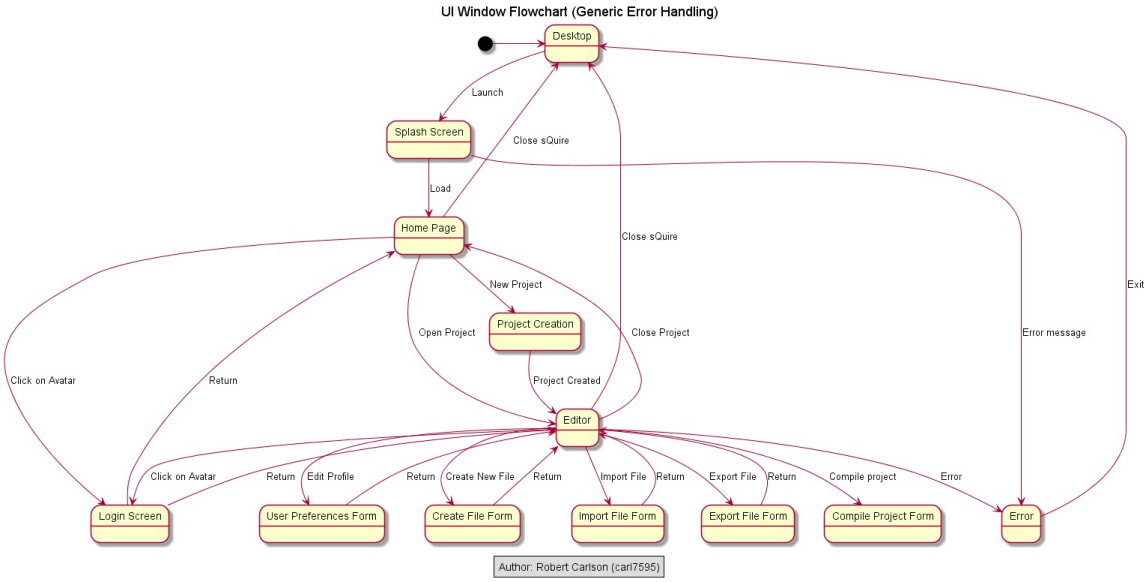


2.4.37 Feature 3: Message User by Name Sequence Diagram



2.5 USER INTERFACE DIAGRAMS

2.5.1 UI Window Flowchart



3 IMPLEMENTATION

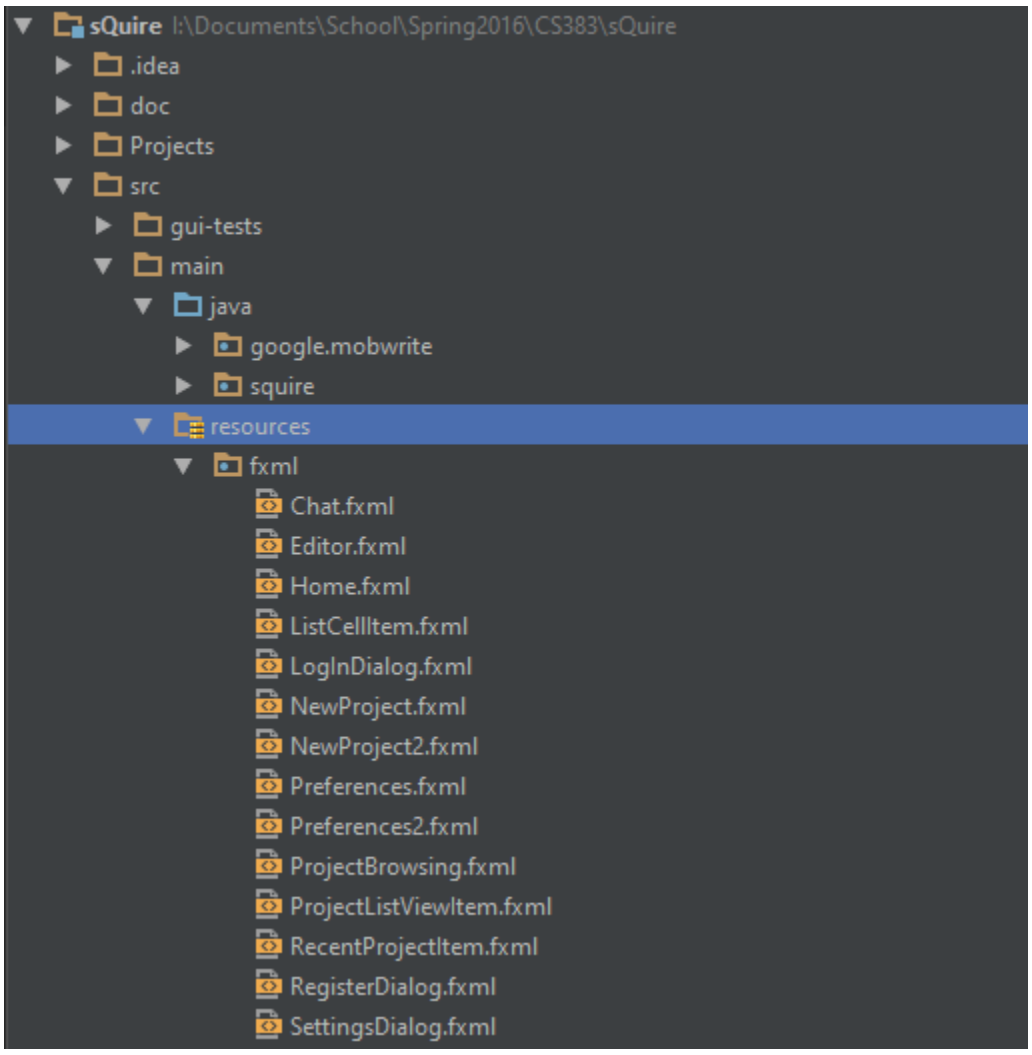
3.1 User Interface

3.1.1 JavaFX Framework

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:



3.2 Server

3.2.1 MobWrite

3.2.2 Chat

3.2.3 Database

3.2.4 Request Handler

3.3 Client

3.3.1 CodeArea

3.3.2 Response Handler

3.4 Deployment

3.4.1 GitHub Repository

3.4.2 Dependencies

4 TESTING

5 METRICS