



# WACHEMO UNIVERSITY

SCHOOL OF COMPUTING AND INFORMATICS

DEPARTMENT OF SOFTWARE ENGINEERING

PROJECT DOCUMENTATION

**Course title:** SOFTWARE ENGINEERING TOOLS AND PRACTICE

**Project title:** Smart Task Controller

## Group members

### Group Members

### id\_no

- |                   |         |
|-------------------|---------|
| 1. Abay Tefera    | 1500569 |
| 2. Diyana Abeba   | 1501839 |
| 3. Nahom Maru     | 1404600 |
| 4. Samuel Birhanu | 1501241 |
| 5. Solomon Tsehay | 1501288 |

**Submitted to:** Feysa Kedir

**Date:** april 25, 2025

## **Table Content**

### **Chapter 1: Requirement Analysis (Use Case)**

1.1 Functional Overview-----	3
1.2 Identifying the Actors-----	3
1.3 Use Case Diagram Components-----	4
1.4 Detailed Use Case Flow-----	4
1.5 Example of Use Case Model -----	5
1.6 Use Case Descriptions and Template Table -----	8

### **Chapter Two: High-Level Design (Sequence Diagram)**

2.1 High-Level Sequence Diagram-----	22
2.2 Sequence Diagram Components-----	22
2.3 Sequence Descriptions by Use Case-----	23
2.4 Tools and Steps Used-----	30

### **Chapter Three: Low-Level Design (Class Diagram)**

3.1 Class Design Overview-----	31
3.2 Main Classes & Roles-----	31
3.3 Key Relationships-----	31
3.4 Example Class Diagram-----	32
3.5 Tools and Steps-----	34

## **Chapter Four: Implementation**

4.1 Implementation Overview	36
4.2 Sample Class Implementations	36
4.3 Tools Used	38
4.4 Deployment	38

## **Chapter Five: Change Management**

5.1 Version Control Tool Used	39
5.2 Git Workflow and Commands	39
5.3 Git Snapshots	40
5.4 Benefits	42

## **Chapter Six: Unit Testing**

6.1 Purpose	43
6.2 Tools Used	43
6.3 Example Test Code	43

## **Chapter Seven: Build**

7.1 Build Process Overview	45
7.2 Build Tools and Steps	45

## **Appendix-----45**

# Chapter One: Requirement Analysis (Use Case)

The initial phase of developing the Smart Task Controller system involves an in-depth analysis of how diverse user types interact with the application. This stage is critical as it lays the foundation for understanding the system's functional requirements and provides a structured visual representation of interactions between users and the system through a use case model. This chapter meticulously identifies each actor's goals, responsibilities, and the system's responses to their actions, ensuring a comprehensive blueprint for the project's development.

## 1.1 Functional Overview

The Smart Task Controller is a sophisticated web-based application designed to streamline task management for both individual users and administrators. It offers a robust set of features, including task creation, editing, searching, and viewing, alongside advanced functionalities such as permission control, calendar synchronization, and customizable reminder alerts. The system aims to enhance productivity by providing an intuitive interface and automated notifications, catering to the needs of users ranging from personal task organizers to organizational administrators.

## 1.2 Identifying the Actors

The system involves two primary actors, each with distinct roles:

- **User:** A registered individual who leverages the application to manage personal or assigned tasks. This actor performs operations such as creating new tasks with detailed descriptions, editing existing tasks, searching for specific tasks using keywords, viewing a comprehensive task list, setting reminders for due dates, modifying reminders, synchronizing tasks with external calendar applications, and receiving timely notifications.
- **Admin:** A privileged user with elevated responsibilities, including managing other users (e.g., adding or removing them), distributing tasks across the user base, setting or modifying user permissions, inviting new users to the system, overseeing task progress, adding comments to tasks, and reviewing user profiles for administrative purposes.

## 1.3 Use Case Diagram Components

**Actors:** User, Admin

**Use Cases:** A detailed list of functionalities includes:

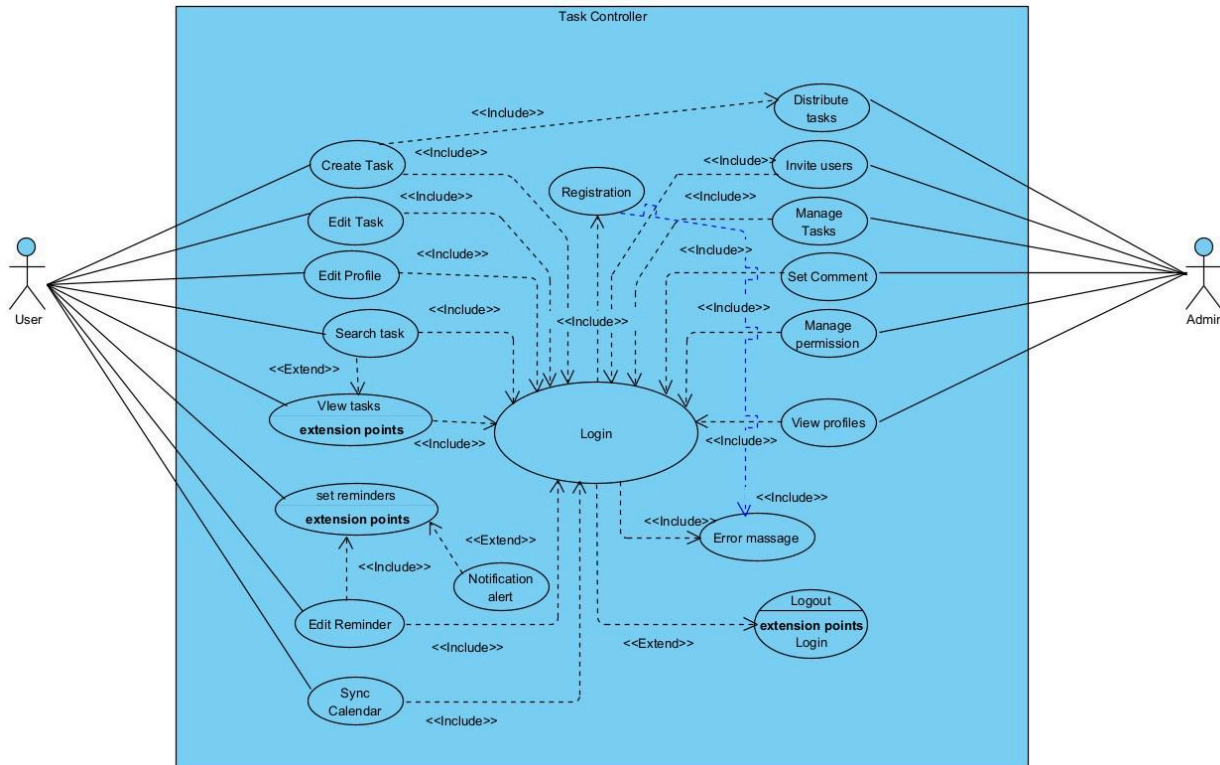
- Login: Authenticate user identity.
- Logout: Terminate the current session.
- Registration: Establish a new user account.
- Create Task: Initiate a new task entry.
- Edit Task: Modify an existing task's details.
- Edit Profile: Update personal account information.
- Search Task: Locate tasks using search criteria (extends View Tasks).
- View Tasks: Display all tasks associated with the user (extension point).
- Set Reminders: Schedule alerts for task deadlines (extension point).
- Edit Reminder: Adjust reminder settings or timing.
- Notification Alert: Deliver alerts when reminders are triggered (extends Set Reminders).
- Sync Calendar: Integrate tasks with an external calendar.
- Error Message: Provide feedback for failed operations.
- Distribute Tasks: Assign tasks to other users (Admin).
- Invite Users: Add new users to the system (Admin).
- Manage Tasks: Monitor and control task statuses (Admin).
- Set Comment: Add notes to tasks (Admin).
- Manage Permission: Define user access levels (Admin).
- View Profiles: Access and review user details (Admin).

## 1.4 Detailed Use Case Flow

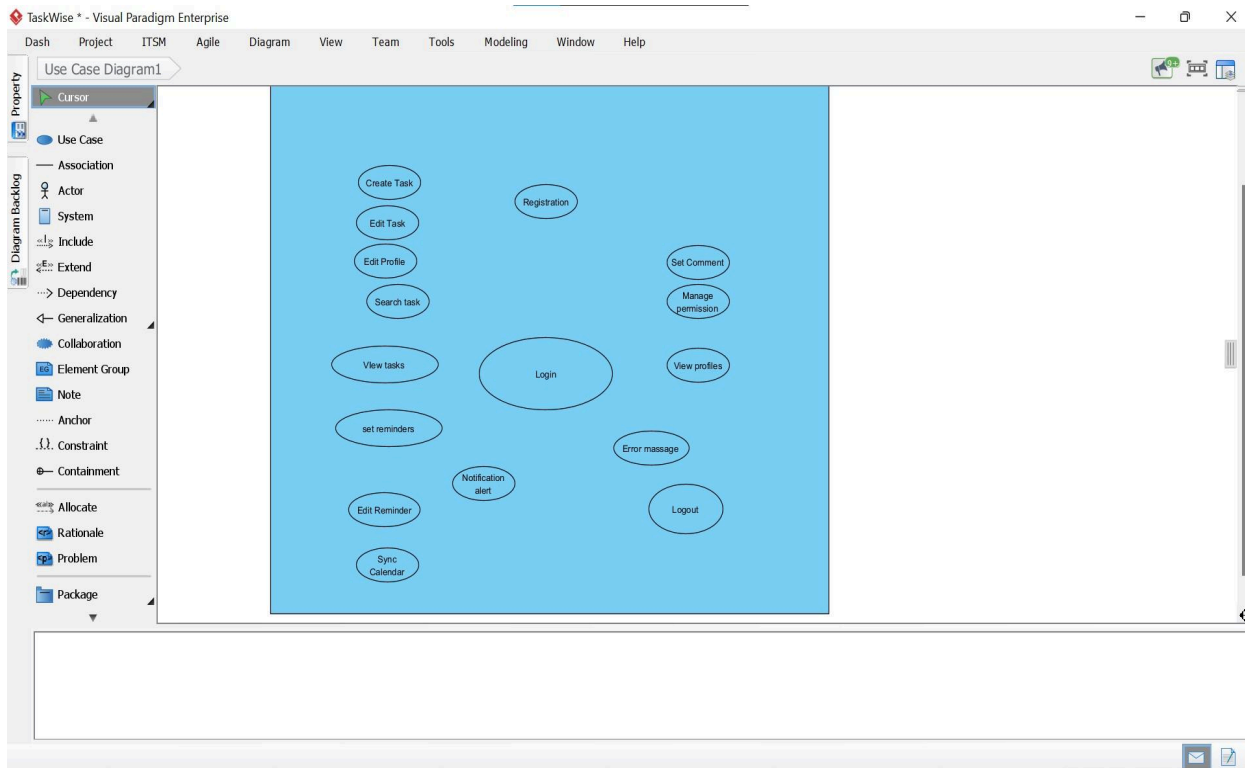
The interaction flow begins when a user or admin accesses the login page to authenticate their identity. If the user is not yet registered, they are directed to the registration process, which includes email verification to ensure account validity. Upon successful login, the user is redirected to a personalized dashboard, serving as the central hub for task management. Users can create, edit, search, and view tasks, set reminders with specific times, synchronize their tasks with external calendar systems like Google Calendar, and receive automated alerts. Administrators have additional privileges, allowing them to manage the user base, distribute tasks equitably, control access permissions, invite new members, monitor task progress, add comments for clarity, and review user profiles to maintain system integrity.

## 1.5 Example of Use Case Model

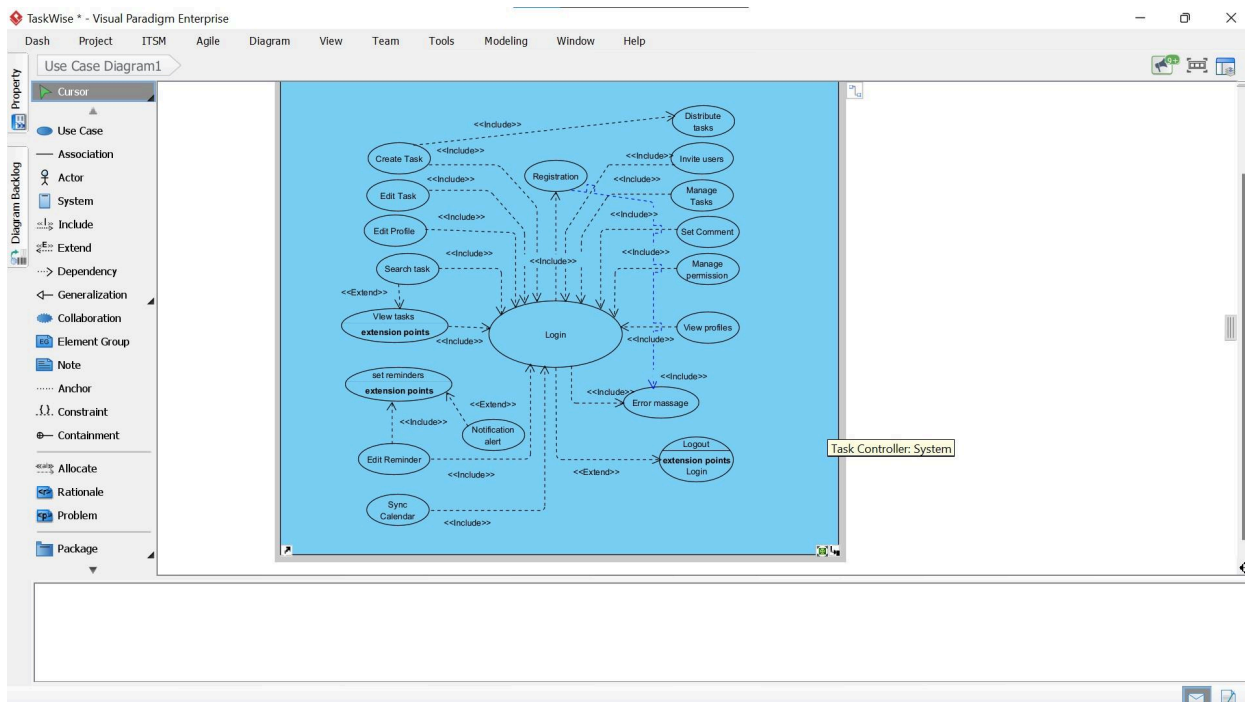
The use case diagram visually represents the dynamic interactions between the User and Admin actors and the Smart Task Controller system. It highlights core functionalities such as Login, Registration, and task management, using "extends" relationships (e.g., Search Task extends View Tasks for advanced filtering) and "includes" relationships (e.g., Login includes Error Message for invalid attempts). This diagram serves as a roadmap for developers, ensuring all user interactions are accounted for.



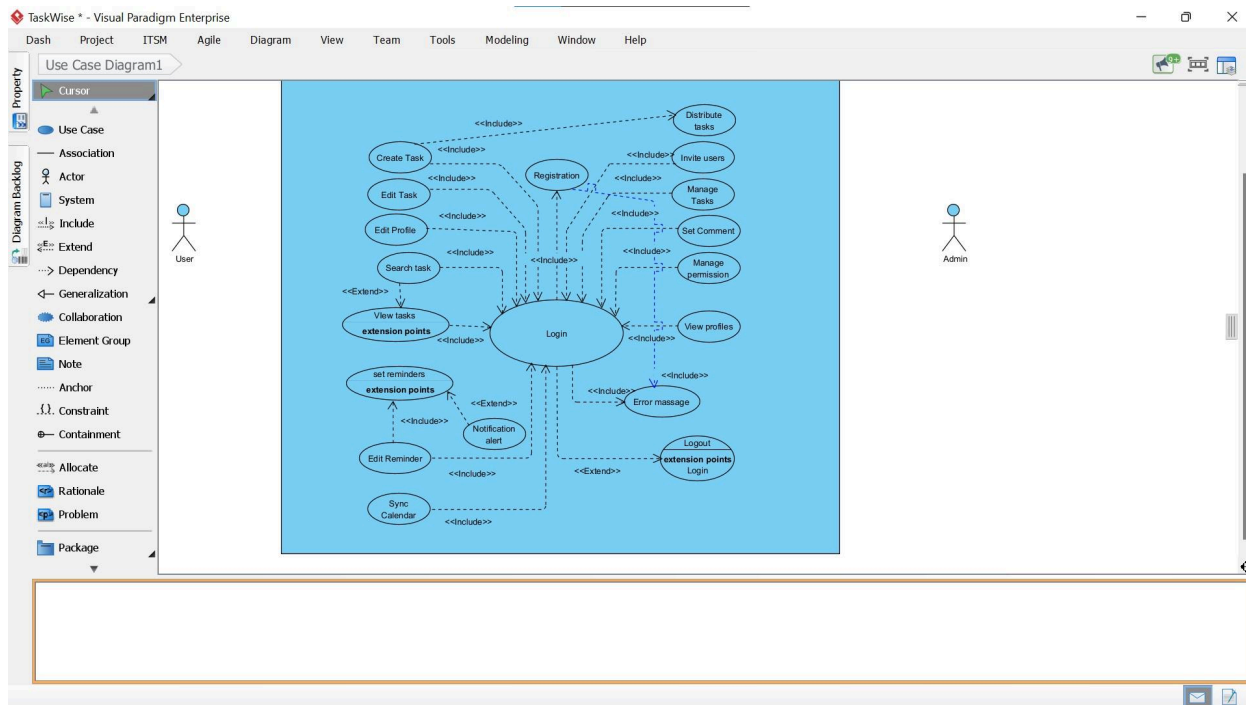
**Step** 1: draw the system and the use cases.



**Step** 2: join them with appropriate conjunction.



**Step 3:** put the users and finally join them in the use case there belongs to.





## 1.6 Use Case Descriptions and Template Table

This section provides detailed narratives for each use case, explaining the purpose, preconditions, postconditions, and potential exceptions to ensure a thorough understanding of system behavior.

<b>Use Case ID</b>	<b>UC-01</b>	
<b>Use Case Name</b>	Login	
<b>Actor</b>	User / Admin	
<b>Summary</b>	This use case describes how a registered User or Admin logs into the Smart Task Controller system by providing valid credentials.	
<b>Precondition</b>	The actor must already have an account with valid credentials.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: The actor opens the login page.  Step 3: The actor enters their email and password  Step 5: The actor clicks "Login".	Step 2: The system displays the login form.  Step 4: The system processes the credentials.  Step 6: The system validates and redirects to the dashboard.
<b>Alternative Scenario</b>	If credentials are incorrect: <ul style="list-style-type: none"><li>• The system displays an error.</li><li>• The actor retries or recovers account.</li></ul>	
<b>Post Condition</b>	The user is logged in and redirected to the dashboard.	

<b>Use Case ID</b>	<b>UC-02</b>	
<b>Use Case Name</b>	Registration	
<b>Actor</b>	User	
<b>Summary</b>	Allows a new user to create an account and verify via email.	
<b>Precondition</b>	The email must not already exist.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: User goes to home page. Step 3: Clicks "Sign Up". Step 5: Enters name, email, password. Step 7: Submits form.	Step 2: Home page loads. Step 4: Registration form displays. Step 6: System validates input. Step 8: System stores data and sends verification email.
<b>Alternative Scenario</b>	If email exists or invalid input: <ul style="list-style-type: none"> <li>Error message prompts user to retry.</li> </ul>	
<b>Post Condition</b>	The account is created and ready for verification.	

<b>Use Case ID</b>	<b>UC-03</b>
--------------------	--------------

<b>Use Case Name</b>	Create Task	
<b>Actor</b>	User	
<b>Summary</b>	Enables a user to add a new task with specific details.	
<b>Precondition</b>	User must be logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	
	Step 1: User clicks "Create Task". Step 3: Fills form with title, due date, etc. Step 5: Submits the form.	Step 2: Task creation form appears. Step 4: System validates. Step 6: Task is saved and confirmed.
<b>Alternative Scenario</b>	Invalid or missing input: <ul style="list-style-type: none"> <li>Error prompts user to correct.</li> </ul>	
<b>Post Condition</b>	Task appears in user's task list.	

<b>Use Case ID</b>	<b>UC-04</b>
<b>Use Case Name</b>	Edit Task
<b>Actor</b>	User
<b>Summary</b>	Modify details of an existing task.

<b>Precondition</b>	Task must exist and be accessible.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Selects a task. Step 3:Edits task fields. Step 5:Submits changes.	Step 2: Form loads with task details. Step 4: System validates input. Step 6: Task is updated and confirmed.
<b>Alternative Scenario</b>	Task not found or invalid input: <ul style="list-style-type: none"> <li>System shows error.</li> </ul>	
<b>Post Condition</b>	Task is updated in the database.	

<b>Use Case ID</b>	<b>UC-05</b>	
<b>Use Case Name</b>	Edit Profile	
<b>Actor</b>	User	
<b>Summary</b>	Allows users to update personal information.	
<b>Precondition</b>	User must be logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	
	Step 1: Clicks on "Edit Profile". Step 3: Edits personal details. Step 5: Submits the form.	Step 2: System displays profile form. Step 4: System validates input. Step 6: System updates profile.

<b>Alternative Scenario</b>	Invalid input: <ul style="list-style-type: none"> <li>Error prompts correction.</li> </ul>
<b>Post Condition</b>	User profile is updated.

<b>Use Case ID</b>	<b>UC-06</b>	
<b>Use Case Name</b>	Search Task	
<b>Actor</b>	User	
<b>Summary</b>	Enables users to search tasks using keywords.	
<b>Precondition</b>	User must be logged in and have tasks.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: User enters keyword. Step 3: Views search results..	Step 2: System filters tasks. Step 4: Displays matching tasks.
<b>Alternative Scenario</b>	No results: <ul style="list-style-type: none"> <li>System displays "No results found."</li> </ul>	
<b>Post Condition</b>	Search results are shown.	

<b>Use Case ID</b>	<b>UC-07</b>	
<b>Use Case Name</b>	View Tasks	
<b>Actor</b>	User	
<b>Summary</b>	Displays all tasks associated with the user.	
<b>Precondition</b>	User must be logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Opens task list.	Step 2: Retrieves and displays tasks.
<b>Alternative Scenario</b>	No tasks found: <ul style="list-style-type: none"> <li>• Message displays "No tasks."</li> </ul>	
<b>Post Condition</b>	Tasks are visible for management.	

<b>Use Case ID</b>	<b>UC-08</b>	
<b>Use Case Name</b>	Set Reminders	
<b>Actor</b>	User	
<b>Summary</b>	Allows users to create reminders for tasks.	

<b>Precondition</b>	Task must exist.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Selects a task. Step 3: Sets reminder details.	Step 2: System shows reminder options. Step 4: Reminder saved.
<b>Alternative Scenario</b>	Invalid time: <ul style="list-style-type: none"> <li>Error prompts correction.</li> </ul>	
<b>Post Condition</b>	Reminder is scheduled.	

<b>Use Case ID</b>	<b>UC-09</b>	
<b>Use Case Name</b>	Edit Reminder	
<b>Actor</b>	User	
<b>Summary</b>	Modify reminder timing or delivery method.	
<b>Precondition</b>	Reminder must exist.	
<b>Basic Scenario</b>	<b>Actor Action</b>	
	Step 1: Selects existing reminder. Step 3: Modifies details.	Step 2: Displays reminder details. Step 4: System saves changes.
<b>Alternative Scenario</b>	<ul style="list-style-type: none"> <li>Reminder not found:               <ul style="list-style-type: none"> <li>Error is displayed.</li> </ul> </li> </ul>	

<b>Post Condition</b>	Reminder is updated.
-----------------------	----------------------

<b>Use Case ID</b>	<b>UC-10</b>	
<b>Use Case Name</b>	Notification Alert	
<b>Actor</b>	System	
<b>Summary</b>	Sends alerts when reminders are triggered.	
<b>Precondition</b>	Active reminder exists.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	(system)	Step 1: Scheduler checks reminders. Step 2: Notification is sent.
<b>Alternative Scenario</b>	Delivery fails: <ul style="list-style-type: none"> <li>○ Retry or fallback is used.</li> </ul>	
<b>Post Condition</b>	Notification is received.	

<b>Use Case ID</b>	<b>UC-11</b>
<b>Use Case Name</b>	Sync Calendar



<b>Actor</b>	User	
<b>Summary</b>	Sync tasks with external calendar.	
<b>Precondition</b>	User must have access to calendar.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Clicks sync option.  Step 3: Confirms permission.	Step 2: Authenticates with calendar.  Step 4: Calendar is updated.
<b>Alternative Scenario</b>	Permission denied: <ul style="list-style-type: none"> <li>• Sync fails and shows error.</li> </ul>	
<b>Post Condition</b>	Tasks synced to external calendar.	

<b>Use Case ID</b>	<b>UC-12</b>
<b>Use Case Name</b>	Logout
<b>Actor</b>	User / Admin
<b>Summary</b>	Ends the user session.
<b>Precondition</b>	User must be logged in.

Basic Scenario	Actor Action	System Response
	Step 1: Clicks logout. Step 3: Confirms logout.	Step 2: System asks for confirmation. Step 4: Session ends and redirects to home.
Alternative Scenario	Cancels logout: <ul style="list-style-type: none"> <li>User stays logged in.</li> </ul>	
Post Condition	User is logged out.	

Use Case ID	UC-13	
Use Case Name	Distribute Tasks	
Actor	Admin	
Summary	Admin assigns tasks to users.	
Precondition	Admin is logged in and data exists.	
Basic Scenario	Actor Action	
	Step 1: Selects task and user.	Step 2: Task is assigned and user notified.
Alternative Scenario	User not found: <ul style="list-style-type: none"> <li>Error is shown.</li> </ul>	
Post Condition	Task is assigned.	

<b>Use Case ID</b>	<b>UC-14</b>	
<b>Use Case Name</b>	Invite Users	
<b>Actor</b>	Admin	
<b>Summary</b>	Adds new users by invitation.	
<b>Precondition</b>	Admin has invite rights.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Enters email and role.	Step 2: Sends invitation or creates user.
<b>Alternative Scenario</b>	Invalid email: <ul style="list-style-type: none"> <li>Error is shown.</li> </ul>	
<b>Post Condition</b>	User is invited.	

<b>Use Case ID</b>	<b>UC-15</b>	
<b>Use Case Name</b>	Manage Tasks	
<b>Actor</b>	Admin	
<b>Summary</b>	Monitors and updates tasks.	

<b>Precondition</b>	Admin is logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	
	Step 1: Opens task management. Step 3: Updates statuses.	Step 2: Displays tasks. Step 4: Changes are saved.
<b>Alternative Scenario</b>	Task not found: <ul style="list-style-type: none"> <li>Error is shown.</li> </ul>	
<b>Post Condition</b>	Task status is updated.	

<b>Use Case ID</b>	<b>UC-16</b>	
<b>Use Case Name</b>	Set Comment	
<b>Actor</b>	Admin	
<b>Summary</b>	Adds comment to a task.	
<b>Precondition</b>	Task must exist.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Opens a task. Step 3: Adds comment.	Step 2: Comment field appears. Step 4: Comment is saved.
<b>Alternative Scenario</b>	Empty comment: <ul style="list-style-type: none"> <li>Error is shown.</li> </ul>	

<b>Post Condition</b>	Comment is stored.
-----------------------	--------------------

<b>Use Case ID</b>	<b>UC-17</b>	
<b>Use Case Name</b>	Manage Permission	
<b>Actor</b>	Admin	
<b>Summary</b>	Changes user roles and access levels.	
<b>Precondition</b>	Admin is logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Selects a user.  Step 3: Updates role.	Step 2: Shows current permissions.  Step 4: System saves changes.
<b>Alternative Scenario</b>	Invalid role: <ul style="list-style-type: none"> <li>Error is displayed.</li> </ul>	
<b>Post Condition</b>	Permissions are updated.	

<b>Use Case ID</b>	<b>UC-18</b>
<b>Use Case Name</b>	View Profiles

<b>Actor</b>	Admin	
<b>Summary</b>	View user details for monitoring.	
<b>Precondition</b>	Admin is logged in.	
<b>Basic Scenario</b>	<b>Actor Action</b>	<b>System Response</b>
	Step 1: Selects a user.	Step 2: User profile is displayed.
<b>Alternative Scenario</b>	User not found: <ul style="list-style-type: none"> <li>Error is shown.</li> </ul>	
<b>Post Condition</b>	Profile is reviewed.	

# Chapter Two: High-Level Design (Sequence Diagram)

## 2.1 High-Level Sequence Diagram

This chapter provides an in-depth exploration of how users and system components interact over time to execute critical operations such as registration, login, task creation, setting reminders, logging out, editing tasks, and managing users. Each sequence diagram captures the dynamic behavior of the system, offering a step-by-step visualization of message exchanges and decision points, which are essential for understanding the system's operational flow.

## 2.2 Sequence Diagram Components

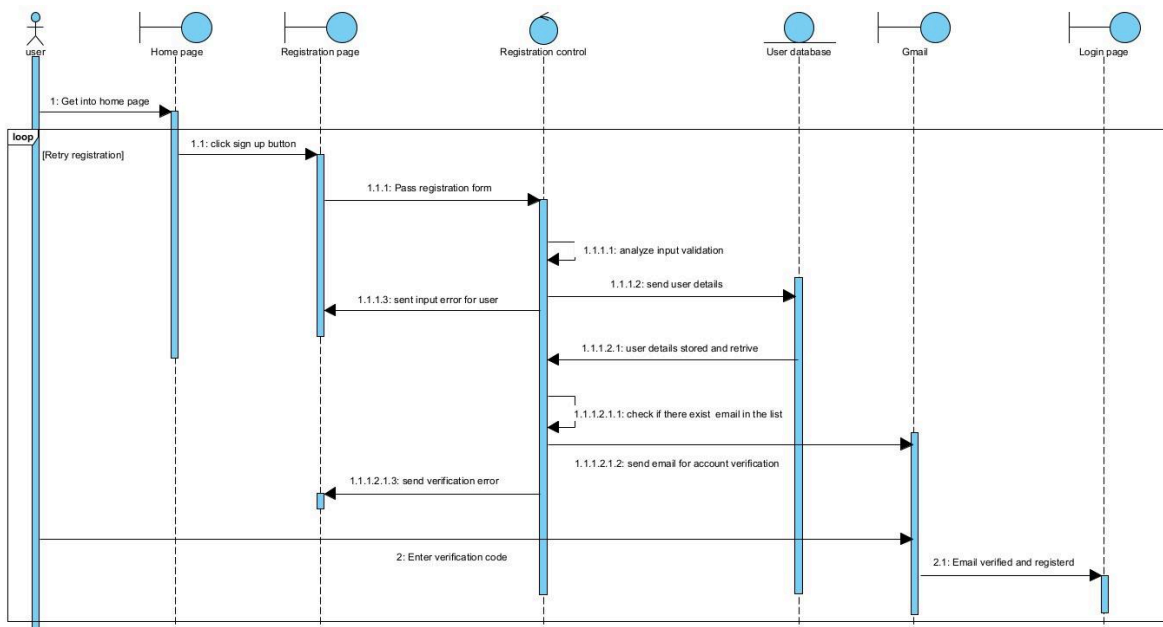
- **Actor/Object:** Represents external entities (e.g., User, Admin) and internal system components (e.g., Task Controller, User Database), each playing a specific role in the interaction.
- **Lifeline:** A vertical dashed line that indicates the existence of an object or actor throughout the scenario, showing when it is active or idle.
- **Messages:** Horizontal arrows representing the communication between components, labeled with the action or data being exchanged (e.g., "Submit login input").
- **Activation Bar:** A thin rectangle on the lifeline that signifies the duration an object is processing a message, providing insight into system responsiveness.
- **Fragments:** Special constructs like loops (e.g., [can create many times]) or alternate flows (e.g., [user confirm Logout]) that depict repetitive or conditional behaviors.

## 2.3 Sequence Descriptions by Use Case

### Registration

1. The user navigates to the home page, the entry point of the application.
2. The user clicks the "Sign Up" button, triggering the display of a detailed registration form requiring name, email, and password.
3. The user fills in the form and submits the data, which is then sent to the system for processing.
4. The system validates the input (e.g., checks for duplicate emails), stores the user data in the database, and initiates an email verification process. If validation fails, an error message is returned.

*Sequence Diagram:* This diagram illustrates the User interacting with the Home Page, Registration Page, Registration Controller, User Database, Gmail, and Login Page. It includes a loop for retrying registration, detailed validation steps, and the email verification workflow.

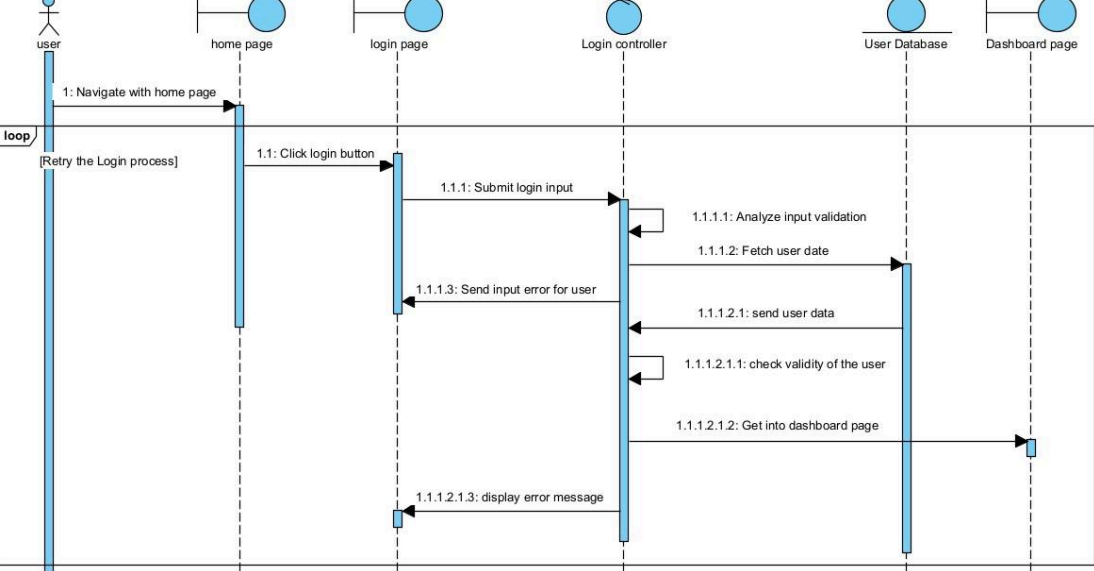


### Login

1. The user enters their email and password into the login form, initiating the authentication process.
2. The system forwards this input to the login controller, which validates the credentials against the database.
3. Upon successful validation, the user is redirected to their personalized dashboard; otherwise, an error alert is triggered to prompt re-entry.

*Sequence Diagram:* This diagram depicts the User interacting with the Home Page, Login



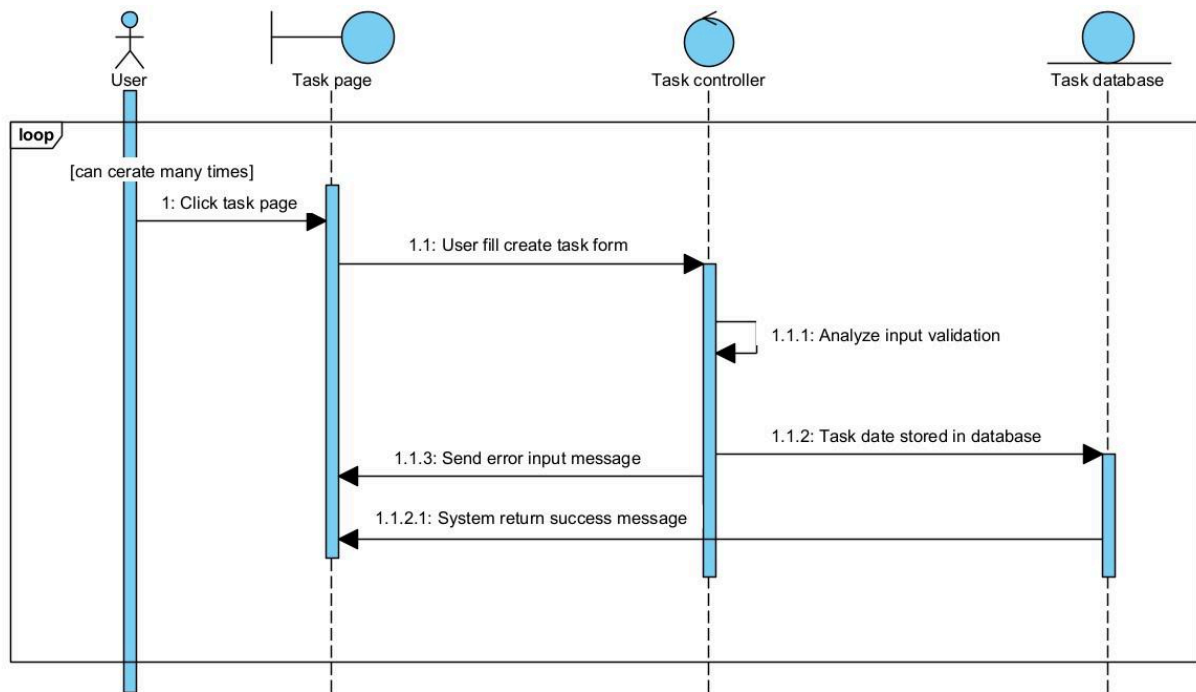


## Create Task

1. The user clicks the "Create Task" button on the task page, signaling their intent to add a new task.
2. The system responds by displaying a form where the user can input details such as title, description, due date, and priority.
3. The user submits the completed form, and the system performs input validation before saving the task to the database.
4. A success message confirms the task has been added to the user's list, or an error message is shown if validation fails.

**Sequence Diagram:** This diagram illustrates the User interacting with the Task Page, Task Controller, and Task Database. It includes input validation, error handling, and a confirmation

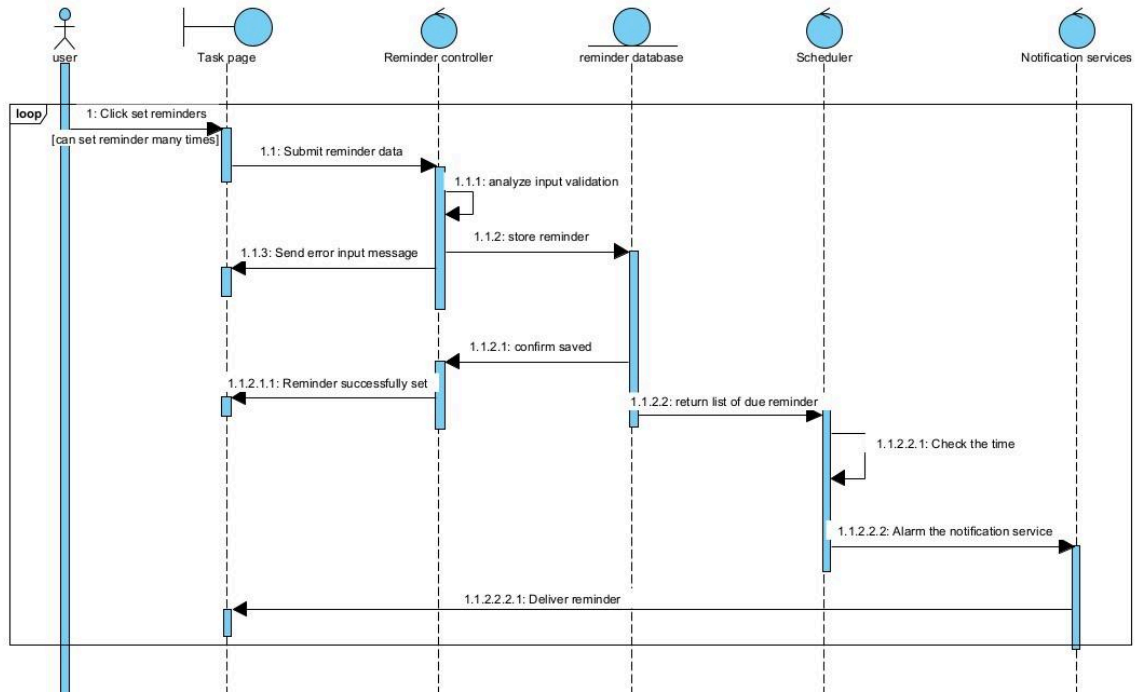
step with a loop for multiple task creations.



## Set Reminder

1. The user selects a task and chooses to set a reminder, specifying a time and notification method (e.g., email or popup).
2. The system saves the reminder details to the database and confirms the action to the user.
3. At the scheduled time, the scheduler checks for due reminders and triggers the notification process.
4. The notification service delivers the alert to the user, ensuring they are reminded of the task deadline.

**Sequence Diagram:** This diagram shows the User interacting with the Task Page, Reminder Controller, Reminder Database, Scheduler, and Notification Service. It includes validation, storage, scheduling, and delivery steps.

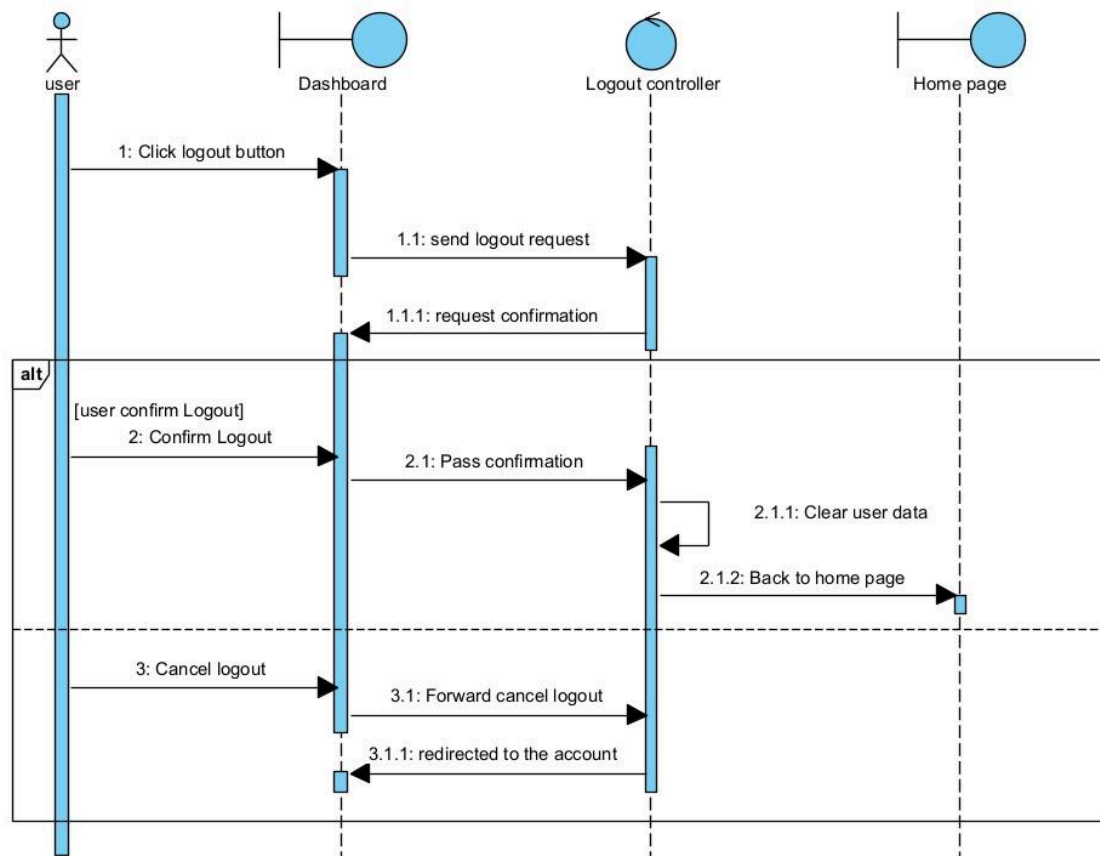


## Logout

1. The user clicks the "Logout" button on the dashboard, initiating the session termination process.
2. The system requests confirmation from the user to prevent accidental logouts.
3. The user confirms the action, and the system clears session data and redirects to the home page; if canceled, the user remains on the dashboard.

*Sequence Diagram:* This diagram depicts the User interacting with the Dashboard, Logout Controller, and Home Page. It includes an alternate flow for confirmation or cancellation with

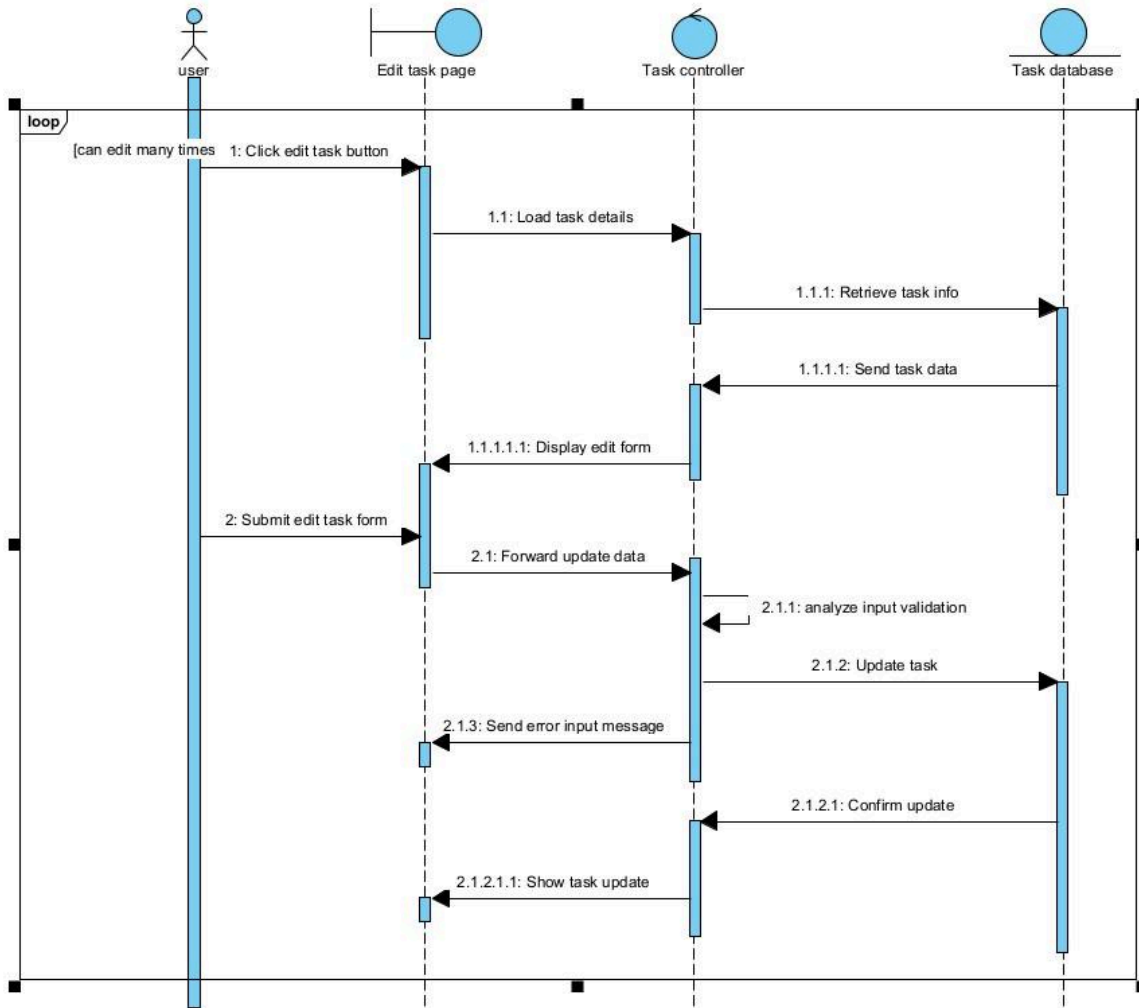
clear redirection logic.



## Edit Task

1. The user clicks the "Edit Task" button, prompting the system to load the task's current details.
2. The system displays an edit form pre-filled with the task's data, allowing modifications.
3. The user submits the updated details, which the system validates before updating the database.
4. A confirmation message is shown to the user, or an error is displayed if the update fails.

*Sequence Diagram:* This diagram illustrates the User interacting with the Edit Task Page, Task Controller, and Task Database. It includes loading, validation, updating, and confirmation steps with a loop for multiple edits.

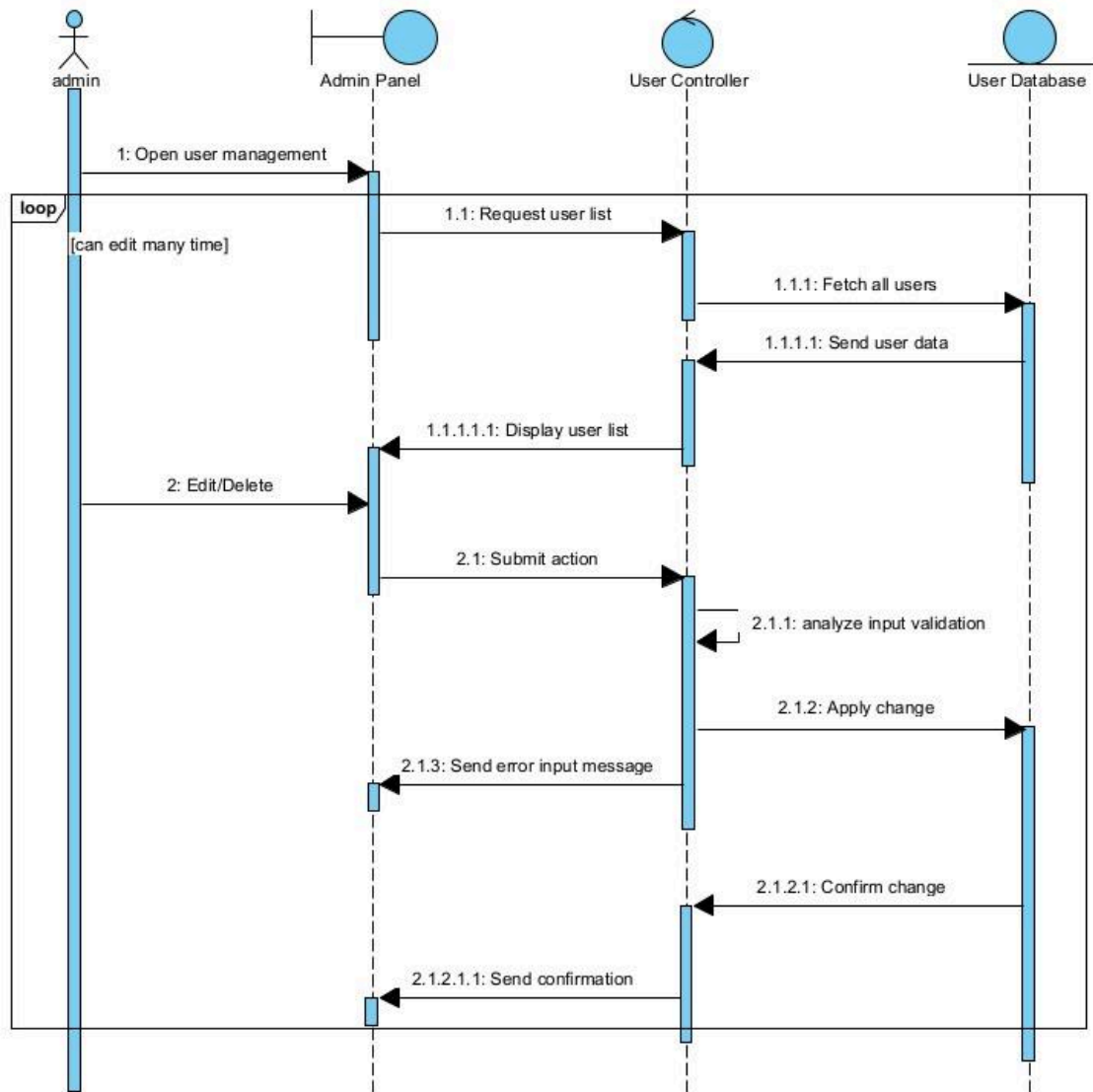


## Manage Users (Admin)

1. The admin opens the user management panel from the dashboard to oversee user activities.
2. The admin requests a list of users, which the system fetches from the database and displays.
3. The admin selects an action (e.g., edit or delete), submits it, and the system applies the change after validation.
4. A confirmation is sent to the admin, or an error message is displayed if the action fails.

*Sequence Diagram:* This diagram shows the Admin interacting with the Admin Panel, User Controller, and User Database. It includes fetching, validation, applying changes, and

confirmation steps with a loop for multiple edits.



## 2.4 Tools and Steps Used

**Tool Used:** Visual Paradigm, a powerful UML modeling tool that supports the creation of detailed sequence diagrams.

**Steps:**

1. Open a new sequence diagram within Visual Paradigm to begin the design process.
2. Add actors (e.g., User, Admin) and system objects (e.g., Task Controller, Database) to represent all entities involved.
3. Define lifelines for each object to show their active periods throughout the scenario.
4. Draw messages as arrows between lifelines, labeling them with specific actions or data (e.g., "Submit reminder data").
5. Incorporate fragments such as loops or alternate flows to handle repetitive or conditional interactions.
6. Review the diagram for accuracy, adjust layouts for clarity, and export it as an image file for inclusion in the documentation.

# Chapter Three: Low-Level Design (Class Diagram)

## 3.1 Class Design Overview

This chapter delves into the internal structure of the Smart Task Controller system, presenting a detailed breakdown of its major classes, their attributes, methods, and the relationships that bind them together. The class diagram serves as a blueprint for the system's implementation, ensuring that every component is well-defined and interconnected logically.

## 3.2 Main Classes & Roles

- **User:** A core class that encapsulates user-specific data such as name, email, and password, along with methods for authentication (e.g., login), profile management (e.g., edit profile), and task-related operations (e.g., search, view tasks).
- **Task:** A class representing individual tasks with attributes like title, due date, and priority, and methods for creation, editing, deletion, and distribution.
- **Reminder:** A class designed to manage task reminders, storing time and status details, with methods to set and retrieve reminder information.
- **Admin:** A specialized class that inherits from User, adding administrative functionalities such as user management, task distribution, and permission control.
- **Notification:** A class responsible for generating and sending alerts to users, with methods to handle notification delivery and deadline checks.
- **Scheduler:** A background process class that monitors reminder times and triggers notifications, maintaining a queue of tasks and their statuses.

## 3.3 Key Relationships

- **User → Task:** A one-to-many association where a single user can create and manage multiple tasks, reflecting the user's productivity scope.
- **Task → Reminder:** A one-to-many association allowing each task to have multiple reminders for different deadlines or stages.
- **Reminder → Notification:** A dependency relationship where a reminder triggers the creation of a notification when its time is due.
- **Admin → User:** An inheritance relationship where Admin extends User, inheriting basic user properties and adding administrative capabilities.
- **Scheduler → Reminder:** A querying relationship where the scheduler periodically checks reminders to ensure timely notifications.



### 3.4 Example Class Diagram

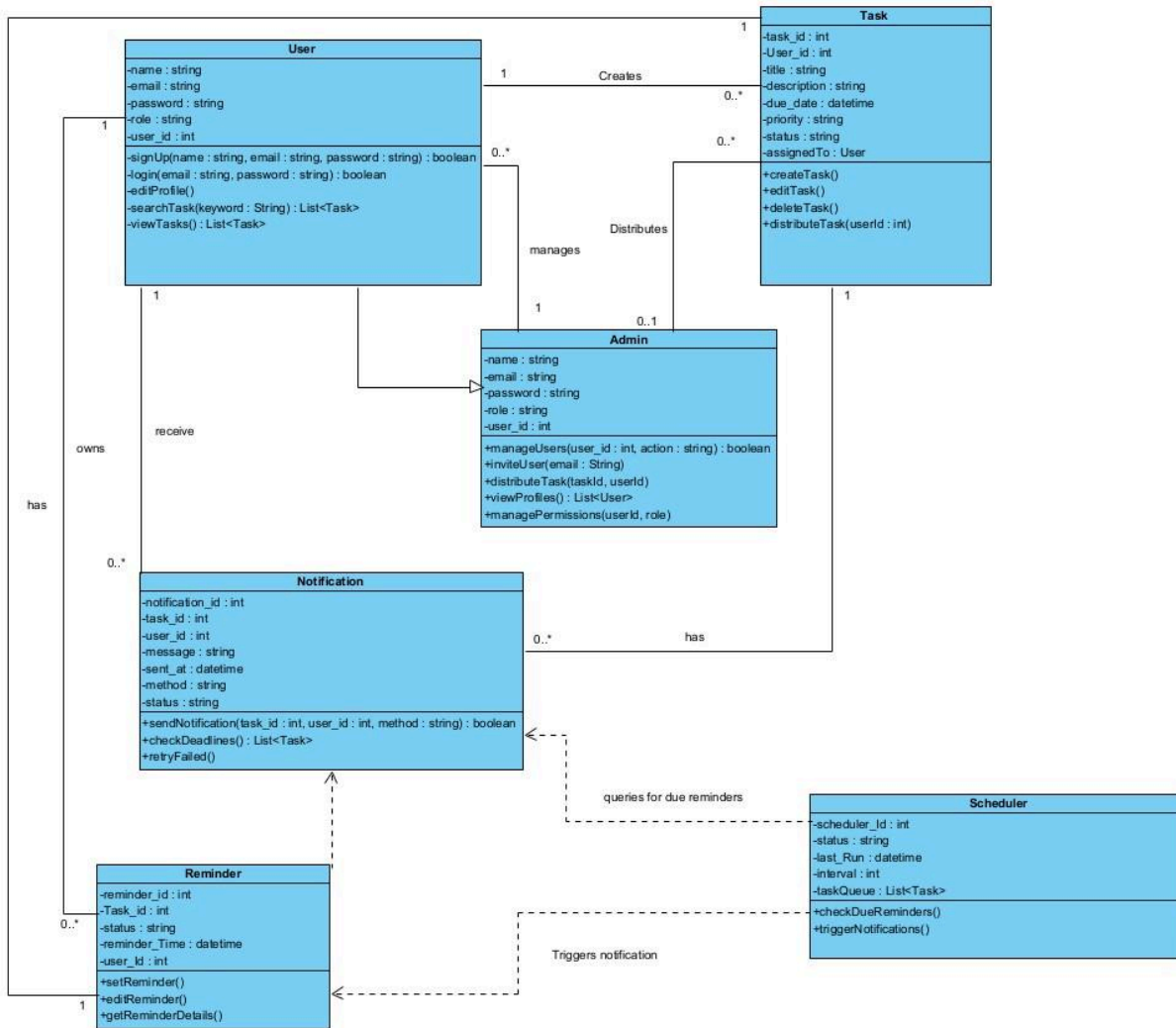
The class diagram provides a visual representation of the system's structure, detailing each class and its interactions:

- **User:**
  - **Attributes:** `name: String` (user's full name), `email: String` (unique email address), `password: String` (secured password), `role: String` (user or admin role), `user_id: int` (unique identifier).
  - **Methods:**
    - `signUp(name: String, email: String, password: String): boolean` - Registers a new user with validation.
    - `login(email: String, password: String): boolean` - Authenticates the user against stored credentials.
    - `editProfile()` - Updates user details with input validation.
    - `searchTask(keyword: String): List<Task>` - Returns a list of tasks matching the keyword.
    - `viewTasks(): List<Task>` - Retrieves and displays all user tasks.
- **Admin** (inherits from User):
  - **Attributes:** Inherits `name`, `email`, `password`, `role`, `user_id` from User.
  - **Methods:**
    - `inviteUser(email: String, user_id: int, action: String): boolean` - Invites a new user with specified action.
    - `manageTasks()` - Oversees task statuses and progress.
    - `managePermissions(user_id: int, role)` - Sets or updates user access levels.
- **Task:**
  - **Attributes:** `task_id: int` (unique task identifier), `user_id: int` (owner's ID), `title: String` (task name), `description: String` (task details), `due_date: datetime` (deadline), `priority: String` (e.g., high, medium), `status: String` (e.g., pending, done), `assignedTo: User` (assigned user object).
  - **Methods:**
    - `createTask()` - Initializes a new task entry.
    - `editTask()` - Modifies task details.
    - `deleteTask()` - Removes a task from the system.
    - `distributeTask(user_id: int)` - Assigns the task to another user.
- **Reminder:**

- **Attributes:** `reminder_id: int` (unique identifier), `task_id: int` (associated task), `status: String` (e.g., active, expired), `reminder_time: datetime` (scheduled time), `user_id: int` (owner's ID).
  - **Methods:**
    - `setReminder()` - Schedules a new reminder.
    - `getReminderDetails()` - Retrieves reminder information.
- **Notification:**
    - **Attributes:** `notification_id: int` (unique identifier), `task_id: int` (related task), `user_id: int` (recipient), `message: String` (alert content), `sent_at: datetime` (send time), `method: String` (e.g., email, popup), `status: String` (e.g., sent, failed).
    - **Methods:**
      - `sendNotification(task_id: int, user_id: int, method: String): boolean` - Delivers the notification.
      - `checkDeadlines(): List<Task>` - Identifies overdue tasks.
      - `retryFailed()` - Attempts to resend failed notifications.
  - **Scheduler:**
    - **Attributes:** `scheduler_id: int` (unique identifier), `status: String` (e.g., running, idle), `last_run: datetime` (last execution time), `interval: int` (check frequency in minutes), `taskQueue: List<Task>` (queued tasks).
    - **Methods:**
      - `checkDueReminders()` - Scans for upcoming reminders.
      - `triggerNotifications()` - Activates notification delivery.

### Relationships:

- A User creates 1-to-many Tasks, reflecting their task management capacity.
- A Task has 0-to-many Notifications, allowing multiple alerts per task.
- A Reminder triggers Notifications, establishing a dependency for alerts.
- A Scheduler queries for due Reminders, ensuring timely reminders.
- An Admin inherits from User and manages Tasks, extending functionality.



### 3.5 Tools and Steps

**Tool Used:** Visual Paradigm, a comprehensive UML modeling tool that supports detailed class diagram creation with advanced features.

**Steps:**

1. Launch the Visual Paradigm application and open a new class diagram to start the design process.
2. Add each class (e.g., User, Task) by dragging it onto the canvas, then define its attributes and methods with appropriate data types and visibility.
3. Establish relationships between classes using association lines (e.g., User to Task), inheritance arrows (e.g., Admin to User), and dependency connectors (e.g., Reminder to Notification).
4. Set multiplicities (e.g., 1-to-many) and visibility modifiers (e.g., + for public, - for private) to enhance clarity.

5. Review the diagram for completeness, adjust the layout for readability, and export it as a high-resolution image for documentation inclusion.

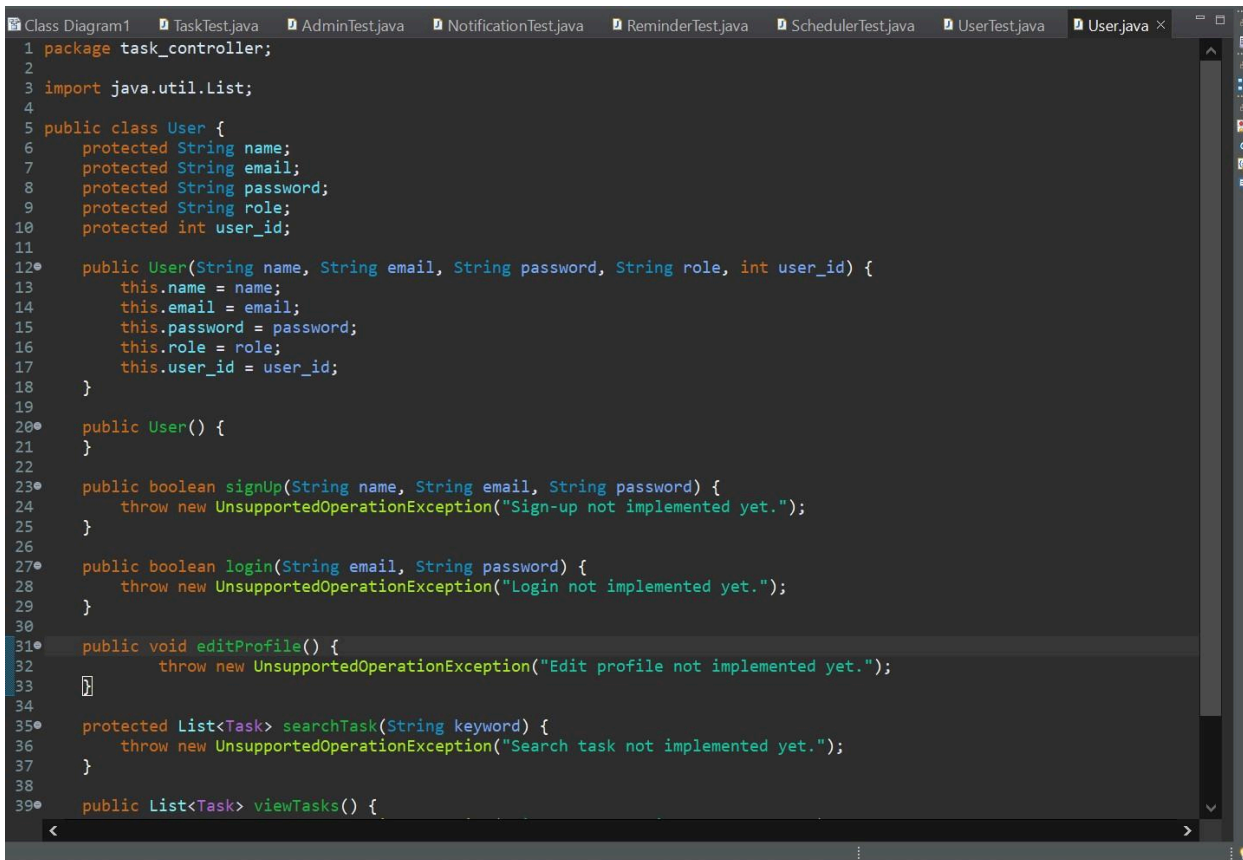
# Chapter Four: Implementation

## 4.1 Implementation Overview

The implementation phase transforms the design specifications into functional code, utilizing the Java programming language within the Eclipse Integrated Development Environment (IDE). This stage focuses on developing the core classes identified in the class diagram, ensuring they align with the system's requirements and provide a robust foundation for task management. The process involves coding, testing, and deploying the application, with Visual Paradigm assisting in generating initial code skeletons.

## 4.2 Sample Class Implementations

User.java:



```
1 package task_controller;
2
3 import java.util.List;
4
5 public class User {
6     protected String name;
7     protected String email;
8     protected String password;
9     protected String role;
10    protected int user_id;
11
12    public User(String name, String email, String password, String role, int user_id) {
13        this.name = name;
14        this.email = email;
15        this.password = password;
16        this.role = role;
17        this.user_id = user_id;
18    }
19
20    public User() {
21    }
22
23    public boolean signUp(String name, String email, String password) {
24        throw new UnsupportedOperationException("Sign-up not implemented yet.");
25    }
26
27    public boolean login(String email, String password) {
28        throw new UnsupportedOperationException("Login not implemented yet.");
29    }
30
31    public void editProfile() {
32        throw new UnsupportedOperationException("Edit profile not implemented yet.");
33    }
34
35    protected List<Task> searchTask(String keyword) {
36        throw new UnsupportedOperationException("Search task not implemented yet.");
37    }
38
39    public List<Task> viewTasks() {
```

## Task.java:

```
Class Diagram1 *Task.java x
1 package task_controller;
2
3 import java.time.LocalDateTime;
4
5 public class Task {
6     private int task_id;
7     private int user_id;
8     private String title;
9     private String description;
10    private LocalDateTime due_date;
11    private String priority;
12    private String status;
13    private String category;
14
15    public Task(int task_id, int user_id, String title, String description, LocalDateTime due_date, String priority,
16               String status, String category, LocalDateTime created_at, LocalDateTime start_time, LocalDateTime end_time) {
17        this.task_id = task_id;
18        this.user_id = user_id;
19        this.title = title;
20        this.description = description;
21        this.due_date = due_date;
22        this.priority = priority;
23        this.status = status;
24        this.category = category;
25    }
26
27    public boolean createTask(LocalDateTime due_date) {
28        throw new UnsupportedOperationException();
29    }
30
31    public boolean editTask(int task_id, String title, String description, LocalDateTime due_date, String priority,
32                           String status, String category) {
33        throw new UnsupportedOperationException();
34    }
35
36    public boolean deleteTask(LocalDateTime start_time, LocalDateTime end_time) {
37        throw new UnsupportedOperationException();
38    }
39
40    public Task distributeTask(int task_id) {
41        throw new UnsupportedOperationException();
42    }
43}
```

## Reminder.java :

```
Class Diagram1 *Task.java User.java Scheduler.java Reminder.java x
1 package task_controller;
2
3 import java.sql.Date;
4
5 public class Reminder {
6     private int reminder_id;
7     private int task_id;
8     private String status;
9     private Date reminder_time;
10
11    public Reminder(int reminder_id, int task_id, String status, Date reminder_time) {
12        this.reminder_id = reminder_id;
13        this.task_id = task_id;
14        this.status = status;
15        this.reminder_time = reminder_time;
16    }
17
18    public Reminder setReminder(int admin_id) {
19        throw new UnsupportedOperationException();
20    }
21
22    public Reminder editReminder(int report_id) {
23        throw new UnsupportedOperationException();
24    }
25 }
```

### 4.3 Tools Used

- **IDE:** Eclipse, a versatile development environment with robust debugging and code generation features.
- **Language:** Java, chosen for its platform independence and extensive library support.
- **Other Tools:** Visual Paradigm, utilized for generating initial code skeletons from class diagrams and ensuring design-code alignment.

### 4.4 Deployment

The application is deployed using Eclipse run configurations, allowing execution in both console mode for testing and GUI mode for user interaction. The deployment process involves compiling the source code, running unit tests, and launching the application on a local server or standalone environment. Future enhancements may include deploying to a web server like Apache Tomcat to support broader access.

# Chapter Five: Change Management

## 5.1 Version Control Tool Used

The project employs **Git**, a distributed version control system, hosted on **GitHub** for collaborative development and version tracking. This tool enables multiple developers to work simultaneously, merge changes, and maintain a history of all modifications.

## 5.2 Git Workflow and Commands

The Git workflow ensures systematic change management with the following commands:

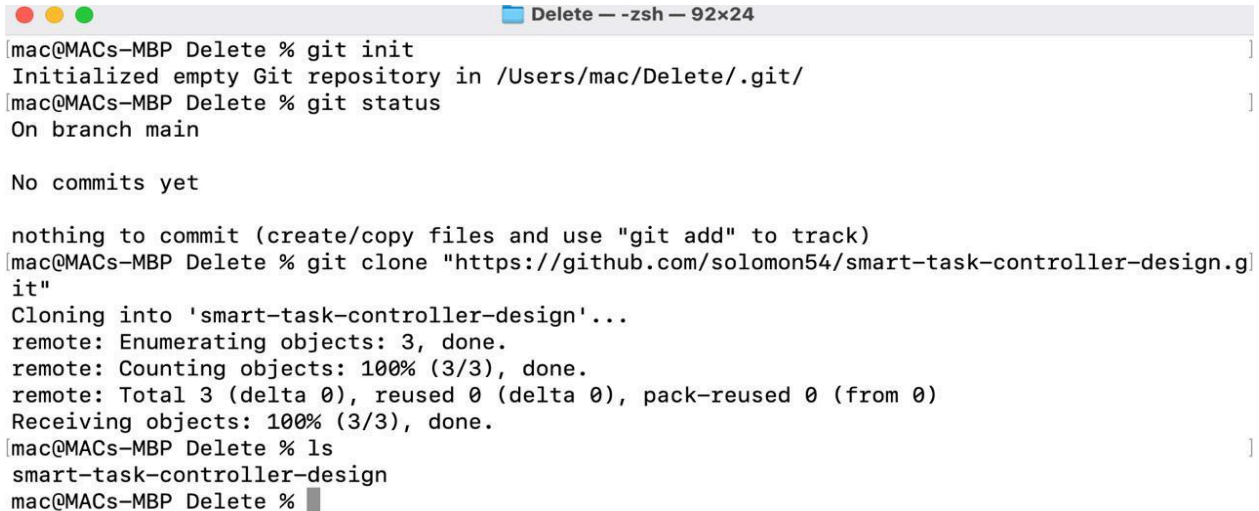
Command	Description
<code>git init</code>	Initializes a new Git repository in the project directory, marking the starting point for version control.
<code>git add</code> <code>&lt;file&gt;</code>	Stages specified files for the next commit, preparing changes for inclusion.
<code>git commit -m</code> <code>"msg"</code>	Commits the staged changes to the local repository with a descriptive message (e.g., "Add User class").
<code>git push</code>	Uploads local commits to the remote GitHub repository, synchronizing changes.
<code>git pull</code>	Downloads the latest changes from the remote repository to the local copy, ensuring alignment.



## 5.3 Git Snapshots

To demonstrate version control in action, screenshots of key Git activities are included, such as commit logs, branch creation, and pull requests on GitHub. These visuals provide evidence of the project's development history and collaboration efforts.

### git init

A terminal window titled "Delete — -zsh — 92x24" showing the execution of Git commands. The user runs 'git init', which initializes an empty repository. Then, they run 'git status', showing they are on the 'main' branch with no commits yet. Finally, they run 'git clone' to clone a repository from GitHub. The terminal output shows the cloning process, including enumerating and counting objects, and receiving the repository data. The directory listing shows the cloned repository files.

```
mac@MACs-MBP Delete % git init
Initialized empty Git repository in /Users/mac/Delete/.git/
mac@MACs-MBP Delete % git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
mac@MACs-MBP Delete % git clone "https://github.com/solomon54/smart-task-controller-design.git"
Cloning into 'smart-task-controller-design'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
mac@MACs-MBP Delete % ls
smart-task-controller-design
mac@MACs-MBP Delete %
```

### git add <file>

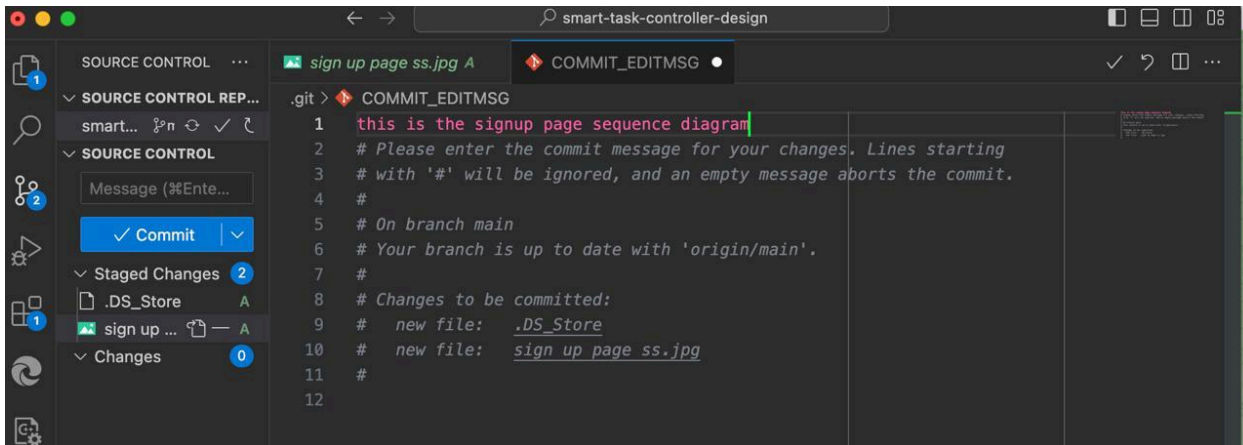
A terminal window showing the execution of 'git status' and 'git commit'. The 'git status' command shows the current branch is 'main' and it is up to date with 'origin/main'. It then lists changes to be committed: a new file '.DS\_Store' and a new file 'sign up page ss.jpg'. The user then runs 'git commit', and the terminal shows a hint to wait for the editor to close the file.

```
mac@MACs-MBP smart-task-controller-design % git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .DS_Store
    new file:   sign up page ss.jpg

mac@MACs-MBP smart-task-controller-design % git commit
hint: Waiting for your editor to close the file...
```

```
git commit -m "msg"
```



## Git logs

```
mac@MACs-MBP smart-task-controller-design % git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
(use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  create task.jpg
  edit task.jpg
  search task.jpg

nothing added to commit but untracked files present (use "git add" to track)
mac@MACs-MBP smart-task-controller-design % git add .
mac@MACs-MBP smart-task-controller-design % git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
(use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   create task.jpg
  new file:   edit task.jpg
  new file:   search task.jpg

mac@MACs-MBP smart-task-controller-design % git commit -m"I'm also adding create task, edit task and search task pages
";
[main 2ae6540] I'm also adding create task, edit task and search task pages
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 create task.jpg
create mode 100644 edit task.jpg
create mode 100644 search task.jpg
mac@MACs-MBP smart-task-controller-design %
```

## 5.4 Benefits

- **Track History:** Every change is logged, allowing reversion to previous states if needed.
- **Easy Collaboration:** Multiple developers can work concurrently with merge conflict resolution.
- **Revert Changes:** Mistakes can be undone using commit history.
- **Branch-Based Development:** Features can be developed in isolation before merging into the main branch.

# Chapter Six: Unit Testing

## 6.1 Purpose

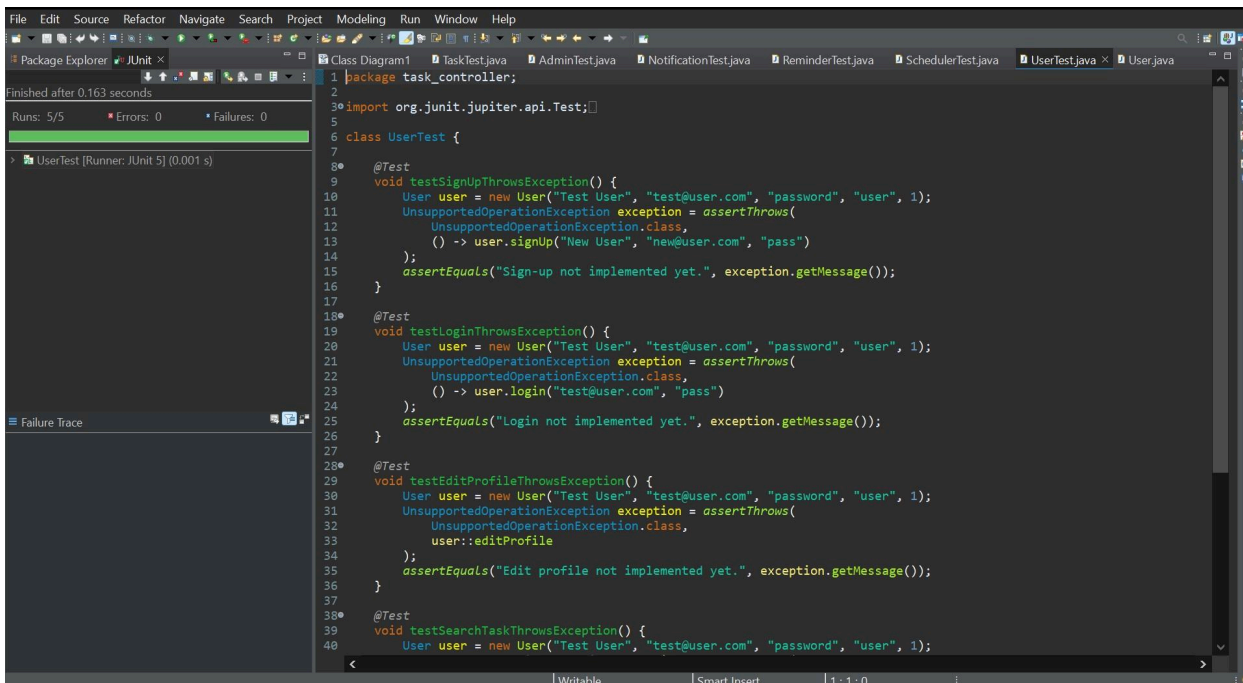
Unit testing is conducted to verify the correctness and reliability of key components within the Smart Task Controller, such as user login authentication, task creation functionality, and reminder setup processes. This step ensures that each module operates as intended before integration, reducing bugs and enhancing system stability.

## 6.2 Tools Used

This project utilizes **JUnit**, a widely adopted testing framework for Java. JUnit provides a set of annotations and assertions that enable developers to create and validate test cases effectively, ensuring code correctness and reliability throughout the development process. Additionally, the project is developed using the **Eclipse IDE**, which offers robust features for code writing, debugging, and seamless integration with JUnit.

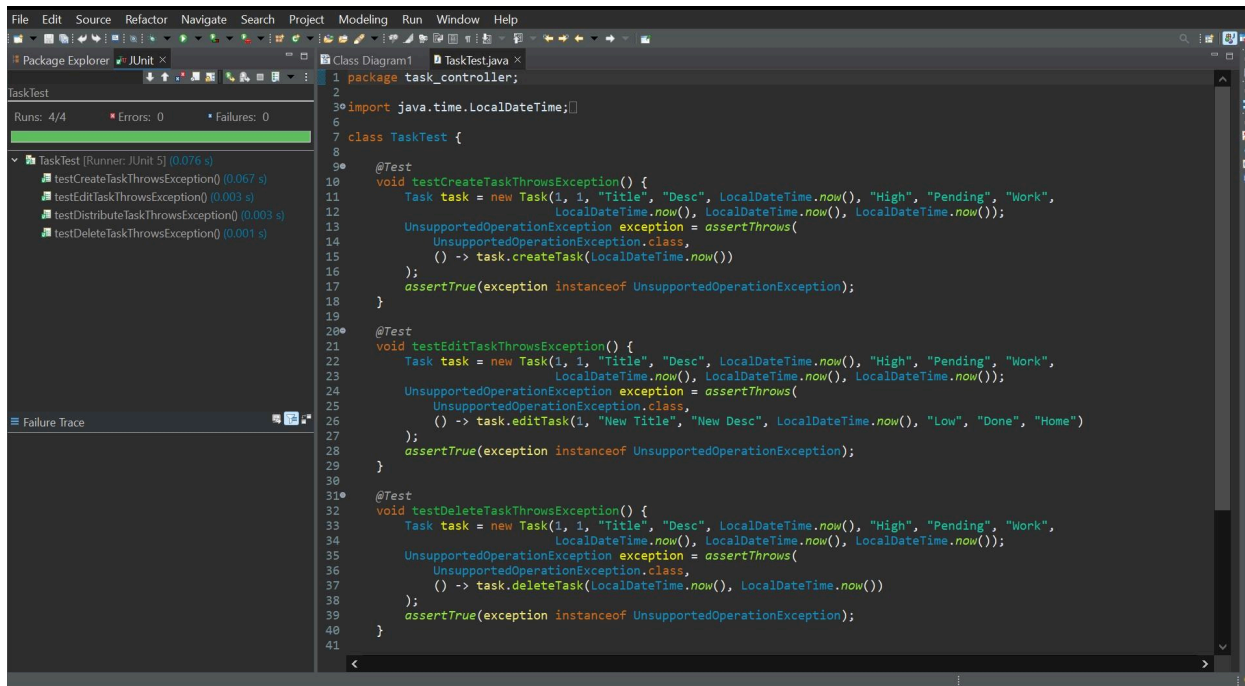
## 6.3 Example Test Code

### User Junit Test:



```
1 package task_controller;
2
3 import org.junit.jupiter.api.Test;
4
5 class UserTest {
6
7     @Test
8     void testSignUpThrowsException() {
9         User user = new User("Test User", "test@user.com", "password", "user", 1);
10        UnsupportedOperationException exception = assertThrows(
11            UnsupportedOperationException.class,
12            () -> user.signUp("New User", "new@user.com", "pass")
13        );
14        assertEquals("Sign-up not implemented yet.", exception.getMessage());
15    }
16
17     @Test
18     void testLoginThrowsException() {
19         User user = new User("Test User", "test@user.com", "password", "user", 1);
20        UnsupportedOperationException exception = assertThrows(
21            UnsupportedOperationException.class,
22            () -> user.login("test@user.com", "pass")
23        );
24        assertEquals("Login not implemented yet.", exception.getMessage());
25    }
26
27     @Test
28     void testEditProfileThrowsException() {
29         User user = new User("Test User", "test@user.com", "password", "user", 1);
30        UnsupportedOperationException exception = assertThrows(
31            UnsupportedOperationException.class,
32            user::editProfile
33        );
34        assertEquals("Edit profile not implemented yet.", exception.getMessage());
35    }
36
37     @Test
38     void testSearchTaskThrowsException() {
39         User user = new User("Test User", "test@user.com", "password", "user", 1);
40    }
```

## Task Junit Test:

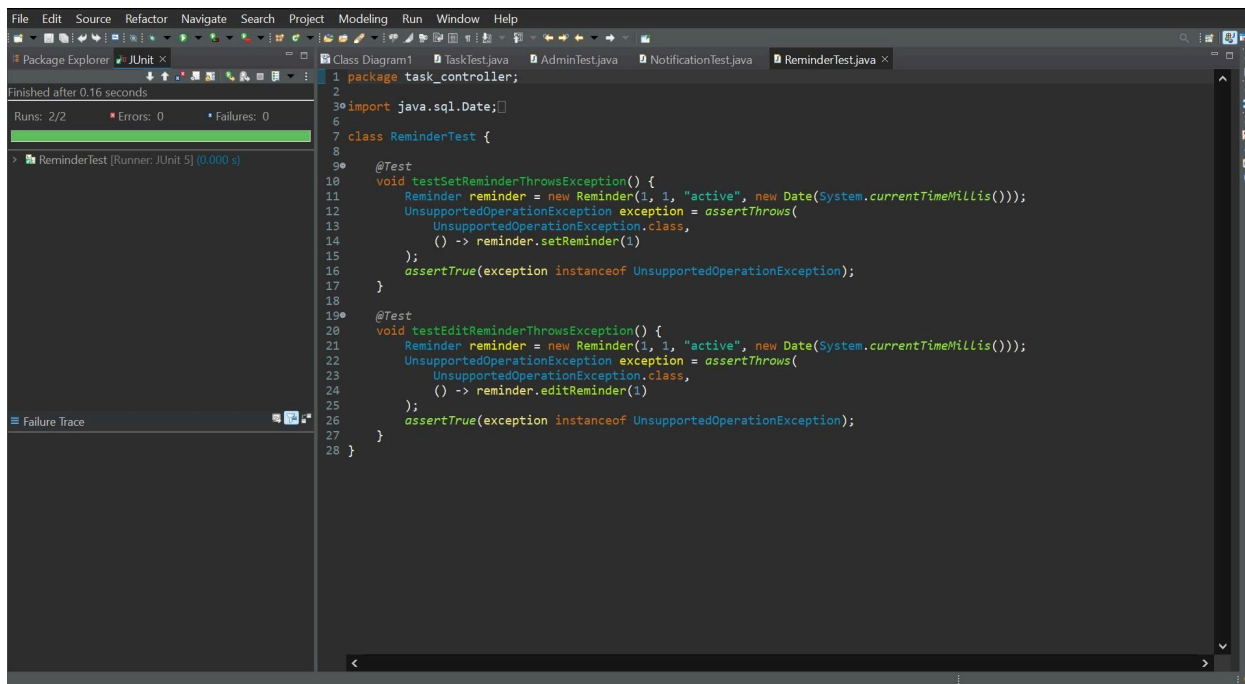


The screenshot shows an IDE with the following components:

- Package Explorer:** Shows the 'TaskTest' class under the 'task\_controller' package. It indicates 4/4 runs, 0 errors, and 0 failures.
- TaskTest.java:** Contains the following code:

```
1 package task_controller;
2
3 import java.time.LocalDateTime;
4
5
6
7 class TaskTest {
8
9     @Test
10    void testCreateTaskThrowsException() {
11        Task task = new Task(1, 1, "Title", "Desc", LocalDateTime.now(), "High", "Pending", "Work",
12                             LocalDateTime.now(), LocalDateTime.now(), LocalDateTime.now());
13        UnsupportedOperationException exception = assertThrows(
14            UnsupportedOperationException.class,
15            () -> task.createTask(LocalDateTime.now())
16        );
17        assertTrue(exception instanceof UnsupportedOperationException);
18    }
19
20    @Test
21    void testEditTaskThrowsException() {
22        Task task = new Task(1, 1, "Title", "Desc", LocalDateTime.now(), "High", "Pending", "Work",
23                             LocalDateTime.now(), LocalDateTime.now(), LocalDateTime.now());
24        UnsupportedOperationException exception = assertThrows(
25            UnsupportedOperationException.class,
26            () -> task.editTask(1, "New Title", "New Desc", LocalDateTime.now(), "Low", "Done", "Home")
27        );
28        assertTrue(exception instanceof UnsupportedOperationException);
29    }
30
31    @Test
32    void testDeleteTaskThrowsException() {
33        Task task = new Task(1, 1, "Title", "Desc", LocalDateTime.now(), "High", "Pending", "Work",
34                             LocalDateTime.now(), LocalDateTime.now(), LocalDateTime.now());
35        UnsupportedOperationException exception = assertThrows(
36            UnsupportedOperationException.class,
37            () -> task.deleteTask(LocalDateTime.now(), LocalDateTime.now())
38        );
39        assertTrue(exception instanceof UnsupportedOperationException);
40    }
41}
```

## Reminder Junit Test:



The screenshot shows an IDE with the following components:

- Package Explorer:** Shows the 'ReminderTest' class under the 'task\_controller' package. It indicates 2/2 runs, 0 errors, and 0 failures.
- ReminderTest.java:** Contains the following code:

```
1 package task_controller;
2
3 import java.sql.Date;
4
5
6
7 class ReminderTest {
8
9     @Test
10    void testSetReminderThrowsException() {
11        Reminder reminder = new Reminder(1, 1, "active", new Date(System.currentTimeMillis()));
12        UnsupportedOperationException exception = assertThrows(
13            UnsupportedOperationException.class,
14            () -> reminder.setReminder(1)
15        );
16        assertTrue(exception instanceof UnsupportedOperationException);
17    }
18
19    @Test
20    void testEditReminderThrowsException() {
21        Reminder reminder = new Reminder(1, 1, "active", new Date(System.currentTimeMillis()));
22        UnsupportedOperationException exception = assertThrows(
23            UnsupportedOperationException.class,
24            () -> reminder.editReminder(1)
25        );
26        assertTrue(exception instanceof UnsupportedOperationException);
27    }
28 }
```

# Chapter Seven: Build

## 7.1 Build Process Overview

The build process compiles the Java source code into executable bytecode, executes unit tests to validate functionality, and generates the application output. This phase is crucial for ensuring the system is ready for deployment and use, providing a final verification step before release.

## 7.2 Build Tools and Steps

- **Tool:** Eclipse IDE, with its built-in compiler and run configurations, or the command-line interface (CLI) for manual builds.
- **Steps:**
  1. Open the project in Eclipse and select "Build Project" from the menu to compile all source files.
  2. Run all JUnit test cases to verify each component's behavior, checking for pass/fail results.
  3. Execute the application using the configured run settings, generating output in the console or GUI window.
  4. Review the output for errors or unexpected behavior, making adjustments as needed.

## Appendix

- **Source Code Repository:** Available on GitHub (optional link or clone instructions).
- **Screenshots:** Includes all class diagrams, sequence diagrams, code snippets, JUnit test results, Git activity, and build outputs for comprehensive documentation.