Wachemo University
College of Engineering and Technology

Department of Software Engineering

Course Title: - Software Engineering Tools and practice Project

Group 3

Name	ID
1. Roza Philipos ————————————————————	1501228
2. Olifan Bededa —————————————————————	1501207
3. Tsedeke Wondimu ———————————————————	1510032
4. Kirubel Wondwossen —————————————————	1501042

Date: May 16, 2025

Acknowledgments

We would like to extend our sincere appreciation to everyone who contributed to the development and success of our Fitness Tracking System.

First and foremost, we are deeply grateful to our team members for their dedication, collaboration, and relentless effort throughout the project. Their technical skills, creativity, and teamwork were essential in designing and implementing the system.

We would also like to thank our mentors and academic advisors for their continuous guidance and constructive feedback. Their expertise and encouragement played a significant role in helping us overcome obstacles and maintain our focus on the project objectives.

Our heartfelt thanks also go to the students and test users who provided helpful feedback during the evaluation of the system. Their input has been vital in improving the overall user experience and functionality.

Finally, we express our gratitude to our families and friends for their constant support, patience, and encouragement during every phase of this project.

Abstract

The fitness tracking system is a web-based application designed to empower users in achieving their health and wellness goals by providing a centralized platform to monitor workouts, nutrition, and overall progress. This document explores the architecture, core functionalities, and system interactions that drive the fitness platform. Through an in-depth analysis of use cases, sequence diagrams, and class diagrams, it highlights essential features such as user registration and login, workout and meal logging, progress tracking, health monitoring, and achievement unlocking. Additionally, it covers admin functionalities including user management and content moderation. The document also discusses key system requirements, design choices, and potential future improvements such as enhanced analytics and motivational features. By examining the fitness tracking system, this document offers valuable insights into the design and implementation of personalized health management platforms.

Contents

Chapter One

1.1. Requirement Analysis

Requirement analysis plays a vital role in systems engineering, software development, and project management. It focuses on identifying, recording, and handling the expectations and demands of stakeholders for a specific system or project.

Requirement Analysis of Fitness Tracking System:

1. **User Registration and Login:**
   • Users must be able to register by providing personal details such as name, email, password, age, weight, height, and fitness goals.
   • Users should securely log in to access their personalized dashboard and system features.
   • The system should provide features to reset or recover forgotten passwords.
   • A secure logout feature should be available to end the session.
2. **Edit Profile:**
   • Users should be able to update their profile information such as weight, height, age, activity level, fitness goals, health conditions, and diet preferences.
   • The system should allow changing the password and uploading a profile picture.
   • All updates must be saved securely and reflect instantly in system calculations and suggestions.
3. **Track Workouts:**
   • Users should be able to log daily workouts by selecting a workout type, entering duration, and calories burned.
   • Notes can be added to individual workouts.
   • The system should allow viewing, editing, or deleting past workout logs.

4. **Set Fitness Goals:**
   • Users can define specific goals like losing weight, gaining muscle, or maintaining fitness.
   • Goals should include target weight/BMI and deadline/timeframe.
   • Progress should be tracked with visual indicators and editable goal options.
5. **Monitor Diet and Nutrition:**
   • Users should be able to log meals, search food items, and enter calories consumed.
   • A daily calorie summary should be presented, along with personalized diet recommendations based on profile data.
   • Nutrition tracking should help users stay within their daily intake limits.
6. **BMI Calculation:**
   • The system should calculate BMI using user profile data (weight and height).
   • It should categorize BMI (e.g., Underweight, Normal, Overweight, Obese).

• Suggestions for gaining or losing weight based on BMI category should be provided.
• Integration with goal-setting and workout planning is essential.

7. **Workout Scheduling:**
   • Users should be able to schedule workouts for the week, selecting specific dates and times.
   • Scheduled sessions can be edited or canceled.
   • A weekly calendar should display all planned workouts.

8. **Track Progress:**
   • The system should generate progress charts showing weight change, BMI trends, and calories burned.
   • Users should be able to compare their current state with starting values.
   • History of goals and achievements should be accessible from this module.
   • BMI calculation is integrated here for continuous health evaluation.

9. **Health Tracking:**
   • Users should be able to enter and monitor health data like blood pressure and sugar levels.
   • Graphs should display historical trends.
   • Alerts must be triggered if values are out of healthy ranges.

10. **Achievements and Rewards:**
    • Users should earn badges or achievements for completing specific tasks (e.g., 10 workouts, reaching a goal).
    • These achievements should be visible on the dashboard.
    • Integration with progress tracking is required to unlock achievements based on system milestones.

11. **Motivational Quotes:**
    • The dashboard should show a random motivational quote each time the user logs in.
    • Users can refresh for a new quote or mark favorites.
    • Quotes are part of the optional dashboard widget features.

12. **"Did You Know" Health Facts:**
    • The dashboard should display fun and informative health or nutrition facts.
    • Users can refresh the fact or see a daily featured fact.
    • These facts are designed to educate and motivate users and can be grouped with dashboard widgets.

13. **Dashboard Widgets:**
    • The system should group fun features like motivational quotes, health facts, and achievement highlights into an optional section of the dashboard.
    • Users can choose to engage with these tools for encouragement and education.

14. **Manage Workouts (Admin):**
    • The admin can view a list of all existing workouts and select any entry to **edit** or **delete**.
    • Editing allows updating workout details like name, type, and calorie values to keep the data accurate and up-to-date.
    • Deleting removes outdated or incorrect workouts from the database, ensuring users only access valid and relevant exercise options.

15. **Add Workouts (Admin):**

- The admin can add new workout types to the system by specifying the workout name, category/type, and estimated calories burned.
- This function ensures the workout database stays relevant, covering a wide range of exercise options for users.
- Once submitted, the new workout is stored in the database and becomes available for users to select during workout logging or scheduling.

1.2. Use Case Diagram Components: Fitness Tracking System

A use case provides a thorough explanation of how a user (or another system) engages with a system to accomplish a particular objective. It is an essential element in software and systems engineering for capturing functional needs and directing system design and implementation. Use cases support a clearer view of the system's actions from the user's standpoint and ensure that every possible interaction is accounted for.

Here are the possible components for the use case diagram:

1. Actors
- **User**: Regular system user who interacts with all fitness-related features.
- **Admin**: System administrator responsible for **adding motivational quotes**, **"Did You Know" facts**, and **new workout types** as well as overseeing system content.

2. Use Cases

2.1. Authentication
- **Register**: New users register in the system.
- **Login**: Required before accessing most system functionalities.
- **Logout**: Extends from the main user session.

2.2. User Functionalities

- **Update Profile**
- **Track Workouts**
- **Workout Scheduling**
- **Set Fitness Goals**
- **Monitor Diet & Nutrition**
- **Health Tracking**
- **Calculate BMI**
- **Track Progress** (includes Calculate BMI, Health Tracking, and other relevant progress metrics)

- **View Dashboard Widgets**
  - **See "Did You Know" Facts** *(<<extend>>)*
  - **View Motivational Quotes** *(<<extend>>)*
- **Achievements** *(<<extend>> from Track Progress)*
- **Manage Workout Schedule**

2.3. Admin Functionalities
- **Add Motivational Quotes** *(<<include>> Login)*
- **Add "Did You Know" Fact** *(<<include>> Login)*
- **Add Workouts** *(<<include>> Login)*
- **Manage Workout** *(<<include>> Login)*

3. Associations
- **User** → connected to all personal fitness-related use cases.
- **Admin** → connected to administrative tasks (**Add Motivational Quotes**, **Add "Did You Know" Fact**, **Add Workouts and Manage Workout**).
- All admin and user actions rely on the **Login** use case.

4. System Boundary
- The boundary is labeled as **Fitness Tracking System**.
- All use cases are inside this system boundary box.
- External actors (User, Admin) are outside and connected via lines to the relevant use cases.

5. Include / Extend / Extension Points
- **<<include>>**:
  - Used where a use case always requires another (e.g., all actions include Login).
- **<<extend>>**:
  - Used for optional behavior like:
    - Logout
    - Manage Schedule
    - Manage Workout
    - Achievements (extends from Track Progress)
- **Extension Points**:
  - Marked on Logout, View Motivational Quotes, See "Did You Know" Facts, Workout Schedule, Track Progress, Workouts.
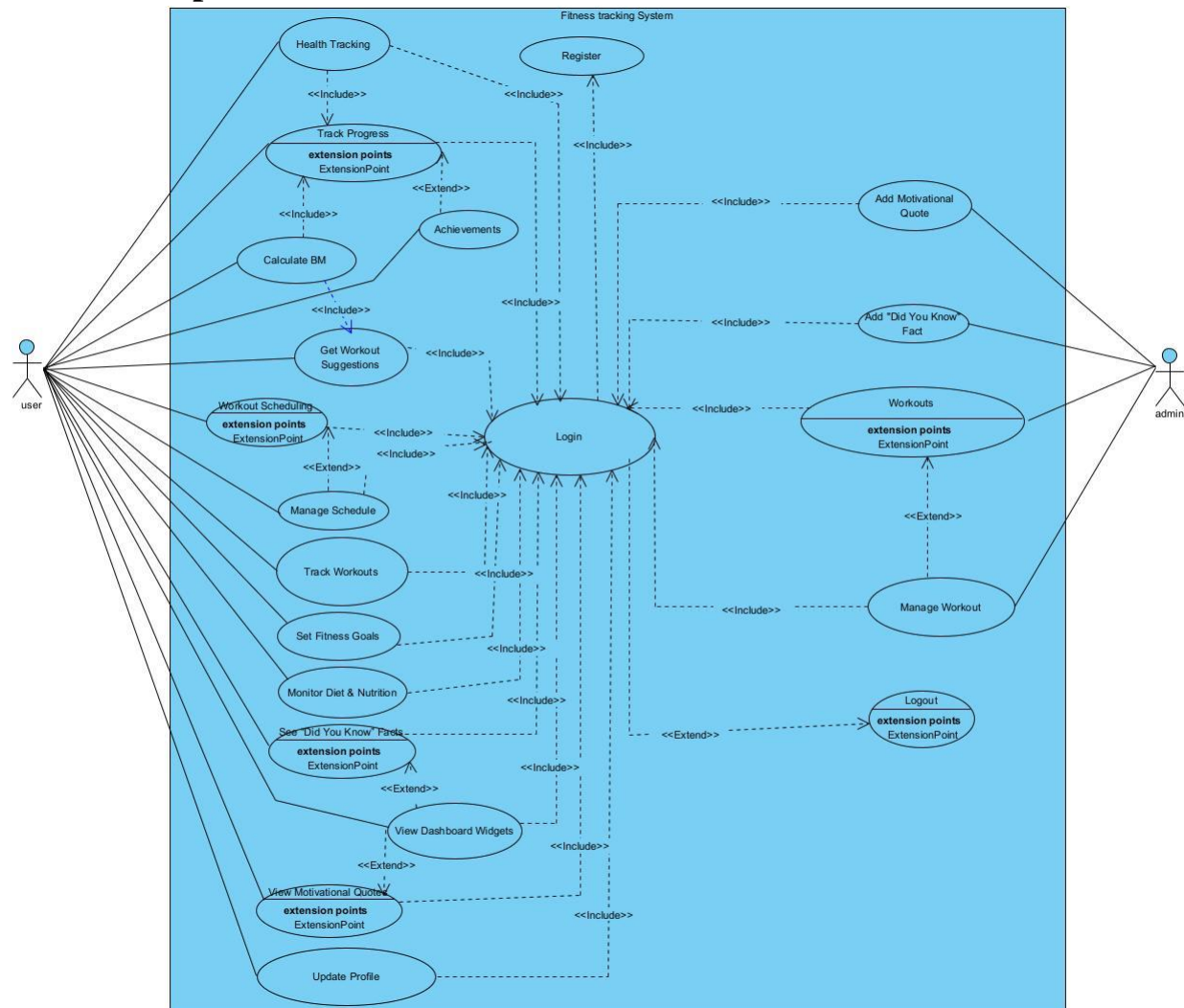
1.3. Example of use case model

Figure 1.3- 1: Use case diagram for fitness tracking system

1.4. Use case Description/template

Use Case ID	UC-1
Use Case Name	Register & Login
Actor	User
Summary	This use case allows a user to register a new account and log in to access their dashboard and features.
Precondition	1. The user must have internet access. 2. The app or website must be accessible.
Basic Scenario	**Actor Action** Step1: User visits the Home Page and navigates to Register Page. Step2: User enters registration details and submits. Step3: System (AuthController) checks for existing email.

Use Case ID	UC-1
	Step4: If valid, saves user info and redirects to Login Page. Step5: User enters login credentials. Step6: System validates and redirects to Dashboard.
Alternative Scenario	Step4A: If email exists, system shows "Email already in use" error. Step6A: If credentials are invalid, system shows "Invalid credentials" error.
Post Condition	The user is registered and/or logged in and redirected to their dashboard.

Table 1.4- 1: Use case description for user registration and login

Use Case ID	UC-2
Use Case Name	Workout Scheduling
Actor	User
Summary	Allows users to schedule workouts by type, day, and time, and view/update weekly plans.
Precondition	1. The user must be logged in. 2. The workout library must be accessible.
Basic Scenario	**Actor Action** Step1: User selects workout type, day, and time on Schedule Page. Step2: System checks for existing conflict. Step3: If none, retrieves user's fitness goal and validates workout balance. Step4: Saves schedule and sets next workout timer. Step5: System displays success, updated calendar, time left, and total weekly duration.
Alternative Scenario	Step2A: If conflict exists, system shows conflict error and stops scheduling.
Post Condition	The schedule is saved, timers are set, and user sees their updated weekly workout plan.

Table 1.4- 2: Use case description for workout scheduling

Use Case ID	UC-3/4
Use Case Name	Manage Workout Schedule
Actor	User
Summary	Allows users to edit or delete their previously scheduled workouts, updating or removing them from the weekly workout calendar.
Precondition	1. User must be logged in. 2. A workout schedule must already exist.
Basic Scenario	**Edit Workout Flow** Step1: User selects a workout and edits the details. Step2: System checks for schedule conflicts.

Use Case ID	UC-3/4
	Step3: If no conflict, system updates the workout schedule. Step4: System adjusts the timer for the new workout time. Step5: Page displays success message. Step6: Weekly calendar and time-to-next-workout are updated. **Delete Workout Flow** Step1: User selects a workout to delete. Step2: System checks if the workout exists. Step3: If it exists, system deletes the workout and removes the timer. Step4: Page displays success message. Step5: Weekly calendar is updated.
Alternative Scenario	**Edit Flow Alternative** Step2A: If the new workout time conflicts with another, system displays a conflict error. **Delete Flow Alternative** Step2B: If the selected workout is not found, system displays an error.
Post Condition	The workout schedule is updated or deleted successfully, and all related data such as timers and calendars are refreshed accordingly.

Table 1.4- 3: Use case description for manage workout schedule

Use Case ID	UC-5
Use Case Name	Track Progress and Health Tracking
Actor	User
Summary	Users log health metrics and progress; the system analyzes and displays feedback.
Precondition	1. User must be logged in. 2. User profile must have height data.
Basic Scenario	**Actor Actions** Step1: User submits weight, BP, sugar on Progress Page. Step2: System saves entry. Step3: Triggers BMI calculation and returns advice. Step4: User sets goal weight. Step5: System evaluates health status and shows progress.
Alternative Scenario	Step5A: If BP/sugar is abnormal, health status is marked "Alert".
Post Condition	Progress and health data are saved and displayed with insights.

Table 1.4- 4: Use case description for track progress and health tracking

Use Case ID	UC-6
Use Case Name	Achievements
Actor	User
Summary	Users view unlocked achievements and progress toward new badges.
Precondition	1. User must be logged in. 2. Some workout or meal logs must exist.

Use Case ID	UC-6
Basic Scenario	**Actor Actions** Step1: User opens Achievements Page. Step2: System fetches achievements. Step3: If milestone reached, badge is generated. Step4: Displays recent and all achievements.
Alternative Scenario	Step3A: If milestone not reached, system shows progress only.
Post Condition	Achievements and badges are shown based on progress.

Table 1.4- 5: Use case description for achievements

Use Case ID	UC-7
Use Case Name	Track Workout
Actor	User
Summary	Users can log their workouts, including type, duration, and calories burned.
Precondition	1. User must be logged in. 2. Workout types (predefined/custom) must be available.
Basic Scenario	**Actor Actions** Step1: User enters workout info (type, duration, calories). Step2: System validates workout ID. Step3: System inserts log into `workout_log`. Step4: If valid, confirms saved; else shows under-burn warning. Step5: Page displays confirmation or warning.
Alternative Scenario	Step4A: If calories < required, system shows a warning message (under-burn).
Post Condition	Workout is logged, and confirmation or warning is displayed to the user.

Table 1.4- 6: Use case description for track workout

Use Case ID	UC-8/9
Use Case Name	Set Fitness Goal and Edit Profile
Actor	User
Summary	Allows users to update personal details and set a fitness goal (e.g., Bulking, Cutting).
Precondition	1. User must be logged in.
Basic Scenario	**Edit Profile Flow** Step1: User updates personal info. Step2: System saves the updated record. Step3: Page shows updated info. **Set Fitness Goal Flow** Step4: User selects a fitness goal. Step5: System saves goal to profile. Step6: Page confirms update.

Use Case ID	UC-8/9
Alternative Scenario	Step2A: If invalid info is submitted, system shows an error.
Post Condition	User profile and fitness goal are updated successfully.

Table 1.4- 7: Use case description for set fitness goal and edit profile

Use Case ID	UC-10
Use Case Name	Monitor Diet & Nutrition
Actor	User
Summary	Users can log meals and view daily calorie summaries and alerts.
Precondition	1. User must be logged in. 2. Daily calorie limit must be defined.
Basic Scenario	**Actor Actions** Step1: User enters meal details (name, calories, time). Step2: System saves the meal log. Step3: If within limit, confirms saved. Step4: Page shows updated daily summary.
Alternative Scenario	Step3A: If calories exceed the daily limit, system shows an over-intake alert.
Post Condition	Meal data is saved and daily summary is updated with alerts if necessary.

Table 1.4- 8: Use case description for monitor diet and nutrition

Use Case ID	UC-10/11
Use Case Name	Manage Workout Schedule
Actor	User
Summary	Allows users to edit or delete their previously scheduled workouts, updating or removing them from the weekly workout calendar.
Precondition	1. User must be logged in. 2. A workout schedule must already exist.
Basic Scenario	**Edit Workout Flow** Step1: User selects a workout and edits the details. Step2: System checks for schedule conflicts. Step3: If no conflict, system updates the workout schedule. Step4: System adjusts the timer for the new workout time. Step5: Page displays success message. Step6: Weekly calendar and time-to-next-workout are updated. **Delete Workout Flow** Step1: User selects a workout to delete. Step2: System checks if the workout exists. Step3: If it exists, system deletes the workout and removes the timer. Step4: Page displays success message. Step5: Weekly calendar is updated.

Use Case ID	UC-10/11
Alternative Scenario	**Edit Flow Alternative** Step2A: If the new workout time conflicts with another, system displays a conflict error. **Delete Flow Alternative** Step2B: If the selected workout is not found, system displays an error.
Post Condition	The workout schedule is updated or deleted successfully, and all related data such as timers and calendars are refreshed accordingly.

Table 1.4- 9: Use case description for manage workout

Use Case ID	UC-12
Use Case Name	Add "Did You Know" / Motivational Quote
Actor	Admin
Summary	Admin adds inspirational or educational content to be displayed on the user dashboard.
Precondition	Admin must be logged in.
Basic Scenario	Step1: Admin navigates to the "Add Content" section. Step2: Admin selects content type ("Did You Know" or "Motivational Quote"). Step3: Admin submits the content text and type. Step4: System saves the content to the database with the selected type. Step5: System displays a success message.
Post Condition	The submitted content is saved and becomes available for display on user dashboards.

Table 1.4- 10: Use case description add did you know facts and motivational quotes

Use Case ID	UC-13
Use Case Name	Manage Workouts
Actor	Admin
Summary	Admin can view, edit, or delete existing workouts in the system to keep the workout library up-to-date.
Precondition	Admin must be logged in. Existing workouts must be present in the database.
Basic Scenario	Step1: Admin navigates to "Manage Workouts". Step2: System displays the list of all workouts. Step3: Admin selects a workout and clicks "Edit" or "Delete". Step4: Admin submits changes. Step5: System updates or removes the workout. Step6: Page shows confirmation message.
Post Condition	Workout record is updated or deleted. The workout library reflects the changes immediately.

Table 1.4- 11: Use case description of manage workout

Use Case ID	UC-14
Use Case Name	Add Workouts
Actor	Admin
Summary	Admin can add new workout types to the database, including calorie burn values.
Precondition	Admin must be logged in.
Basic Scenario	Step1: Admin navigates to "Add Workout". Step2: Admin submits workout name, type, and calories. Step3: System saves workout to database. Step4: Page shows confirmation message.
Post Condition	New workout is saved and available for user selection during workout logging or scheduling.

Table 1.4- 12: Use case description of add workouts

1.5. Tools and steps to draw Use Case

1.5.1. Tools Used

To create the use case diagram, we used Visual Paradigm, a user-friendly UML tool known for its intuitive drag-and-drop interface and support for all standard UML diagrams. It also offers customizable templates and export options, making it a great choice for both academic and professional use.

Steps in Creating the Use Case Diagram

1. **Understand the System Requirements**

   Begin by reviewing the functional requirements of the **Fitness Tracking System**,

   identifying all key features, user interactions, and system responses.

2. **Identify Actors**

   Determine the external entities that interact with the system. For this system:

   o **User**: The end-user interacting with fitness features.

   o **Admin**: System administrator responsible for **adding motivational quotes**, **"Did You Know" facts**, and **new workout types** as well as overseeing system content.
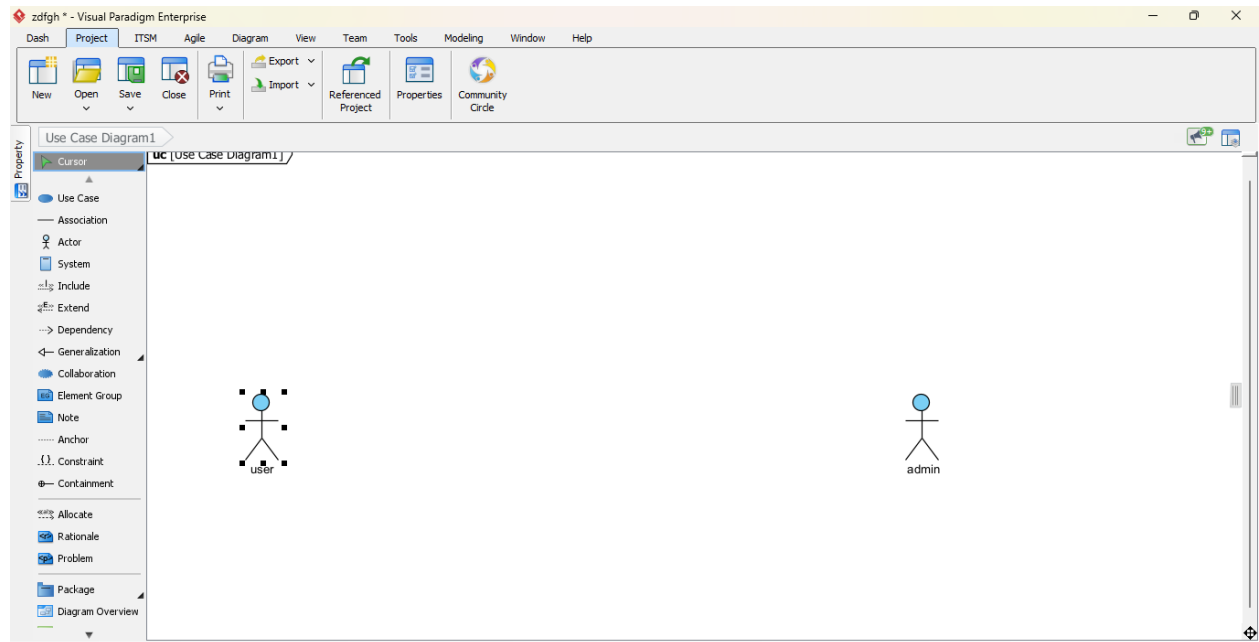
Figure 1.5- 1: Identify actors

3. <u>Define Use Cases</u>

3.1. <u>Authentication</u>

- **Register** – Create a new user account.
- **Login** – Access the system.
- **Logout** – End session (shown as an extension point in the diagram).

3.2. <u>Fitness Features (User Functionalities)</u>

- **Health Tracking** – General health monitoring.
- **Track Progress** – Monitor fitness and health data over time.
- **Calculate BMI** – Measure body mass index.
- **Get Workout Suggestions** – Receive tailored workout plans.
- **Workout Scheduling** – Plan and organize workouts.
- **Track Workouts** – Log workout sessions.
- **Set Fitness Goals** – Define personal fitness objectives.
- **Monitor Diet & Nutrition** – Track food intake and calories.
- **See "Did You Know" Facts** – Display motivational or educational health facts.

- **View Dashboard Widgets** – Visualize key data and content on a central panel.

- **View Motivational Quotes** – Get inspired through daily motivational content.

- **Update Profile** – Edit user account or health information.

- **Achievements** – Earn and view fitness-related accomplishments.

3.3. Administrative Tasks (Admin Functionalities)

- **Add Motivational Quotes** – Admins can create and manage motivational quotes displayed on user dashboards.
- **Add "Did You Know" Fact** – Admins can input informative health or nutrition facts for dashboard display.
- **Add Workouts** – Admins can define new workout types that users can select when tracking their exercise routines.
- **Manage Workouts** – Admin edit or delete a workout.
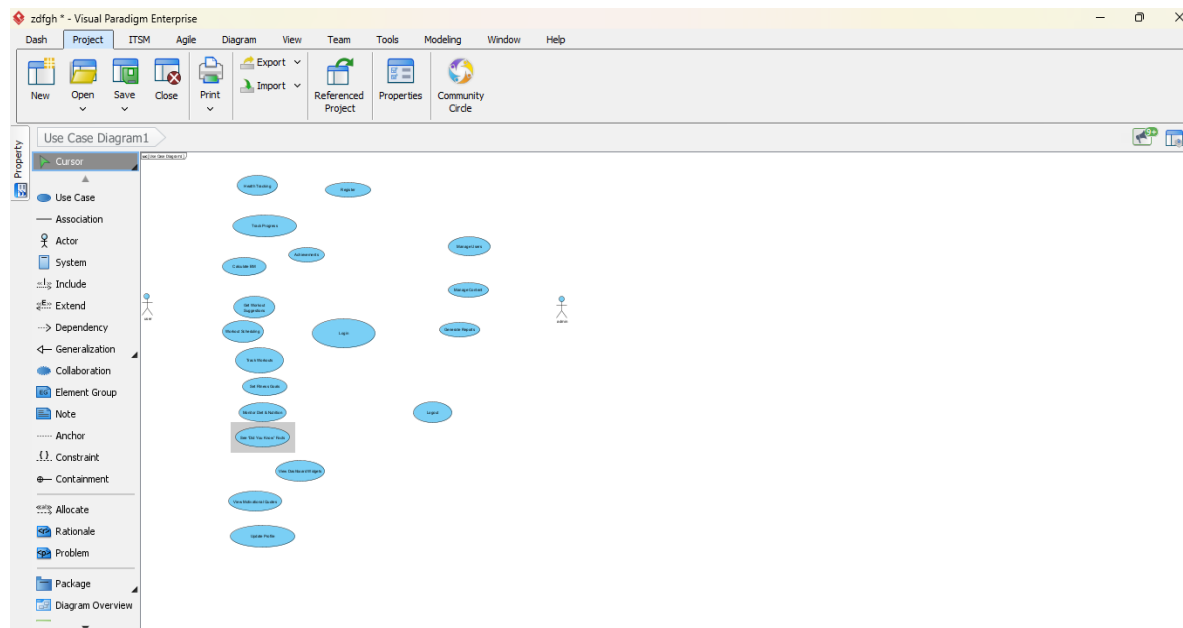
Figure 1.5- 2: Define use cases

4. Draw the System Boundary

- o Encapsulate all use cases within a labeled box titled **"Fitness Tracking System"** to represent the scope of the system.
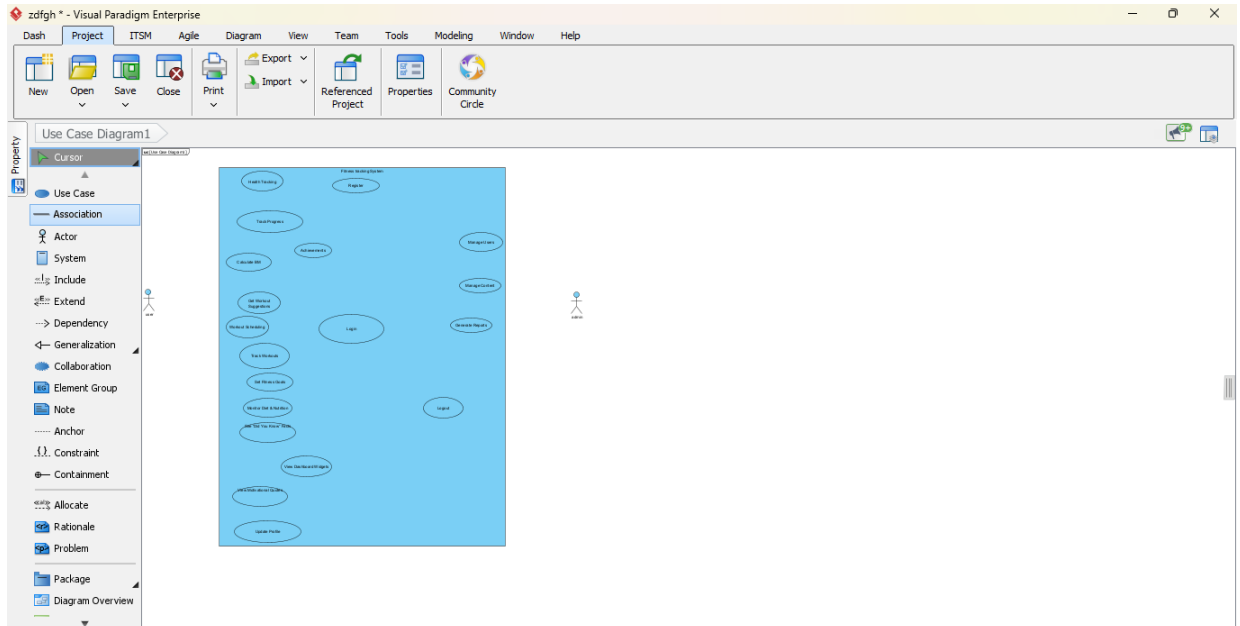
Figure 1.5- 3: Draw the System Boundary

5. Establish Relationships

- o Use **association lines** to connect actors with the use cases they participate in.
- o Apply **<<include>>** relationships for use cases that are always called as part of another.
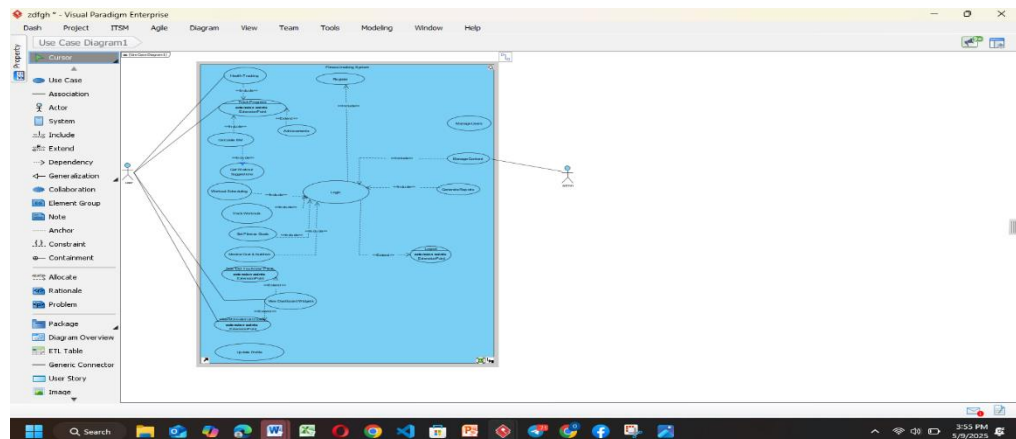- o Apply **<<extend>>** for optional behaviors or conditional processes.

Figure 1.5- 4: Establish relationships

6. Review and Validate

Ensure that all functional requirements are represented and that relationships between actors and use cases are accurate. Validate against the original requirement specification.
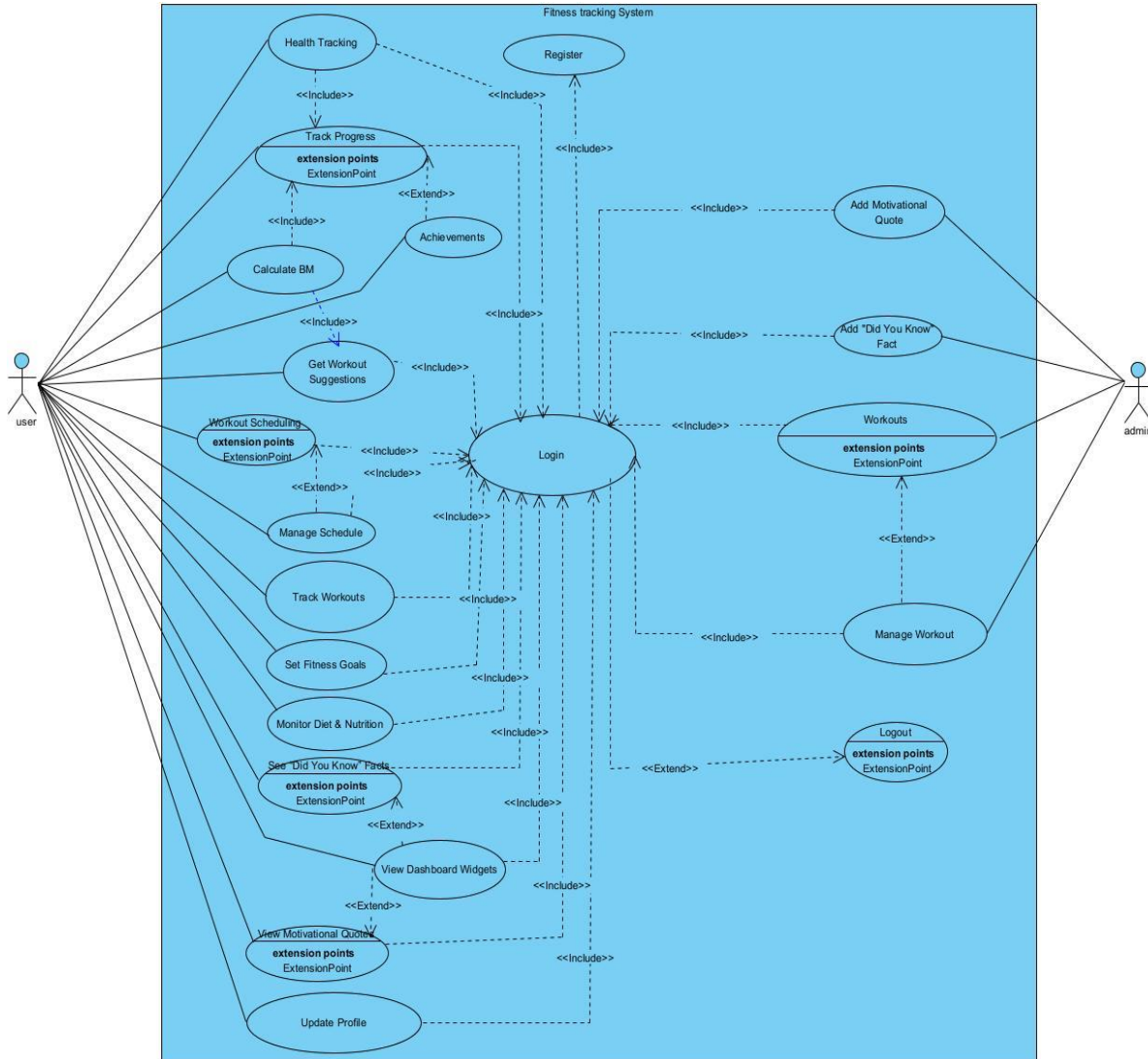
Figure 1.5- 5: Review use case

Chapter Two

2.1. High Level Sequence Diagram

A high-level sequence diagram is a UML (Unified Modeling Language) tool used to visualize the overall flow of interactions in a system. It emphasizes the communication between various objects or system components and the chronological sequence of messages exchanged. This type of diagram typically illustrates the key actors, system boundaries, and the main steps involved in completing a process, offering a simplified yet structured view of system behavior.

Explanation of the components typically found in a high-level sequence diagram:

1. Objects/Components:

    - Objects or components participating in the interactions are represented as vertical lines (lifelines) on the diagram. Each lifeline represents an instance of an object or a component.

2. Messages:

    - Messages or actions exchanged between the objects/components are depicted as arrows between the lifelines. The arrows indicate the flow of communication or control between the objects/components.

    - Messages can be labelled to indicate the type of message, such as method calls, signals, or events. They may also include any parameters or arguments being passed.

3. Lifeline Ordering:

    - The vertical ordering of the lifelines on the diagram represents the sequence of interactions. Lifelines are typically arranged in the order in which the messages are exchanged, indicating the flow of control or data between the objects/components.

4. Activation Bars:

    - Activation bars, also known as activation rectangles, can be used to represent the duration or lifespan of an operation or method call. They show the period during which an object/component is actively processing a specific message.

5. System Boundary (Optional):

    - In some cases, a high-level sequence diagram may include a system boundary or scope box. This box visually represents the system being modelled and helps to define its boundaries.

2.2. Component of High-level Sequence Diagram

A high-level sequence diagram typically includes the following key components:

- **Actors**: Represent external entities or users that interact with the system. Depicted as stick figures or labeled boxes, actors initiate the flow of events in the system.
- **Lifelines**: Indicate the participating objects or system components. Shown as vertical dashed lines, each lifeline corresponds to an object or component involved in the sequence.
- **Messages**: Represent the communication between lifelines. These are illustrated as arrows and show the flow of information or control. Messages can be synchronous or asynchronous and may include parameters or return values.
- **Activation Bars**: Also known as execution occurrences, these bars appear on lifelines to show when an object is actively performing an operation. They are vertical rectangles and indicate processing periods triggered by received messages.
- **Control Flow**: Demonstrates the chronological order in which messages are exchanged. This flow is represented by the sequence and positioning of messages across lifelines, helping to visualize the logical dependencies and timing of actions.
- **Optional Fragments**: Used to model alternative paths, conditions, or repeated actions. These are enclosed in frames labeled with keywords like `alt`, `opt`, or `loop`, and represent specific logic or branches in the interaction flow.
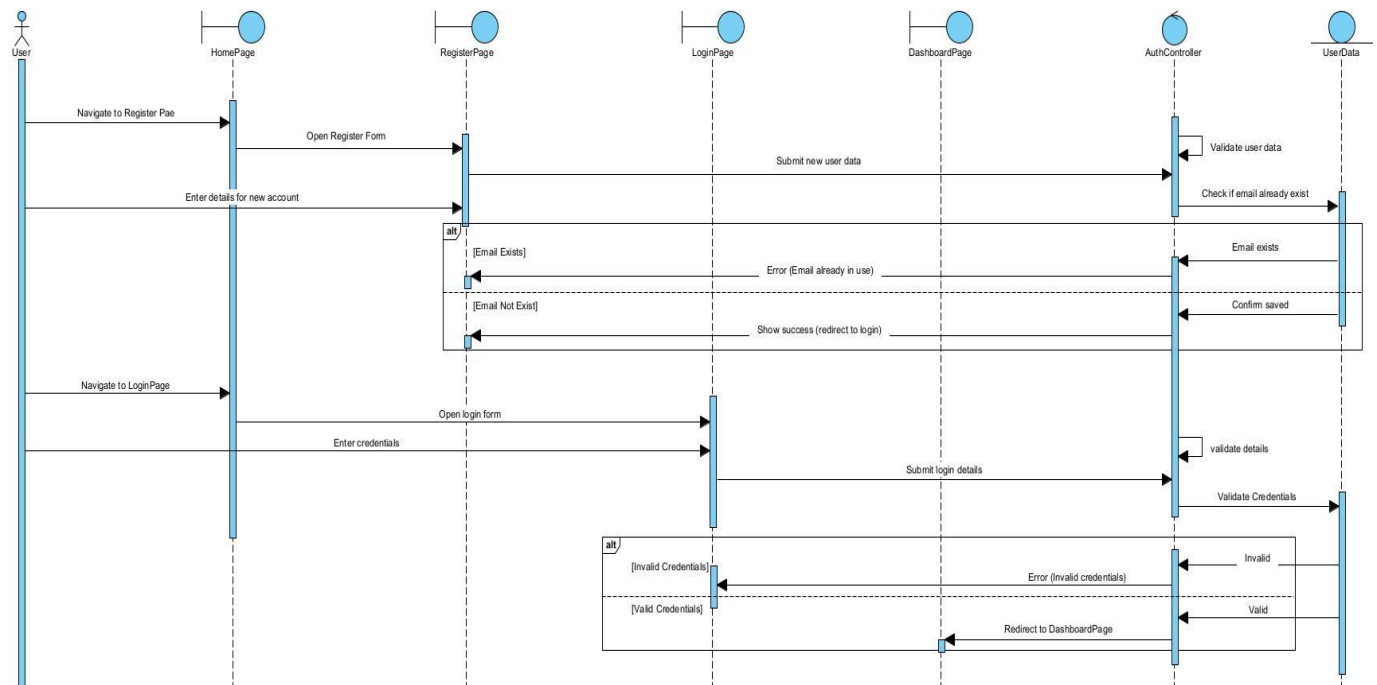
2.3. Example of High Level Sequence

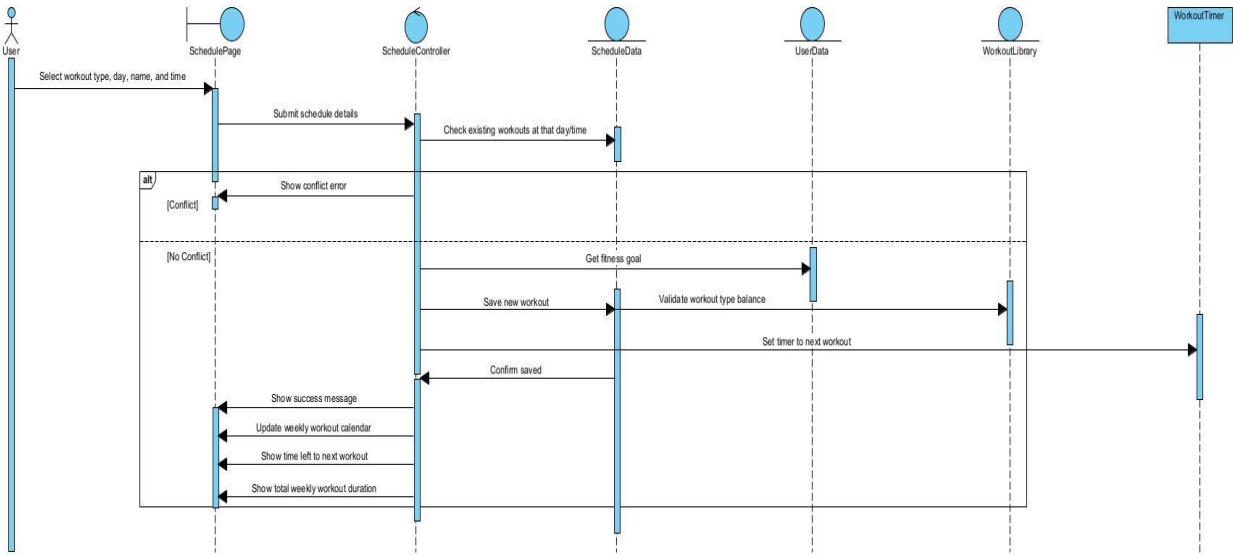**Figure 2.3- 1**: Registration and login
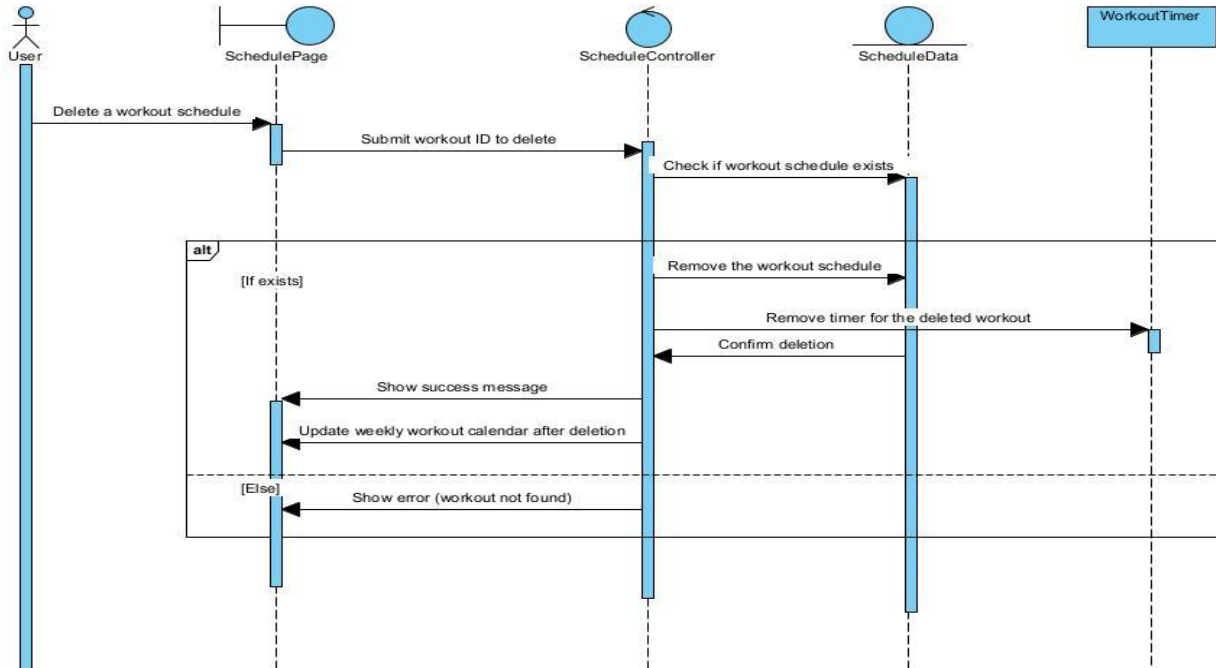
Figure 2.3- 2: Workout scheduling

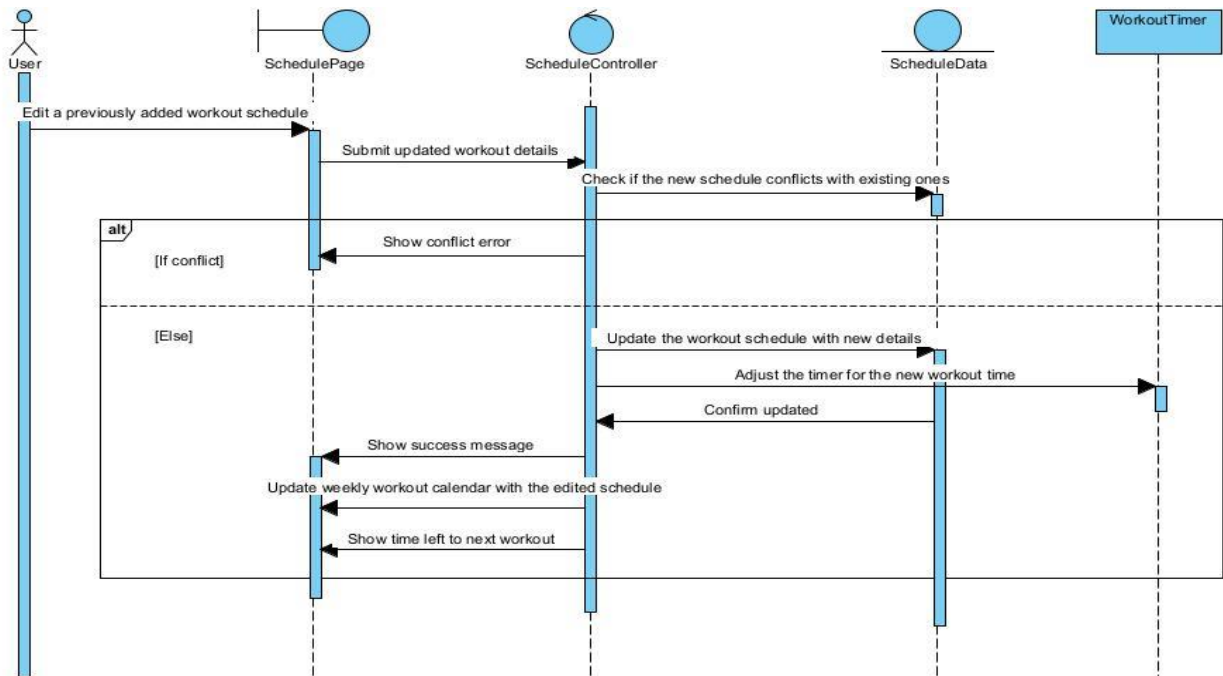**Figure 2.3- 3**: Delete schedule

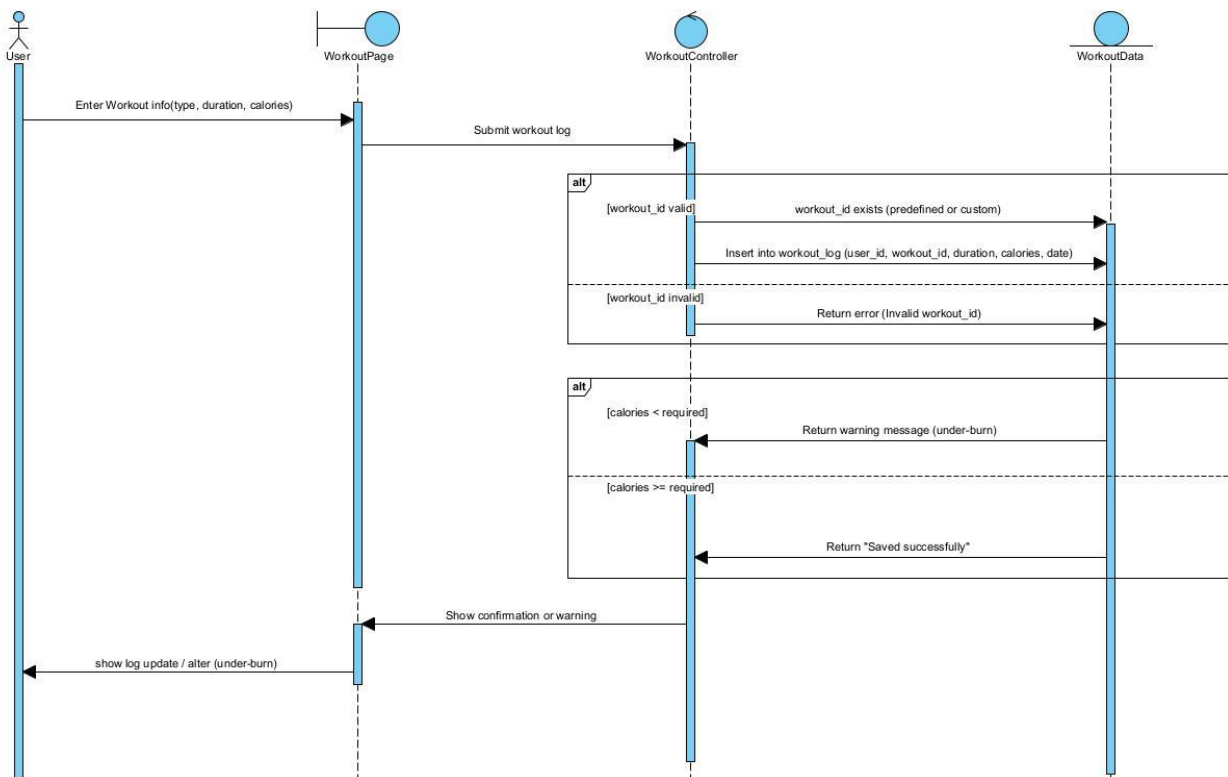**Figure 2.3- 4:** Edit schedule
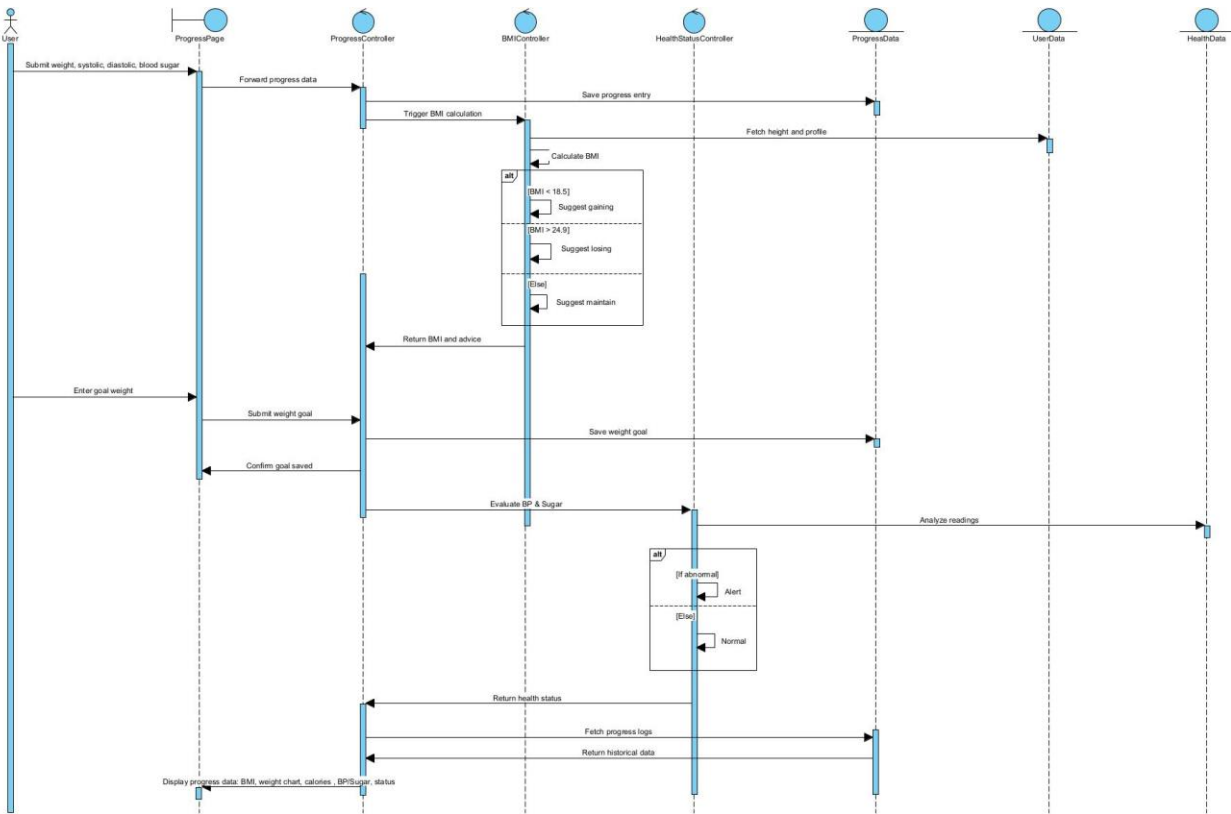

Figure 2.3- 5: Track workout

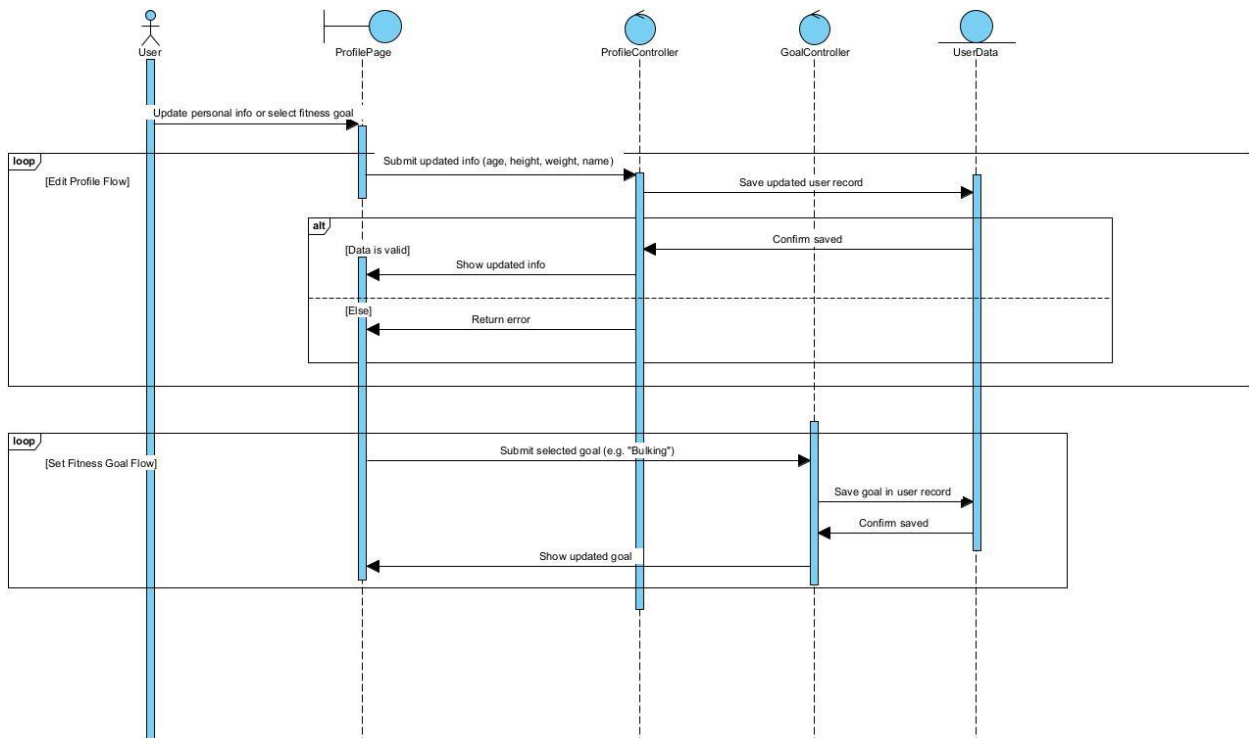**Figure 2.3- 6:** Track progress and health

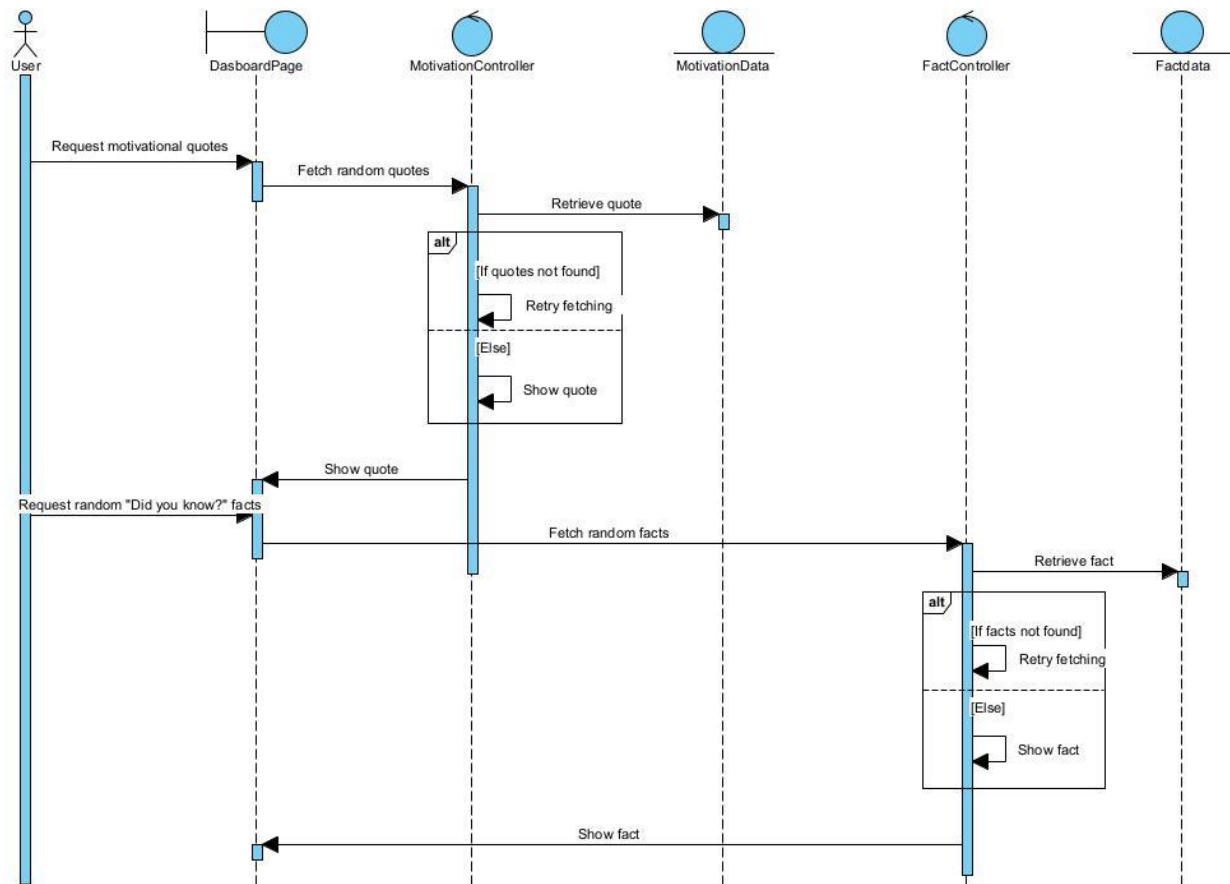**Figure 2.3- 7:** Set fitness and edit profile
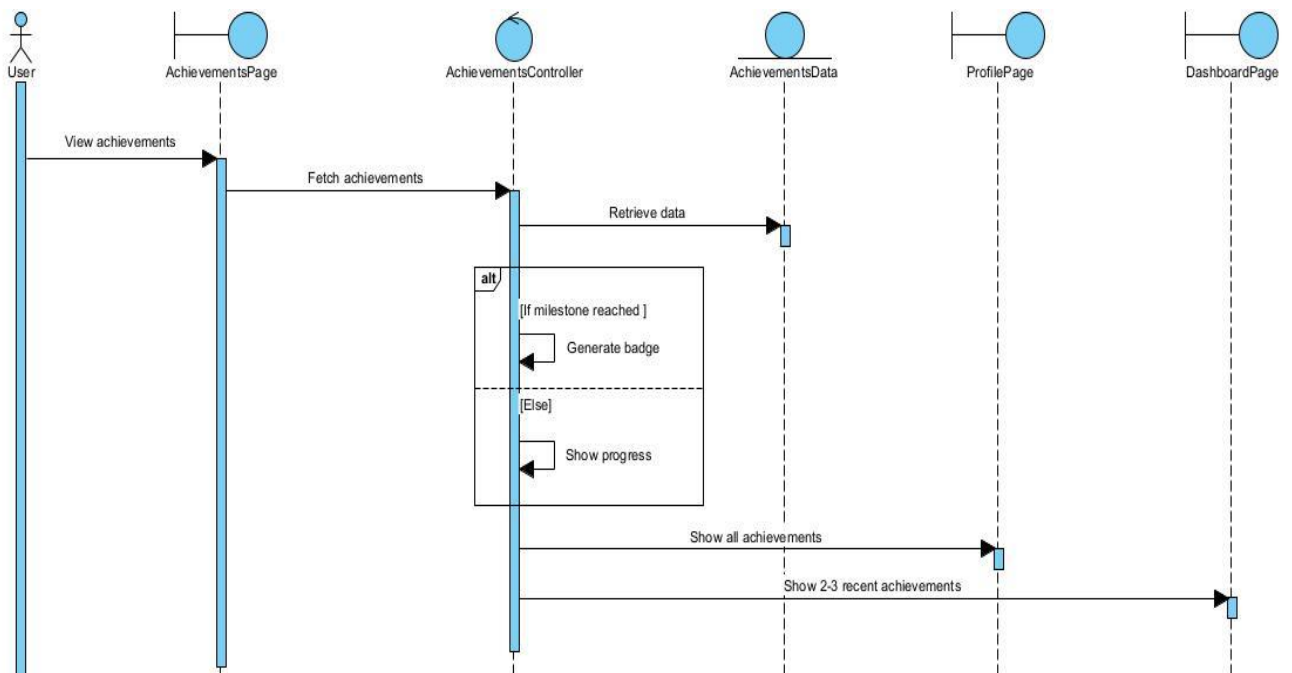
Figure 2.3- 8: Did you know and motivation
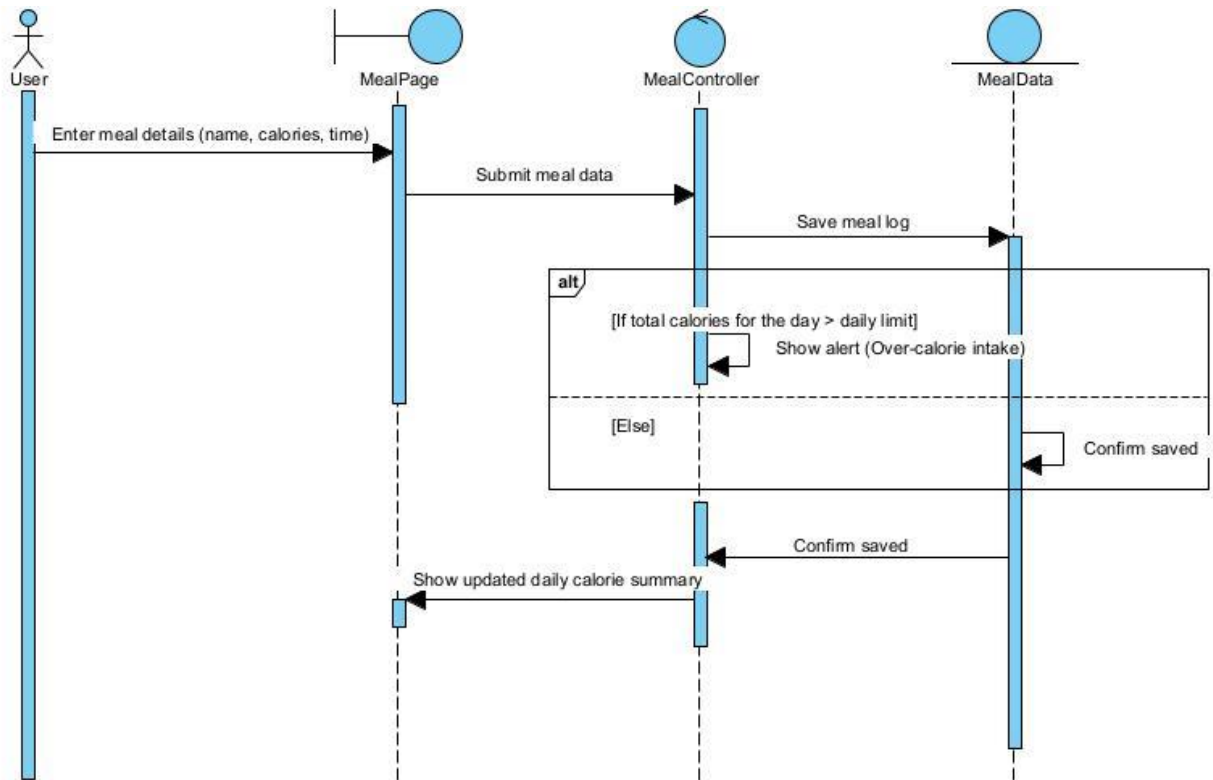

Figure 2.3- 9: Achievements
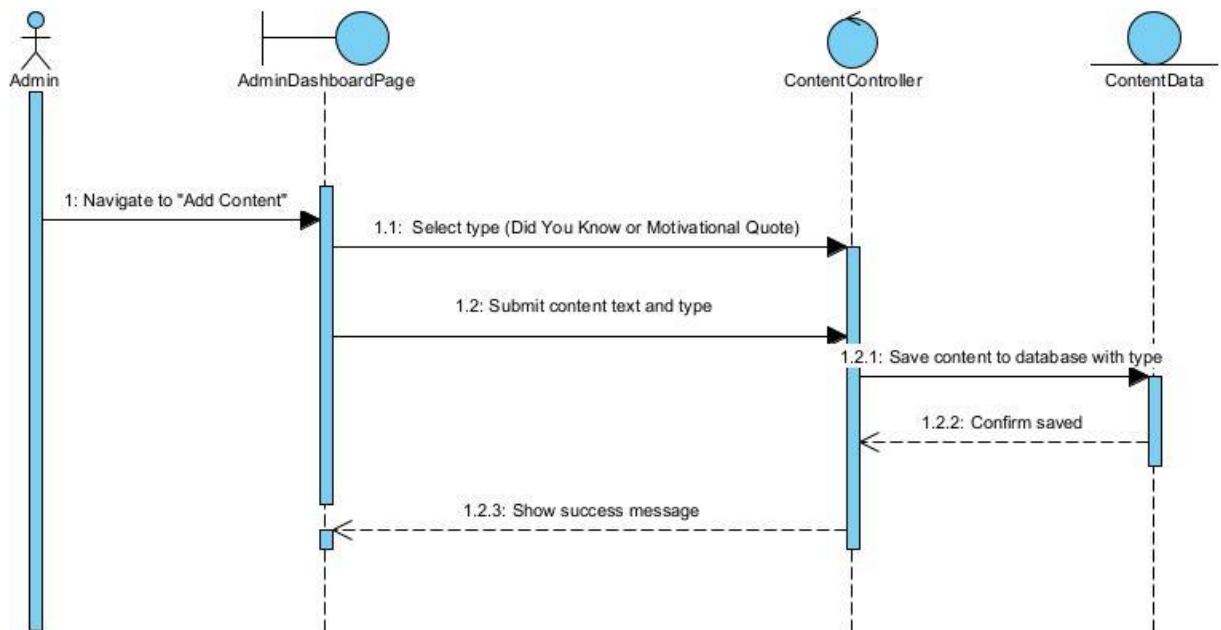
Figure 2.3- 10: Monitor diet

Figure 2.3- 1: Add content
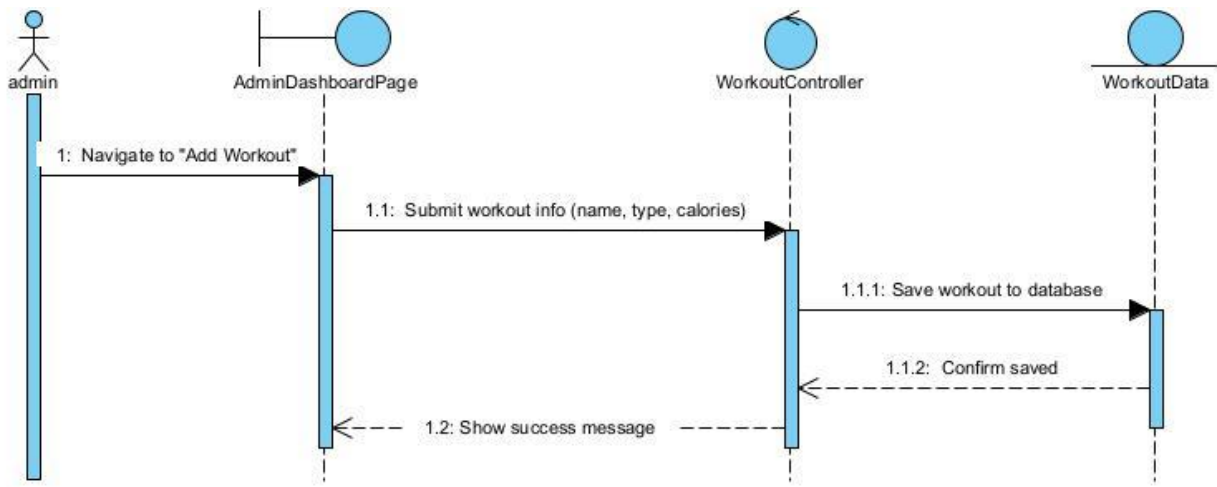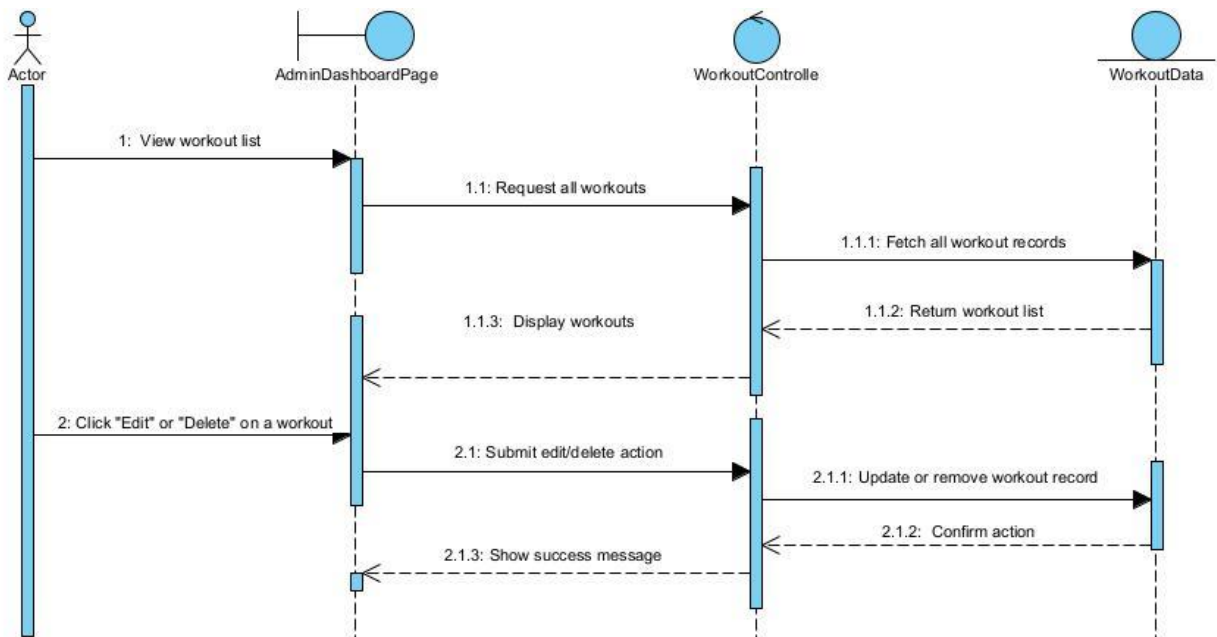
Figure 2.3- 2: Add workout

Figure 2.3- 3: Manage workout

2.4. Tools and Steps to Draw High Level Sequence Diagram

Tool Used

- **Visual Paradigm**: A professional UML modeling tool that supports various diagram types including sequence diagrams. It provides drag-and-drop features, easy formatting, and clear lifeline/message management for accurate and neat UML design.

Steps to Create the Diagram:.

1. **Add Actors and Lifelines**
   o Drag and drop **actors** (e.g., Admin, User) to the left side.
   o Add **lifelines** (e.g., AdminDashboardPage, Controller, Data Entity) to represent system components.
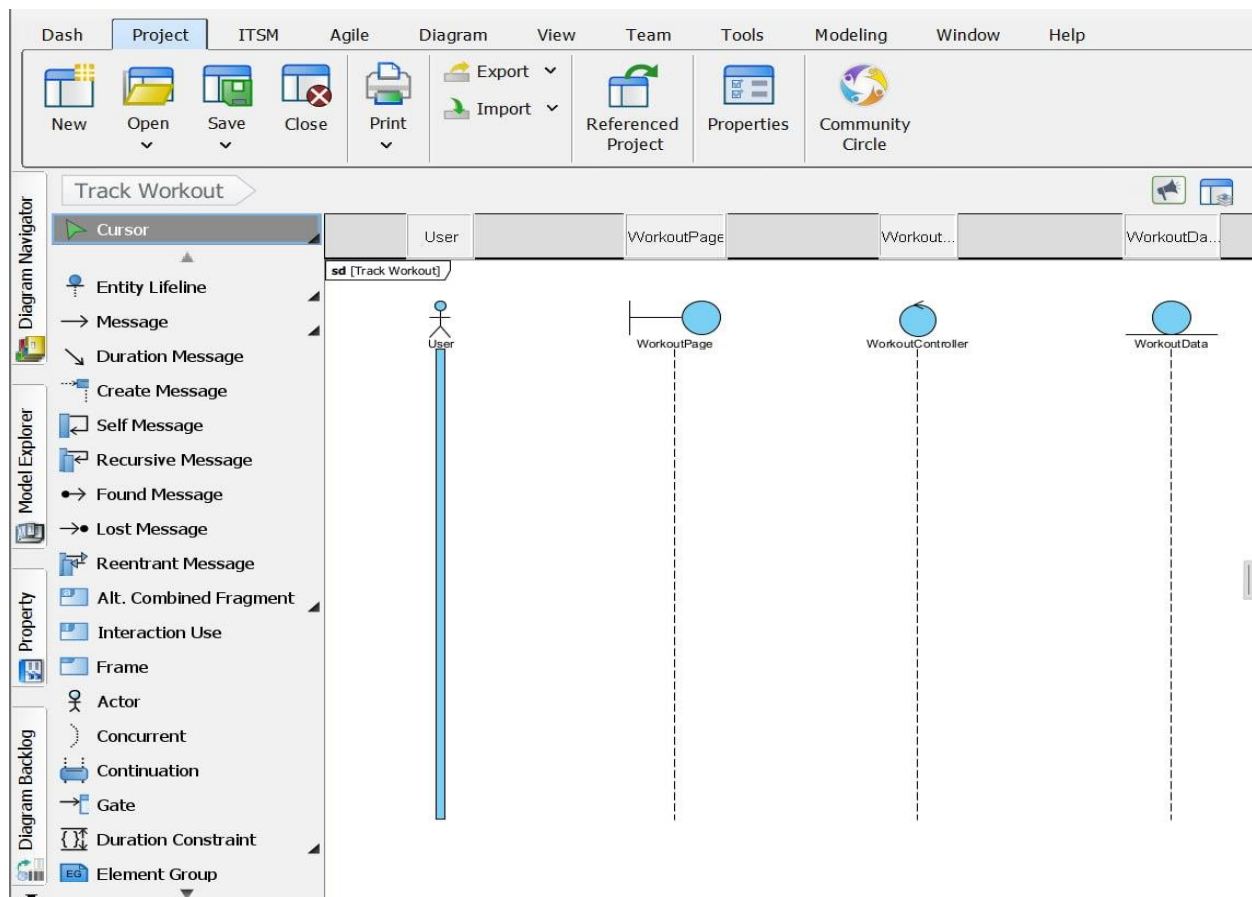
Figure 2.4- 1: Add lifelines

2. **Insert Messages**

- Use **arrows** to show the flow of messages between lifelines.
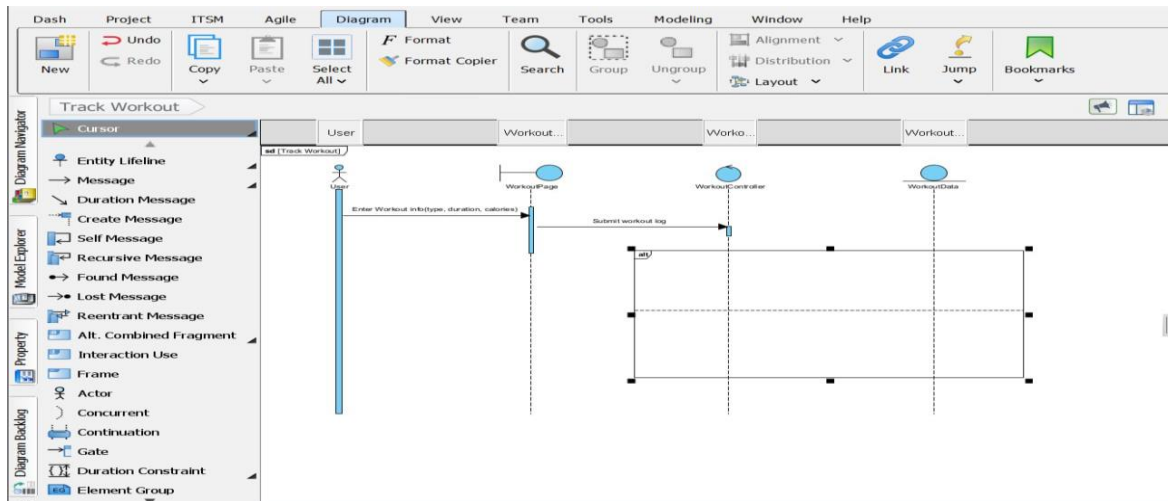
Figure 2.4- 2: Insert messages

3. **Use Activation Bars**
   Add **activation bars** to lifelines to represent periods of execution when an object or component is performing an action.
4. **Organize the Control Flow**
   Ensure all messages are arranged top-down in chronological order. Use **return arrows** if needed to show responses or confirmations.
5. **Add Optional Fragments (if applicable)**

* Use fragments like **alt**, **opt**, or **loop** to show conditional flows or repeated actions (e.g., optional success messages or error handling).

Figure 2.4- 3: Finished sequence diagram

Chapter Three

3.1 Low-level (Detail) Design (class design)

Low-level design often referred to as detailed design or class design, is a phase in software development where the specifications and architecture defined in the high-level design are refined into detailed design documents.

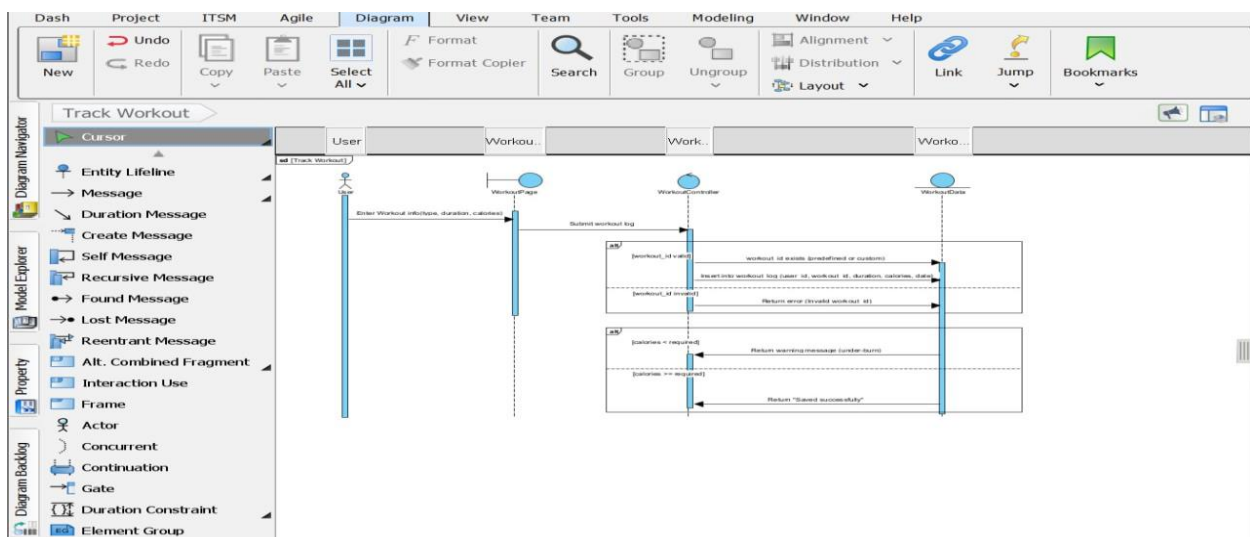| Concept	Description
Classes and Objects	Define the essential classes and objects for the system, each with a clear responsibility and encapsulated behavior.
Attributes and Methods	Identify key attributes (data members) and methods (functions) for each class to perform required actions and store relevant data.
Visibility	Assign appropriate visibility modifiers (public, private, protected) to attributes and methods to enforce encapsulation and maintain data integrity.
Inheritance	Establish parent-child class relationships to promote code reuse and enable polymorphism where subclasses inherit from base classes.
Association	Describe the logical relationships between classes, including one-to-one, one-to-many, and many-to-many connections.
Aggregation and Composition	Define whole-part relationships where **aggregation** allows parts to exist independently, while **composition** implies dependent lifecycle of parts.
Dependencies	Explain how one class relies on another to function, typically for method calls or object usage, indicating coupling within the system.
Design Patterns	Apply well-known patterns like Singleton, Factory, and Observer to address common design challenges and improve maintainability and scalability.
Interfaces and Abstract Classes	Use interfaces and abstract classes to define common methods and structures, allowing flexible implementation and adherence to defined contracts.

Table 3. 1: Key concepts low-level diagram

3.2. Components of Class Diagram

1. Classes:
• User
• Admin
• Workout
• UserWorkout
• WorkoutLog
• WorkoutSchedule

- Meal
- MealLog
- UserProgress
- Achievement
- UserAchievement
- MotivationalContent

2. Attributes:
- User: user_id, name, email, password, age, height, weight, disease_history, goal
- Admin: admin_id, name, email, password
- Workout: workout_id, name, type, calories_burned_per_hour
- UserWorkout: id, user_id, name, type, calories_burned_per_hour
- WorkoutLog: log_id, user_id, workout_id, workout_day, date, duration, calories_burned
- WorkoutSchedule: schedule_id, user_id, workout_id, schedule_date, time
- Meal: meal_id, name, calories, protein, fat, carbs
- MealLog: meal_log_id, user_id, meal_id, quantity, date
- UserProgress: progress_id, user_id, weight, blood_pressure, sugar_level, date
- Achievement: achievement_id, title, description, type
- UserAchievement: id, user_id, achievement_id, date_achieved
- MotivationalContent: content_id, content_type, text, admin_id, created_at

3. Methods:
- User: register(), login(), updateProfile(), calculateBMI(), trackProgress(), logWorkout(), logMeal(), viewAchievements(), setGoal(), viewDashboardWidgets()
- Admin: login(), manageUsers(), addWorkout(), addMotivationalContent(), editUserProfile(), deleteUser()
- Workout: createWorkout(), updateWorkout(), deleteWorkout(), getAllWorkouts()
- UserWorkout: createCustomWorkout(), getCustomWorkouts()
- WorkoutLog: logWorkout(), calculateCaloriesBurned(), getLogsByUser(), updateWorkoutLog()
- WorkoutSchedule: scheduleWorkout(), updateSchedule(), deleteSchedule(), checkScheduleConflict(), getUserSchedule()
- Meal: createMeal(), updateMeal(), getAllMeals()
- MealLog: logMeal(), updateMealLog(), getDailyCalorieIntake(), checkOverCalorieLimit()
- UserProgress: updateProgress(), viewProgressGraph(), getWeeklyProgress()
- Achievement: defineAchievement(), getAllAchievements()
- UserAchievement: unlockAchievement(), getAchievementsByUser(), hasAchievement()
- MotivationalContent: addContent(), editContent(), deleteContent(), getContentByType()

4. Relationships:
- Aggregation:
- User -- WorkoutLog
- User -- MealLog
- User -- WorkoutSchedule

• Composition:
• User -- UserWorkout
• User -- UserProgress
• User -- UserAchievement

• Association:
• User -- MotivationalContent
• Admin -- User
• Admin -- MotivationalContent
• Admin -- Workout
• Workout -- WorkoutLog
• Workout -- WorkoutSchedule
• Meal -- MealLog
• Achievement – UserAchievement
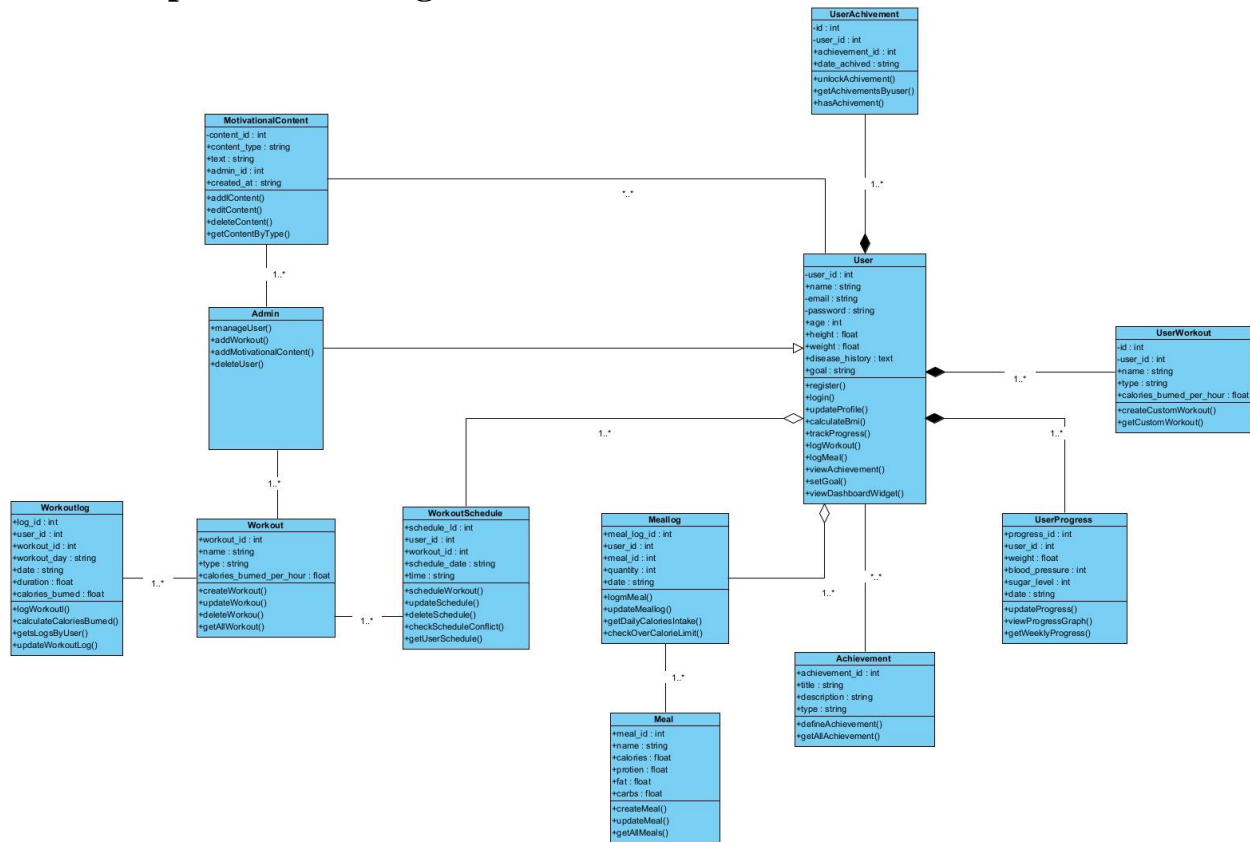
3.3. Example of Class Diagram

UserAchievement
-id : int
-user_id : int
+achievement_id : int
+date_achived : string
+unlockAchievement()
+getAchivementsByuser()
+hasAchievement()

MotivationalContent
-content_id : int
+content_type : string
+text : string
+admin_id : int
+created_at : string
+addlContent()
+editContent()
+deleteContent()
+getContentByType()

User
-user_id : int
+name : string
-email : string
-password : string
+age : int
+height : float
+weight : float
+disease_history : text
+goal : string
+register()
+login()
+updateProfile()
+calculateBmi()
+trackProgress()
+logWorkout()
+logMeal()
+viewAchievement()
+setGoal()
+viewDashboardWidget()

UserWorkout
-id : int
-user_id : int
+name : string
+type : string
+calories_burned_per_hour : float
+createCustomWorkout()
+getCustomWorkout()

Admin
+manageUser()
+addWorkout()
+addMotivationalContent()
+deleteUser()

Workoutlog
+log_id : int
+user_id : int
+workout_id : int
+workout_day : string
+date : string
+duration : float
+calories_burned : float
+logWorkout()
+calculateCaloriesBurned()
+getsLogsByUser()
+updateWorkoutLog()

Workout
+workout_id : int
+name : string
+type : string
+calories_burned_per_hour : float
+createWorkout()
+updateWorkou()
+deleteWorkout()
+getAllWorkout()

WorkoutSchedule
+schedule_id : int
+user_id : int
+workout_id : int
+schedule_date : string
+time : string
+scheduleWorkout()
+updateSchedule()
+deleteSchedule()
+checkScheduleConflict()
+getUserSchedule()

MealLog
+meal_log_id : int
+user_id : int
+meal_id : int
+quantity : int
+date : string
+logmMeal()
+updateMeallog()
+getDailyCaloriesIntake()
+checkOverCalorieLimit()

Meal
+meal_id : int
+name : string
+calories : float
+protien : float
+fat : float
+carbs : float
+createMeal()
+updateMeal()
+getAllMeals()

Achievement
+achievement_id : int
+title : string
+description : string
+type : string
+defineAchievement()
+getAllAchievement()

UserProgress
+progress_id : int
+user_id : int
+weight : float
+blood_pressure : int
+sugar_level : int
+date : string
+updateProgress()
+viewProgressGraph()
+getWeeklyProgress()

Figure 3.3- 1: Class diagram

3.4. Tools and steps to draw low level design

1. Identify Classes

- Review the requirements or use cases of the fitness tracking system to identify the main classes needed to implement the functionality. Each class should represent a distinct entity or concept in the system.
- Examples: **User**, **Admin**, **Workout**, **UserWorkout**, **WorkoutLog**, **WorkoutSchedule**, **Meal**, **MealLog**, **UserProgress**, **Achievement**, **UserAchievement**, **MotivationalContent**

2. Define Attributes

- For each class, define the attributes (properties or data) that describe its state. Consider what information each object of the class needs to store.
- Examples:
    - **User**: userID, name, email, password, age, height, weight, diseaseHistory, goal
    - **Admin**: adminID, name, email, password
    - **Workout**: workoutID, name, type, caloriesBurnedPerHour
    - **UserWorkout**: id, userID, name, type, caloriesBurnedPerHour
    - **WorkoutLog**: logID, userID, workoutID, date, duration, caloriesBurned
    - **WorkoutSchedule**: scheduleID, userID, workoutID, scheduleDate, time
    - **Meal**: mealID, name, calories, protein, fat, carbs
    - **MealLog**: mealLogID, userID, mealID, quantity, date
    - **UserProgress**: progressID, userID, weight, bloodPressure, sugarLevel, date
    - **Achievement**: achievementID, title, description, type
    - **UserAchievement**: id, userID, achievementID, dateAchieved
    - **MotivationalContent**: contentID, contentType (quote, tip, fact), text, adminID, createdAt

3. Determine Methods

- Determine the methods (behaviors or operations) that objects of each class can perform. Think about what actions or tasks need to be implemented for the class to fulfill its responsibilities.
- Examples:
    - **User**: register(), login(), updateProfile(), calculateBMI(), logWorkout(), logMeal(), trackProgress(), setGoal(), viewAchievements()
    - **Admin**: login(), manageUsers(), addMotivationalContent(), manageWorkouts(), manageMeals()
    - **Workout**: createWorkout(), updateWorkout(), deleteWorkout()
    - **UserWorkout**: createCustomWorkout(), viewCustomWorkouts()
    - **WorkoutLog**: logWorkout(), calculateCaloriesBurned(), updateLog()
    - **WorkoutSchedule**: scheduleWorkout(), updateSchedule(), cancelSchedule()
    - **Meal**: createMeal(), updateMeal(), deleteMeal()

- **MealLog**: logMeal(), getTotalCaloriesForDate(), updateMealLog()
- **UserProgress**: updateProgress(), viewProgressGraph(), getWeeklyProgress()
- **Achievement**: defineAchievement(), getAllAchievements()
- **UserAchievement**: unlockAchievement(), getAchievementsByUser()
- **MotivationalContent**: addContent(), editContent(), deleteContent(), getRandomContent()

4. Identify Relationships

- Identify the relationships between classes. This includes associations, aggregations, compositions, inheritance, and dependencies. Consider how classes interact with each other and share information.
- Examples:
  - **User** logs **WorkoutLog** and **MealLog**
  - **User** has **UserWorkout**, **UserProgress**, **UserAchievement**, and **WorkoutSchedule**
  - **Admin** manages **MotivationalContent**, **Workout**, and **Meal**
  - **Workout** and **Meal** are used in logs and schedules
  - **Achievement** is related to **UserAchievement**
  - **User** views **MotivationalContent**

5. Refine Class Responsibilities

- Review and refine the responsibilities of each class. Ensure that each class has a clear and cohesive purpose and that its attributes and methods are relevant to that purpose.
- Example responsibilities:
  - **User**: Handles user registration, login, profile updates, and logs workout/meal data
  - **Admin**: Manages content and user-related actions at a system level
  - **Workout**: Defines general workout types and calories burned info
  - **UserWorkout**: Represents user-customized workouts
  - **WorkoutLog**: Records actual performed workouts with stats
  - **WorkoutSchedule**: Manages future workout planning
  - **Meal**: Defines meal items and nutritional values
  - **MealLog**: Records food consumption
  - **UserProgress**: Stores health data over time
  - **Achievement**: Defines milestones users can unlock
  - **UserAchievement**: Links users with their earned achievements
  - **MotivationalContent**: Provides quotes, facts, and tips for user motivation

6. Draw the Class Diagram

- Using your chosen diagramming tool, start drawing the class diagram based on the identified classes, attributes, methods, and relationships. Arrange the classes and relationships in a logical and organized manner.
- Example tools: Visual Paradignm

7. Add Details

- Add additional details to the class diagram as needed, such as:
    - **Visibility modifiers**: + (public), - (private), # (protected)
    - **Data types** for attributes
    - **Method signatures**
    - **Multiplicity** for associations

8. Review and Iterate

- Review the class diagram to ensure that it accurately represents the design of the fitness tracking system. Make any necessary iterations or refinements based on feedback or changes to requirements.
    - Solicit feedback from stakeholders or instructors
    - Verify alignment with documented use cases and system requirements
    - Update diagram as project scope evolves

Chapter Four

4.1 Implementation

Implementation refers to the process of translating a plan, idea, model, or design into an operational form. In the context of software development, it means converting system designs, diagrams, and specifications into executable code. This involves writing the actual code that performs the functions and processes defined in the design documents and class diagrams.

```java
public class Admin extends User {

  public void manageUser() {
    // TODO - implement Admin.manageUser
    throw new UnsupportedOperationException();
  }

  public void addWorkout() {
    // TODO - implement Admin.addWorkout
    throw new UnsupportedOperationException();
  }

  public void addMotivationalContent() {
    // TODO - implement Admin.addMotivationalContent
    throw new UnsupportedOperationException();
  }

  public void deleteUser() {
    // TODO - implement Admin.deleteUser
    throw new UnsupportedOperationException();
  }

}
```

Figure 4.1- 1: Admin class

```
public class User {

    public int user_id;
    public string name;
    public string email;
    public string password;
    public int age;
    public float height;
    public float weight;
    public text disease_history;
    public string goal;

    public void register() {
        // TODO - implement User.register
        throw new UnsupportedOperationException();
    }

    public void login() {
        // TODO - implement User.login
        throw new UnsupportedOperationException();
    }

    public void updateProfile() {
        // TODO - implement User.updateProfile
        throw new UnsupportedOperationException();
    }

    public void calculateBmi() {
        // TODO - implement User.calculateBmi
        throw new UnsupportedOperationException();
    }

    public void trackProgress() {
        // TODO - implement User.trackProgress
        throw new UnsupportedOperationException();
    }

    public void logWorkout() {
        // TODO - implement User.logWorkout
        throw new UnsupportedOperationException();
    }

    public void logMeal() {
        // TODO - implement User.logMeal
        throw new UnsupportedOperationException();
    }

    public void viewAchievement() {
        // TODO - implement User.viewAchievement
        throw new UnsupportedOperationException();
    }

    public void setGoal() {
        // TODO - implement User.setGoal
        throw new UnsupportedOperationException();
    }

    public void viewDashboardWidget() {
```

Figure 4.1- 2: User class

```
1   public class Achievement {
2
3       public int achievement_id;
4       public string title;
5       public string description;
6       public string type;
7
8       public void defineAchievement() {
9           // TODO - implement Achievement.defineAchievement
10          throw new UnsupportedOperationException();
11      }
12
13      public void getAllAchievement() {
14          // TODO - implement Achievement.getAllAchievement
15          throw new UnsupportedOperationException();
16      }
17
18  }
```

Figure 4.1- 3: Achievements class

```
public class Meal {

    public int meal_id;
    public string name;
    public float calories;
    public float protien;
    public float fat;
    public float carbs;

    public void createMeal() {
        // TODO - implement Meal.createMeal
        throw new UnsupportedOperationException();
    }

    public void updateMeal() {
        // TODO - implement Meal.updateMeal
        throw new UnsupportedOperationException();
    }

    public void getAllMeals() {
        // TODO - implement Meal.getAllMeals
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 4: Meal class

```
public class Meallog {

    public int meal_log_id;
    public int user_id;
    public int meal_id;
    public int quantity;
    public string date;

    public void logmMeal() {
        // TODO - implement Meallog.logmMeal
        throw new UnsupportedOperationException();
    }

    public void updateMeallog() {
        // TODO - implement Meallog.updateMeallog
        throw new UnsupportedOperationException();
    }

    public void getDailyCaloriesIntake() {
        // TODO - implement Meallog.getDailyCaloriesIntake
        throw new UnsupportedOperationException();
    }

    public void checkOverCalorieLimit() {
        // TODO - implement Meallog.checkOverCalorieLimit
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 5: Meal log class

```
public class MotivationalContent {

    public int content_id;
    public string content_type;
    public string text;
    public int admin_id;
    public string created_at;

    public void addlContent() {
        // TODO - implement MotivationalContent.addlContent
        throw new UnsupportedOperationException();
    }

    public void editContent() {
        // TODO - implement MotivationalContent.editContent
        throw new UnsupportedOperationException();
    }

    public void deleteContent() {
        // TODO - implement MotivationalContent.deleteContent
        throw new UnsupportedOperationException();
    }

    public void getContentByType() {
        // TODO - implement MotivationalContent.getContentByType
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 6: Motivational content class

```
public class UserAchivement {

  public int id;
  public int user_id;
  public int achievement_id;
  public string date_achived;

  public void unlockAchivement() {
      // TODO - implement UserAchivement.unlockAchivement
      throw new UnsupportedOperationException();
  }

  public void getAchivementsByuser() {
      // TODO - implement UserAchivement.getAchivementsByuser
      throw new UnsupportedOperationException();
  }

  public void hasAchivement() {
      // TODO - implement UserAchivement.hasAchivement
      throw new UnsupportedOperationException();
  }

}
```

Figure 4.1- 7: User achievement class

```
public class UserProgress {

    public int progress_id;
    public int user_id;
    public float weight;
    public int blood_pressure;
    public int sugar_level;
    public string date;

    public void updateProgress() {
        // TODO - implement UserProgress.updateProgress
        throw new UnsupportedOperationException();
    }

    public void viewProgressGraph() {
        // TODO - implement UserProgress.viewProgressGraph
        throw new UnsupportedOperationException();
    }

    public void getWeeklyProgress() {
        // TODO - implement UserProgress.getWeeklyProgress
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 8: User progress class

```
public class Workout {

    public int workout_id;
    public string name;
    public string type;
    public float calories_burned_per_hour;

    public void createWorkout() {
        // TODO - implement Workout.createWorkout
        throw new UnsupportedOperationException();
    }

    public void updateWorkou() {
        // TODO - implement Workout.updateWorkou
        throw new UnsupportedOperationException();
    }

    public void deleteWorkou() {
        // TODO - implement Workout.deleteWorkou
        throw new UnsupportedOperationException();
    }

    public void getAllWorkout() {
        // TODO - implement Workout.getAllWorkout
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 9: Workout class

```
public class UserWorkout {

    public int id;
    public int user_id;
    public string name;
    public string type;
    public float calories_burned_per_hour;

    public void createCustomWorkout() {
        // TODO - implement UserWorkout.createCustomWorkout
        throw new UnsupportedOperationException();
    }

    public void getCustomWorkout() {
        // TODO - implement UserWorkout.getCustomWorkout
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 10: User workout

```
public class Workoutlog {

    public int log_id;
    public int user_id;
    public int workout_id;
    public string workout_day;
    public string date;
    public float duration;
    public float calories_burned;

    public void logWorkoutl() {
        // TODO - implement Workoutlog.logWorkoutl
        throw new UnsupportedOperationException();
    }

    public void calculateCaloriesBurned() {
        // TODO - implement Workoutlog.calculateCaloriesBurned
        throw new UnsupportedOperationException();
    }

    public void getsLogsByUser() {
        // TODO - implement Workoutlog.getsLogsByUser
        throw new UnsupportedOperationException();
    }

    public void updateWorkoutLog() {
        // TODO - implement Workoutlog.updateWorkoutLog
        throw new UnsupportedOperationException();
    }

}
```

Figure 4.1- 11: Workout log class

```
public class WorkoutSchedule {

    public int schedule_Id;
    public int user_id;
    public int workout_id;
    public string schedule_date;
    public string time;

    public void scheduleWorkout() {
        // TODO - implement WorkoutSchedule.scheduleWorkout
        throw new UnsupportedOperationException();
    }

    public void updateSchedule() {
        // TODO - implement WorkoutSchedule.updateSchedule
        throw new UnsupportedOperationException();
    }

    public void deleteSchedule() {
        // TODO - implement WorkoutSchedule.deleteSchedule
        throw new UnsupportedOperationException();
    }

    public void checkScheduleConflict() {
        // TODO - implement WorkoutSchedule.checkScheduleConflict
        throw new UnsupportedOperationException();
    }

    public void getUserSchedule() {
        // TODO - implement WorkoutSchedule.getUserSchedule
        throw new UnsupportedOperationException();
    }
}
```

Figure 4.1- 12: Workout Schedule class

4.2 Steps to Generate Code from Class Diagram

Step 1: Identify Classes and Relationships
Begin by reviewing the class diagram to understand the system's structure. Identify the main classes along with their attributes and methods. Also, pay attention to how classes relate to each other through associations, inheritance, or other relationships.

Step 2: Define Classes:
Create each class in your chosen programming language. Every class from the diagram should be defined as a separate class in the code, following proper naming conventions and object-oriented design principles.

Step 3: Add Attributes:
Within each class, define the attributes (also called variables or fields) as specified in the class diagram. Use appropriate data types and visibility modifiers (like `private` or `protected`) to encapsulate the data.

Step 4: Implement Methods:
Implement the methods (functions) for each class that describe its behavior. These methods should perform the operations or responsibilities outlined in the class diagram and be aligned with your system's use cases.

Step 5: Establish Relationships:
Establish the relationships between classes in the code. This includes implementing associations using object references or lists, and using inheritance (`extends`) where one class inherits the features of another. For example, a subclass may inherit common properties and methods from a superclass.

Step 6: Write Constructors and Destructors:
Add constructors to initialize objects with their attributes upon creation. Although Java does not have explicit destructors, if your system involves managing resources like files or database connections, you should implement proper resource cleanup using techniques like `try-with-resources` or closing methods.

Chapter Five

5.1 Change Management (version control using Git)

Version control also known as source control, is a system that records changes to a file or set of files over time so that you can recall specific versions later. It is widely used in software development to manage and track changes to code, but it can be applied to any set of files. Here are the key concepts and benefits of version control:

1. **Repository**: A database storing the files, along with the history of changes made to them. Repositories can be local (on your own machine) or remote (on a server).

2. **Commit**: A record of what changes were made to the repository at a specific point in time. Commits often include a message describing the changes.

3. **Branch**: A separate line of development. Branches allow multiple developers to work on different features or bug fixes simultaneously without interfering with each other's work.

4. **Merge**: The process of combining changes from different branches. This can be straightforward or complex, depending on how much the branches have diverged.

5. **Conflict**: Occurs when changes from different branches are incompatible. Conflicts must be resolved manually.

6. **Tag**: A marker used to denote specific points in the repository's history, often used to mark releases or significant changes.

Benefits of Version Control

1. **Collaboration**: Multiple developers can work on the same project simultaneously without interfering with each other.

2. **History Tracking**: Every change is recorded, so you can understand what was changed, why, and by whom.

3. **Backup**: The repository acts as a backup. If a file is lost, it can be restored from the repository.

4. **Versioning**: You can create branches and tags to manage different versions of your project, such as development, testing, and production versions.

5. **Conflict Resolution**: When changes conflict, the system can help identify and resolve these conflicts.

6. **Revert Changes**: Mistakes can be undone by reverting to a previous state of the project.

Basic Workflow in Git

Clone: Copying a repository from a remote server to your local machine.

git clone https://github.com/fira-j/WEBCAMPSOFT.git
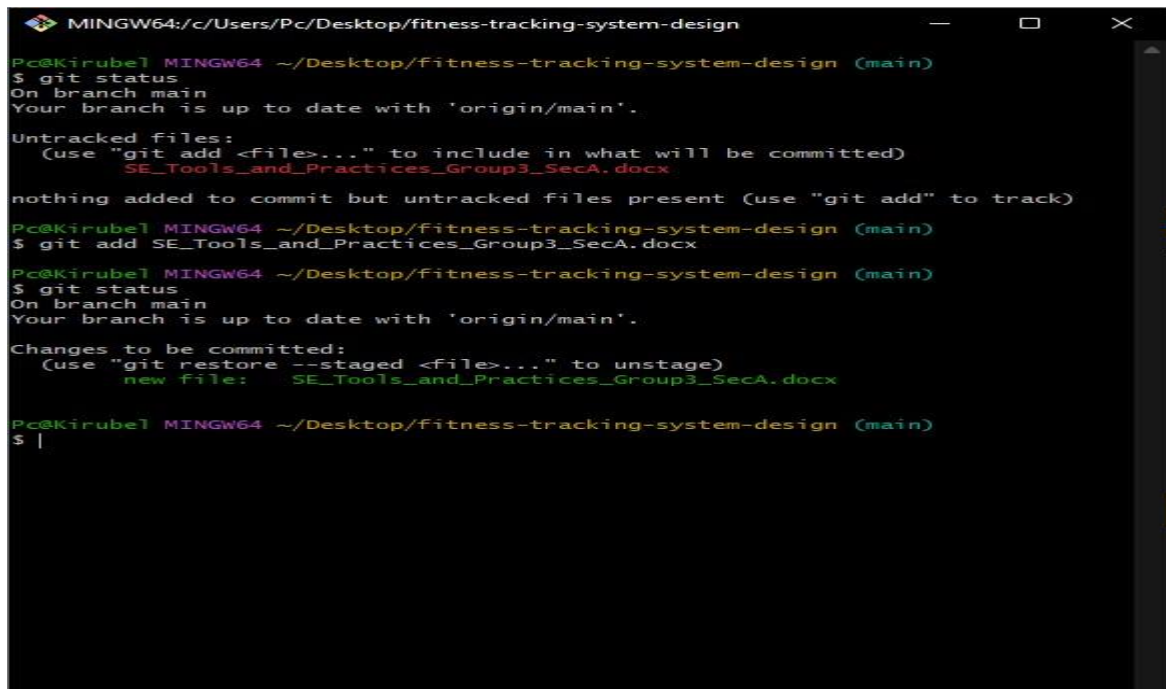
Making Changes: Editing files in your working directory and preparing them for commit.

git add gursha.html

Committing Changes: Saving snapshots of your changes.

git commit -m "message "

Pushing Changes: Uploading your commits to a remote repository.

git push origin feature-branch

Pull Requests: Submitting your branch for review and merging.

- On platforms like GitHub, you create a pull request via the website.
- Team members review the changes, discuss, and suggest modifications.
- Once approved, the branch is merged into the main branch.
- Merging: Combining branches, often from a feature branch into the main branch.

5 .2 Steps and tools that we use in our project to implements git

Step 1:

Clone the repository from GitHub in git bash terminal using the command, git clone <url> as shown in the figure.

Figure 5.2- 1: Git clone

Step 2:

Add changes in the working directory to the staging area. This is a necessary step before committing the changes to the repository, in the terminal using the command git add <file> as shown in the figure.

Figure 5.2- 2: Git add

Step 3:

Save our changes to the local repository. When we commit, we record the changes that we've staged (using git add) and add a descriptive message explaining what changes were made. Here's a detailed overview of how to use git commit effectively in the figure.

Figure 5.2- 3: Git commit

Step 4:

Upload local repository content to a remote repository. It is one of the key commands in Git for collaboration and synchronization with remote repositories. With the command git push as shown in the figure.

Figure 5.2- 4: Git push

Chapter Six

6.1 Unit Test

Unit testing is a software testing technique that focuses on verifying individual components or units of code in isolation from the rest of the application. Its main goal is to ensure that each unit functions correctly on its own. A unit is usually the smallest testable part of an application, such as a function, method, or class.

Key Concepts of Unit Testing

Unit testing is based on several key principles that help ensure effective and reliable tests:

- **Isolation:** Each unit is tested separately from the rest of the application. To achieve this, dependencies are often replaced with mock objects or stubs, allowing the test to focus solely on the behavior of the unit under test.
- **Automation:** Unit tests are typically automated, enabling frequent and consistent execution throughout the development cycle. Automation provides developers with immediate feedback on the stability and correctness of their code.
- **Repeatability:** A good unit test should be repeatable, producing the same result every time it is run. This consistency ensures that tests are reliable and that results are not influenced by external factors.
- **Coverage:** Unit tests should aim to cover all logical paths through the code, including common scenarios and edge cases. Comprehensive test coverage helps detect bugs early in development.
- **Speed:** Since unit tests deal with small, isolated pieces of code, they are designed to run quickly. This allows for rapid feedback and efficient test cycles during development.

Writing Effective Unit Tests

To write unit tests that are both useful and maintainable, consider the following best practices:

- **Keep Tests Small and Focused:** Each test should target a specific aspect of the unit's behavior. This makes the tests easier to read, understand, and maintain.
- **Use Descriptive Names:** Name your test methods in a way that clearly describes what they are verifying. This improves readability and helps developers quickly grasp the purpose of each test.
- **Use the Arrange-Act-Assert Pattern:** Structure tests using the Arrange-Act-Assert (AAA) pattern. First, arrange the necessary setup and inputs. Next, act by calling the unit being tested. Finally, assert that the outcome matches the expected result.
- **Isolate Dependencies:** Replace any dependencies not under test with mock objects or stubs. This ensures the test remains focused on the unit's behavior, not on external systems.
- **Test Edge Cases:** Be sure to include tests that cover edge cases and boundary conditions. This helps verify that the unit can handle unexpected or extreme input gracefully.

JUnit Testing

JUnit is a widely used testing framework for Java applications, primarily designed for unit testing. It offers a simple and standardized way to write and run tests, helping developers ensure their code is correct and reliable. JUnit supports test-driven development (TDD), making it a fundamental tool in many Java development workflows.

Key Concepts of JUnit Testing

One of the core features of JUnit is its use of **annotations** to manage the test lifecycle and structure. These annotations make it easy to define what a test should do and when certain code should run. Here are some of the most commonly used JUnit annotations:

- `@Test`: Marks a method as a test case.
- `@Before`: Runs before each test method. It's typically used to set up any common test data or configurations.
- `@After`: Runs after each test method to perform cleanup tasks.
- `@BeforeClass`: Executes once before any test methods in the class. This is used for setting up resources needed for all tests.
- `@AfterClass`: Executes once after all test methods have run. It's useful for releasing shared resources.
- `@Ignore`: Tells JUnit to skip the annotated test method during execution.

These annotations help keep tests organized, reusable, and easy to manage, making JUnit a powerful tool for maintaining high-quality Java code.

```java
import static org.junit.Assert.*;
public class UserTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {}
    @AfterClass
    public static void tearDownAfterClass() throws Exception {}
    @Before
    public void setUp() throws Exception {}
    @After
    public void tearDown() throws Exception {}
    @Test
    public void testRegister() {fail("Not yet implemented");}
    @Test
    public void testLogin() {fail("Not yet implemented");}
    @Test
    public void testUpdateProfile() {fail("Not yet implemented");}
    @Test
    public void testCalculateBmi() {fail("Not yet implemented");}
    @Test
    public void testTrackProgress() {fail("Not yet implemented");}
    @Test
    public void testLogWorkout() {fail("Not yet implemented");}
    @Test
    public void testLogMeal() {fail("Not yet implemented");}
    @Test
    public void testViewAchievement() {fail("Not yet implemented");}
    @Test
    public void testSetGoal() {fail("Not yet implemented");}
    @Test
    public void testViewDashboardWidget() {fail("Not yet implemented");}
}
```

Figure 6.1- 1: User class test case

```java
import static org.junit.Assert.*;

public class AdminTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testManageUser() {
        fail("Not yet implemented");
    }
    @Test
    public void testAddWorkout() {
        fail("Not yet implemented");
    }
    @Test
    public void testAddMotivationalContent() {
        fail("Not yet implemented");
    }
    @Test
    public void testDeleteUser() {
        fail("Not yet implemented");
    }
}
```

Figure 6.1- 2: Admin class test case

```java
import static org.junit.Assert.*;

public class MealTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testCreateMeal() {
        fail("Not yet implemented");
    }

    @Test
    public void testUpdateMeal() {
        fail("Not yet implemented");
    }

    @Test
    public void testGetAllMeals() {
        fail("Not yet implemented");
    }

}
```

Figure 6.1- 3: Meal class test case

```java
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
public class MeallogTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {}
    @AfterClass
    public static void tearDownAfterClass() throws Exception {}
    @Before
    public void setUp() throws Exception {}
    @After
    public void tearDown() throws Exception {}
    @Test
    public void testLogmMeal() {
        fail("Not yet implemented");}
    @Test
    public void testUpdateMeallog() {
        fail("Not yet implemented");}
    @Test
    public void testGetDailyCaloriesIntake() {
        fail("Not yet implemented");
    }
    @Test
    public void testCheckOverCalorieLimit() {
        fail("Not yet implemented");
    }

}
```

Figure 6.1- 4: Meal log class test case

```java
import static org.junit.Assert.*;


public class AchievementTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testDefineAchievement() {
        fail("Not yet implemented");
    }

    @Test
    public void testGetAllAchievement() {
        fail("Not yet implemented");
    }

}
```

Figure 6.1- 5: Achievements class test case

```java
import static org.junit.Assert.*;
public class MotivationalContentTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testAddlContent() {
        fail("Not yet implemented");
    }
    @Test
    public void testEditContent() {
        fail("Not yet implemented");
    }
    @Test
    public void testDeleteContent() {
        fail("Not yet implemented");
    }
    @Test
    public void testGetContentByType() {
        fail("Not yet implemented");
    }
}
```

Figure 6.1- 6: Motivational class test case

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
public class UserAchivementTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testUnlockAchivement() {
        fail("Not yet implemented");
    }
    @Test
    public void testGetAchivementsByuser() {
        fail("Not yet implemented");
    }
    @Test
    public void testHasAchivement() {fail("Not yet implemented");}

}
```

Figure 6.1- 7: User achievements class test case

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
public class UserProgressTest {
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
    @Before
    public void setUp() throws Exception {
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testUpdateProgress() {
        fail("Not yet implemented");
    }
    @Test
    public void testViewProgressGraph() {
        fail("Not yet implemented");
    }
    @Test
    public void testGetWeeklyProgress() {
        fail("Not yet implemented");
    }
```

Figure 6.1- 8: User progress class test case

```java
import static org.junit.Assert.*;

public class WorkoutTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

}
```

Figure 6.1- 9: Workout class test case

```java
import junit.framework.TestCase;

public class UserWorkoutTest extends TestCase {

    public void testCreateCustomWorkout() {
        fail("Not yet implemented");
    }

    public void testGetCustomWorkout() {
        fail("Not yet implemented");
    }

}
```

Figure 6.1- 10: User workout test case

```
⊕ import static org.junit.Assert.*;☐
  public class WorkoutScheduleTest {
⊖      @BeforeClass
       public static void setUpBeforeClass() throws Exception {
       }
⊖      @AfterClass
       public static void tearDownAfterClass() throws Exception {
       }
⊖      @Before
       public void setUp() throws Exception {
       }
⊖      @After
       public void tearDown() throws Exception {
       }
⊖      @Test
       public void testScheduleWorkout() {
           fail("Not yet implemented");
       }
⊖      @Test
       public void testUpdateSchedule() {
           fail("Not yet implemented");
       }
⊖      @Test
       public void testDeleteSchedule() {
           fail("Not yet implemented");
       }
⊖      @Test
       public void testCheckScheduleConflict() {fail("Not yet implemented");}
⊖      @Test
       public void testGetUserSchedule() {
           fail("Not yet implemented");
       }
```

Figure 6.1- 11: Workout schedule class test case

6.2 Steps and Tools Used in Unit Test

Step 1: Set up your development environment

Ensure that you have a development environment with JUnit properly configured. You can use build tools like Gradle or Maven to manage dependencies.

Step 2: Identify the class to be tested

Determine the Java class or method that you want to test.

Step 3: Create a new test class

Create a new Java class with a name that reflects the class or method being tested. For example, if you are testing a class called Calculator, the test class could be named CalculatorTest.

Step 4: Import necessary dependencies

Import the necessary JUnit classes and any additional classes required for testing.

Step 5: Write test methods

Inside the test class, write test methods. Each test method should be annotated with @Test to indicate that it is a test method.

Step 6: Set up the test environment

If your test requires a specific setup, such as creating objects or initializing variables, use the @Before annotation on a method that will be executed before each test method.

Chapter Seven

7.1 Build (prepare build script for compilation, unit test, jar file creation)

In software development, the term "build" refers to the process of converting source code files into standalone software artifacts that can be run on a computer. This process typically includes compiling the source code, running automated tests, packaging the compiled code into executable files (such as JAR files in Java), and preparing the software for deployment.

Build Process

1. **Compilation**:

   - The source code is transformed into executable code by a compiler. For Java projects, this means converting **.java** files into **.class** files.

2. **Unit Testing**:

   - Automated tests are run to ensure that the individual units of code (e.g., methods, classes) function correctly. This step helps in identifying and fixing bugs early in the development cycle.

3. **Packaging**:

   - The compiled code and other resources are packaged into a distributable format, such as a JAR (Java ARchive) file for Java applications. This package can then be deployed and executed on a target environment.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="FitnessTrackingApp" default="run-tests" basedir=".">

    <property name="src.dir"        value="src"/>
    <property name="test.dir"       value="test"/>
    <property name="build.dir"      value="build"/>
    <property name="classes.dir"    value="${build.dir}/classes"/>
    <property name="test.classes.dir" value="${build.dir}/test-classes"/>
    <property name="lib.dir"        value="lib"/>
    <property name="report.dir"     value="report"/>

    <path id="classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${build.dir}"/>
        <delete dir="${report.dir}"/>
    </target>

    <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}" includeantruntime="false">
            <classpath refid="classpath"/>
        </javac>
    </target>

    <target name="compile-tests" depends="compile">
        <mkdir dir="${test.classes.dir}"/>
        <javac srcdir="${test.dir}" destdir="${test.classes.dir}" includeantruntime="false">
            <classpath>
                <path refid="classpath"/>
                <pathelement path="${classes.dir}"/>
            </classpath>
        </javac>
    </target>

    <target name="run-tests" depends="compile-tests">
        <mkdir dir="${report.dir}"/>

        <junit printsummary="on" haltonfailure="false" fork="true">
            <classpath>
                <path refid="classpath"/>
                <pathelement path="${classes.dir}"/>
                <pathelement path="${test.classes.dir}"/>
            </classpath>
            <formatter type="plain"/>
            <formatter type="xml"/>

            <batchtest todir="${report.dir}">
                <fileset dir="${test.classes.dir}">
                    <include name="**/*Test.class"/>
                </fileset>
            </batchtest>
        </junit>
    </target>
```

Figure 7.1- 1: Build xml

7.2 Steps and Tools Used in Our Project to Implement the Build

We followed a structured build process in our project, using **Eclipse** as the main development tool. Below are the key steps:

Step 1: Source Code Management
We used Eclipse to manage and organize our source code. It provided an integrated workspace where we could write, navigate, and maintain the codebase efficiently.
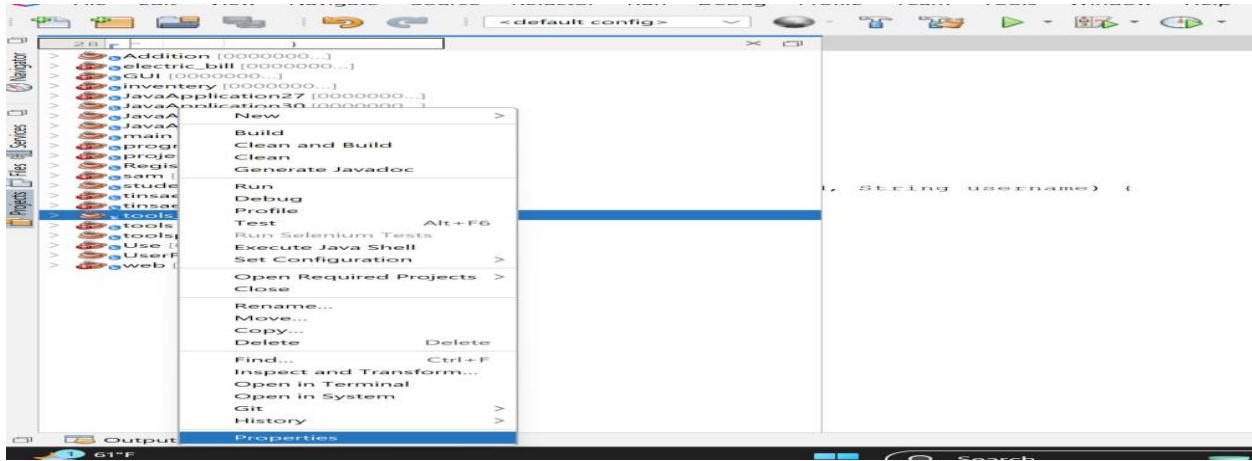
Figure 7.2- 1: step 1 of implementing build

Step 2: Build and Compile

Eclipse handled the compilation of our source code and managed dependencies automatically. Its built-in features simplified the build process, ensuring the application compiled correctly with all necessary libraries.
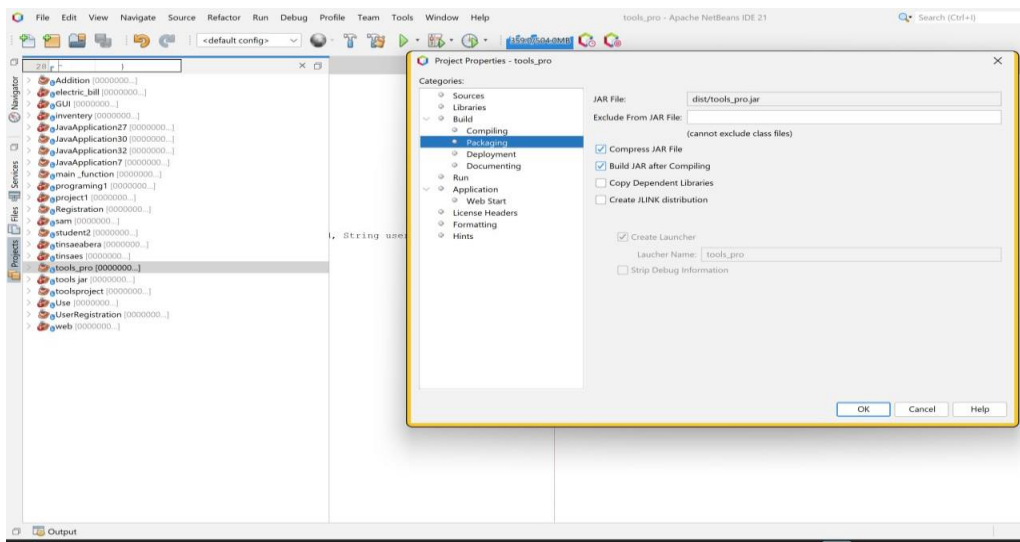

Figure 7.2- 2: Packaging

Step 3: Testing

For testing, we utilized Eclipse's integration with JUnit to run unit tests. This allowed us to validate individual components of the application and catch errors early in the development cycle.
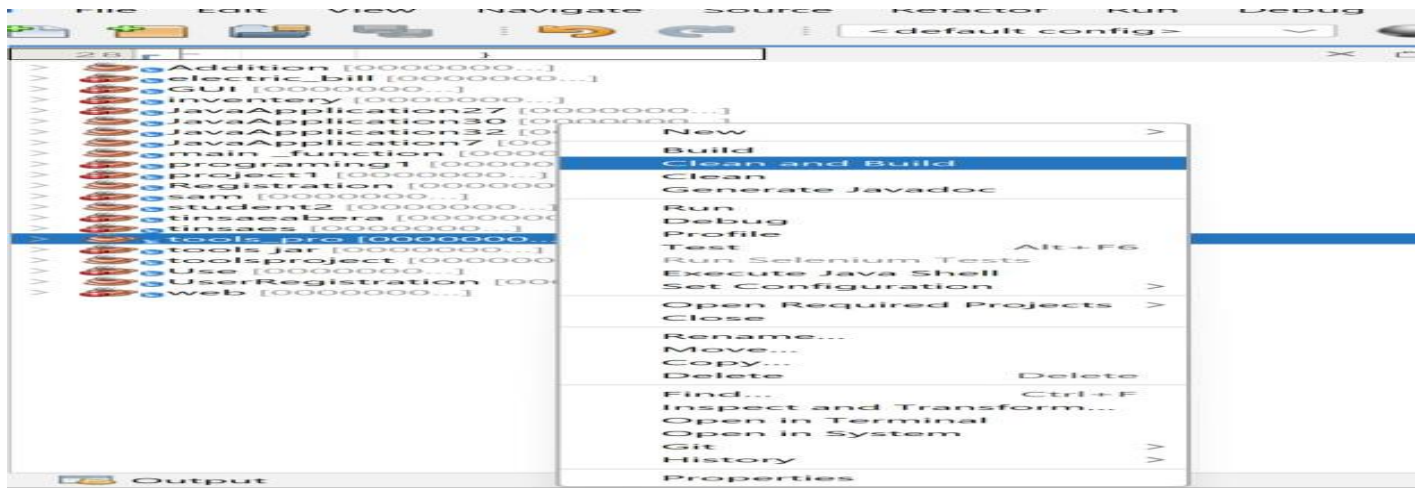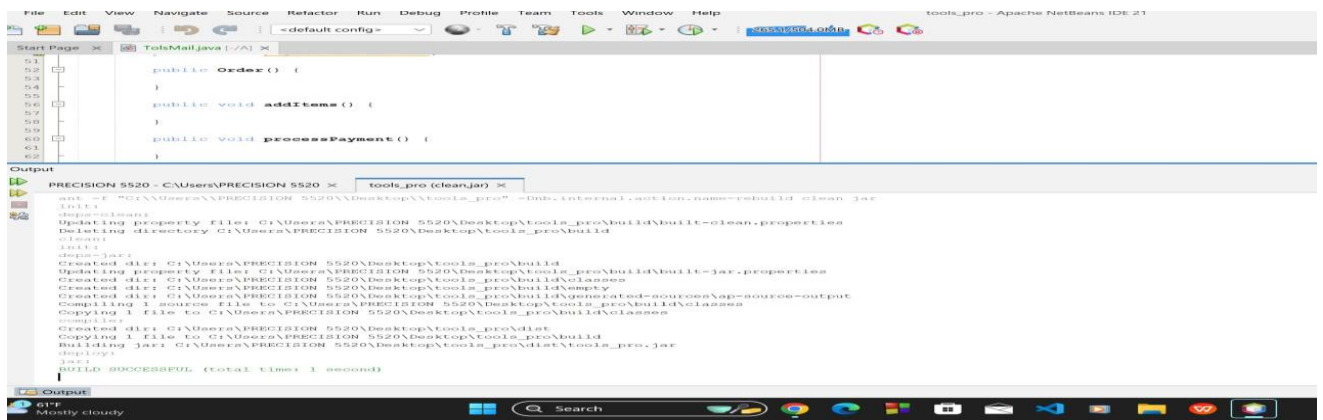
Figure 7.2- 3: Test

Figure 7.2- 4: Build success