**Cairo University  
Faculty of Computers and Artificial Intelligence** 

**CS251**

**Introduction to Software Engineering**

Infinite loop

Software Design Specifications

Version 3.0

| **ID** | **Name** | **Email** |
| --- | --- | --- |
| 20230610 | Abdelrahman Ashraf Mohamed | abdelrahmannashraf21@gmail.com |
| 20230267 | Omar Mostafa Saad Mohamed | omar16220061622006@gmail.com |
| 20230515 | Yousef Hesham Ali | yousefelzayann@gmail.com |

April 2025

**Contents**

[Team 3](#_heading=h.juvxw4k2wg1m)

[Document Purpose and Audience 3](#_heading=h.gi31whzd8dwz)

[System Models 4](#_heading=h.le58o6rjvwo3)

[I. Architecture Diagram 4](#_heading=h.xd4193gqvg1x)

[II. Class Diagram(s) 6](#_heading=h.6o0ypcn4wgb1)

[III. Class Descriptions 6](#_heading=h.8nf0pj3evw0h)

[IV. Sequence diagrams 8](#_heading=h.f3z90joltffx)

[Class - Sequence Usage Table 11](#_heading=h.icbhiu5kog02)

[V. State Diagram 11](#_heading=h.cna1u94p7q1w)

[VI. SOLID Principles 12](#_heading=h.e43nx0q38zm0)

[VII. Design Patterns 15](#_heading=h.6yr4r851oviv)

[Tools 16](#_heading=h.4trys871i5eu)

[Ownership Report 17](#_heading=h.wqoc1o3o3p0o)

# Team

| **ID** | **Name** | **Email** | **Mobile** |
| --- | --- | --- | --- |
| 20230610 | Abdelrahman Ashraf Mohamed | [abdelrahmannashraf21@gmail.com](mailto:abdelrahmannashraf21@gmail.com) |  |
| 20230267 | Omar Mostafa Saad Mohamed | [omar16220061622006@gmail.com](mailto:omar16220061622006@gmail.com) |  |
| 20230515 | Yousef Hesham Ali | [yousefelzayann@gmail.com](mailto:yousefelzayann@gmail.com) |  |

# Document Purpose and Audience

This document is the Software Design Specification (SDS) for the project.

It provides detailed design specifications, including:

* System architecture
* Class diagrams and descriptions
* Sequence diagrams
* State diagrams
* Application of SOLID principles
* Use of design patterns
* Tools used for development

The target audience includes:

1. Developers : To implement the system based on the provided design.
2. Software Designers : To review and refine the system architecture and design.
3. Project Managers : To ensure alignment with project requirements and timelines.
4. Quality Assurance (QA) Engineers : To create test cases based on the design and verify functionality.
5. Stakeholders/Customers : To understand how the system will meet their requirements.
6. Team Members : To collaborate effectively during implementation and ensure consistency across the project.

# System Models

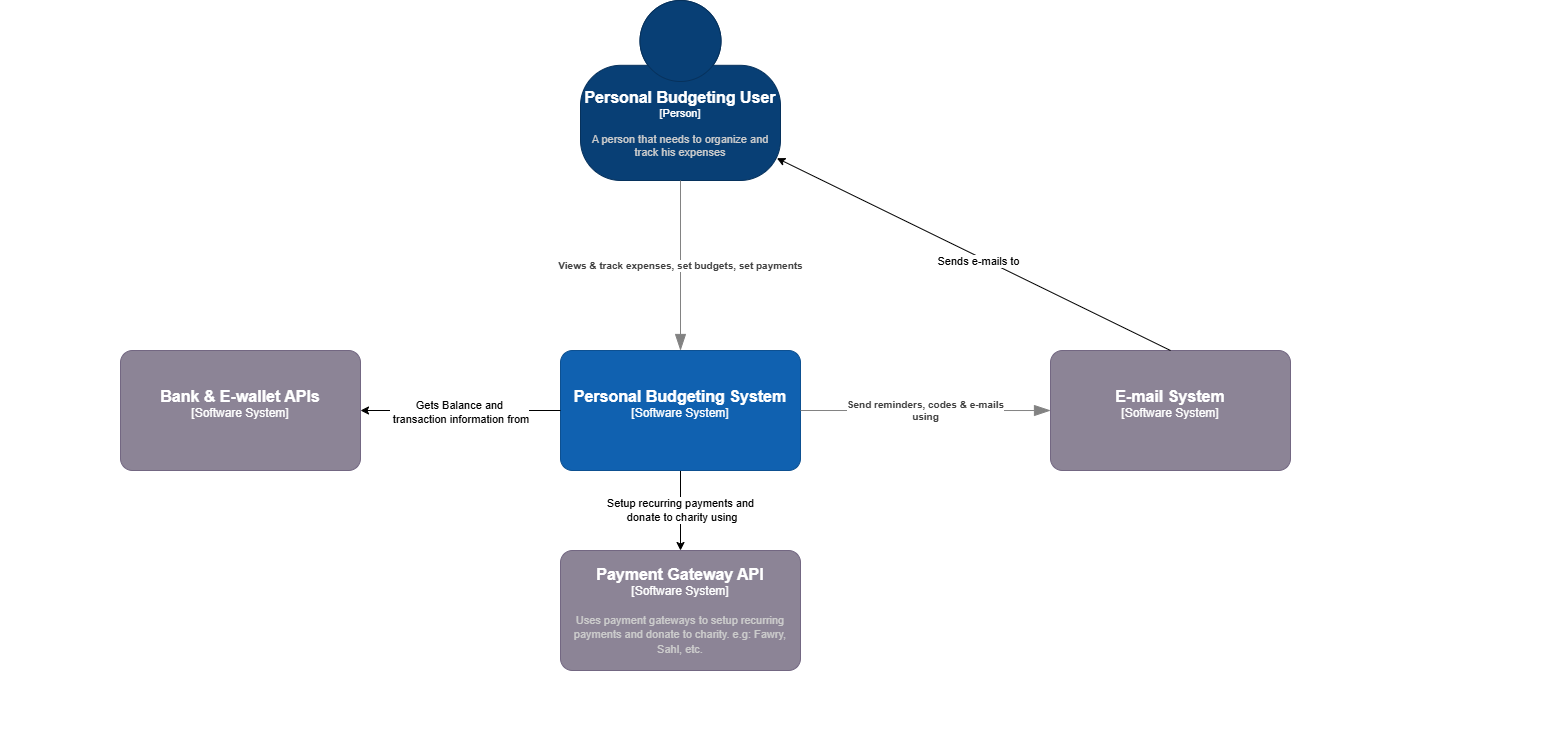
## I. Architecture Diagram

**The Personal Budgeting system uses a Client-Server architecture. In this model, the client application (Mobile and Web app) interacts with the backend server through APIs. The server handles authentication, budgeting logic, transaction management, setting payments, and communication with external services like banks and payment gateways.**

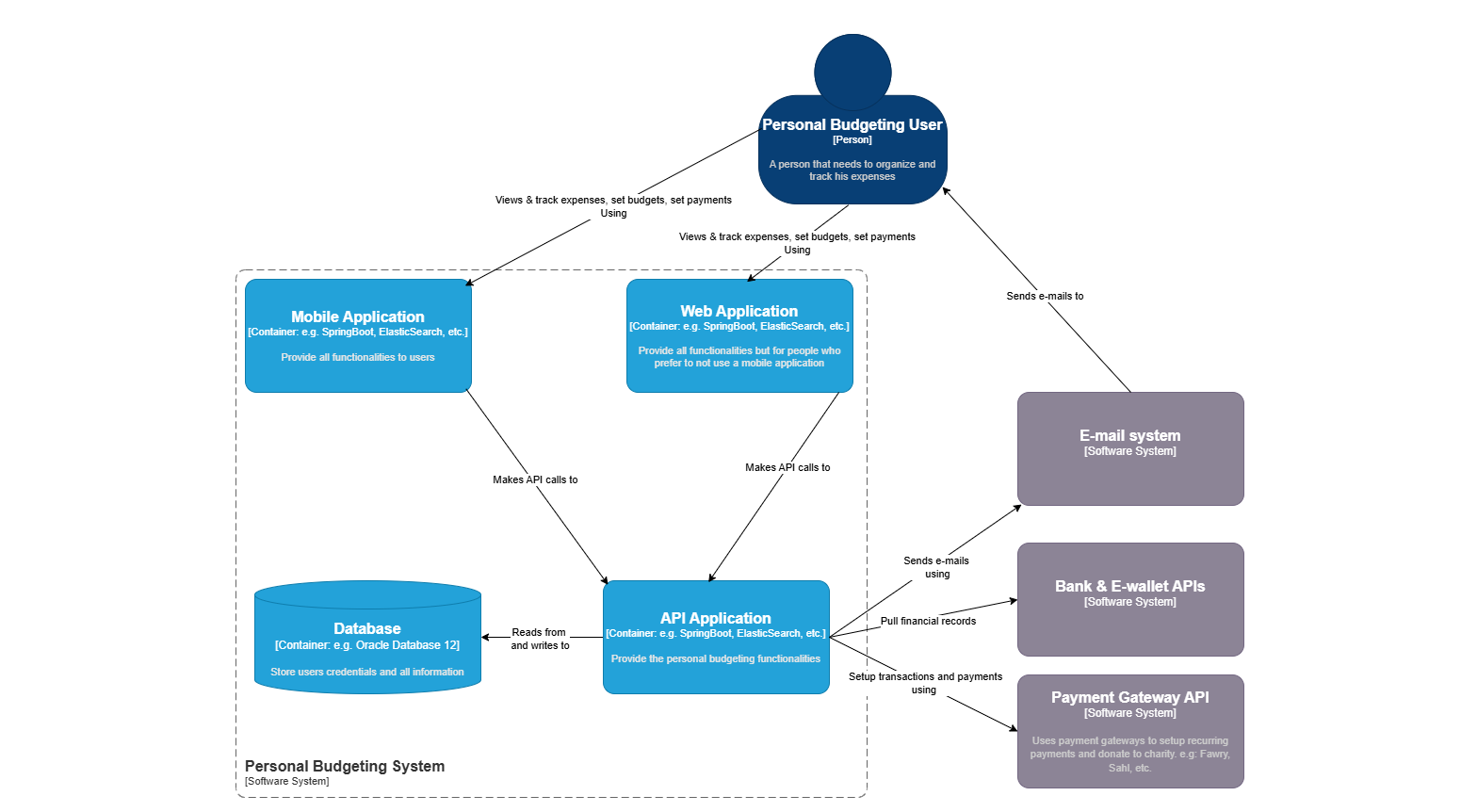
**This architecture was chosen because it provides:**

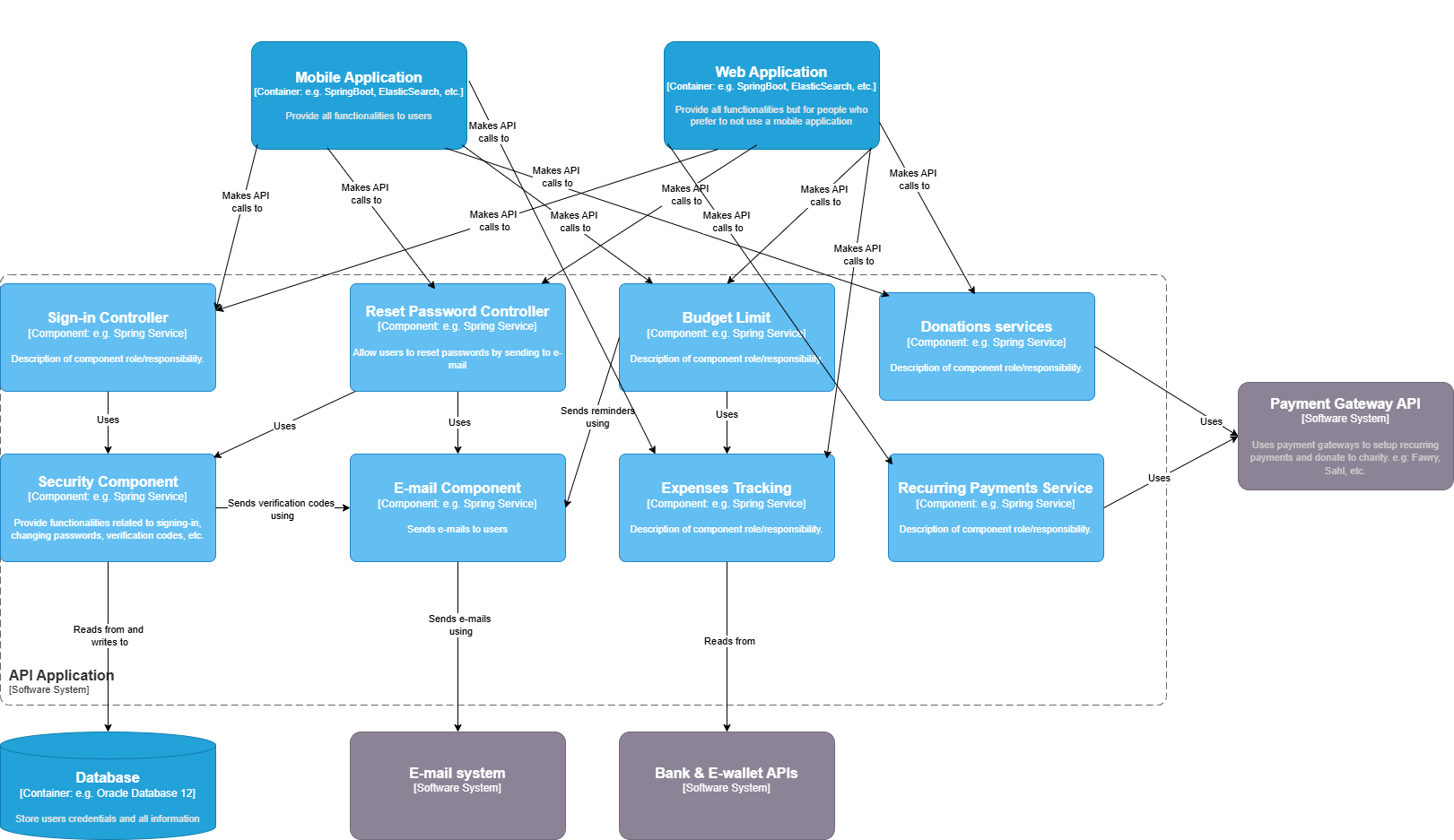
* **Clear separation between the user interface (frontend) and business logic (backend),**
* **Better security, since sensitive operations (e.g., budget calculations, bank API calls) happen on the server,**
* **Easier scalability as user load increases,**
* **Simplified maintenance because backend services can evolve independently from the client app.**

**Level 1:**

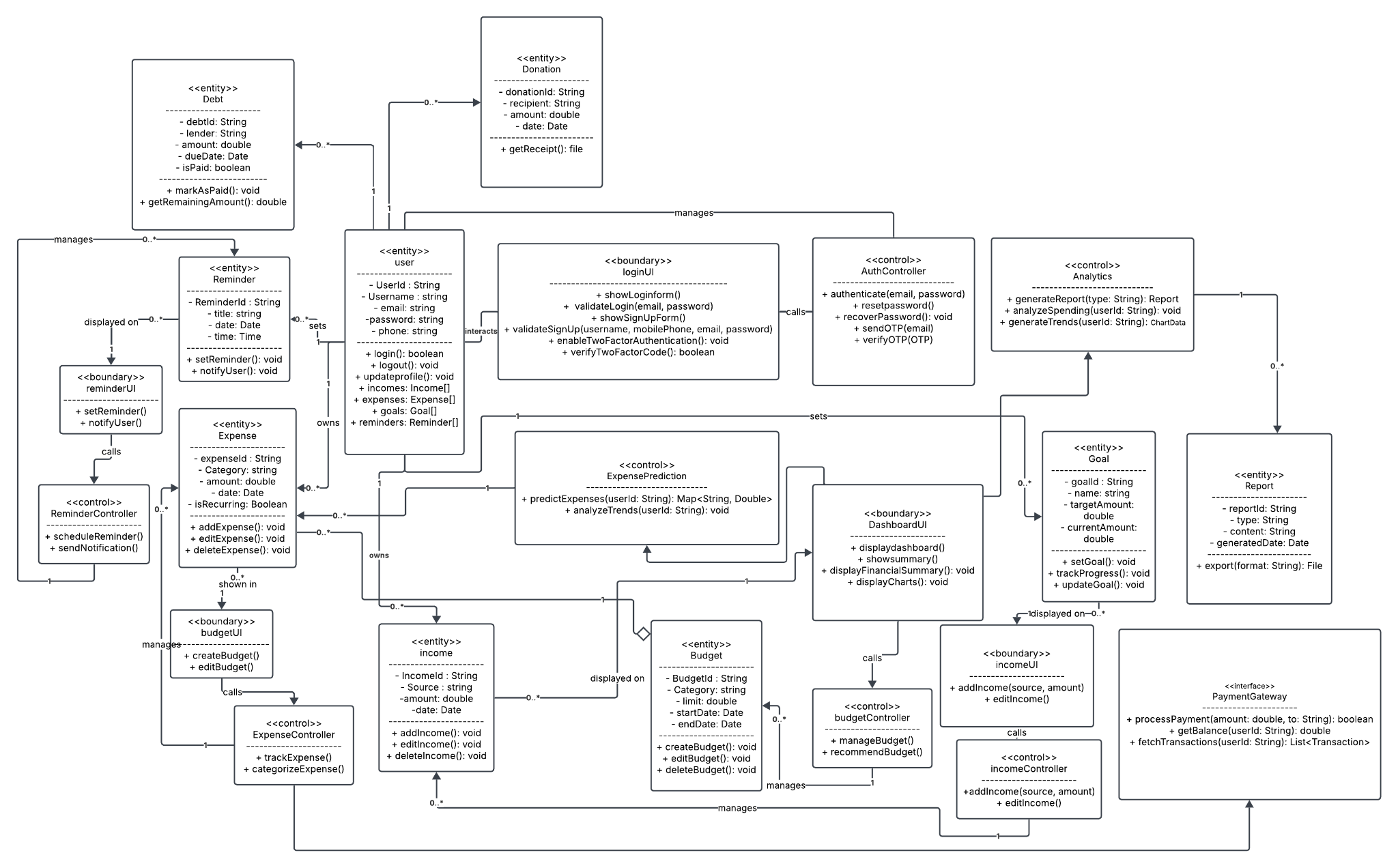
****

**Level 2:**

****

**Level 3:**

## II. Class Diagram(s)

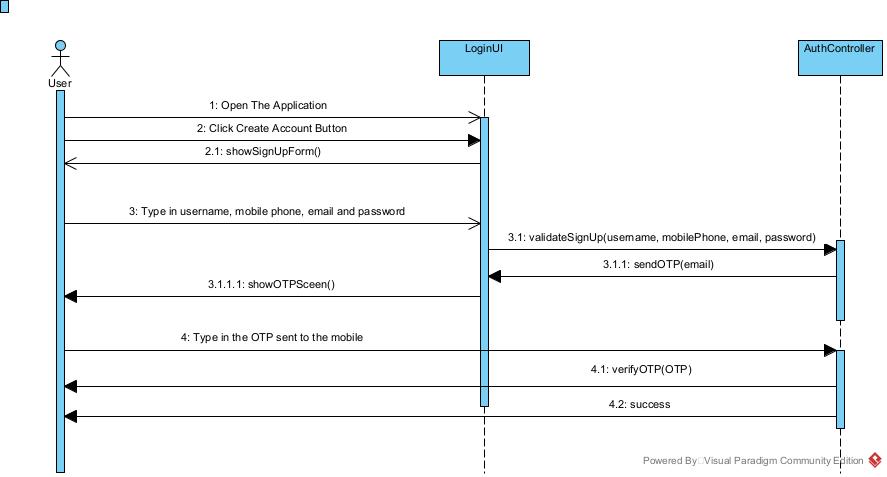


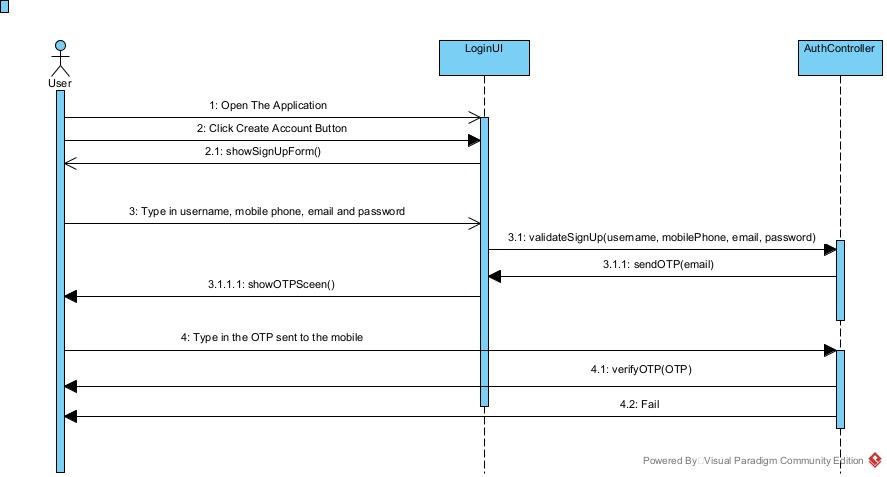
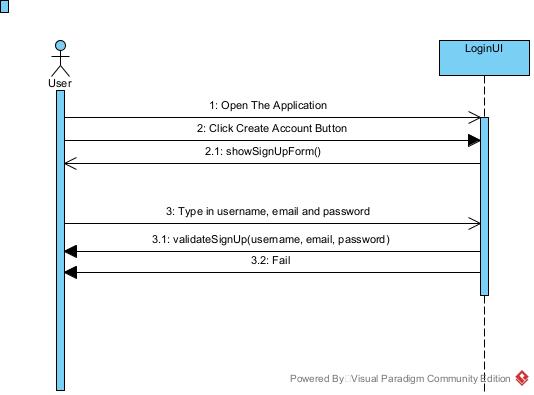
## III. Class Descriptions

| **Class ID** | **Class Name** | **Description & Responsibility** |
| --- | --- | --- |
| 1 | User | Represents a system user; responsible for authentication, storing personal details, incomes, expenses, goals, reminders, debts, and donations. |
| 2 | Income | Models user’s income records; responsible for storing income details and supporting add/edit/delete operations |
| 3 | Expense | Models user’s expenses; tracks categories, amounts, dates, recurring status, and supports CRUD functions |
| 4 | Budget | Represents spending plans; responsible for limits, periods, and tracking associated expenses. |
| 5 | Goal | Models financial objectives (e.g., savings target); tracks target amount and progress |
| 6 | Reminder | Represents reminders for user activities/events; notifies users of important dates or financial deadlines. |
| 7 | Debt | Represents borrowed money or liabilities; tracks lender, amount, repayment due dates, and payment status. |
| 8 | Donation | Models charitable donations by the user; keeps record of recipients, amounts, and dates. |
| 9 | Report | Stores analytical results and financial reports; can be exported and is generated by Analytics. |
| 10 | LoginUI | User interface for login/authentication screens, including two-factor authentication. |
| 11 | DashboardUI | Main interface showing user’s financial summaries, including reports and analytics. |
| 12 | BudgetUI | Interface for budget creation and editing; displays budget information and related expenses. |
| 13 | ReminderUI | Interface for managing reminders; allows users to set, view, and receive notifications. |
| 14 | IncomeUI | Interface for adding/editing/viewing income records. |
| 15 | AuthController | Handles authentication, login, and password recovery logic. |
| 16 | ExpenseController | Controls management and categorization of user expenses. |
| 17 | BudgetController | Oversees budgets, including management and generating recommendations. |
| 18 | ReminderController | Handles business logic for reminders, including notification scheduling. |
| 19 | IncomeController | Manages operations related to user income records. |
| 20 | Analytics | Analyzes user’s spending, trends, and generates financial reports & insights for display or export. |
| 21 | ExpensePrediction | Uses AI to predict user’s future expenses and trends based on historical data. |
| 22 | PaymentGateway | Interface defining external payment integration methods (process payments, check balance, fetch transactions) |

## IV. Sequence diagrams

**User Story #1**

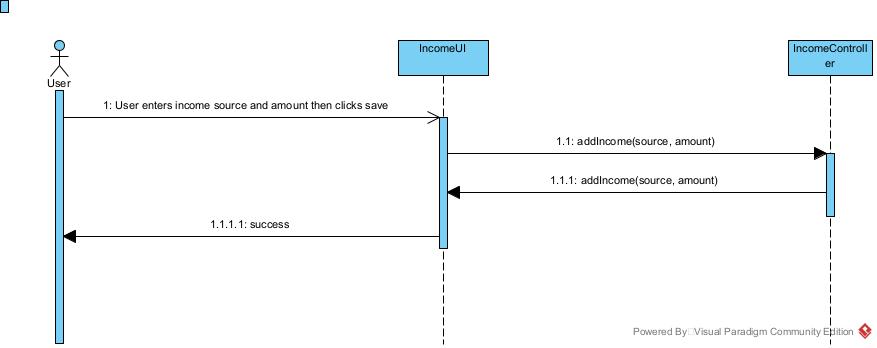
****



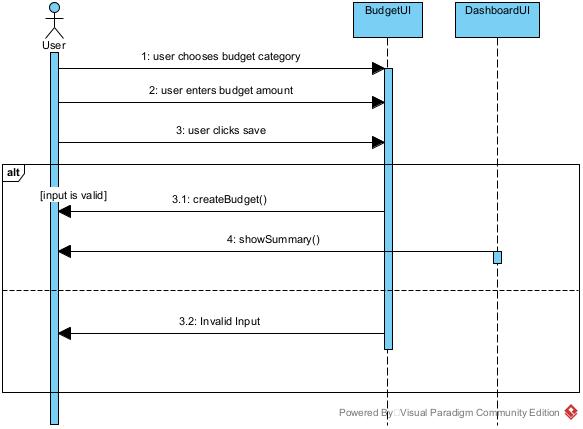
**User Story #2**

### 

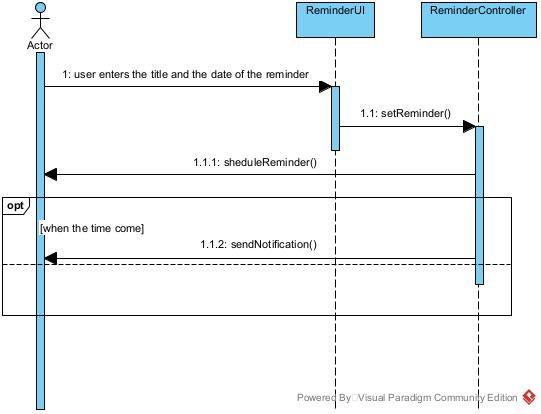
### 

**User Story #3**

**User Story #4**

****

**User Story #5**

****

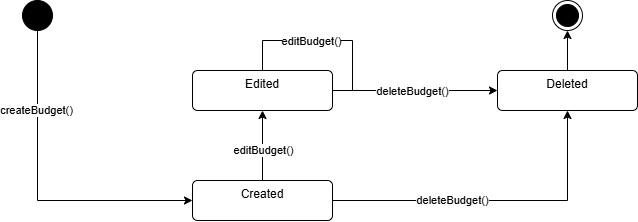
**User Story #6**

### 

### Class - Sequence Usage Table

| **Sequence Diagram** | **Classes Used** | **All Methods Used** |
| --- | --- | --- |
| 1. Sign In | AuthUI  AuthController | showLoginForm()  validateLogin(email, password)  —-----------------------------------------------  authenticate(email, password) |
| 1. Sign Up | AuthUI  AuthController | showSignupForm()  validateSignup(username, email,  mobilePhone, password)  —-------------------------------------------------  sendOTP(email)  verifyOTP(OTP) |
| 1. Add Income | IncomeUI  IncomeController | addIncome(source, amount)  —----------------------------------------  addIncome(source, amount) - at income controller. |
| 1. Add Budget | budgetUI  DashboardUI | createBudget()  —---------------------------------  showSummary() |
| 1. Add Reminder | ReminderUI  ReminderController | setReminder()  —---------------------------------  sheduleReminder()  sendNotification() |
| 1. Add Expense | DashboardUI  Expense | displayFinancialSummary()  —-------------------------------------------  addExpense() |

## V. State Diagram



Since the "Budget" class is essential to the financial planning system, it was selected as the object for the state diagram. The diagram models the budget's lifecycle, starting from creation, allowing optional edits, and potentially ending with deletion. This helps developers understand the allowed transitions based on user actions.

* After calling "createBudget()," a budget moves into the "Created" state.
* It can optionally be updated using “editBudget()” to enter the “Edited” state.
* The user has the option to call "deleteBudget()" at any time after creation or editing, which puts the object in the "Deleted" state.
* The “Deleted” state leads to the “End” state, indicating the budget is no longer in use.

## 

## VI. SOLID Principles

1. Single Responsibility Principle (SRP)

Explanation:

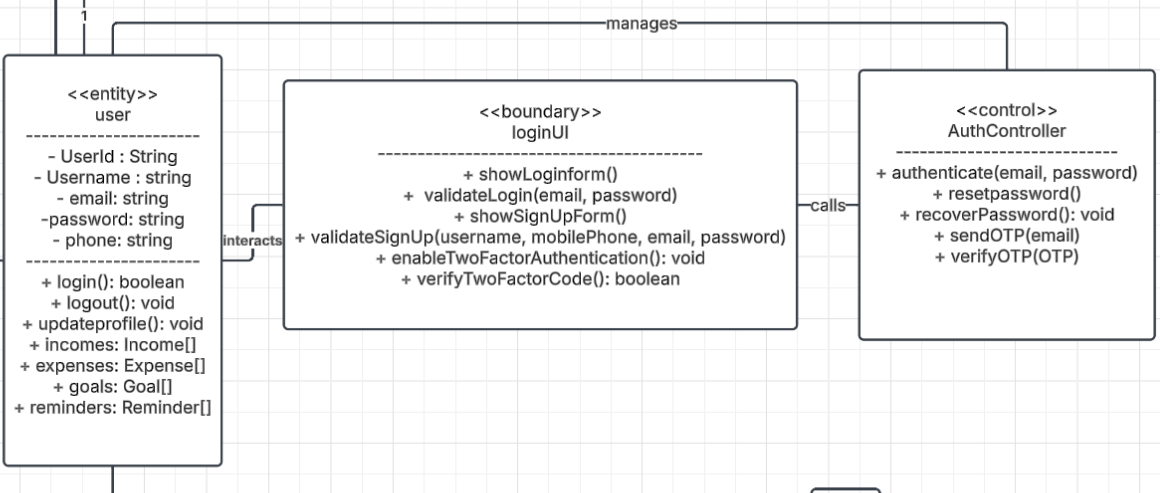
The Single Responsibility Principle states that a class should have only one reason to change. Each class in the design is responsible for a single, well-defined task.

Application in the Design:

- Applied in: AuthController Class

- The AuthController class is solely responsible for handling authentication-related operations, such as validating login credentials (validateLogin(email, password)), resetting passwords (resetPassword()), enabling two-factor authentication (enableTwoFactorAuthentication()), and verifying OTPs (verifyOTP(OTP)).

- Why SRP is Applied: Authentication logic is isolated from other functionalities like income or expense management, ensuring that changes to authentication do not affect other parts of the system.



2. Open/Closed Principle (OCP)

Explanation:

The Open/Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification. This means you can add new functionality without changing existing code.

-Applied in: PaymentGateway Interface

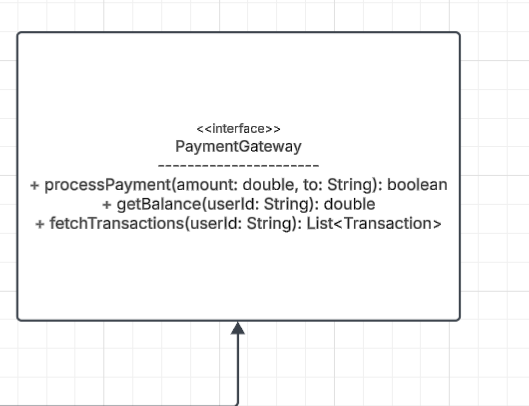
- The PaymentGateway class is designed as an interface that defines methods like processPayment(), getBalance(), and fetchTransactions().

- Specific implementations (e.g., VodafoneCashGateway, OrangeCashGateway) extend this interface to handle payment processing for different digital wallets.

- Why OCP is Applied: If a new payment method (e.g., PayPal) needs to be added, you can create a new class (e.g., PayPalGateway) that implements the PaymentGateway interface without modifying the existing payment processing logic.

- Benefit:

- Adding new payment integrations does not require changes to the core system, making the design more flexible and maintainable.



3. Dependency Inversion Principle (DIP)

Explanation:

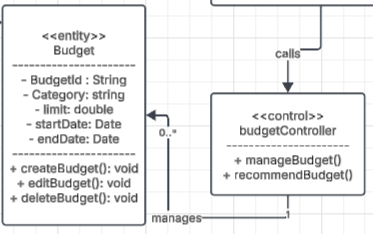
The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

Applied in: BudgetController and Budget

- The BudgetController class depends on the Budget abstraction rather than directly interacting with specific implementations of budgets.

- Why DIP is Applied: The BudgetController uses an interface or abstract class (Budget) to manage budgets, allowing flexibility to introduce new types of budgets (e.g., monthly budgets, yearly budgets) without modifying the controller.

Diagram Reference:



## VII. Design Patterns

1. **The Bridge Design Pattern:**

**Where we used it ?**

We used it in the PaymentGateway interface and the different implementation of it.

**What was the benefit of using it ?**

It allowed us to decouple the PaymentGateway abstract functionality that is needed in our application and the different implementations of it.

1. **The Observer Design Pattern:**

**Where we used it ?**

We used the Observer design pattern a lot throughout all the application in more than one place

1. **Reminder, ReminderUI, ReminderController and the User classes:**

**What was the benefit of using it ?**

it allowed the user class - as a subscriber - to get notified about the new reminders added by the user in the ReminderUI - as a publisher.

1. **Expense, budgetUI, ExpenseController and the User classes:**

**What was the benefit of using it ?**

it allowed the user class - as a subscriber - to get notified about the new expenses added by the user in the budgetUI - as a publisher.

1. **Goal, IncomeUI, IncomeController and the User classes:**

**What was the benefit of using it ?**

it allowed the user class - as a subscriber - to get notified about the new goals added by the user in the IncomeUI - as a publisher.

1. **Income, DashboardUI and the User classes:**

**What was the benefit of using it ?**

it allowed the user class - as a subscriber - to get notified about the new incomes added by the user in the IncomeUI - as a publisher.

1. **The Singleton Design Pattern:**

**Where was it used ?**

It will be used for creating objects from the implementations of PaymentGateway interface.

**What is the benefit from using it ?**

While it might not be apparent in the class diagram, we will use these global objects to communicate with 3rd parties throughout all the application, for you will not duplicate integration points.

# Tools

* **Lucidchart**
* **Draw.io**
* **Visual Paradigm**

# Ownership Report

| **Item** | **Owners** |
| --- | --- |
| **Yousef Hesham Ali** | *Class diagram, Class Descriptions, SOLID Principles* |
| **Abdelrahman Ashraf Mohamed** | *Architecture Diagram, State Diagram* |
| **Omar Mostafa Saad Mohamed** | *Sequence Diagrams, Sequence Descriptions and Design Patterns* |