UE22CS352B – Object Oriented Analysis and Design

Mini Project Report

Personal Finance Tracker

Submitted by:

Shreya M Hegde : PES1UG22CS573

Shreya Mittal : PES1UG22CS575

6 J

Dr. Bhargavi Mokashi

January – May 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under
Karnataka Act No. 16 of
2013) 100ft Ring Road,
Bengaluru – 560 085,
Karnataka, India

Problem Statement:

The Personal Finance Tracker aims to provide a comprehensive solution that centralizes financial management, offering intuitive tools to track income and expenses, plan budgets, set financial goals, and generate meaningful insights to improve users' financial well-being.

Key Features:

Major Features:

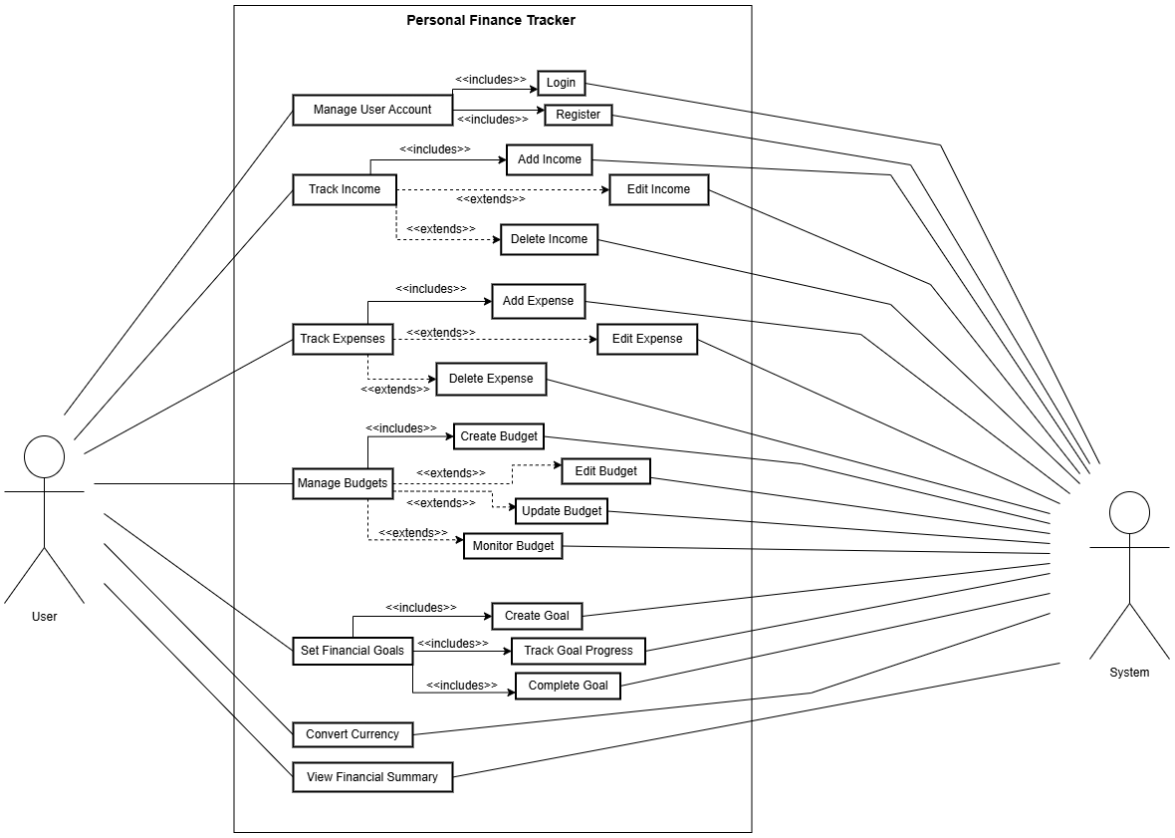
1. **Comprehensive Expense Management**
    - Detailed categorization system for expenses (Housing, Transportation, Food, etc.)
    - Support for recurring expense entries
    - Category-based expense analysis with visual representations
2. **Income Tracking & Analysis**
    - Multiple income source recording (Salary, Freelance, Investments, Business, Other)
    - Income categorization (Fixed vs. Variable)
    - Recurring income entry support for regular payments
3. **Budget Planning & Monitoring**
    - Category-specific budget creation with customizable time frames
    - Spending limit definition across various expense categories
    - Real-time monitoring of actual spending against budget limits
    - Visual indicators showing budget status and potential overspending
4. **Financial Goal Setting**
    - Creation of personalized financial goals with target amounts
    - Progress tracking toward defined goals
    - Status monitoring (In Progress, Completed, Cancelled)
    - Timeline visualization for goal achievement

Minor Features:

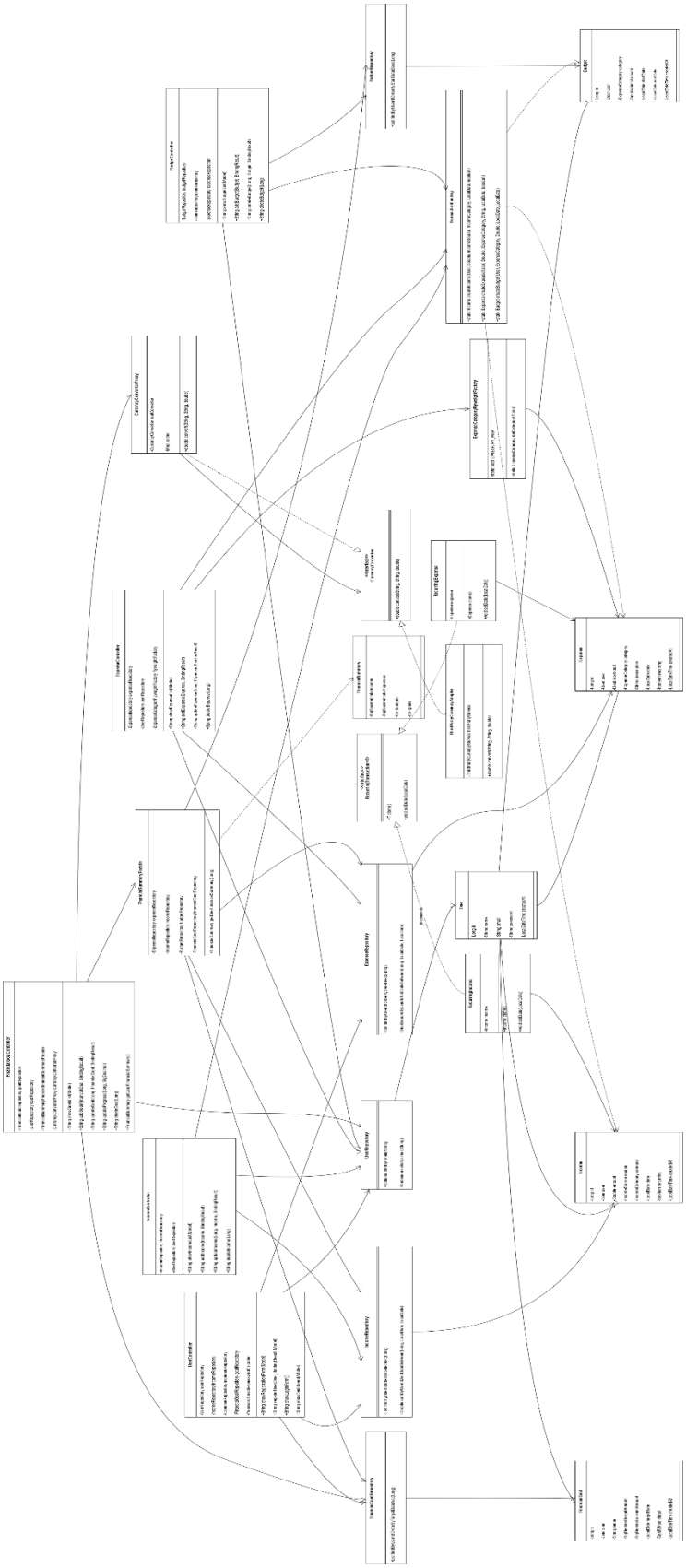
1. **Financial Dashboard and Reporting**
    - Centralized dashboard with comprehensive financial overview
    - Monthly and yearly financial summaries showing trends
2. **Currency Conversion**
    - Conversion between different currencies for global finance management
    - Access to real-time exchange rates
    - Cached conversion results for improved performance
3. **Secure User Management**
    - Registration and authentication system with strong security
    - User profiles with personalized financial overviews
    - Password encryption and secure data handling
4. **Recurring Transaction Management**
    - Configuration of recurring income and expenses
    - Automatic transaction generation for regular financial activities
    - Ability to modify or cancel recurring transactions as needed

Models

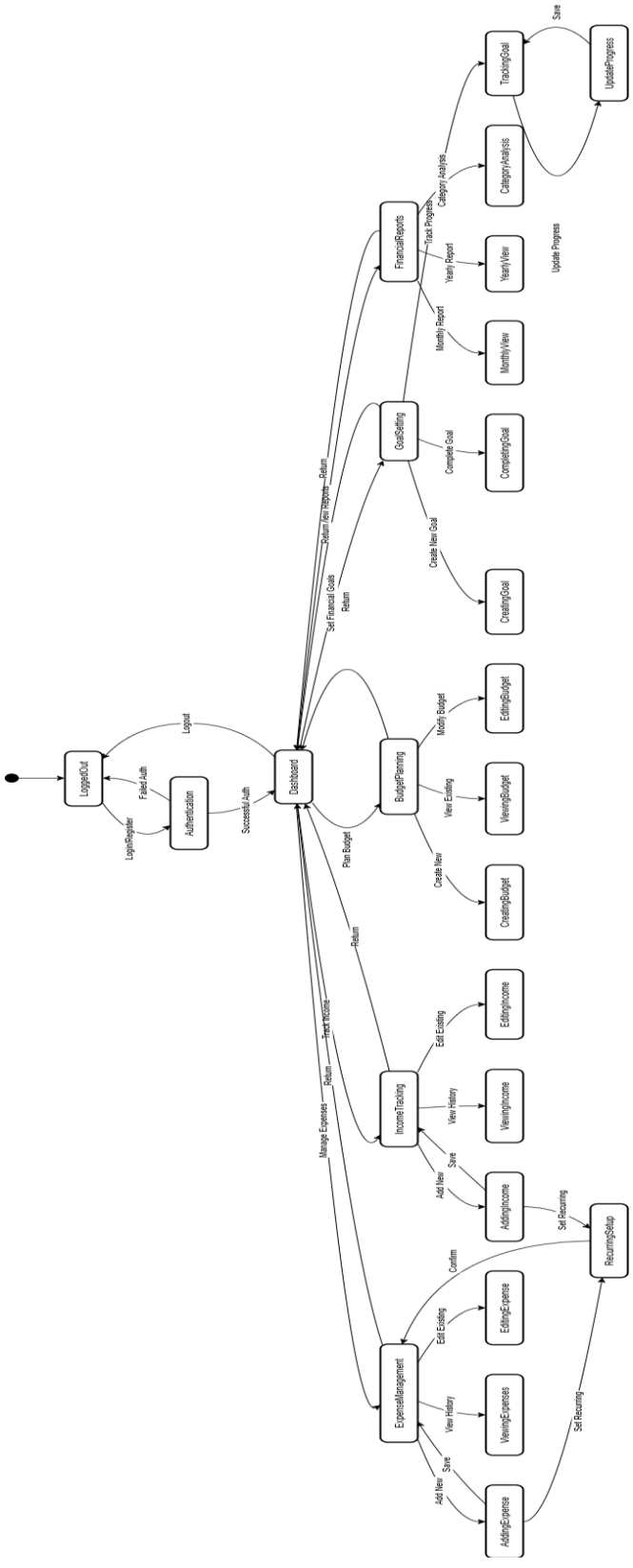
Use Case Diagram:



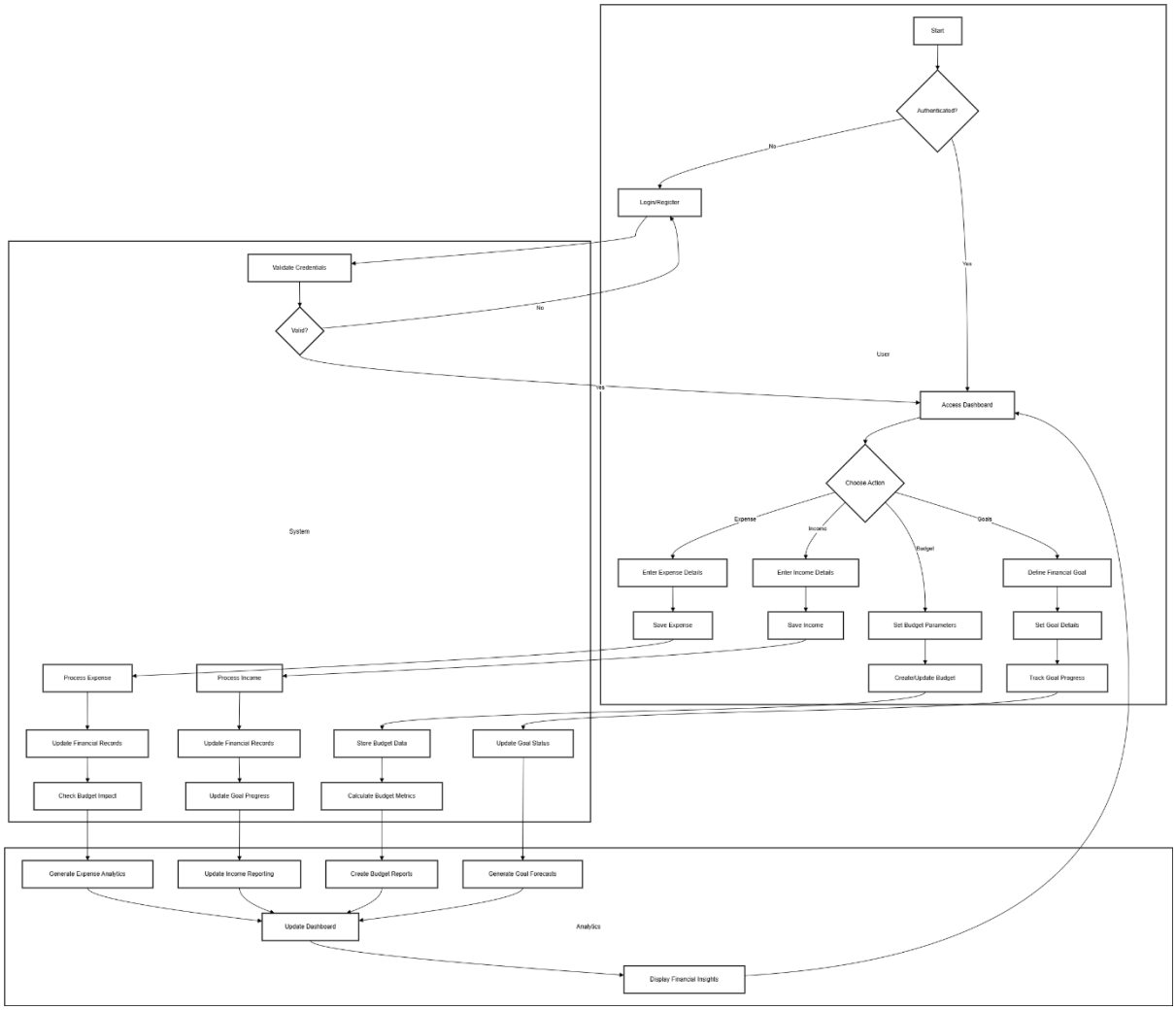
Class Diagram:



State Diagram:



Activity Diagram:



Architecture Patterns, Design principles, and Design Patterns:

Architecture Patterns

Model – View – Controller Pattern (MVC)

Design Principles and patterns

Creational Design Pattern Used

1. Factory Pattern

Purpose: Encapsulates object creation logic, allowing for flexible and consistent instantiation of complex objects.

Where: TransactionFactory.java

How to View:

- Used throughout the app to create Expense, Income, and Budget objects.
- **Files:** TransactionFactory.java, ExpenseController.java, IncomeController .java, BudgetController.java

Code Example:

```
// In ExpenseController, IncomeController, etc.
Expense newExpense = TransactionFactory.createExpense(
   user, amount, category, description, date, recurring
);
Income newIncome = TransactionFactory.createIncome(
   user, amount, source, category, date, recurring
);
Budget newBudget = TransactionFactory.createBudget(
   user, category, limitAmount, startDate, endDate
);
```

2. Prototype Pattern

Purpose: Create new objects by copying (cloning) existing ones, useful for recurring or templated data.

Where: prototype/RecurringTransaction.java, prototype/RecurringExpense.java , prototype/RecurringIncome.java

How to View:

- Used for recurring transactions (expenses/incomes) to efficiently create new instances based on a template.
- **Files:** RecurringTransaction.java, RecurringExpense.java, RecurringInco me.java

Code Example:

```
// Cloning a recurring expense
RecurringExpense recurring = new RecurringExpense(existingExpense);
Expense nextMonth = recurring.clone(); // New Expense with same fields
```

3. Singleton Pattern
Purpose: Ensure a class has only one instance and provide a global point of access to it.
Where: Spring-managed beans by default (e.g., services annotated with @Component, @Service, @Repository)
How to View:

- All Spring beans are singletons unless explicitly marked otherwise.
- **Files:** Any class with @Component, @Service, or @Repository annotation (e.g., FinancialSummaryFacade.java, CurrencyConverterProxy.java)

Code Example:

```
@Component
public class FinancialSummaryFacade {
   // This bean is a singleton by default in Spring
}
```

Structural Design Patterns Used

1. Facade Pattern
Purpose: Simplifies complex subsystem interactions by providing a unified interface.
Where: FinancialSummaryFacade.java
How to View:

- Used in FinancialGoalController to aggregate data from incomes, expenses, budgets, and goals.
- **Endpoint:** /goals/summary (returns a JSON summary for the logged-in user)

Code Example:

```
// In FinancialGoalController
@Autowired
private FinancialSummaryFacade financialSummaryFacade;

@GetMapping("/goals/summary")
@ResponseBody
public FinancialSummary getSummary(Authentication auth) {
   User user = userRepository.findByEmail(auth.getName()).orElseThrow();
   return financialSummaryFacade.getUserFinancialSummary(user.getId());
}
```

2. Adapter Pattern
Purpose: Allows incompatible interfaces to work together.
Where:
- ThirdPartyCurrencyService.java (simulated external API)
- CurrencyConverter.java (target interface)
- ThirdPartyCurrencyAdapter.java (adapter)

How to View:
- Used for currency conversion in the dashboard UI.
- **UI:** Currency Converter card on the dashboard (/dashboard)
- **Endpoint:** /goals/convert-currency?amount=1000&from=INR&to=USD

Code Example:
```
// Interface
public interface CurrencyConverter {
   double convert(String fromCurrency, String toCurrency, double amount);
}

// Adapter
@Component
public class ThirdPartyCurrencyAdapter implements CurrencyConverter {
   private final ThirdPartyCurrencyService thirdPartyService = new
ThirdPartyCurrencyService();
   public double convert(String from, String to, double amount) {
      return thirdPartyService.convert(from, to, amount);
   }
}
```
3. Flyweight Pattern
Purpose: Reduces memory usage by sharing common objects (e.g., enums) instead of creating many duplicates.
Where: ExpenseCategoryFlyweightFactory.java
How to View:
- Used when creating or updating expenses to ensure all categories are shared instances.
- **Files:** ExpenseController.java, TransactionFactory.java

Code Example:
```
// In ExpenseController
Expense.ExpenseCategory flyweightCategory =
ExpenseCategoryFlyweightFactory.getCategory(expense.getCategory().name())
;
Expense newExpense = TransactionFactory.createExpense(
   user, expense.getAmount(), flyweightCategory, ...
);
```
4. Proxy Pattern
Purpose: Adds extra functionality (like caching, logging, security) to an object

without changing its interface.
Where: CurrencyConverterProxy.java
How to View:

- All currency conversions now go through this proxy for caching and logging.
- **UI:** Currency Converter card on the dashboard
- **Endpoint:** /goals/convert-currency
- **Logging:** Check your server logs for messages like:
  - [PROXY] Cache hit for conversion: INR-USD-1000.0
  - [PROXY] Cache miss for conversion: INR-USD-1000.0. Delegating to real converter.

Code Example:

```
@Component
public class CurrencyConverterProxy implements CurrencyConverter {
   private final CurrencyConverter realConverter;
   private final Map<String, Double> cache = new HashMap<>();
   public double convert(String from, String to, double amount) {
      String key = from + "-" + to + "-" + amount;
      if (cache.containsKey(key)) {
         // log: cache hit
         return cache.get(key);
      }
      // log: cache miss
      double result = realConverter.convert(from, to, amount);
      cache.put(key, result);
      return result;
   }
}
```

Github link to the Codebase:
https://github.com/shreyahegde3/Financial-Tracker-