**Finance Management System**

-Bavatharani S

**Project Goal**

To build a **modular, testable, and secure personal finance tracking system** using Python, with MySQL for storage, following OOP principles and complete with exception handling and unit testing.

**Requirements Breakdown**

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Module / Feature | Task | Tech Used |
| 1 | Database Setup | Design schema: Users, ExpenseCategories, Expenses | MySQL |
| 2 | Entity Classes (Models) | Create classes: User, Expense, ExpenseCategory | Python (OOP) |
| 3 | DAO Implementation | Implement DB operations: CRUD for users and expenses | Python + MySQL Connector |
| 4 | Utility Layer | Add DB connection utils and config loader | Python + File I/O |
| 5 | Exception Handling | Custom exceptions: UserNotFoundException, ExpenseNotFoundException | Python Exception Classes |
| 6 | Menu-Driven Console Interface | Terminal for registering, logging in, adding, viewing expenses | Python |
| 7 | Expense Management | Add, update, delete, view expenses; link to category and user | Full stack |
| 8 | Report Generation | Generate reports by date range, category | SQL + Python logic |
| 9 | Unit Testing | Write test cases for user creation, expense creation, exceptions | unittest / pytest |
| 10 | GitHub Integration | Push code to GitHub repository | Git / GitHub |

**M – Must Have (Essential for MVP)**

|  |  |  |
| --- | --- | --- |
| Requirement | Category | Reason |
| User Registration | Functional | Enables new user access. |
| User Login & Authentication | Functional | Secure user access to system. |
| Add Expense | Functional | Core feature for tracking spending. |
| View Expenses | Functional | Allows users to review spending. |
| Update Expense | Functional | Enables corrections. |
| Delete Expense | Functional | Data management and cleanup. |
| Expense Categorization | Functional | Organizes expense types. |
| View Expense History | Functional | Tracks and reviews past data. |
| Exception Handling (Custom Exceptions) | Non-Functional | Prevents system crashes. |
| Unit Testing for Users and Expenses | Non-Functional | Ensures reliability of core functions. |
| CLI Menu Interface | Functional | Provides user interaction layer. |
| Modularity & OOP Code Structure | Non-Functional | Enhances maintainability and clarity. |
| Performance (Fast Execution) | Non-Functional | Smooth and efficient operations. |
| Security (Sensitive Data Handling) | Non-Functional | Prevents data breaches. |
| Reliability (DB Access & Exception Safety) | Non-Functional | Stable application behavior. |

**S – Should Have (Important but not Critical for MVP)**

|  |  |  |
| --- | --- | --- |
| Requirement | Category | Reason |
| Generate Expense Reports | Functional | Useful insights for users. |
| Search/Filter Expenses | Functional | Helps in quickly locating data. |
| Testability Support (Pytest/Unittest) | Non-Functional | Improves quality assurance. |
| Usability (User-Friendly CLI) | Non-Functional | Enhances experience but MVP can function without it. |

**Prioritized Requirements (High → Low)**

**High Priority (Core System Functionality)**

These are essential for MVP (Minimum Viable Product):

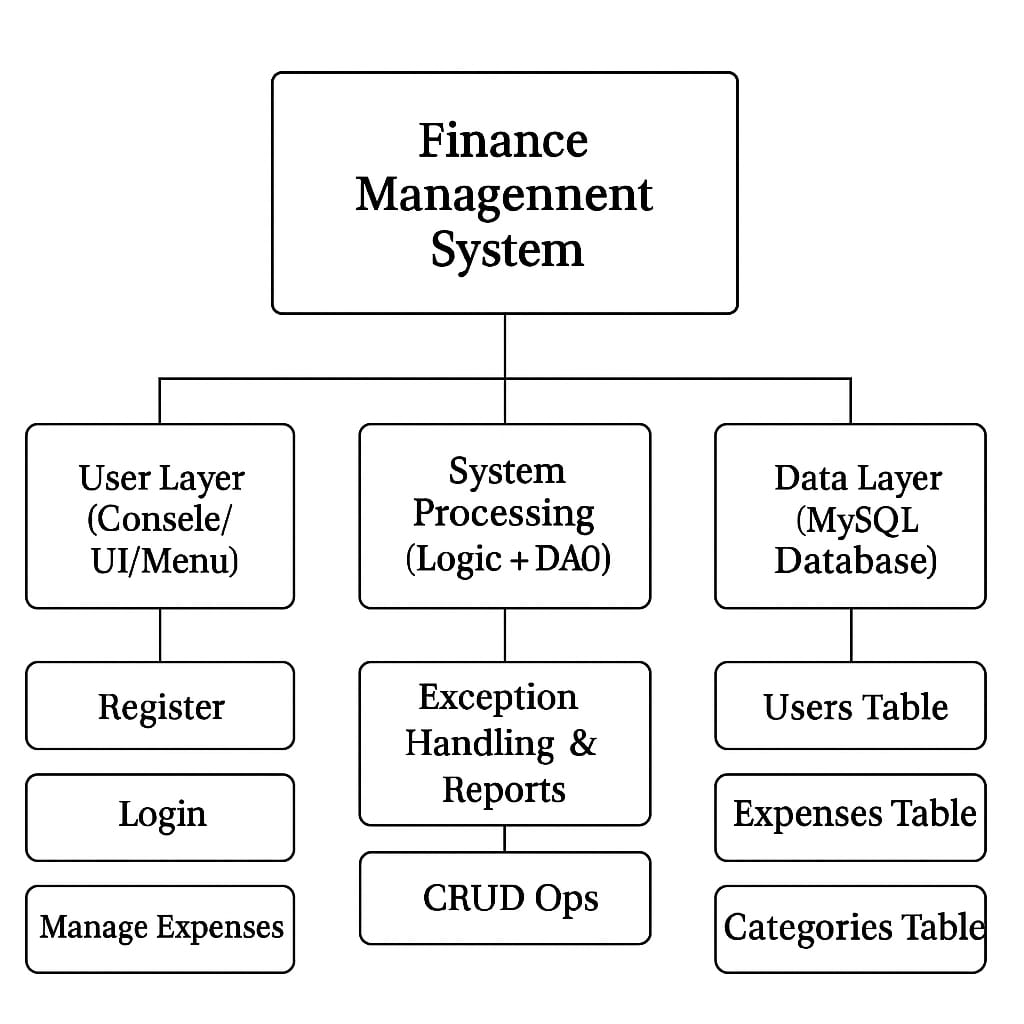
1. Database Schema Design
2. Model Classes (User, Expense, Category)
3. DAO Layer (CRUD operations)
4. Expense Management
5. User Registration/Login
6. Menu-Driven Console Interface

**Medium Priority (Enhancements and Reliability)**

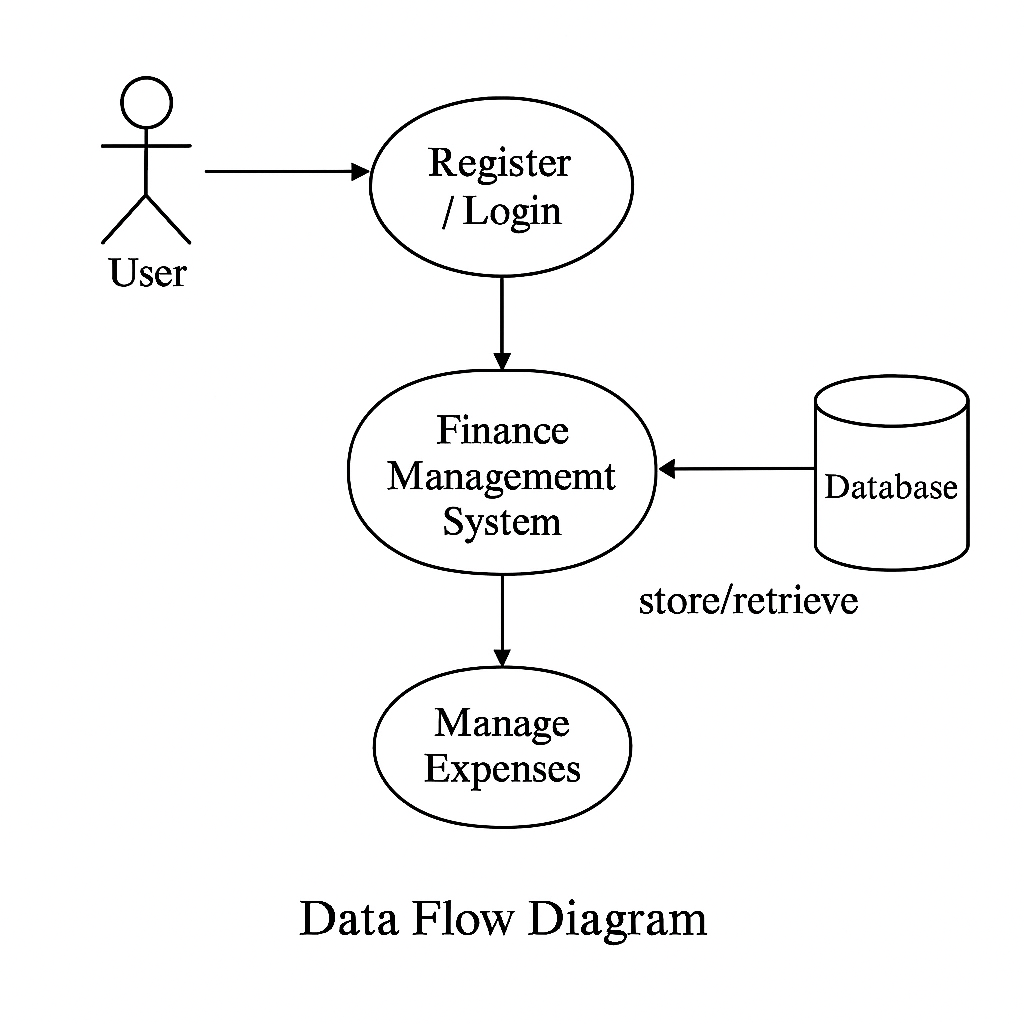
Add usability, and maintainability:

1. Custom Exception Handling
2. DB Utility Setup (DBConnUtil, DBPropertyUtil)
3. Report Generation
4. Unit Testing
5. GitHub Code Submission

**System Flow Chart**



**Data Flow Diagram**



**Expected Output per Module**

**entity/**

* Classes: User, Expense, ExpenseCategory
* Contains: Fields, constructors, getters/setters

**dao/**

* Interfaces and implementations for IFinanceRepository, FinanceRepositoryImpl

**exception/**

* Custom exceptions like UserNotFoundException, ExpenseNotFoundException

**util/**

* DBConnUtil: Returns DB connection
* DBPropertyUtil: Reads database config

**main/**

* FinanceApp.py: Menu-driven console interface

**test/**

* Test cases for CRUD operations, exception scenarios

**config/**

* db.properties: DB config like host, port, username, password

**Exception Handling**

The system includes custom exception classes to manage unexpected conditions gracefully and maintain user-friendly interactions.

1. **UserNotFoundException**

* Raised when: Attempting operations (view/edit/delete) on a user ID that does not exist.
* Handled in: DAO layer and caught in the main application to print friendly error messages.

1. **ExpenseNotFoundException**

* Raised when: An invalid or non-existent expense ID is accessed for viewing or deletion.
* Handled in: DAO or service layer and caught in the menu-driven interface.

**Unit Testing**

Unit tests ensure that key functionalities behave correctly under various conditions, including edge cases and invalid inputs.

**Key Test Cases:**

|  |  |
| --- | --- |
| Test Case | Description |
| test\_user\_creation() | Checks that a user is added successfully to the database |
| test\_expense\_creation() | Verifies correct insertion of expense details |
| test\_expense\_retrieval() | Confirms that expenses related to a user are fetched correctly |
| test\_user\_not\_found\_exception() | Ensures exception is raised for non-existent user operations |
| test\_expense\_not\_found\_exception() | Validates error handling for invalid expense ID usage |

**Conclusion**

The Finance Management System is a comprehensive and modular Python application developed to help users efficiently manage their personal finances. It leverages object-oriented programming principles to create structured, reusable code and ensures robust database interactions through MySQL. The system allows users to securely register, log in, and manage their expenses by adding, updating, or deleting entries while categorizing them appropriately. With the inclusion of custom exception handling, it gracefully manages invalid user operations and system errors. Furthermore, unit testing has been implemented to validate the reliability and correctness of core functionalities such as user and expense management. The system follows a well-organized package structure, separating entities, data access logic, utility classes, and test modules, which enhances maintainability and scalability.