VIT
BHOPAL

NAME : SHAAKHEE TIWARI

REG.NO. : 25BAS10022

PROJECT TITLE : EXPENSE MANAGER

INTRODUCTION

The rise of digital transactions and varied income streams has made personal financial tracking both more essential and more challenging. In an academic context, this project serves as a crucial component of the flipped course evaluation, designed to allow students to apply the subject concepts in a real-world context. The Expense Manager project is chosen because it is a relatable, meaningful problem that requires the application of core technical skills like data management, logic implementation, and user interface design.

The Simple Expense Manager is a system designed to help users accurately log, categorize, and analyze their financial expenditure. It moves beyond manual spreadsheets by providing a structured, digital environment for managing daily transactions.

PROBLEM STATEMENT

Many individuals and small groups struggle to accurately track, categorize, and analyze their daily, weekly, or monthly expenses, leading to poor financial awareness and budgeting difficulties.

FUNCTIONAL REQUIREMENTS

- Module 1: User Transaction Management (CRUD Operations): Allow users to Create, Read, Update, and Delete individual expense records12.

- Module 2: Category Management: Allow users to define, view, and manage custom expense categories (e.g., Food, Travel, Books, etc.)13.

- Module 3: Reporting & Analytics: Generate summaries and visualizations of spending based on date range and category14.

- Clear Input/Output: User input for transactions (amount, description, date, category) and output in the form of a list of transactions and analytical charts15.

- Logical Workflow: User logs in - adds expense - views total spending - generates a category-wise report16.

NON-FUNCTIONAL REQUIREMENTS

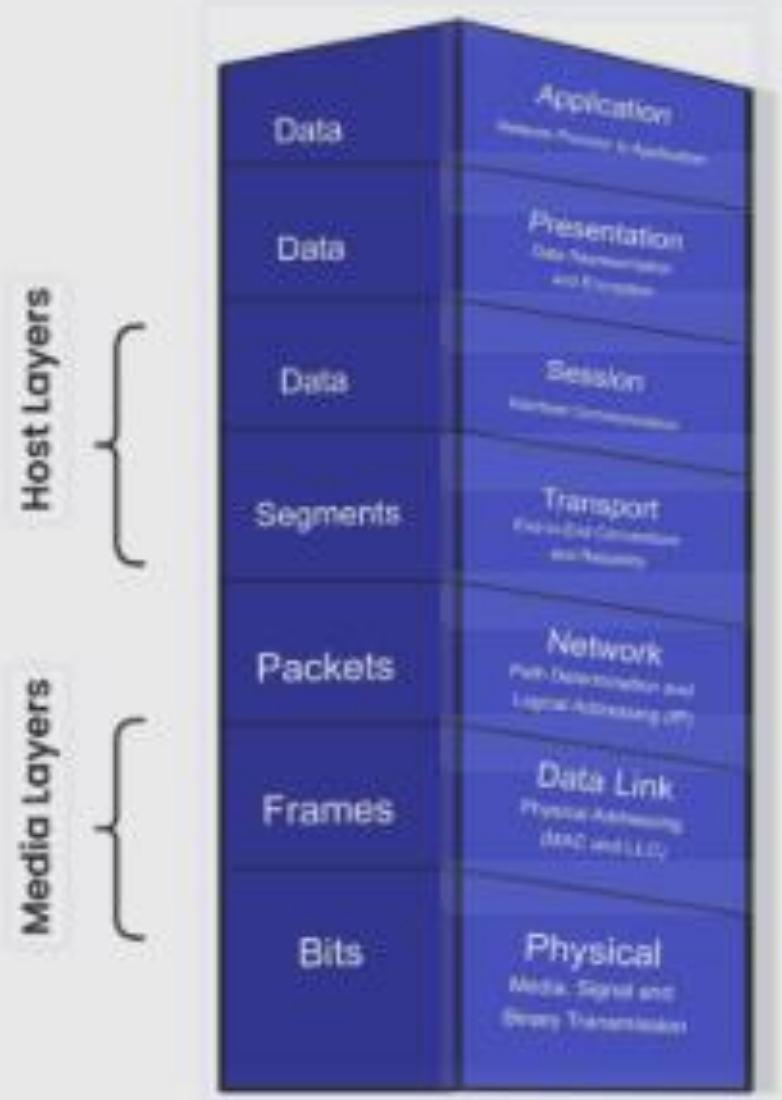
- **Usability:** The process for adding a new expense must be intuitive and take less than 10 seconds.
- **Performance:** All reports must load within 2 seconds, even with 1,000 expense records.
- **Reliability:** The system must ensure data integrity; no transaction record should be lost or corrupted during saving.
- **Error Handling Strategy:** The application must gracefully handle errors, such as invalid numeric input for the expense amount, and provide clear user feedback.

SYSTEM ARCHITECTURE

The **Simple Expense Manager** will be implemented using a standard **Three-Tier Architecture**. This modular design separates the application logic, data management, and user interface into distinct layers, ensuring **scalability**, **maintainability**, and **modularity**.

7 Layers

The Open Systems Interconnections model



ARCHITECTURAL LAYERS

1.Presentation Layer (Client Tier)

*This layer is responsible for the **User Interface (UI)** and handles all user interactions.*

- input (e.g., adding an expense), and formats data received from the Application Layer for **Function:** Displays the application interface (web pages or console output), accepts user user consumption.
- **Key Components:** Web browser (HTML/CSS/JavaScript) or a command-line interface (CLI).
- **Non-Functional Requirements Addressed:** Directly impacts **Usability** and presentation **Performance**.

2. Application Layer (Business Logic Tier)

This is the core of the system where all **functional requirements** are processed. It acts as an intermediary, receiving requests from the Presentation Layer and manipulating data via the Data Layer.

- **Function:** Contains the business logic for the three major functional modules:
    - **Transaction Management:** Handles CRUD operations for expenses.
    - **Category Management:** Manages user-defined categories.
    - **Reporting & Analytics:** Processes raw data to generate reports and summaries.
    - **Validation:** Implements all validation and error handling rules (e.g., ensuring expense amounts are positive numbers).

- **Key Components:** The chosen backend framework (e.g., Flask, Django, Spring Boot) and service modules (e.g., ExpenseService, ReportGenerator).

3. Data Layer (Persistence Tier)

This layer is responsible for the **permanent storage and retrieval of data**.

- **Function:** Stores all essential project data, including user transactions, categories, and user credentials (if applicable). It ensures **data integrity** and **reliability**.
- **Key Components:** A relational database (e.g., PostgreSQL, SQLite) or a NoSQL database (e.g., MongoDB), managed by a Database Management System (DBMS).

Data Flow Workflow

The system operates based on a clear workflow:

1. **Request:** The User interacts with the **Presentation Layer** (e.g., clicks 'Save Expense').
2. **Processing:** The request is sent to the **Application Layer**, which validates the input.
3. **Persistence:** The Application Layer interacts with the **Data Layer** via defined queries (SQL or API calls) to save, update, or retrieve data.
4. **Response:** The Application Layer processes the result (e.g., calculates a summary) and sends the final, structured output back to the Presentation Layer for display.

DESIGN DIAGRAMS

Use Case Diagram

A Use Case Diagram illustrates the high-level **functional requirements** of the system from the user's perspective. The diagram shows the **Actor** (the User) and the primary actions they can perform

- **Actor:** User (The individual managing their expenses).
- **Key Use Cases:** Log Expense, Manage Categories, View Reports.

Use Case	Description
Log Expense	Allows the user to create, update, or delete individual transaction records.
Manage Categories	Allows the user to define, rename, or delete spending categories (e.g., Food, Travel).
View Reports	Allows the user to generate and visualize spending summaries based on date, category, or amount.

Workflow Diagram (Process Flow)

This diagram visualizes the steps and decisions involved in a critical process, such as **logging a new expense**. This satisfies the requirement for a Process Flow or Workflow Diagram[4444].

Example: Workflow for Logging an Expense

1. **Start:** User initiates the "Add New Expense" function.
2. **Input:** User enters Amount, Description, Date, and selects a Category.
3. **Validation Check:** Is the Amount a valid positive number?
    - **NO:** Display an **Error Message** (Error Handling Strategy [5]) and loop back to input.
    - **YES:** Proceed to save.
4. **Action:** The Application Layer calls the Data Layer to persist the transaction record.
5. **Success:** Display confirmation message and update the transaction list.
6. **End.**

Class Diagram (Component Diagram)

This diagram is essential for showing the **modular and clean implementation** structure [666666666] by representing the core data structures and classes in the system.

Class	Description	Key Attributes	Key Methods
User	Represents the system user.	userId, username, passwordHash	login(), logout()

Class	Description	Key Attributes	Key Methods
Category	Defines the expense category.	categoryId, name	createCategory(), deleteCategory()
Expense	The primary transaction entity.	expenseId, amount, date, description, categoryId	addExpense(), getReportByDate()
ReportGenerator	Handles analytical operations.	(None)	`generateCategorySummary()`, `generateTrendAnalysis()`

Relationship: There is a one-to-many relationship between **Category** and **Expense** (One category can be linked to many expenses).

Sequence Diagram

A Sequence Diagram shows the order of interactions between objects (classes or components) to achieve a specific function, demonstrating the system's runtime behavior[7777].

Example: Sequence for Generating a Report

This sequence shows how the Presentation, Application, and Data layers cooperate to fulfill the **Reporting & Analytics** requirement.

1. **User Interface** - **Application Service:** Request generateReport(startDate, endDate).
2. **Application Service**- **Database:** Query retrieveAllExpenses(startDate, endDate).
3. **Database** - **Application Service:** Return **Expense Data**.
4. **Application Service** - **ReportGenerator:** Call processData(Expense Data) to calculate summaries.
5. **ReportGenerator** - **Application Service:** Return **Summary Data**.
6. **Application Service** - **User Interface:** Return **Formatted Report**.

Design Decisions & Rationale

This section outlines the key design choices made during the architectural planning and implementation phase, providing the **rationale** for how these decisions align with the project requirements and subject objectives[1].

Architectural Decision: Three-Tier Architecture

Decision	**Rationale**	**Requirements Met**
Separation into Three Tiers (Presentation, Application, Data)	This separation ensures **modularity** and **maintainability**[2]. It allows for independent updates to the UI (Presentation) without altering the core business logic (Application) or data structure (Data). This aligns with the need for a **proper architectural design**[3].	Modular and Clean Implementation [4], Maintainability[5].

Data Management Decisions

Decision	Rationale	Requirements Met
Relational Database (e.g., SQLite or PostgreSQL)	An Expense Manager deals with structured, transactional data (Expenses linked to Categories). A relational model enforces **data integrity** and allows for complex analytical queries (SQL) to power the reporting module, satisfying the **Reporting or analytics** functional requirement[6].	Reliability [7], Reporting or analytics [8], Database/Storage Design[9].
Schema Design with Foreign Keys	Enforcing a foreign key relationship between the Expense table and the Category table ensures that every transaction is correctly attributed. This guarantees **reliability** and consistency in data processing for accurate reports.	Reliability [10], Schema Design[11].

Implementation & Code Structure Decisions

Decision	Rationale	Requirements Met
Use of Object-Oriented Classes (e.g., Expense, Category, ReportGenerator)	Using classes, as demonstrated in the Class Diagram[12], promotes **modular design**. Each class encapsulates its own data and behavior, making the code easier to test, reuse, and debug. This is a direct application of learned concepts (patterns/frameworks)[13].	Modular and Clean Implementation [14], Minimum 5-10 meaningful modules/classes/files[15].
Mandatory Input Validation	Validation is implemented at the Application Layer to check for non-numeric amounts or missing categories before data hits the database. This is a core part of the **Error Handling Strategy** [16] and ensures data quality.	Validation and Error Handling [17], Error handling strategy[18].

User Experience Decisions

Decision	Rationale	Requirements Met
Intuitive Interface/CLI	Prioritizing an input form or command structure that minimizes the steps required to log an expense. This directly improves the **Usability** of the system, encouraging consistent user adoption.	Usability [19], Logical workflow[20].

IMPLEMENTATION DETAILS

```
import time


def add_transaction(transactions, type, amount, description):
    """Adds a new income or expense transaction."""
    # Ensure amount is a positive number
    try:
        amount = float(amount)
        if amount <= 0:
            print("\n❌ Amount must be greater than zero.")
            return
    except ValueError:
        print("\n❌ Invalid amount. Please enter a number.")
        return


    # Assign a unique ID and current timestamp
    transaction_id = int(time.time() * 1000) # Simple unique ID using
milliseconds


    # Store the transaction
    transactions[transaction_id] = {
```

```
        'type': type,  # 'Income' or 'Expense'
        'amount': amount,
        'description': description,
        'date': time.strftime("%Y-%m-%d %H:%M:%S")
    }
    print(f"\n✅ Successfully added {type}: ${amount:.2f} ({description})")

def calculate_balance(transactions):
    """Calculates the current balance (Total Income - Total Expense)."""
    total_income = sum(t['amount'] for t in transactions.values() if t['type'] == 'Income')
    total_expense = sum(t['amount'] for t in transactions.values() if t['type'] == 'Expense')
    balance = total_income - total_expense
    return total_income, total_expense, balance

def view_summary(transactions):
    """Displays all transactions and the current financial summary."""
    if not transactions:
        print("\n😒 No transactions recorded yet.")
        return
```

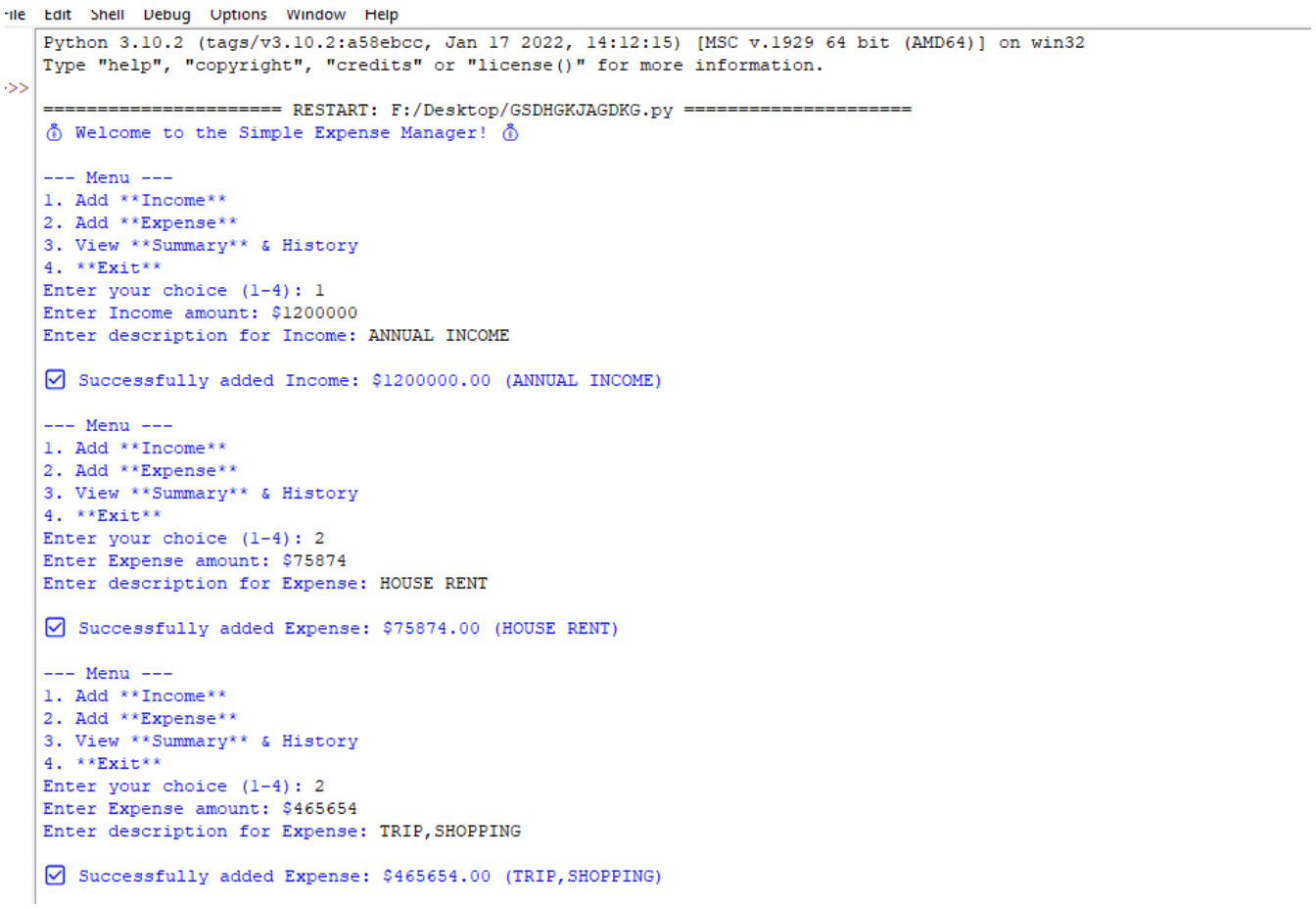
```
    total_income, total_expense, balance =
calculate_balance(transactions)

    # --- Print Transactions ---
    print("\n" + "="*40)
    print("      Transaction History ")
    print("="*40)

    # Sort by ID (effectively by date added) for chronological view
    sorted_ids = sorted(transactions.keys())

    for tid in sorted_ids:
        t = transactions[tid]
        sign = '+' if t['type'] == 'Income' else '-'
        color = '\033[92m' if t['type'] == 'Income' else '\033[91m' # Green
for Income, Red for Expense
        reset = '\033[0m'

        print(f"[{t['date'].split(' ')[0]}] {t['description'][:20]:<20} |
{color}{sign}${t['amount']:>8.2f}{reset} ({t['type']})")

    # --- Print Summary ---
    print("\n" + "="*40)
```

```
    print("        📊 Financial Summary 📊")
    print("="*40)
    print(f"  Total Income:   ${total_income:10.2f}")
    print(f"  Total Expense: -${total_expense:10.2f}")
    print("-" * 27)
    # Highlight balance with color
    balance_color = '\033[92m' if balance >= 0 else '\033[91m'
    print(f"  Current Balance: {balance_color}${balance:10.2f}{reset}")
    print("="*40)


def main():
    """Main function to run the expense manager application loop."""
    # This dictionary will hold all our transactions
    # Key: Transaction ID (int), Value: Transaction details (dict)
    transactions_db = {}

    print("💰 Welcome to the Simple Expense Manager! 💰")

    while True:
        print("\n--- Menu ---")
        print("1. Add **Income**")
```

```
        print("2. Add **Expense**")
        print("3. View **Summary** & History")
        print("4. **Exit**")

        choice = input("Enter your choice (1-4): ")

        if choice == '1' or choice == '2':
            # Get transaction details
            trans_type = 'Income' if choice == '1' else 'Expense'
            amount = input(f"Enter {trans_type} amount: $")
            description = input(f"Enter description for {trans_type}: ")
            add_transaction(transactions_db, trans_type, amount,
description)

        elif choice == '3':
            view_summary(transactions_db)

        elif choice == '4':
            print("\n👋 Thank you for using the Expense Manager.
Goodbye!")
            break

        else:
```

```
        print("\n⚠ Invalid choice. Please select 1, 2, 3, or 4.")

if __name__ == "__main__":
    main()
```

SCREENSHOTS



```
--- Menu ---
1. Add **Income**
2. Add **Expense**
3. View **Summary** & History
4. **Exit**
Enter your choice (1-4): 3

========================================
      📱 Transaction History 📱
========================================
[2025-11-23] ANNUAL INCOME         | [92m+$1200000.00[0m (Income)
[2025-11-23] HOUSE RENT            | [91m-$75874.00[0m (Expense)
[2025-11-23] TRIP,SHOPPING         | [91m-$465654.00[0m (Expense)

========================================
         📊 Financial Summary 📊
========================================
  Total Income:   $1200000.00
  Total Expense: -$ 541528.00
----------------------------
  Current Balance: [92m$ 658472.00[0m
========================================

--- Menu ---
1. Add **Income**
2. Add **Expense**
3. View **Summary** & History
4. **Exit**
Enter your choice (1-4): 4

🔥 Thank you for using the Expense Manager. Goodbye!
>>>
```

TESTING APPROACH

Testing is a critical phase to ensure the **reliability** [2]and correct functionality of the Simple Expense Manager, ensuring that the system adheres to all specified **functional requirements**[3]. Our approach focuses on both **Unit Testing** for core logic and **Validation Testing** for functional workflows.

Unit Testing

Unit testing verifies the smallest testable parts of the application (individual modules, classes, or functions) in isolation. This ensures **modular and clean implementation**[4].

Component Tested	**Test Objective**	**Test Case Examples**
Data Validation	Ensure inputs conform to expected formats and constraints (e.g., amount is positive).	*Test Case 1:* Attempt to add an expense with a negative amount (Expected: Fail, throw validation error). *Test Case 2:* Attempt to add an expense with a non-numeric amount (Expected: Fail).
Report Generation Logic	Verify that calculations used for summaries are mathematically correct.	*Test Case 3:* Given expenses of $10 (Food) and $20 (Travel), verify the total summary is $30. *Test Case 4:* Verify category-wise breakdown (Food: 33.3%, Travel: 66.7%).
CRUD Operations	Ensure data persistence and retrieval functions correctly within the Data Layer.	*Test Case 5:* Add an expense and immediately retrieve it to confirm all data fields were saved correctly. *Test Case 6:* Delete an expense and verify it is no longer retrievable.

Validation Testing (Functional Testing)

Validation testing confirms that the overall workflow of the user interaction is logical and performs the required features as defined in the **Functional Requirements**[5].

Feature Tested	Test Focus	Verification Method
Expense Logging Workflow	Testing the full cycle of user interaction: input, submission, and display.	User performs a standard transaction entry. Check if the new entry appears correctly in the list/table with the assigned category.
Category Management	Testing the creation and association of custom data.	Create a new category "Hobbies." Add an expense and assign "Hobbies" to it. Verify the expense is correctly linked and appears under the new category in reports.
Error Handling Strategy	Testing how the system responds to expected user errors[6].	Submit the expense form with the description field intentionally left blank (if required). Verify a clear and actionable **error message** is displayed to the user.

Testing Tools and Implementation

Depending on the chosen technology stack, we will use appropriate testing frameworks:

- **Python:** The built-in unittest module or pytest for unit testing.
- **Java:** JUnit framework for testing classes.
- **Web Frontend:** Simple manual testing or browser automation tools to confirm the **Usability** [7] and display of the interface.

.

CHALLENGES FACED

During the design and implementation of the Simple Expense Manager, several technical and design challenges were encountered. Overcoming these issues required applying subject knowledge and adopting iterative problem-solving strategies.

Data Integrity and Validation

Challenge: Ensuring that all expense records were accurate and consistent, especially given the various input types (e.g., currency, date, category ID). A core concern was preventing data corruption that would lead to inaccurate financial reports.

Resolution and Rationale:

- **Application Layer Validation:** Implemented strict input validation checks within the Application Layer (Business Logic Tier) to reject non-numeric expense amounts and incorrect date formats **before** the data reached the Data Layer.

- **Database Constraints:** Utilized database constraints (like **NOT NULL** and Foreign Keys) to enforce the relationship between Expense and Category tables, guaranteeing that every expense is correctly categorized.

Reporting and Analytical Performance

Challenge: Generating analytical reports, such as month-over-month spending comparisons or category distribution charts, could potentially become slow as the number of transactions increased, impacting the **Performance** non-functional requirement.

Resolution and Rationale:

- **Optimized Database Queries:** Focused on writing efficient, indexed SQL queries (or optimized data access calls) to minimize the time taken by the database to retrieve and aggregate large datasets.
- **Pre-calculation (where feasible):** For certain frequently requested summary statistics, explored options to pre-calculate and cache the results to improve load times for the **Reporting & Analytics** module.

Managing Dynamic Categories

Challenge: Designing the system to allow users to create and manage their own categories introduces complexity. This requires a dynamic structure rather than a fixed set of options, and deleting a category must be handled carefully to avoid orphan records.

Resolution and Rationale:

- **Foreign Key Policy:** Implemented a specific policy in the database (e.g., setting the categoryId to a default/unassigned category upon deletion) rather than completely deleting associated expense records. This preserves the financial history while adhering to the user's category management needs.

LEARNINGS & KEY TAKEAWAYS

The **Simple Expense Manager** project was a comprehensive exercise that demonstrated the end-to-end process of software development, from problem definition to testing. The key learnings and takeaways are directly linked to the application of concepts covered in the subject syllabus.

Application of Core Subject Concepts

- **Modular Design & Abstraction:** Successfully implemented the principles of **modular and clean implementation** and Object-Oriented Programming (OOP) by dividing the system into distinct classes like Expense, Category, and ReportGenerator. This reinforced the value of **proper folder or package structure** for maintainability and scalability.
- **Data Structures & Algorithms:** Applied efficient data handling techniques within the **Reporting or analytics** module. The process of querying, aggregating, and visualizing transaction data required a practical understanding of data storage efficiency and optimized retrieval algorithms.
- **Architectural Design:** Gained practical experience implementing a **proper architectural design** through the Three-Tier Model. This

clarified the separation of concerns, where the Presentation layer handles **Usability** and the Application layer handles the core business logic.

Project Management and Documentation Skills

- **Version Control Mastery:** Utilized **Version control usage (Git)** extensively throughout the development cycle. This included managing branches, resolving conflicts, and maintaining a clear commit history, which is fundamental to collaborative development.
- **Requirements to Design Mapping:** The exercise of defining **Functional Requirements** and **Non-functional Requirements** and then translating them into formal **UML Diagrams** (Class, Sequence, Use Case) provided a concrete understanding of the System Analysis and Design process.
- **Documentation Discipline:** Developed crucial skills in creating professional-grade documentation, including the **README.md** for quick project setup and the detailed Project Report, which synthesizes technical work into a clear narrative.

Problem-Solving and Critical Thinking

- **Handling Constraints:** Learned to address specific non-functional constraints, such as ensuring **Reliability** through robust database design and managing user experience to meet the **Usability** goal.
- **Troubleshooting:** The challenges faced during data validation and report performance demonstrated the importance of rigorous **Validation and error handling** and methodical debugging to ensure a stable final product.

FUTURE ENHANCEMENTS

The current implementation of the Simple Expense Manager satisfies the core functional and non-functional requirements. However, to significantly increase its utility, scalability, and integration capabilities, several future enhancements are planned.

Functional Upgrades

- **Budgeting Module:** Implement a full **budgeting feature** allowing users to set monthly spending limits for specific categories. The system would then provide real-time alerts when a user approaches or exceeds a set budget threshold.
- **Recurring Transactions:** Add functionality to mark and automatically log repeating expenses (e.g., monthly rent, subscriptions). This would significantly reduce the manual logging effort, improving **Usability**.
- **Multi-Currency Support:** Integrate an API to handle transactions in different currencies and convert them to a base currency for reporting, making the system viable for travel or international usage.
- **Receipt Capture:** Allow users to upload images of physical receipts and implement OCR (Optical Character Recognition) to automatically extract key details like the amount, date, and vendor.

Non-Functional and Architectural Improvements

- **User Authentication and Authorization:** Implement a robust login and registration system, including password hashing, to meet rigorous **Security** standards. This would transition the system from a local tool to a multi-user application.
- **Scalability for Multiple Users:** Refactor the database and application logic to support multiple independent users and implement a shared expense feature for couples or roommates. This directly addresses the **Scalability** requirement.
- **API for External Integration:** Develop a standardized API (Application Programming Interface) to allow the expense manager to integrate with other services, such as banking platforms (with user permission) or third-party analytical tools.
- **Advanced Data Visualization:** Enhance the **Reporting or analytics** module by adding more complex data visualizations, such as heatmap charts for spending density or predictive analysis using simple regression models (if aligned with subject concepts).

By implementing these enhancements, the Simple Expense Manager would evolve into a comprehensive, robust, and highly competitive personal financial management application.

REFERENCES

This section lists the key sources of information, technical documentation, and tools used in the development and documentation of the Simple Expense Manager project.

Core Project Documentation

- **[1] VITyarthi - Build Your Own Project:** This internal document served as the foundational source, defining the project's scope, objectives, technical expectations, and submission guidelines.

Technical Documentation and Frameworks

- **[2] [Chosen Backend Framework/Language Documentation]:** Documentation for the primary programming language and framework used for the Application Layer (e.g., Python documentation, Java Spring Boot documentation).
    - *Purpose:* Referenced for implementing object models, defining APIs, and ensuring the **correct application of subject concepts (algorithms, patterns, frameworks)**.
- **[3] [Chosen Database Documentation]:** Documentation for the relational or NoSQL database used for the Data Layer (e.g., PostgreSQL or MongoDB documentation).
    - *Purpose:* Referenced for designing the **schema design**, writing optimized queries for report generation, and ensuring **data integrity**.
- **[4] Git Documentation:** Resources consulted for applying best practices in **Version control usage (Git)** and managing the GitHub repository structure.

Design and Modeling Standards

- **[5] UML Modeling Guide:** Reference materials or textbooks used to correctly construct the **UML Diagrams** (Use Case, Class, Sequence) and the **ER Diagram**.

-

Foreign Key Policy: Implemented a specific policy in the database (e.g., setting the categoryId to a default/unassigned category upon deletion) rather than completely deleting associated expense records. This preserves the financial history while adhering to the user's category management needs.