

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY

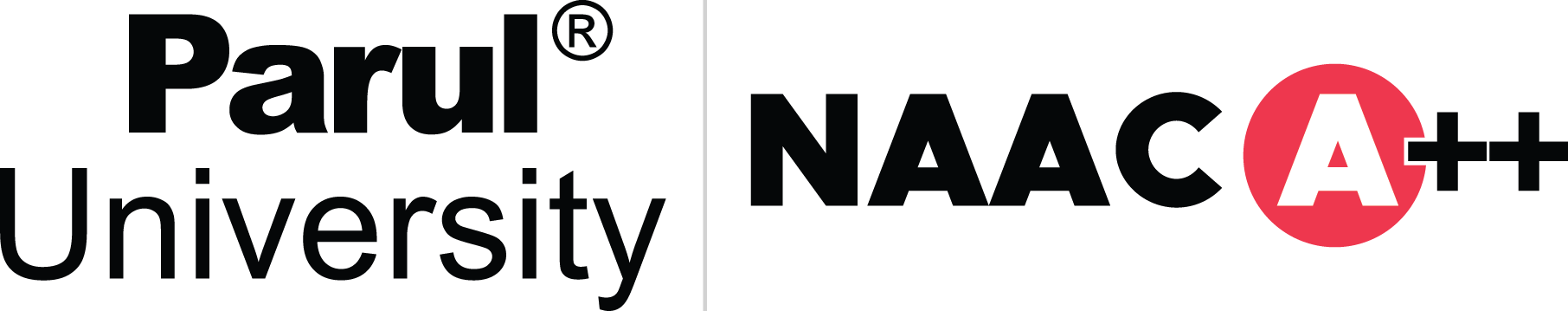
BACHELOR OF TECHNOLOGY

SOFTWARE ENGINEERING LABORATORY

(303105254)

V SEMESTER

Computer Science & Engineering Department



Laboratory Manual

Session 2024-25

CERTIFICATE

This is to Certify that

Mr./Ms. ㅤㅤㅤㅤㅤ VIJAY KUMAR ㅤㅤㅤㅤㅤ ㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤ

With enrolment no. ㅤ 2203051050620ㅤhas successfully completed his/her

Laboratory experiments in SOFTWARE ENGINEERING (303105254) From the

department of ㅤㅤㅤCOMPUTER SCIENCE AND ENGINEERING ㅤ ㅤㅤduring the academic year 2024 - 2025ㅤㅤ



Date of Submission : ………………………. Staff In Charge: ……………………….

Head of department: ……………………….

**TABLE OF CONTENT**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr. No** | **Experiment Title** | **Page No** | | **Date of Completion** | **Sign** | **Marks (out of 10)** |
| **From** | **To** |
| **1.** | Project Definition and objective of the specified module and Perform Requirement Engineering Process. |  |  |  |  |  |
| **2.** | Identify Suitable Design and Implementation model from the different software engineering models. |  |  |  |  |  |
| **3.** | Prepare Software Requirement Specification (SRS) for the selected module. |  |  |  |  |  |
| **4.** | Develop Software project management planning (SPMP) for the specified module. |  |  |  |  |  |
| **5.** | Do Cost and Effort Estimation using different Software Cost Estimation models. |  |  |  |  |  |
| **6.** | Prepare System Analysis and System Design of identified Requirement specification using structure design as DFD with data dictionary and Structure chart for the specific module. |  |  |  |  |  |
| **7.** | Designing the module using Object Oriented approach including Use case Diagram with scenarios, Class Diagram and State Diagram, Collaboration Diagram, Sequence Diagram and Activity Diagram |  |  |  |  |  |
| **8.** | Defining Coding Standards and walk through. |  |  |  |  |  |
| **9.** | Write the test cases for the identified module. |  |  |  |  |  |
| **10.** | Demonstrate the use of different Testing Tools with comparison |  |  |  |  |  |
| **11.** | Define security and quality aspects of the identified module. |  |  |  |  |  |

### **Practical No - 1**

### **Aim -** Project Definition and objective of the specified module and Perform Requirement Engineering

### Process.

### 1. **Context**

Managing personal finances can be a daunting task without proper tools to keep track of income, expenses, and savings. In today's world, where individuals manage multiple transactions daily, an efficient system is necessary to record, categorize, and analyze these transactions to make informed financial decisions. The Personal Finance Management System addresses this need by providing users with a simple platform to log their financial transactions, categorize them, and generate useful reports.

The project is a part of the software engineering coursework, aimed at providing students practical experience in building functional software systems using modern technologies like Flask, SQLAlchemy, and HTML templates.

### 2. **Information**

The Personal Finance Management System is a web-based application built using Flask, a lightweight Python web framework. The system allows users to:

* Add financial transactions with details such as the name, description, category (e.g., income, expense), and amount.
* View a list of all their transactions.
* Generate reports that summarize their financial activities.

The system stores transaction data in a SQLite database, ensuring persistent storage and easy retrieval. The application's interface is built using HTML templates, and its logic is handled by Flask's routing system.

### 3. **Objective**

The main objective of this project is to create a personal finance management tool that enables users to efficiently manage their income and expenses. The project aims to:

* Simplify the process of tracking financial transactions.
* Allow users to view and analyze their spending patterns through reports.
* Provide an intuitive and user-friendly interface for managing finances.
* Implement a functional web application demonstrating the use of Flask for backend development and SQLAlchemy for database management.

### 4. **Functions**

The Personal Finance Management System includes the following core functionalities:

1. **Add Transaction**: Users can log new financial transactions by providing details such as the transaction name, description, category (e.g., income, expense), and amount.
2. **View Transactions**: The system displays a list of all recorded transactions, allowing users to keep track of their financial activities.
3. **Generate Report**: The application offers a report feature where users can view an overview of their financial performance, including total income, total expenses, and other summary statistics.
4. **Data Persistence**: All transaction data is stored in a SQLite database, allowing users to persist their data between sessions.
5. **Responsive Design**: The web application is designed to work on different screen sizes, making it accessible from both desktop and mobile devices.

### 5. **Performance**

The performance of the Personal Finance Management System can be evaluated based on the following criteria:

* **Efficiency**: The application performs CRUD operations (Create, Read, Update, Delete) efficiently using SQLAlchemy. All operations (adding, viewing, and generating reports) occur within milliseconds.
* **User Interface**: The interface is simple and intuitive, designed using HTML templates with a clean navigation system that ensures ease of use.
* **Scalability**: While the project uses SQLite for simplicity, the system can easily be scaled to use a more robust database like MySQL or PostgreSQL with minor configuration changes.
* **Responsiveness**: The system is responsive, meaning it adjusts to different screen sizes, ensuring a smooth user experience on both mobile and desktop.

### 6. **Problem Decomposition**

The project was divided into several modules to break down the complexity and ensure smooth development. Each module addresses specific functionality:

1. **Transaction Module**: Handles the addition, storage, and retrieval of financial transactions. This module interacts with the database (SQLAlchemy) and is responsible for CRUD operations.
2. **Database Module**: This module defines the database schema, including the Transaction table that stores all transaction data. It handles migrations and interactions with the SQLite database.
3. **Report Generation Module**: Provides summary reports of financial activities. This module calculates metrics such as total income and expenses based on the stored transactions.
4. **User Interface Module**: The UI is developed using HTML templates and is responsible for rendering the front-end. Flask's Jinja2 template engine is used to dynamically inject transaction data into the web pages.
5. **Routing and Logic Module**: Flask's routing system defines the endpoints for different actions (e.g., adding transactions, viewing reports). This module handles the user requests and connects the front-end with the back-end logic.

### **Practical No - 2**

### **Aim -** Identifying Suitable Design and Implementation Model for the Personal Finance Management System

### 1. **Planning and Requirement Analysis**

In the initial phase, we performed planning and requirement analysis to define the scope and functionality of the project. For the Personal Finance Management System, the following key requirements were identified:

* The system should allow users to add, view, and categorize financial transactions.
* Users should be able to generate reports summarizing their financial activities.
* The application must store transaction data in a persistent database (e.g., SQLite).
* The interface must be intuitive and accessible on both desktop and mobile devices.

Based on these requirements, we decided to follow the Waterfall Model for development. This model suits small to medium-sized projects with clear requirements and predictable development timelines.

### 2. **Defining Requirements**

In the requirements definition stage, the project was broken down into several functional components. These include:

* **User Interface Requirements**: A simple, user-friendly interface where users can easily input financial transactions and view reports.
* **Database Requirements**: A relational database to store transaction data persistently. Each transaction includes attributes like name, description, category, and amount.
* **Reporting Requirements**: The system should be able to generate summary reports showing the total income and expenses over a period.

These requirements were defined clearly before the development started to avoid scope creep and ensure smooth development.

### 3. **Designing the Product Architecture**

In the design phase, the following architecture was chosen for the system:

1. **Front-end Design**: The front end is built using HTML, CSS, and Jinja2 templates. This ensures a clean and responsive user interface.
2. **Back-end Design**: Flask was chosen as the back-end framework due to its simplicity and suitability for small projects. Flask routes handle user requests, such as adding transactions and viewing reports.
3. **Database Design**: SQLAlchemy is used as the Object-Relational Mapping (ORM) tool to interact with an SQLite database. A Transaction table stores all transaction details, including name, description, category, and amount.
4. **Layered Architecture**: The system follows a layered architecture:
   * **Presentation Layer**: Manages user interaction.
   * **Business Logic Layer**: Handles transaction processing and report generation.
   * **Data Layer**: Manages data storage and retrieval from the database.

### 4. **Building or Developing the Product**

During the development phase, the architecture designed in the previous step was implemented. Key components include:

* **Flask Application**: The Flask framework was used to build the web application. Flask handles routing, connecting different pages, and processing user inputs.
* **HTML Templates**: HTML and Jinja2 templates were used to create dynamic pages where users can add transactions and view reports.
* **Database Integration**: SQLAlchemy ORM was integrated to interact with the SQLite database. All transaction data was stored and retrieved using ORM methods.

This phase also included setting up the project structure, creating necessary files, and writing code to meet functional requirements.

### 5. **Testing the Product**

After development, the system underwent rigorous testing. Both functional and non-functional testing were carried out:

* **Unit Testing**: Each function (e.g., adding a transaction, viewing transactions) was tested individually to ensure correctness.
* **Integration Testing**: After individual components were tested, they were integrated, and the interaction between them was tested. For instance, adding a transaction and ensuring it appears in the transaction list.
* **User Acceptance Testing**: The system was tested for usability and responsiveness. Users should be able to navigate the application easily, and the UI should adjust to different screen sizes.
* **Database Testing**: Tests were conducted to verify that data is correctly stored in the database and can be retrieved for reports.

### 6. **Deployment in Market and Maintenance**

Once the system passed all tests, it was ready for deployment. The Flask application was deployed locally for testing purposes. For larger-scale use, it could be deployed on cloud platforms like Heroku or AWS.

The project also considers maintenance, ensuring the following:

* **Bug Fixes**: Any issues discovered during testing or user feedback can be addressed through version updates.
* **Future Enhancements**: The system is designed to be scalable, allowing future features like user authentication or advanced reporting.

### 7. **Advantages of the Waterfall Model**

* **Clear Structure**: The Waterfall Model provides a well-defined structure. Each phase is completed before moving to the next, ensuring that all requirements are met before development starts.
* **Easy to Understand**: The model is straightforward, making it easier to manage and execute for small projects with clear requirements like this one.
* **Well-documented Process**: Each phase has its own deliverables and milestones, ensuring proper documentation and progress tracking.

### 8. **Drawbacks of the Waterfall Model**

* **Inflexibility**: The Waterfall Model does not handle changes easily. Once a phase is completed, going back to make changes is difficult and costly.
* **Late Testing**: Since testing is done only after development, any critical issues discovered in the later phases can require significant rework.
* **Assumes Perfect Requirements**: The model assumes that all requirements are known from the beginning, which may not always be true in dynamic projects.

### 9. **Justification for Choosing the Waterfall Model**

The Waterfall Model was chosen for this project because:

1. **Clear and Well-defined Requirements**: In the case of the Personal Finance Management System, the requirements were clear from the beginning. The system needed to handle basic financial transactions, store them in a database, and generate reports.
2. **Small-scale Project**: Since this is a small academic project with limited scope, the Waterfall Model provides a simple, structured approach. There is no need for iterative cycles or frequent changes, making it suitable for this project.
3. **Ease of Implementation**: The model allowed for straightforward planning and development. By following a sequential approach, we were able to focus on one phase at a time, ensuring that each phase was completed fully before moving on to the next.

### **Practical No – 3**

### **Aim -** Software Requirement Engineering: Personal Finance Management System

**Name of System**:  
Personal Finance Management System

### **Assumptions**

1. The system will be used by individual users to track their personal income and expenses.
2. Users will have a basic understanding of web applications and how to input financial data.
3. The system will be deployed locally or on a cloud platform and accessible via web browsers on both desktop and mobile devices.
4. Users will manually enter their transactions and rely on the system to categorize and report on their financial data.
5. Data will be stored locally (using SQLite for the academic version), and no cloud storage is assumed at this stage.
6. The system will not require user authentication for the academic project, though it may be added in future versions.
7. No sensitive or high-risk financial information will be stored, as the system is designed for personal use only.

### **Software Requirements Specification (SRS)**

The SRS document describes all functional and non-functional requirements for the system. Requirements are divided into different categories:

### 1. **Functional Requirements**

Functional requirements define the core operations the system must perform to meet its objectives.

#### 1.1 **Transaction Management**

* The system shall allow users to add new transactions.
* The system shall allow users to input the following data for each transaction:
  + Transaction name (mandatory)
  + Description (optional)
  + Amount (mandatory)
  + Category (e.g., income, expense) (mandatory)
  + Date (optional; defaults to current date)

#### 1.2 **Transaction Display**

* The system shall display a list of all transactions in a tabular format.
* The system shall allow users to filter transactions by:
  + Date range
  + Category (income, expense)

#### 1.3 **Transaction Categorization**

* The system shall automatically categorize transactions as either "income" or "expense" based on user input.

#### 1.4 **Report Generation**

* The system shall generate summary reports based on stored transactions.
* The report shall include:
  + Total income over a given period
  + Total expenses over a given period
  + Net balance (income minus expenses)

#### 1.5 **Data Persistence**

* The system shall store all transaction data in a persistent database (SQLite).
* The system shall allow for the retrieval of past transactions based on user-specified criteria.

### 2. **Non-Functional Requirements**

Non-functional requirements define the performance, usability, security, and other aspects of the system.

#### 2.1 **Usability**

* The system shall have an intuitive and user-friendly interface.
* The user interface shall be accessible on both desktop and mobile devices (responsive design).

#### 2.2 **Performance**

* The system shall respond to user actions within 2 seconds.
* The system shall be capable of handling up to 100 transactions efficiently without noticeable performance degradation.

#### 2.3 **Reliability**

* The system shall ensure that data is stored reliably and retrieved correctly from the database.

#### 2.4 **Security**

* As the project is an academic prototype, no user authentication or encryption is required at this stage. However, the system shall be designed to ensure that data integrity is maintained.

#### 2.5 **Scalability**

* The system architecture shall allow for easy migration from SQLite to a more scalable database such as MySQL or PostgreSQL if needed in future versions.

#### 2.6 **Maintainability**

* The codebase shall be well-structured and documented, ensuring ease of future maintenance and upgrades.

#### 2.7 **Portability**

* The system shall be deployable on any server supporting Flask and SQLite. It should also be easily portable to a cloud platform like Heroku or AWS with minimal changes.

### 3. **User Requirements**

User requirements capture the goals and needs of the end-users.

#### 3.1 **User Characteristics**

* The users of this system will primarily be individual users looking to manage their personal finances. They are assumed to have basic computer literacy.

#### 3.2 **User Interface Requirements**

* The system should present a clean and intuitive user interface where users can input, view, and manage transactions easily.
* The interface should include:
  + A home page with an option to add transactions
  + A page to view all transactions
  + A report generation page with financial summaries

#### 3.3 **Input Requirements**

* The system should provide simple forms for users to input financial transaction data. Input validation should be performed to ensure all required fields are completed correctly.

### 4. **System Requirements**

System requirements define the technical specifications for the system.

#### 4.1 **Software Requirements**

* Flask (for the web application framework)
* SQLAlchemy (for database ORM)
* SQLite (for the database)
* HTML, CSS (for frontend)
* Jinja2 (for templating)

#### 4.2 **Hardware Requirements**

* The system requires a local server environment or cloud hosting environment to run Flask applications.
* Basic system requirements for running the application:
  + CPU: 2 GHz processor or higher
  + RAM: 2 GB or higher
  + Storage: 500 MB (for local data storage)

#### 4.3 **Platform Requirements**

* The system shall run on any operating system that supports Flask (e.g., Windows, macOS, Linux).

### 5. **Domain Requirements**

These requirements relate to the specific domain of personal finance management.

#### 5.1 **Financial Data**

* The system should allow users to track their income and expenses effectively.
* Financial data stored in the system should follow standard accounting practices (e.g., negative values for expenses).

#### 5.2 **Categories**

* Income and expense categories should be predefined but allow for user customization in future versions.

### 6. **External Interface Requirements**

These requirements define how the system interacts with external systems and users.

#### 6.1 **User Interface**

* The system will interact with users via a web-based interface, using HTML for structure and CSS for styling.

#### 6.2 **Database Interface**

* The system will interact with an SQLite database using SQLAlchemy as the ORM. All transaction data will be stored and retrieved through this interface.

### 7. **Future Requirements**

In future versions of the project, the following enhancements could be added:

* **User Authentication**: Add login and registration functionality to secure user data.
* **Graphical Reports**: Display income and expenses using charts or graphs for better visualization.
* **Multi-User Support**: Extend the system to support multiple users with individual accounts.
* **Cloud Storage**: Store data on a cloud database for scalability and remote access.

### **Practical No - 4**

### **Aim -** Software Project Management Planning (SPMP) for the Personal Finance Management System

### **1. Objective**

The objective of the Personal Finance Management System project is to develop a web-based application that allows individual users to track their personal financial transactions. The system will enable users to:

* Add, view, and categorize transactions.
* Generate reports summarizing income, expenses, and net balance.
* Maintain persistent data storage for transactions using an SQLite database.

The project aims to complete the development, testing, and deployment of the system within the timeline set for the academic semester, adhering to the project scope and requirements defined in the Software Requirement Specification (SRS).

### **2. Project Scheduling and Tracking**

Project scheduling and tracking ensure that the project milestones are achieved within the deadlines. Below is the planned schedule for the project development process:

#### **Key Milestones**:

* **Week 1-2**: Requirement Gathering and Analysis
* **Week 3-4**: Design of the system architecture and database schema
* **Week 5-6**: Frontend development (UI/UX design)
* **Week 7-8**: Backend development (Flask framework setup, routes, and logic implementation)
* **Week 9-10**: Integration and database connection
* **Week 11-12**: Testing (unit testing, integration testing, user acceptance testing)
* **Week 13-14**: Final deployment and project documentation

#### **Tracking Progress**:

* **Weekly Checkpoints**: Each task will be reviewed weekly to assess progress against the project timeline. Any delays or roadblocks will be addressed immediately to ensure that the project stays on track.
* **Task Breakdown**: The project tasks are divided into smaller, manageable units and assigned deadlines. Each task is tracked individually, ensuring that deadlines are met without overwhelming any particular phase of development.

### **3. Scheduling Method**

The project will use the Work Breakdown Structure (WBS) scheduling method, where the overall project is broken down into smaller, manageable tasks. Each task is then assigned a time frame, and dependencies between tasks are identified. This method ensures that critical tasks are completed first and allows for easy monitoring and adjustment of the schedule.

We will also employ the Critical Path Method (CPM) to identify the longest sequence of tasks that must be completed on time for the entire project to finish by the deadline. Tasks on the critical path are closely monitored to avoid delays.

### **4. Gantt Chart**

The Gantt chart provides a visual representation of the project schedule. Below is a Gantt chart for the Personal Finance Management System project:

| **Task** | **Start Date** | **End Date** | **Duration (Weeks)** | **Dependencies** |
| --- | --- | --- | --- | --- |
| Requirement Gathering and Analysis | Week 1 | Week 2 | 2 Weeks | None |
| System Architecture Design | Week 3 | Week 4 | 2 Weeks | Requirement Gathering |
| Database Schema Design | Week 3 | Week 4 | 2 Weeks | System Architecture |
| UI/UX Design (Frontend) | Week 5 | Week 6 | 2 Weeks | Architecture Design |
| Backend Development (Flask) | Week 7 | Week 8 | 2 Weeks | UI/UX Design |
| Database Integration | Week 9 | Week 10 | 2 Weeks | Backend Development |
| Testing | Week 11 | Week 12 | 2 Weeks | Backend Development |
| Final Deployment and Documentation | Week 13 | Week 14 | 2 Weeks | Testing |

### 

### **5. Experiment Demonstration**

#### **Purpose of the Experiment**:

The experiment aims to demonstrate the functionality of the Personal Finance Management System by tracking and testing key features like adding transactions, generating reports, and categorizing income and expenses.

#### **Experiment Setup**:

* **Step 1**: Open the web application interface on a desktop or mobile browser.
* **Step 2**: Add sample transactions with details such as name, description, amount, category, and date.
* **Step 3**: Generate a report summarizing the total income and expenses for a specific date range.
* **Step 4**: View the list of all transactions and filter them by category (income/expense).
* **Step 5**: Validate that all inputted transactions are saved correctly in the SQLite database.

#### **Success Criteria**:

* The system should successfully store, retrieve, and display transactions.
* Generated reports should reflect accurate financial data.
* The web interface should be responsive and functional across different devices.

#### **Data Recording**:

For each experiment, log the transaction data entered and the system's output (e.g., report summary, transaction list). Any errors or inconsistencies will be noted and rectified during testing.

### **Practical No - 5**

### **Aim -** Cost and Effort Estimation forPersonal Finance Management System

### **1. Objectives of Cost and Effort Estimation**

The primary goal of cost and effort estimation is to:

* Estimate the total effort required to complete the Personal Finance Management System.
* Predict the cost of project development, including resources and time.
* Identify potential risks in terms of underestimation or overestimation.

### **2. Software Cost Estimation Models**

Several cost estimation models are commonly used in software engineering. For this practical, we will focus on:

1. **Lines of Code (LOC)-Based Estimation**
2. **COCOMO (Constructive Cost Model)**
3. **Function Point Analysis (FPA)**

### **3. Lines of Code (LOC)-Based Estimation**

The LOC model estimates the effort and cost based on the number of lines of code in the software. The estimated LOC for this project is **1500 lines** (based on the scope of the project).

#### Estimation Parameters:

* **Average Effort per LOC**: 30 LOC per hour (industry standard for academic projects).
* **Developer Salary**: $25/hour.

#### Calculations:

* **Effort (in hours)** = Estimated LOC / LOC per hour  
  = 1500 LOC / 30 LOC per hour  
  = **50 hours**
* **Cost** = Effort (in hours) \* Developer Salary  
  = 50 hours \* $25/hour  
  = **$1250**

Thus, using the LOC model, the estimated effort is **50 hours**, and the estimated cost is **$1250**.

### **4. COCOMO Model**

The Constructive Cost Model (COCOMO) is a widely used model that estimates the effort required based on project size. COCOMO has three categories: Basic, Intermediate, and Detailed. We will use the **Basic COCOMO** model for this project, which uses a formula based on the size of the project in terms of Kilo Lines of Code (KLOC).

#### COCOMO Formula:

For **organic projects** (small, simple projects like the Personal Finance Management System): Effort (person-months)=2.4×(KLOC)1.05\text{Effort (person-months)} = 2.4 \times \text{(KLOC)}^{1.05}Effort (person-months)=2.4×(KLOC)1.05

#### Estimation Parameters:

* Estimated KLOC = 1.5 KLOC (1500 LOC)

#### Calculations:

* **Effort (in person-months)** = 2.4 × (1.5)^1.05  
  = 2.4 × 1.54  
  = **3.696 person-months**
* Convert person-months to hours:  
  = 3.696 person-months × 152 hours/month  
  = **561.79 hours**
* **Cost** = Effort (in hours) × Developer Salary  
  = 561.79 hours × $25/hour  
  = **$14,044.75**

Thus, using the COCOMO model, the estimated effort is **561.79 hours**, and the estimated cost is **$14,044.75**.

### **5. Function Point Analysis (FPA)**

FPA is used to estimate the size of software projects by quantifying the functionality provided to the user. It takes into account various functional components like inputs, outputs, user interactions, files, and interfaces. For this project, we assume the following based on the project scope:

#### Functional Units:

* **External Inputs (EI)**: 10 (forms for adding transactions, filters, etc.)
* **External Outputs (EO)**: 8 (transaction list, reports, etc.)
* **Internal Logical Files (ILF)**: 5 (transaction data, categories, etc.)
* **External Interface Files (EIF)**: 2 (interaction with the database)
* **External Queries (EQ)**: 4 (report generation, filtering transactions)

#### Complexity Weighting:

| **Function Type** | **Complexity Weight** | **Number of Functions** | **Total Points** |
| --- | --- | --- | --- |
| External Inputs (EI) | 3 | 10 | 30 |
| External Outputs (EO) | 4 | 8 | 32 |
| Internal Logical Files (ILF) | 7 | 5 | 35 |
| External Interface Files (EIF) | 5 | 2 | 10 |
| External Queries (EQ) | 3 | 4 | 12 |
| **Total** |  |  | **119** |

#### Function Point Calculation:

1. **Total Unadjusted Function Points** = 119
2. **Adjustment Factor**: Assume a standard adjustment factor of 1.0 (since it's an academic project with moderate complexity).

#### Final Function Point Value:

* Adjusted Function Points = 119 × 1.0 = **119 Function Points**

#### Effort and Cost Estimation:

* Average effort per function point = **5 hours** (for academic projects)
* Developer Salary = $25/hour
* **Effort (in hours)** = 119 function points × 5 hours  
  = **595 hours**
* **Cost** = 595 hours × $25/hour  
  = **$14,875**

Thus, using Function Point Analysis, the estimated effort is **595 hours**, and the estimated cost is **$14,875**.

### **6. Comparison of Models**

| **Model** | **Effort (in hours)** | **Estimated Cost (in USD)** |
| --- | --- | --- |
| **LOC-Based Estimation** | 50 hours | $1250 |
| **COCOMO Model** | 561.79 hours | $14,044.75 |
| **Function Point Analysis** | 595 hours | $14,875 |

### **7. Conclusion**

* The **LOC-Based Estimation** provides a very basic estimate, with a low cost and effort estimation. This method is simplistic and may not capture the full complexity of the project.
* The **COCOMO Model** offers a more realistic estimation, especially for organic projects like this one. It estimates much higher effort and cost due to its calculation based on lines of code and project complexity.
* **Function Point Analysis** provides a detailed estimation based on the functionality offered by the system. This method also predicts a higher effort and cost, as it factors in user interactions and system functionality in a more granular way.

### **Practical No - 6**

### **Aim -** System Analysis and System Design for Personal Finance Management System

### **1. Introduction to System Analysis and Design**

**System Analysis** involves understanding the business needs and defining the software system's functional and non-functional requirements. The purpose of system analysis is to gather, analyze, and document all the requirements and to create a clear path for the system design phase.

**System Design** involves transforming the gathered requirements into a blueprint for constructing the software. This phase includes creating system architecture, data flow diagrams (DFDs), and identifying data entities and relationships.

In this project, we will use **Structured Design** to create Data Flow Diagrams (DFDs) and data dictionaries for the Personal Finance Management System.

### **2. Data Flow Diagram (DFD)**

A **Data Flow Diagram (DFD)** is a graphical representation of how data moves through a system. It helps to visualize the flow of information from external entities into the system, how that data is processed, and where it is stored.

* **External Entity**: Anything outside the system that interacts with the system.
* **Process**: Actions or tasks performed within the system to transform inputs into outputs.
* **Data Store**: Storage where data is retained.
* **Data Flow**: The flow of data between entities, processes, and data stores.

### **3. DFD Levels**

There are different levels of DFD:

* **Level 0 (Context Diagram)**: Represents the entire system as a single process and its interaction with external entities.
* **Level 1 DFD**: Breaks down the main process into sub-processes, showing more detailed interactions.

### **4. DFD for Personal Finance Management System**

#### **Context Diagram (Level 0 DFD)**

In the context diagram, we represent the entire system as a single process. The external entities interact with the system through input (adding transactions) and receiving output (viewing reports, transactions).

**Entities**:

1. **User**: The user interacts with the system to add and view transactions, generate reports, and track expenses.

**Process**:

1. **Personal Finance Management System**: This represents the system that performs the core operations like adding a transaction, generating a report, and viewing transaction history.

**Data Store**:

* **Transaction Data Store**: Stores all financial transactions of the user.

**Data Flow**:

* **Input Flow**: User inputs transaction details.
* **Output Flow**: User retrieves reports or views transactions.

#### **Level 0 DFD for Personal Finance Management System**

plaintext

Copy code

-----------------------------------------

| External Entity: User |

| |

| |

| +------------------------+ |

| | | |

| | Personal Finance | |

| | Management System | |

| +------------------------+ |

| ^ | |

| | Input: Transactions | |

| v v |

| +------------------------+ +--------------------+

| | Transaction Data | | Output: Reports |

| | Store | +--------------------+

| +------------------------+

-----------------------------------------

#### **Level 1 DFD (Breaking the system into processes)**

1. **Process 1: Add Transaction**
   * User inputs transaction data (name, amount, category, date).
   * Data is stored in the transaction database.
2. **Process 2: View Transactions**
   * User requests to view their transaction history.
   * System retrieves and displays transaction data from the database.
3. **Process 3: Generate Report**
   * User requests a summary of income and expenses over a specified time period.
   * System processes the data and outputs a report.

plaintext

Copy code

---------------------------------------------------------

| External Entity: User |

| |

| +------------------------+ +--------------------+ |

| | Add Transaction | | View Transactions | |

| +------------------------+ +--------------------+ |

| ^ | |

| | v |

| +----------------------------+ +---------------+ |

| | Transaction Data Store | | Generate Report |

| +----------------------------+ +-----------------+

---------------------------------------------------------

### **5. Data Dictionary**

A **Data Dictionary** is a structured repository of information about the data that flows within the system. It describes each data element and its attributes.

| **Data Element** | **Description** | **Type** | **Length** | **Example** |
| --- | --- | --- | --- | --- |
| **Transaction ID** | Unique identifier for each transaction | Integer | 10 | 101 |
| **Transaction Name** | Name or title of the transaction | String | 50 | "Groceries" |
| **Amount** | Amount of money spent or earned | Decimal | 8, 2 | 120.50 |
| **Category** | Category of the transaction (e.g., income, expense) | String | 20 | "Expense" |
| **Date** | Date when the transaction occurred | Date | 10 | "2024-10-03" |
| **User ID** | Unique identifier for the user | Integer | 10 | 5001 |
| **Report Type** | Type of report requested (e.g., daily, monthly, yearly) | String | 10 | "Monthly" |

### **6. Structure Chart**

A **Structure Chart** is a hierarchical diagram that shows the breakdown of the system's processes. It provides a top-down view of the system's functionality.

#### Structure Chart for Personal Finance Management System:

plaintext

Copy code

--------------------------------------------------

| Personal Finance Management System |

|------------------------------------------------|

| |

| +--------------------------------------------+ |

| | Add Transaction | |

| +--------------------------------------------+ |

| |

| +--------------------------------------------+ |

| | View Transactions | |

| +--------------------------------------------+ |

| |

| +--------------------------------------------+ |

| | Generate Report | |

| +--------------------------------------------+ |

--------------------------------------------------

### **7. Example of a Data Flow Diagram for Order Processing**

In a typical **Order Processing System**, we can have the following DFD:

#### **Level 0 DFD for Order Processing**

**External Entities**:

1. **Customer**: Places an order.
2. **Supplier**: Provides the ordered items.

**Processes**:

1. **Order Processing**: Handles the customer’s order and forwards it to the supplier.

**Data Store**:

* **Order Database**: Stores order details.

plaintext

Copy code

-----------------------------------------

| External Entity: Customer |

| |

| +------------------------+ |

| | Order Processing | |

| +------------------------+ |

| ^ | |

| | Input: Order Details | |

| v v |

| +------------------------+ +--------------------+

| | Order Database | | Output: Order |

| +------------------------+ | Confirmation |

-----------------------------------------

### **8. Experiment Demonstration**

#### **Objective**:

The experiment demonstrates the working of the Personal Finance Management System, showcasing how data flows through the system.

#### **Steps**:

1. **Step 1**: Start the web application and navigate to the "Add Transaction" page.
2. **Step 2**: Input transaction details such as name, amount, category, and date, and submit.
3. **Step 3**: The system processes the input and stores the transaction in the database.
4. **Step 4**: Navigate to the "View Transactions" page and verify that the added transaction is displayed.
5. **Step 5**: Go to the "Generate Report" page and request a report for a specific date range. The system processes the data and outputs the requested report.

#### **Expected Output**:

* The system successfully stores and retrieves transaction data.
* The user can view the report generated based on the input.

### **Practical No - 7**

**Aim -** Designing the module using Object Oriented approach including Use case Diagram with scenarios, Class Diagram and State Diagram, Collaboration Diagram, Sequence Diagram and Activity Diagram

### **1. Objective**

The main objective of using the Object-Oriented approach is to design a system that is modular, maintainable, and scalable. For the Personal Finance Management System:

* **Modularity**: The system is divided into smaller, manageable components (objects).
* **Maintainability**: The system is easy to update as changes are isolated to individual components.
* **Reusability**: Objects can be reused across different modules.
* **Scalability**: Objects and classes can be extended easily for future needs.

### **2. Components**

The following components will form the core of our object-oriented design for the PFMS:

1. **User**: An individual who interacts with the system.
2. **Transaction**: The financial record representing income or expense.
3. **Report**: The summary of transactions over a specified period.
4. **Category**: Classification of transactions (e.g., income, expense, groceries, etc.).
5. **System**: The overall finance management system responsible for coordinating user actions, processing transactions, and generating reports.

### **3. Use Case Diagram with Scenario**

The **Use Case Diagram** represents the interaction between the user and the system. Each use case corresponds to a task that a user can perform.

#### **Use Case Diagram**

plaintext

Copy code

+-------------------------------+

| Personal Finance |

| Management System |

+-------------------------------+

/ | \

/ | \

+------------------+ +----------------+ +----------------+

| Add Transaction | | View Report | | View Transaction|

+------------------+ +----------------+ +----------------+

\ | / /

\ | / /

+-------------------------------+

| |

+----+-------+----+

| External Entity: User |

+----------------------------+

#### **Scenario**

**Use Case: Add Transaction**

* **Actor**: User
* **Description**: The user adds a new transaction to the system.
* **Preconditions**: The user must be authenticated.
* **Postconditions**: The transaction is successfully stored in the database.
* **Basic Flow**:
  1. User navigates to the "Add Transaction" page.
  2. User enters the transaction details (e.g., amount, category).
  3. The system validates and stores the transaction.
  4. Confirmation is displayed to the user.

**Use Case: View Transactions**

* **Actor**: User
* **Description**: The user views their past transactions.
* **Basic Flow**:
  1. User navigates to the "View Transactions" page.
  2. System retrieves all the past transactions from the database.
  3. Transactions are displayed to the user.

### **4. Class Diagram**

The **Class Diagram** shows the structure of the system by representing classes and their relationships.

#### **Class Diagram for PFMS**

plaintext

Copy code

+-------------------+ +-----------------+

| User | | Transaction |

+-------------------+ +-----------------+

| - userId: int | | - transId: int |

| - name: String | | - amount: float |

| - email: String | | - date: Date |

+-------------------+ | - category: str |

| + login() | +-----------------+

| + addTransaction()| | + save() |

+-------------------+ +-----------------+

| |

v v

+------------------+ +--------------------+

| Report | | Category |

+------------------+ +--------------------+

| - reportId: int | | - categoryId: int |

| - dateRange: Date| | - name: String |

+------------------+ +--------------------+

| + generate() | | + getCategory() |

+------------------+ +--------------------+

#### Explanation:

* **User**: Represents the user who interacts with the system. The user can log in, add a transaction, or view reports.
* **Transaction**: Stores transaction details such as amount, date, and category.
* **Report**: Generates a report based on transaction data over a selected date range.
* **Category**: Defines the category of a transaction (e.g., groceries, entertainment).

### **5. State Diagram**

The **State Diagram** describes the states of an object and the transitions between those states.

#### **State Diagram for Transaction Object**

plaintext

Copy code

+---------------------+

| New Transaction |

+---------------------+

|

v

+---------------------+

| Transaction Validated|

+---------------------+

|

v

+---------------------+

| Transaction Saved |

+---------------------+

|

v

+---------------------+

| Transaction Closed |

+---------------------+

#### Explanation:

* **New Transaction**: A new transaction is created.
* **Transaction Validated**: The system checks the transaction details for validity.
* **Transaction Saved**: The transaction is stored in the database.
* **Transaction Closed**: The transaction is successfully completed.

### **6. Sequence Diagram**

The **Sequence Diagram** shows the interaction between objects over time.

#### **Sequence Diagram for Adding a Transaction**

plaintext

Copy code

User System Transaction

| | |

|-- addTransaction() -->| |

| |-- create() ------>|

| | |

| |-- save() -------->|

| |<--- confirmation--|

<---- success message ------|

#### Explanation:

1. The user initiates the addTransaction() process.
2. The system creates a new transaction object.
3. The transaction object is saved to the database.
4. The system sends a confirmation message back to the user.

### **7. Collaboration Diagram**

The **Collaboration Diagram** shows the interaction between objects, focusing on the messages passed between them.

#### **Collaboration Diagram for Generating Report**

plaintext

Copy code

User ----> System ----> TransactionDataStore

| |

|<----- Report<----|

#### Explanation:

* The user requests a report from the system.
* The system interacts with the TransactionDataStore to fetch the required data.
* The system then generates the report and sends it back to the user.

### **8. Activity Diagram**

The **Activity Diagram** represents the workflow of activities in the system.

#### **Activity Diagram for Adding a Transaction**

plaintext

Copy code

+-----------------------------------+

| Add Transaction Process |

+-----------------------------------+

| | |

Enter -> Validate -> Save -> Success

Transaction Details Transaction Transaction

#### Explanation:

1. The user enters transaction details.
2. The system validates the transaction data.
3. If the transaction is valid, it is saved to the database.
4. A success message is shown to the user.

### **9. Conclusion**

The Object-Oriented Design for the Personal Finance Management System includes:

* **Use Case Diagram**: Showcases interactions between the user and the system.
* **Class Diagram**: Defines the relationships and attributes of each object.
* **State Diagram**: Tracks the different states of transactions.
* **Sequence Diagram**: Illustrates the flow of events when adding a transaction.
* **Collaboration Diagram**: Shows object interactions during report generation.
* **Activity Diagram**: Outlines the process of adding a transaction.

### **Practical No - 8**

**Aim -** Defining Coding Standards and walk through.

### **1. Naming Conventions**

Using consistent and meaningful names for variables, functions, classes, and other elements is crucial for readability.

* **Variables**: Use descriptive names that clearly indicate their purpose. Prefer camelCase for variables and snake\_case for constants.
  + Example:

python

Copy code

totalExpenses = 0

CATEGORY\_INCOME = "Income"

* **Functions**: Use verbs that describe the action. Stick to camelCase for naming functions.
  + Example:

python

Copy code

def addTransaction(transaction):

pass

* **Classes**: Class names should follow PascalCase.
  + Example:

python

Copy code

class TransactionReport:

pass

### **2. Code Formatting**

Consistent code formatting makes the codebase easier to read and review. This includes the use of proper indentation, spacing, and line breaks.

* **Indentation**: Use 4 spaces per indentation level (Python default).
  + Example:

python

Copy code

def calculateTotalExpenses(transactions):

total = 0

for transaction in transactions:

total += transaction.amount

return total

* **Line Length**: Keep lines within 80 characters for better readability.
* **Blank Lines**: Use blank lines to separate logical sections of code, making it easier to understand the flow.

### **3. Commenting**

Good comments explain the **why** and **what** of the code. However, avoid over-commenting or stating the obvious.

* **Function Documentation**: Each function should have a docstring that explains its purpose, parameters, and return value.
  + Example:

python

Copy code

def generateReport(transactions):

"""

Generates a report for a list of transactions.

Args:

transactions (list): List of Transaction objects.

Returns:

report (dict): Summary report of total income and expenses.

"""

pass

* **Inline Comments**: Add comments where the logic might not be immediately obvious.
  + Example:

python

Copy code

# Adding up all expenses for the month

total\_expenses = sum([t.amount for t in transactions if t.type == 'expense'])

### **4. Error Handling**

Robust error handling prevents unexpected crashes and provides useful feedback.

* **Use try-except Blocks**: Catch exceptions to prevent the application from crashing.
  + Example:

python

Copy code

try:

totalExpenses = calculateTotalExpenses(transactions)

except ValueError as e:

print(f"Error calculating expenses: {e}")

* **Graceful Degradation**: If a function fails, it should return a meaningful error message instead of just breaking the application.
  + Example:

python

Copy code

def calculateTotalExpenses(transactions):

if not transactions:

return "No transactions available"

# Proceed with calculations

### **5. Modularization**

Breaking code into reusable and logical modules makes it easier to manage, test, and scale.

* **Small, Single Responsibility Functions**: Each function should do one thing and do it well.
  + Example:

python

Copy code

def addTransaction(amount, category, date):

transaction = Transaction(amount, category, date)

saveTransaction(transaction)

def saveTransaction(transaction):

# Logic to save transaction to database

* **Separate Concerns**: Keep your business logic (e.g., managing finances) separate from presentation logic (HTML templates). This can be achieved through frameworks like Flask.
  + Example:

python

Copy code

# Logic layer

def generateReport():

pass

# View layer (HTML template rendering)

return render\_template("report.html", report=generateReport())

### **6. Security Practices**

For any application dealing with financial data, security is paramount.

* **Input Validation**: Ensure that all user inputs are validated to prevent SQL injection, cross-site scripting (XSS), and other attacks.
  + Example:

python

Copy code

@app.route('/add\_transaction', methods=['POST'])

def add\_transaction():

# Validate input to avoid malicious data

if not validate\_transaction\_data(request.form):

return "Invalid data", 400

* **Sensitive Data Protection**: Never expose sensitive data (like passwords, account details) in logs or error messages.
  + Example:

python

Copy code

# Avoid displaying sensitive details in errors

try:

save\_transaction(transaction)

except Exception as e:

logging.error(f"Transaction save failed: {str(e)}")

return "Transaction failed", 500

## **Code Walkthrough: Adding a Transaction**

Here's a walkthrough of how the coding standards apply in the Personal Finance Management System.

### **Code Example**

python

Copy code

# Import necessary libraries

from flask import Flask, render\_template, request, redirect

# Define Flask app

app = Flask(\_\_name\_\_)

class Transaction:

"""

A class to represent a financial transaction.

"""

def \_\_init\_\_(self, amount, category, date):

self.amount = amount

self.category = category

self.date = date

# Global variable to store transactions

transactions = []

@app.route('/')

def index():

"""

Home page to show the list of transactions and option to add a new one.

"""

return render\_template('index.html', transactions=transactions)

@app.route('/add\_transaction', methods=['GET', 'POST'])

def add\_transaction():

"""

Handles the process of adding a new transaction.

GET: Renders the 'add\_transaction' form.

POST: Adds the transaction to the system and redirects to the home page.

"""

if request.method == 'POST':

# Retrieve form data

amount = request.form['amount']

category = request.form['category']

date = request.form['date']

# Input validation

if not amount or not category or not date:

return "All fields are required", 400

# Add the transaction to the list

transaction = Transaction(amount, category, date)

transactions.append(transaction)

return redirect('/')

return render\_template('add\_transaction.html')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

### **Explanation:**

1. **Naming Conventions**: The class Transaction and function names like add\_transaction() follow consistent naming standards.
2. **Commenting**: Each function has docstrings explaining its purpose, input, and output.
3. **Modularization**: The logic for handling transactions is separated into its own function (add\_transaction()), ensuring single-responsibility and better organization.
4. **Error Handling**: The POST request checks for valid data before processing the transaction. If any data is missing, it returns an error message with a 400 status code.
5. **Security**: User input is validated to ensure that all fields are filled out, preventing incorrect or harmful data from being processed.

### **Practical No - 9**

**Aim -** Write the test cases for the identified module.

#### **Objective**

The objective of writing test cases is to ensure that the Personal Finance Management System (PFMS) operates as intended, with accurate calculations, proper data handling, and correct functionality in different scenarios. Each test case will verify a specific aspect of the system, including the ability to add, manage, and report financial transactions.

The test cases for this module will cover:

1. Adding new transactions.
2. Generating reports (summary of income and expenses).
3. Validation of input data (correct formats and required fields).

### **Basics of Testing**

In software testing, the goal is to validate that the software meets the specified requirements and to ensure that it functions as expected under various conditions.

* **Test Case**: A set of inputs, execution conditions, and expected outcomes developed to verify the correctness of a software feature.
* **Expected Result**: The output or behavior the system should exhibit when the test case is executed.
* **Actual Result**: The output or behavior the system exhibits when the test case is executed, which will be compared against the expected result.

### **Goals of Testing**

1. **Ensure Correctness**: Verify that the system behaves as expected for all defined use cases.
2. **Prevent Bugs**: Catch potential issues early in development, preventing them from reaching production.
3. **Validate Business Logic**: Ensure that the core logic of adding, calculating, and reporting transactions works correctly.
4. **Enhance Security**: Ensure that invalid data cannot be submitted, protecting the system from malicious input.

### **Typical Tests for PFMS Module**

1. **Unit Testing**: Test individual functions (e.g., adding a transaction, calculating totals).
2. **Integration Testing**: Verify that different components work together (e.g., adding a transaction updates the report).
3. **System Testing**: Test the entire application for correctness under different inputs and scenarios.
4. **User Acceptance Testing (UAT)**: Ensure the application meets the end-user requirements and expectations.

### **Test Cases for Personal Finance Management System**

| **Test Case ID** | **Test Case Description** | **Preconditions** | **Test Steps** | **Expected Output** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- | --- | --- |
| TC-001 | Add a valid transaction to the system | System is running, form is accessible | 1. Navigate to the "Add Transaction" page.  2. Enter valid transaction details.  3. Click "Submit". | The transaction should be added and visible on the transactions list. |  |  |
| TC-002 | Attempt to add a transaction with missing required fields | System is running, form is accessible | 1. Navigate to the "Add Transaction" page.  2. Leave one or more required fields empty.  3. Click "Submit". | The system should display an error message and prevent the transaction from being added. |  |  |
| TC-003 | Calculate total income and expenses for the month | Transactions exist in the system | 1. View the transaction report for the current month.  2. Verify the calculations for income and expenses. | The report should display correct totals for income and expenses. |  |  |
| TC-004 | Input invalid data (e.g., negative transaction amount) | System is running, form is accessible | 1. Navigate to the "Add Transaction" page.  2. Enter a negative amount.  3. Click "Submit". | The system should display an error message indicating invalid input. |  |  |
| TC-005 | Check if transaction data persists across different sessions | Transaction exists in the system | 1. Add a transaction.  2. Close and reopen the application.  3. Check if the transaction is still listed. | The transaction should persist and still be visible after reopening the app. |  |  |
| TC-006 | View a detailed report for a selected date range | Transactions exist in the system | 1. Navigate to the "Reports" page.  2. Select a specific date range.  3. View the detailed report. | The system should display a report showing transactions within the selected range. |  |  |
| TC-007 | Test for concurrent access when two users add transactions | System allows multi-user access | 1. Open two sessions of the application.  2. Add different transactions from both sessions. | Both transactions should be added and visible in the combined report. |  |  |

### **Example Test Case Walkthrough**

#### **Test Case ID**: TC-001

**Test Case Name**: Add a Valid Transaction  
**Test Description**: This test ensures that a valid transaction is successfully added to the system and displayed in the list of transactions.  
**Preconditions**: The system is running, and the "Add Transaction" form is accessible.  
**Test Steps**:

1. Open the PFMS system.
2. Navigate to the "Add Transaction" page.
3. Fill in all required fields with valid data:
   * Amount: 500
   * Category: "Groceries"
   * Date: "2024-10-03"
4. Click the "Submit" button.

**Expected Output**: The system should redirect to the transactions page and display the newly added transaction with the correct details.

### **Experiment Demonstration**

**Objective**: The goal of the experiment is to validate the module by running test cases and comparing the **Actual Result** with the **Expected Output**. Any discrepancies should be addressed by fixing bugs or modifying the test case logic.

**Procedure**:

1. Setup the testing environment by ensuring the application is running in development mode.
2. Run each test case manually or using an automated testing framework like **unittest** or **pytest** in Python.
3. For each test, document the actual result and compare it with the expected result.
4. Track test status:
   * **Pass**: The actual result matches the expected result.
   * **Fail**: The actual result does not match the expected result, requiring further investigation.

**Example Code for Testing**: Using **pytest**, here is an example of how to test the functionality:

python

Copy code

import pytest

from app import app

def test\_add\_transaction():

client = app.test\_client()

# Simulate adding a transaction

response = client.post('/add\_transaction', data={

'amount': '500',

'category': 'Groceries',

'date': '2024-10-03'

})

# Check if the response is a redirect

assert response.status\_code == 302

# Check if the transaction is listed

response = client.get('/')

assert b'Groceries' in response.data

assert b'500' in response.data

**Results**: After running the tests, you will compile the outcomes (pass/fail) for each test case and record them in your project report.

### **Practical No - 10**

**Aim -** Demonstrate the use of different Testing Tools with comparison

#### **Objective**

The main objective of this demonstration is to explore various testing tools, their features, and why automated testing is important in software development. This report will compare popular testing tools, discuss the benefits of automated testing, and provide insights into the use of testing tools like **VS Code**, **TestComplete**, and others.

### **Why Automated Testing?**

Automated testing is essential in modern software development due to several reasons:

1. **Efficiency**: Automated tests can run repeatedly with minimal human intervention, reducing the time spent on manual testing.
2. **Accuracy**: Automated tests reduce human error, providing consistent and accurate results.
3. **Cost-effective**: Although initial setup may require time and resources, automated testing saves money in the long run due to fewer manual efforts.
4. **Scalability**: Automated tests can easily be scaled to cover large test suites, making it easier to test complex applications.
5. **Continuous Integration (CI)**: Automated tests are often integrated into CI pipelines, ensuring that new code does not break existing functionality.

### **Comparison of Testing Tools**

| **Tool** | **Description** | **Key Features** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- | --- |
| **VS Code (with pytest)** | A lightweight IDE with built-in support for running tests using extensions like pytest. | - Lightweight  - Debugging tools  - Extension support | - Open-source  - Easy to integrate  - Cost-effective | - Lacks out-of-the-box GUI for testing  - Limited support for complex testing |
| **TestComplete** | A commercial automated UI testing tool for web, desktop, and mobile applications. | - Supports multiple languages (JavaScript, Python)  - Cross-browser testing  - Scriptless testing available | - Powerful UI testing tool  - Visual Object recognition | - Expensive  - Requires high initial learning curve |
| **Selenium** | Open-source automated testing tool for web applications. | - Supports multiple programming languages  - Works across different browsers | - Free and open-source  - Flexible and customizable | - Requires more effort to set up  - Cannot handle non-browser applications |
| **JUnit** | A unit testing framework for Java applications. | - Integration with CI tools  - Built-in assertions | - Works seamlessly with Java projects  - Easy to use | - Java-specific  - Limited functionality beyond unit tests |
| **Cucumber** | A testing tool that supports Behavior-Driven Development (BDD). | - Gherkin language for writing test cases  - Focuses on collaboration | - Great for BDD  - Easy for non-technical users | - Requires good knowledge of Gherkin  - Not suited for non-BDD projects |

### **Why TestComplete is Good**

**TestComplete** is considered one of the best tools for automated testing, especially for UI testing of desktop, web, and mobile applications. Its strengths include:

* **Cross-platform support**: TestComplete allows for testing on multiple platforms, including Windows, macOS, iOS, and Android.
* **Ease of use**: It provides a **scriptless option** for beginners, making it easier for users who are not proficient in coding to create automated tests.
* **AI-powered object recognition**: TestComplete can recognize dynamic user interface elements, which is useful for testing applications with complex UI structures.
* **Integration with CI/CD**: TestComplete can easily be integrated into continuous integration and deployment pipelines, providing seamless automated testing in a CI environment.

### **Where’s the Catch?**

While **TestComplete** is a great tool, it does come with its drawbacks:

1. **Cost**: It’s a commercial tool, and licenses are expensive, making it less accessible for startups or small teams.
2. **Learning Curve**: Although it provides a scriptless testing option, mastering the full capabilities of TestComplete requires a steep learning curve, especially for scripting in multiple languages like Python or JavaScript.
3. **Limited Open-source Community**: Unlike Selenium or pytest, TestComplete does not have a large open-source community, which means fewer resources and online support.

### **Experiment Demonstration**

#### **Objective**

To demonstrate how testing tools like **VS Code** (using pytest) and **TestComplete** can be used in automated testing for the Personal Finance Management System (PFMS).

#### **Procedure**

1. **VS Code with pytest**:
   * Install **pytest** using pip install pytest.
   * Write unit test cases for the PFMS system.
   * Run the test cases in VS Code using the pytest extension.
   * View test results in the VS Code terminal.

Example test case:

python

Copy code

def test\_add\_transaction():

response = add\_transaction('Groceries', 100, '2024-10-01')

assert response == 'Transaction added successfully'

1. **TestComplete**:
   * Create a new project in TestComplete.
   * Set up a test for the PFMS web interface by recording a transaction entry.
   * Use **AI-powered visual recognition** to identify buttons, input fields, and messages.
   * Execute the test and observe the result.
   * Automate UI interactions and validate that the correct data is displayed in the transaction list.

### **Practical No - 11**

**Aim -** Define security and quality aspects of the identified module

#### **1. Security Aspects**

Ensuring the security of the Personal Finance Management System is crucial, as it handles sensitive financial information. Here are the key security aspects to consider:

**a. Data Encryption**

* **Description**: All sensitive data, including user credentials and financial transactions, should be encrypted both in transit and at rest.
* **Implementation**: Use protocols like TLS (Transport Layer Security) for data in transit and AES (Advanced Encryption Standard) for data at rest.

**b. User Authentication and Authorization**

* **Description**: Implement a robust authentication mechanism to ensure that only authorized users can access their accounts.
* **Implementation**: Use multi-factor authentication (MFA) to enhance security and enforce strong password policies.

**c. Input Validation**

* **Description**: Validate all user inputs to prevent injection attacks (e.g., SQL injection, cross-site scripting).
* **Implementation**: Use whitelisting techniques to ensure only valid inputs are processed and reject any harmful inputs.

**d. Session Management**

* **Description**: Properly manage user sessions to prevent session hijacking or fixation attacks.
* **Implementation**: Implement session timeouts, regenerate session IDs upon login, and use secure cookies.

**e. Secure API Access**

* **Description**: If the PFMS exposes APIs, ensure they are secure and protected against unauthorized access.
* **Implementation**: Use OAuth or API tokens for authentication and validate incoming requests to prevent unauthorized access.

**f. Regular Security Audits**

* **Description**: Conduct regular security audits and vulnerability assessments to identify and mitigate potential threats.
* **Implementation**: Use automated tools for scanning vulnerabilities and manual penetration testing to uncover weaknesses.

#### **2. Quality Aspects**

Quality in software development ensures that the system meets user requirements and performs well. Here are the key quality aspects to focus on:

**a. Functional Quality**

* **Description**: The system must meet all specified functional requirements, providing accurate and reliable financial data handling.
* **Implementation**: Conduct thorough functional testing, including unit tests, integration tests, and system tests.

**b. Performance**

* **Description**: The PFMS should perform efficiently under varying loads, handling multiple users and transactions seamlessly.
* **Implementation**: Use load testing tools to simulate high traffic and assess response times, resource usage, and scalability.

**c. Usability**

* **Description**: The user interface must be intuitive and user-friendly, allowing users to navigate the system effortlessly.
* **Implementation**: Conduct usability testing with real users to gather feedback on the interface design and improve user experience.

**d. Reliability**

* **Description**: The system should be dependable, with minimal downtime and error-free operation.
* **Implementation**: Implement logging and monitoring to detect issues promptly and ensure that the system is resilient to failures.

**e. Maintainability**

* **Description**: The PFMS codebase should be easy to maintain and update, with clear documentation and modular design.
* **Implementation**: Follow coding standards and best practices, and provide comprehensive documentation for future developers.

**f. Security Testing**

* **Description**: Regularly test the system for security vulnerabilities to ensure that it is secure against potential threats.
* **Implementation**: Conduct security testing techniques, such as penetration testing and static code analysis, to identify and fix vulnerabilities.