SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 1st semester
SOFTWARE ENGINEERING (PCA20D03J)

Lab Manual

Lab 1: Identifying Project Objective and Scope

Aim: To understand and define the core objectives and boundaries of a software project.

Procedure:

1. **Understand the Problem:** Begin by thoroughly understanding the problem that the software aims to solve. This involves discussions with stakeholders, reviewing existing systems, and identifying pain points.
2. **Define Project Objectives:** Clearly state what the software project intends to achieve. Objectives should be SMART (Specific, Measurable, Achievable, Relevant, Time-bound).
3. **Identify Stakeholders:** List all individuals or groups who will be affected by or have an interest in the project (e.g., users, clients, development team, management).
4. **Define Project Scope (In-Scope):** List all functionalities, features, and components that *will* be part of the software system. Be as detailed as possible.
5. **Define Project Scope (Out-of-Scope):** List all functionalities, features, and components that *will not* be part of the software system. This is crucial to prevent scope creep.
6. **Document Assumptions and Constraints:** Note any assumptions made during the definition phase and any limitations or restrictions (e.g., budget, technology, time).
7. **Review and Validate:** Present the defined objectives and scope to stakeholders for review and obtain their agreement and sign-off.

Source Code: *This lab is conceptual and does not involve writing source code. The output is documentation.*

Input:

- Initial problem description (e.g., "Develop a system to manage library books").
- Stakeholder interviews/requirements.

Expected Output:

- A clear document outlining:
    - Project Title: Library Management System
    - Project Objectives:
        - To automate book cataloging by [Date].
        - To enable online book search for users.
        - To track book borrowings and returns efficiently.
    - In-Scope Features: Book cataloging, user registration, search functionality, borrowing/returning, overdue notifications.

- Out-of-Scope Features: Payment gateway integration, advanced analytics, mobile application.
- Assumptions: Users have internet access.
- Constraints: Project must be completed within 6 months.

Lab 2: Selection of Suitable Software Process Model for the Suggested System

Aim: To analyze different software process models and select the most appropriate one for a given software project.

Procedure:

1. **Understand Project Characteristics:** Analyze the project's size, complexity, clarity of requirements, risk level, team experience, and client involvement.
2. **Research Software Process Models:** Review common models such as Waterfall, Agile (Scrum, Kanban), Spiral, V-Model, Incremental, and Prototyping. Understand their strengths, weaknesses, and applicability.
3. **Evaluate Models Against Project Characteristics:** For each relevant model, assess how well it aligns with the project's specific attributes.
    - *Waterfall:* Suitable for well-understood, stable requirements.
    - *Agile:* Suitable for evolving requirements, high client involvement, iterative development.
    - *Spiral:* Suitable for high-risk projects, large-scale systems.
4. **Consider Organizational Context:** Take into account the organization's culture, existing processes, and available resources.
5. **Propose and Justify Selection:** Based on the evaluation, select the most suitable model and provide a clear justification for the choice, highlighting how it mitigates risks and optimizes development.
6. **Document the Decision:** Record the chosen model and the rationale behind it.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a decision document.*

Input:

- Project objective and scope document (from Lab 1).
- Information on team size, client availability, and technology stack.

Expected Output:

- A document detailing:
    - Project Name: [e.g., Library Management System]
    - Project Characteristics Summary: (e.g., Requirements are somewhat stable but may evolve, medium-sized project, client wants regular feedback).
    - Analysis of Candidate Models: (e.g., Waterfall - less suitable due to potential requirement changes; Agile - suitable due to iterative nature and client feedback).
    - Selected Process Model: Agile (Scrum).
    - Justification: Scrum allows for flexibility in requirements, promotes continuous client feedback through sprints, and enables early delivery of working software, which aligns with the client's desire for visibility and adaptability.

Lab 3: Problem Statement Preparation

Aim: To articulate a clear, concise, and comprehensive problem statement for a software project.

Procedure:

1. **Identify the Core Problem:** What is the fundamental issue or need that the software aims to address?
2. **Define the Current Situation:** Describe the existing state, including any inefficiencies, challenges, or gaps.
3. **Identify the Impact:** Explain the negative consequences of the problem if it remains unsolved (e.g., financial loss, decreased productivity, user dissatisfaction).
4. **State the Desired Outcome:** Describe the ideal future state once the software solution is implemented.
5. **Formulate the Problem Statement:** Combine the above elements into a concise paragraph or two. A good problem statement often follows a structure like: "The problem is [current situation]. This leads to [impact]. A solution is needed to [desired outcome]."
6. **Refine and Review:** Ensure the statement is unambiguous, specific, and focuses on the problem, not the solution.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a written problem statement.*

Input:

- Initial project idea or client request.
- Discussions and observations about the existing system or process.

Expected Output:

- **Problem Statement Example (for Library Management System):** "Currently, the university library manages its book cataloging, borrowing, and returning processes manually, relying on physical registers and spreadsheets. This manual system leads to frequent errors in record-keeping, difficulty in tracking overdue books, and a time-consuming search process for users. As a result, librarians spend excessive time on administrative tasks, and users experience frustration due to delays and inaccurate information. A robust, automated system is needed to streamline library operations, improve data accuracy, and enhance the overall user experience by providing efficient book management and search capabilities."

Lab 4: Project Planning

Aim: To develop a comprehensive project plan, including tasks, timelines, resources, and risk management.

Procedure:

1. **Define Project Scope (Revisit):** Confirm the scope established in Lab 1.
2. **Break Down Work (WBS):** Create a Work Breakdown Structure (WBS) by decomposing the project into smaller, manageable tasks and sub-tasks.
3. **Estimate Effort and Duration:** For each task, estimate the effort (person-hours/days) required and its duration.
4. **Identify Dependencies:** Determine which tasks must be completed before others can start.
5. **Allocate Resources:** Assign team members and other resources (e.g., tools, hardware) to tasks.
6. **Develop Schedule:** Create a project schedule using tools like Gantt charts or network diagrams, incorporating dependencies and durations.
7. **Identify Risks:** Brainstorm potential risks (technical, operational, financial, schedule) that could impact the project.
8. **Plan Risk Mitigation:** For each identified risk, devise strategies to prevent, reduce, or respond to it.
9. **Define Quality Assurance Activities:** Plan how quality will be ensured throughout the project lifecycle.
10. **Establish Communication Plan:** Define how and when stakeholders will be updated on project progress.
11. **Document the Plan:** Compile all planning elements into a formal project plan document.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a project plan document.*

Input:

- Project scope and objectives.
- Team availability and skill sets.
- Organizational policies and procedures.

Expected Output:

- A Project Plan document including:
    - **Project Schedule:** (e.g., Gantt chart snippet showing phases like Requirements Gathering, Design, Development, Testing, Deployment with timelines).
    - **Resource Allocation:** (e.g., Developer A assigned to Module X, Tester B assigned to Module Y).
    - **Risk Register:**
        - Risk: Requirements change frequently. Mitigation: Implement Agile methodology with regular client feedback.
        - Risk: Key developer leaves. Mitigation: Cross-training, comprehensive documentation.
    - **Communication Plan:** Weekly team meetings, bi-weekly stakeholder updates.

Lab 5: Performing Various Requirement Analysis

Aim: To gather, analyze, and validate functional and non-functional requirements for a software system.

Procedure:

1. **Elicitation Techniques:**
    - **Interviews:** Conduct structured interviews with stakeholders to gather their needs.
    - **Surveys/Questionnaires:** Distribute surveys to a wider audience for quantitative data.
    - **Workshops (JAD/RAD):** Facilitate collaborative sessions to define requirements.
    - **Prototyping:** Develop mock-ups or prototypes to get early feedback.
    - **Observation:** Observe users performing tasks in their environment.
    - **Document Analysis:** Review existing documentation (manuals, reports, system specifications).
2. **Categorize Requirements:** Classify requirements into functional (what the system *does*) and non-functional (qualities like performance, security, usability).
3. **Analyze and Prioritize:**
    - **Conflict Resolution:** Identify and resolve conflicting requirements.
    - **Feasibility Study:** Assess the technical and economic feasibility of implementing requirements.
    - **Prioritization:** Rank requirements based on importance and urgency (e.g., MoSCoW: Must, Should, Could, Won't).
4. **Specify Requirements:** Document requirements clearly, unambiguously, and verifiably. Use techniques like Use Cases, User Stories, or detailed functional specifications.
5. **Validate Requirements:** Review requirements with stakeholders to ensure they are complete, consistent, and accurately reflect their needs.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a collection of analyzed requirements.*

Input:

- Problem statement.
- Stakeholder list.
- Existing system documentation.

Expected Output:

- **Functional Requirements (Examples):**
    - The system shall allow users to search for books by title, author, or ISBN.
    - The system shall allow librarians to add new books to the catalog.
    - The system shall record book borrowing and return dates.
- **Non-Functional Requirements (Examples):**
    - **Performance:** The system shall respond to search queries within 2 seconds for up to 10,000 books.
    - **Security:** User authentication shall be required for all library staff.
    - **Usability:** The user interface shall be intuitive and easy to navigate for first-time users.
- **Prioritized List:**
    - Must Have: Search, Add Book, Borrow/Return.

- Should Have: Overdue notifications.
- Could Have: User reviews.

Lab 6: Develop Software Requirement Specification Sheet (SRS)

Aim: To create a formal Software Requirements Specification (SRS) document that comprehensively describes the software system's functional and non-functional requirements.

Procedure:

1. **Standard Structure:** Follow a recognized SRS standard (e.g., IEEE 830) or a defined organizational template. Common sections include:
    - Introduction (Purpose, Scope, Definitions)
    - Overall Description (Product Perspective, User Characteristics, Constraints, Assumptions)
    - Specific Requirements (Functional Requirements, Non-Functional Requirements, External Interface Requirements)
    - Appendices (Glossary, Index)
2. **Detail Functional Requirements:** For each functional requirement, provide a clear description, inputs, outputs, pre-conditions, post-conditions, and error handling. Use cases or user stories can be detailed here.
3. **Specify Non-Functional Requirements:** Quantify non-functional requirements where possible (e.g., performance in seconds, security levels, availability uptime).
4. **Define External Interfaces:** Describe how the system interacts with other systems, hardware, or users (e.g., APIs, user interface components).
5. **Review and Iterate:** Conduct internal reviews with the development team and external reviews with stakeholders to ensure accuracy, completeness, and clarity.
6. **Version Control:** Maintain different versions of the SRS as requirements evolve.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a formal SRS document.*

Input:

- Analyzed requirements from Lab 5.
- Project scope and objectives.

Expected Output:

- A multi-page SRS document for the chosen project (e.g., Library Management System), containing:
    - **Section 1: Introduction**
        - 1.1 Purpose: To define requirements for the Library Management System.
        - 1.2 Scope: Covers book management, user management, borrowing/returning.
    - **Section 2: Overall Description**
        - 2.1 Product Perspective: Standalone web application.
        - 2.2 User Characteristics: Librarians, Students, Faculty.
    - **Section 3: Specific Requirements**
        - 3.1 Functional Requirements:
            - FR1: Search Books (details: inputs, outputs, pre/post-conditions).
            - FR2: Add New Book (details).
        - 3.2 Non-Functional Requirements:

        - NFR1: Performance (e.g., 90% of searches complete in < 2s).
        - NFR2: Security (e.g., Role-based access control).
    - 3.3 External Interface Requirements:
        - User Interface: Web-based, responsive design.
        - Software Interfaces: Integration with university LDAP for user authentication (if applicable).

Lab 7: Draw Function Oriented Diagram

Aim: To represent the functional decomposition of a system using diagrams like Data Flow Diagrams (DFDs) or Structure Charts.

Procedure:

1. **Understand the System's Processes:** Identify the main functions or processes the system performs.
2. **Identify Data Flows:** Determine what data moves between these processes, external entities, and data stores.
3. **Draw Context Diagram (Level 0 DFD):** Represent the entire system as a single process, showing its interactions with external entities.
4. **Draw Level 1 DFD:** Decompose the main process into its major sub-processes, showing data flows between them, external entities, and data stores.
5. **Draw Lower Level DFDs (if necessary):** Further decompose complex sub-processes into more detailed processes until the desired level of granularity is reached.
6. **Draw Structure Chart (Alternative/Complementary):** If focusing on module structure, create a structure chart showing the hierarchical arrangement of program modules and the data/control passing between them.
7. **Review and Refine:** Ensure the diagrams are consistent, complete, and accurately reflect the system's functionality.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a set of diagrams.*

Input:

- SRS document (especially functional requirements).
- Understanding of system processes.

Expected Output:

- **Data Flow Diagram (DFD) Example (for Library Management System):**
  - **Context Diagram (Level 0):**
    - External Entities: User, Librarian.
    - Process: Library Management System.
    - Data Flows: Search Query (User to System), Search Results (System to User), Book Details (Librarian to System), Book Record (System to Librarian).
  - **Level 1 DFD:**
    - Processes: User Management, Book Management, Borrowing/Returning.
    - Data Stores: User Database, Book Catalog, Borrowing Records.
    - Data Flows between processes and data stores.
- **Structure Chart Example:**
  - Hierarchical representation of modules like "Main Library System" calling "Manage Books," "Manage Users," and "Process Transactions," with data couples and control flags shown.

Lab 8: User's View Analysis

Aim: To analyze the system from the perspective of different user roles, understanding their goals, tasks, and interactions.

Procedure:

1. **Identify User Roles/Personas:** Determine the different types of users who will interact with the system (e.g., student, faculty, librarian, administrator). Create personas for each.
2. **Define User Goals:** For each user role, identify their primary goals when using the system.
3. **Map User Tasks:** Break down each goal into specific tasks the user will perform.
4. **Create Use Cases/User Stories:**
    - **Use Case:** Describe a sequence of actions between an actor (user) and the system to achieve a specific goal. Include pre-conditions, post-conditions, and alternative flows.
    - **User Story:** A short, simple description of a feature told from the perspective of the user, stating "As a [type of user], I want [some goal] so that [some reason]."
5. **Develop Scenarios:** Create specific instances of use cases or user stories to illustrate typical interactions.
6. **Analyze User Interface Needs:** Based on tasks and goals, identify the necessary UI elements and workflows.
7. **Validate with Users:** Present the analysis to actual users or representatives to ensure accuracy and completeness.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a set of use cases, user stories, or user scenarios.*

Input:

- SRS document.
- Stakeholder interviews.

Expected Output:

- **User Roles/Personas:**
    - **Student User:** Goal: Find and borrow books. Tasks: Search, View Details, Request Borrow.
    - **Librarian User:** Goal: Manage library inventory. Tasks: Add Book, Process Borrow/Return, Manage Users.
- **Use Case Example:**
    - **Use Case:** Borrow a Book
    - **Actor:** Student User
    - **Pre-conditions:** Student is registered, book is available.
    - **Main Flow:**
        1. Student searches for a book.
        2. System displays search results.
        3. Student selects a book and requests to borrow.
        4. System checks availability.
        5. System records borrowing event.
        6. System updates book status to 'Borrowed'.
    - **Alternative Flow:** Book not available, Student not registered.
- **User Story Example:**

- As a **Student**, I want to **search for books by author** so that I can **find all books by my favorite writer easily**.
- As a **Librarian**, I want to **add new books to the catalog** so that **the library inventory is always up-to-date**.

Lab 9: Structure View Diagram

Aim: To represent the static structure of a software system, often using UML Class Diagrams or Component Diagrams.

Procedure:

1. **Identify Key Entities/Classes:** From the requirements, identify the main concepts or objects in the system (e.g., Book, User, Loan, Librarian).
2. **Define Attributes:** For each entity/class, determine its properties or attributes (e.g., for Book: title, author, ISBN, publication year).
3. **Define Operations/Methods:** Identify the behaviors or functions associated with each entity/class (e.g., for Book: `borrow()`, `return()`, `updateStatus()`).
4. **Identify Relationships:** Determine how classes interact with each other (e.g., association, aggregation, composition, inheritance).
    - **Association:** A general relationship between two classes.
    - **Aggregation:** A "has-a" relationship where one class is part of another, but can exist independently.
    - **Composition:** A "has-a" relationship where the part cannot exist without the whole.
    - **Inheritance:** An "is-a" relationship where one class derives from another.
5. **Draw Class Diagram:** Use UML notation to draw the classes, their attributes, operations, and relationships.
6. **Draw Component Diagram (if applicable):** If focusing on the physical structure, illustrate the system's components and their dependencies.
7. **Review and Refine:** Ensure the diagram accurately reflects the system's structure and is consistent with other design artifacts.

Source Code: *This lab is conceptual and does not involve writing source code. The output is a set of diagrams.*

Input:

- SRS document.
- User's view analysis.

Expected Output:

- **UML Class Diagram Example (for Library Management System):**
    - **Classes:**
        - `Book` (attributes: `title`, `author`, `ISBN`, `status`; operations: `borrow()`, `return()`)
        - `User` (attributes: `userID`, `name`, `contactInfo`; operations: `login()`, `logout()`)
        - `Librarian` (inherits from `User`; operations: `addBook()`, `registerUser()`)
        - `Loan` (attributes: `loanID`, `loanDate`, `dueDate`, `returnDate`; operations: `createLoan()`, `closeLoan()`)
    - **Relationships:**
        - `User` has many `Loan`s (one-to-many association).
        - `Book` has many `Loan`s (one-to-many association).
        - `Librarian` is a `User` (inheritance).
- **UML Component Diagram Example:**

- Components like "User Interface," "Business Logic," "Database Access Layer," and "External Services" showing their interfaces and dependencies.

Lab 10: Test Case Design for Unit Testing

Aim: To design effective test cases for individual software units (e.g., functions, methods, classes) to ensure they work correctly in isolation.

Procedure:

1. **Identify Unit Under Test (UUT):** Select a specific function, method, or class to test.
2. **Understand UUT Functionality:** Review the requirements and design specifications for the UUT.
3. **Identify Inputs and Outputs:** Determine all possible inputs to the UUT and their expected outputs.
4. **Apply Test Case Design Techniques:**
    - **Equivalence Partitioning:** Divide input data into partitions where all values in a partition are expected to be processed in the same way. Select one representative value from each partition.
    - **Boundary Value Analysis:** Test values at the boundaries of equivalence partitions (e.g., minimum, maximum, just inside/outside the boundary).
    - **Decision Table Testing:** For complex logic with multiple conditions and actions, create a table mapping combinations of conditions to actions.
    - **State Transition Testing:** For units with different states, design tests to cover all states and transitions.
    - **Error Guessing:** Use intuition and experience to guess potential error-prone inputs.
5. **Define Test Steps:** Outline the precise steps to execute the test case.
6. **Specify Expected Results:** Clearly state what the UUT should output or what state it should be in after execution.
7. **Create Test Data:** Prepare any necessary input data for the test case.
8. **Document Test Cases:** Record all details in a structured format.

Source Code: *This lab involves designing test cases, not writing the code itself. The "Source Code" section here refers to the conceptual code of the UUT.*

Input:

- Function/method specification (e.g., `calculateDiscount(price, quantity)`).
- Requirements for the unit.

Expected Output:

- **Test Cases for `calculateDiscount(price, quantity)`:**
    - **UUT:** `calculateDiscount` function
    - **Description:** Calculates discount based on price and quantity. Discount is 10% for quantity >= 10, else 0%.
    - **Test Case 1 (Equivalence Partitioning - Valid):**
        - **Input:** `price = 100`, `quantity = 5`
        - **Expected Output:** `discount = 0`
    - **Test Case 2 (Equivalence Partitioning - Valid):**
        - **Input:** `price = 100`, `quantity = 15`
        - **Expected Output:** `discount = 10`
    - **Test Case 3 (Boundary Value Analysis - Lower Boundary):**
        - **Input:** `price = 100`, `quantity = 9`

        - **Expected Output:** `discount = 0`
    - **Test Case 4 (Boundary Value Analysis - Upper Boundary):**
        - **Input:** `price = 100`, `quantity = 10`
        - **Expected Output:** `discount = 10`
    - **Test Case 5 (Error Guessing - Invalid Input):**
        - **Input:** `price = -50`, `quantity = 10`
        - **Expected Output:** Error/Exception (e.g., "Invalid price")

Lab 11: Test Case Design for Integration Testing

Aim: To design test cases that verify the interactions and interfaces between integrated software modules or components.

Procedure:

1. **Identify Integration Strategy:** Determine the approach for integrating modules (e.g., Big Bang, Top-Down, Bottom-Up, Sandwich).
2. **Identify Interfaces:** For the selected modules, identify all points where they interact or exchange data.
3. **Define Integration Scenarios:** Based on the interfaces, define scenarios that involve the combined functionality of the integrated modules.
4. **Apply Test Case Design Techniques:**
    - **Interface Testing:** Focus on data passing, parameter lists, and return values between modules.
    - **Functional Decomposition:** Test the integrated functionality from an end-to-end perspective.
    - **Path Testing:** Ensure critical paths through the integrated modules are exercised.
    - **Error Handling:** Test how the integrated system handles errors or invalid data passed between modules.
5. **Define Test Steps:** Outline the sequence of actions required to execute the integration test.
6. **Specify Expected Results:** Clearly state the expected behavior of the integrated system.
7. **Create Test Data:** Prepare data that simulates real-world interactions between modules.
8. **Document Test Cases:** Record all details in a structured format.

Source Code: *This lab involves designing test cases for integrated modules. The "Source Code" section here refers to the conceptual interaction between modules.*

Input:

- Design documents (e.g., DFDs, structure charts, class diagrams).
- Unit-tested modules.

Expected Output:

- **Test Cases for Library Management System Integration:**
    - **Modules to Integrate:** User Management Module, Book Management Module, Borrowing/Returning Module.
    - **Integration Strategy:** Top-Down.
    - **Test Case 1 (User Registration and Book Borrowing):**
        - **Description:** Verify that a newly registered user can successfully borrow an available book.
        - **Test Steps:**
            1. Register a new user (via User Management Module).
            2. Log in as the new user.
            3. Search for an available book (via Book Management Module).
            4. Initiate borrowing the book (via Borrowing/Returning Module).
        - **Expected Result:** User is registered, book status changes to 'Borrowed', borrowing record is created, and user receives confirmation.
    - **Test Case 2 (Invalid Book ID during Borrowing):**

- **Description:** Verify system handles an attempt to borrow a non-existent book.
- **Test Steps:**
    1. Log in as a valid user.
    2. Attempt to borrow a book with an invalid/non-existent Book ID.
- **Expected Result:** System displays an error message "Book not found" and does not create a borrowing record.

Lab 12: Performing Testing and Debugging for a Sample Code

Aim: To execute test cases on a sample code, identify defects, and debug the code to fix the issues.

Procedure:

1. **Understand Sample Code:** Review the provided sample code and its intended functionality.
2. **Design Test Cases:** Based on the code's logic, design unit test cases (as learned in Lab 10) to cover various scenarios, including valid inputs, invalid inputs, and edge cases.
3. **Execute Test Cases:** Run the sample code with the designed test inputs.
4. **Record Test Results:** Document the actual output for each test case and compare it with the expected output.
5. **Identify Defects:** If actual output differs from expected output, a defect is identified.
6. **Debug the Code:**
    - **Reproduce the Defect:** Ensure the defect can be consistently reproduced.
    - **Isolate the Problem:** Use debugging tools (e.g., print statements, IDE debuggers with breakpoints, step-through execution) to pinpoint the exact location in the code where the error occurs.
    - **Analyze the Cause:** Understand why the error is happening (e.g., incorrect logic, off-by-one error, wrong variable assignment).
    - **Propose a Fix:** Determine the necessary code changes to resolve the defect.
7. **Implement Fix:** Apply the proposed changes to the code.
8. **Retest:** Rerun the failing test cases to confirm the fix. Also, perform regression testing (rerun previously passed test cases) to ensure the fix hasn't introduced new defects.

Source Code: *This lab involves actual coding and testing.*

Input:

- A sample code snippet with potential bugs.

Expected Output:

- **Sample Code (Python example with a bug):**

```
# buggy_calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    # Bug: Does not handle division by zero
    return a / b

def power(base, exp):
    # Bug: Incorrect calculation for power
    result = 1
    for _ in range(exp):
```

```
        result *= base
    return result
```

- **Test Cases and Results (before debugging):**
  - **Test Case:** `divide(10, 0)`
    - **Expected Output:** Error/Exception (e.g., `ZeroDivisionError`)
    - **Actual Output:** `ZeroDivisionError` (Identified as a missing explicit check/handling, but Python raises it).
  - **Test Case:** `power(2, 3)`
    - **Expected Output:** `8`
    - **Actual Output:** `8` (This specific test case passes, but the `power` function has a bug for `exp=0` or negative `exp`).
  - **Test Case:** `power(5, 0)`
    - **Expected Output:** `1`
    - **Actual Output:** `1` (Still passes due to `range(0)` being empty, `result` remains `1`).
  - **Test Case:** `power(2, -2)`
    - **Expected Output:** Error/Specific behavior for negative exponent
    - **Actual Output:** `1` (Incorrect, loop `range(-2)` is empty).
- **Sample Code (Python example after debugging):**

```
# fixed_calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero") # Fixed: Added zero division handling
    return a / b

def power(base, exp):
    # Fixed: Corrected calculation for power, especially for exp=0 and negative exp
    if exp < 0:
        return 1 / power(base, -exp) # Handle negative exponent recursively
    elif exp == 0:
        return 1
    else:
        result = 1
        for _ in range(exp):
            result *= base
        return result
```

- **Test Cases and Results (after debugging):**
  - **Test Case:** `divide(10, 0)`
    - **Expected Output:** `ValueError("Cannot divide by zero")`
    - **Actual Output:** `ValueError("Cannot divide by zero")` (Pass)
  - **Test Case:** `power(5, 0)`

        - **Expected Output:** `1`
        - **Actual Output:** `1` (Pass)
    - **Test Case:** `power(2, -2)`
        - **Expected Output:** `0.25`
        - **Actual Output:** `0.25` (Pass)

Lab 13: Preparation of Timeline Charts and Tracking the Scheduling

Aim: To create and manage project schedules using timeline charts (e.g., Gantt charts) and to track progress against the planned schedule.

Procedure:

1. **Review WBS and Dependencies:** Use the Work Breakdown Structure and task dependencies defined in Lab 4.
2. **Select a Tool:** Choose a suitable project management tool (e.g., Microsoft Project, Jira, Asana, or even a spreadsheet for simple projects) to create the Gantt chart.
3. **Input Tasks and Durations:** Enter all tasks, their estimated durations, and assigned resources into the tool.
4. **Define Dependencies:** Link tasks according to their dependencies (e.g., Finish-to-Start).
5. **Set Milestones:** Identify key project milestones and mark them on the chart.
6. **Establish Baseline:** Once the initial plan is complete, save it as a baseline for future comparison.
7. **Track Progress:**
    - **Regular Updates:** Periodically update the chart with actual progress (e.g., percentage complete, actual start/end dates).
    - **Identify Variances:** Compare actual progress against the baseline to identify schedule variances (tasks ahead or behind schedule).
    - **Critical Path Analysis:** Identify the critical path – the sequence of tasks that determines the shortest possible project duration. Delays on the critical path directly impact the project end date.
8. **Adjust Schedule (if necessary):** If significant variances occur, analyze the impact and make necessary adjustments to the plan (e.g., reallocate resources, re-prioritize tasks).
9. **Communicate Status:** Regularly report on schedule status to stakeholders.

Source Code: *This lab is conceptual and involves using project management tools, not writing source code. The output is a visual chart and status reports.*

Input:

- Project plan (WBS, task estimates, dependencies) from Lab 4.
- Actual progress updates from the development team.

Expected Output:

- **Gantt Chart Snippet:**
    - A visual representation showing tasks, their durations, dependencies, and a timeline.
    - Example:
        - Task A: [-----] (Planned)
        - Task B: [-----] (Planned)
        - Task A: [==---] (Actual, 50% complete)
        - Task B: [-----] (Not started yet)
- **Schedule Status Report:**
    - Project Name: Library Management System
    - Reporting Period: Week 5
    - Overall Status: On Track / Slightly Delayed

- Key Milestones Achieved: Requirements Finalized
- Tasks Behind Schedule: UI Design (2 days delay)
- Tasks Ahead of Schedule: Database Schema Design (1 day ahead)
- Next Steps: Focus on completing UI Design.

Lab 14: Estimation of Effort and Risk Identification

Aim: To estimate the effort required for a software project and to systematically identify and categorize potential risks.

Procedure:

1. **Effort Estimation:**
    - **Decomposition Technique:** Break down the project into smaller components (e.g., modules, features) and estimate effort for each. Sum them up.
    - **Analogy:** Compare the current project to similar past projects to estimate effort.
    - **Expert Judgment:** Consult experienced team members or subject matter experts.
    - **Parametric Models:** Use formulas or models (e.g., COCOMO) that use project characteristics (e.g., lines of code, function points) to estimate effort.
    - **Three-Point Estimation (PERT):** For each task, estimate Optimistic (O), Most Likely (M), and Pessimistic (P) durations. Calculate Expected Duration = (O + 4M + P) / 6.
    - **Document Assumptions:** Clearly state any assumptions made during estimation.
2. **Risk Identification:**
    - **Brainstorming:** Team members collectively identify potential risks.
    - **Checklists:** Use predefined lists of common software risks.
    - **Interviews:** Talk to stakeholders and experts about potential issues.
    - **SWOT Analysis:** Analyze Strengths, Weaknesses, Opportunities, and Threats.
    - **Root Cause Analysis:** Investigate past failures to identify underlying risks.
3. **Risk Categorization:** Group identified risks (e.g., technical, schedule, budget, operational, personnel).
4. **Risk Analysis (Qualitative/Quantitative):**
    - **Qualitative:** Assess probability (likelihood) and impact (consequence) for each risk (e.g., High, Medium, Low).
    - **Quantitative:** Assign numerical values to probability and impact to calculate risk exposure.
5. **Risk Prioritization:** Rank risks based on their severity (probability x impact).
6. **Document Risks:** Create a Risk Register.

Source Code: *This lab is conceptual and involves analysis and documentation, not writing source code.*

Input:

- Project scope, requirements, and design.
- Historical project data (if available).

Expected Output:

- **Effort Estimation Report:**
    - **Total Estimated Effort:** 1200 person-hours.
    - **Breakdown:**
        - Requirements: 150 hours
        - Design: 250 hours
        - Development: 600 hours
        - Testing: 150 hours
        - Deployment: 50 hours

    - **Assumptions:** Team members are fully available, no major technology changes.
- **Risk Register:**
    - **Risk ID:** R001
    - **Description:** Key developer leaves the project.
    - **Category:** Personnel
    - **Probability:** Medium
    - **Impact:** High
    - **Mitigation Strategy:** Cross-training, comprehensive documentation, succession planning.
    - **Contingency Plan:** Hire a new developer, reallocate tasks.
    - **Risk ID:** R002
    - **Description:** Scope creep due to new client requirements.
    - **Category:** Scope
    - **Probability:** Medium
    - **Impact:** Medium
    - **Mitigation Strategy:** Strict change control process, clear scope definition.
    - **Contingency Plan:** Renegotiate timeline/budget, defer features to next phase.

Lab 15: Software Quality Assurance Components

Aim: To understand and apply various components and activities involved in ensuring software quality throughout the development lifecycle.

Procedure:

1. **Understand Quality Concepts:** Differentiate between Quality Control (QC - finding defects) and Quality Assurance (QA - preventing defects).
2. **Identify QA Activities in SDLC:** Map QA activities to each phase of the Software Development Life Cycle:
    - **Requirements Phase:** Requirements review, traceability matrix.
    - **Design Phase:** Design reviews, architectural reviews.
    - **Development Phase:** Code reviews, static code analysis, unit testing.
    - **Testing Phase:** Integration testing, system testing, acceptance testing, performance testing, security testing.
    - **Deployment/Maintenance Phase:** Release management, post-implementation review, defect tracking.
3. **Explore QA Tools:** Research and understand tools that support QA activities (e.g., test management tools, defect tracking systems, automation testing frameworks, static analysis tools).
4. **Define Quality Metrics:** Identify relevant metrics to measure software quality (e.g., defect density, test coverage, mean time to failure).
5. **Establish Quality Standards:** Understand and apply industry standards or organizational guidelines for quality.
6. **Role of Configuration Management:** Understand how configuration management (version control of code, documents) contributes to quality.
7. **Role of Process Improvement:** Recognize the importance of continuous process improvement (e.g., CMMI, Six Sigma) in enhancing quality.
8. **Document QA Plan:** Create a plan outlining the QA activities, responsibilities, tools, and metrics for the project.

Source Code: *This lab is conceptual and focuses on processes and documentation, not writing source code. The "Source Code" section here refers to the conceptual application of QA within a project.*

Input:

- Software project plan and requirements.
- Organizational quality policies.

Expected Output:

- **Software Quality Assurance Plan:**
    - **Introduction:** Purpose and scope of the QA plan.
    - **QA Activities by Phase:**
        - **Requirements:** Formal review meetings, SRS walkthroughs.
        - **Design:** Design document peer reviews, architectural compliance checks.
        - **Development:** Mandatory code reviews (e.g., 2 reviewers per module), static analysis using SonarQube, unit test coverage target (e.g., 80%).
        - **Testing:**
            - Integration Testing: Performed after each module integration.

        - System Testing: Performed by dedicated QA team.
        - User Acceptance Testing (UAT): Conducted by key stakeholders.
        - Performance Testing: Using JMeter for load testing.
        - Security Testing: Using OWASP ZAP for vulnerability scanning.
    - **Deployment:** Pre-release checklist, post-deployment smoke tests.
- **Quality Metrics:**
    - Defect Density: Number of defects per KLOC (Kilo Lines of Code).
    - Test Case Pass Rate: Percentage of passed test cases.
    - Requirements Traceability: Percentage of requirements linked to test cases.
- **Tools:** Jira (defect tracking), Git (version control), Selenium (automation testing).
- **Roles and Responsibilities:** QA Lead, Test Engineers, Developers.
- **Reporting:** Weekly QA status reports, defect trend analysis.