

Experiment No. 07: Use project management tool to prepare Gantt Chart for the project

Learning Objective:

Students will be able to draw Gantt Chart for the project and understand its role in project scheduling and tracking.

Tools Used:

- **Conceptual Tools:** Gantt Chart principles, Work Breakdown Structure (WBS), Task Dependencies, Milestones, Critical Path Method (related concept often used with scheduling).
- **Software Tools (Examples):** Microsoft Project, Jira (with specific plugins/views), Asana, Trello (with Gantt chart power-ups), Excel (using templates), Draw.io (manual drawing), dedicated Gantt chart software (e.g., GanttProject, TeamGantt).
- **Mentioned in Document:** Gantt Chart, PERT (Program Evaluation and Review Technique) - *It's crucial to understand PERT as well, as it's often compared with Gantt charts.*

1. Full Theory:

a. Project Scheduling and Tracking:

- **Definition:** Project Scheduling is the process of defining project activities, sequencing them in a logical order, estimating their duration, and assigning resources, ultimately creating a timeline or schedule for project completion. Project Tracking involves monitoring the progress of these activities against the planned schedule, identifying deviations, and taking corrective actions.
- **Purpose:** The primary aim is to ensure the project is completed on time and within the allocated resources. Effective scheduling provides a roadmap, facilitates communication, helps manage dependencies, and allows for proactive risk management. Tracking ensures the project stays on course.
- **Key Elements for Effective Scheduling (as per document):**
 - **Task Decomposition:** Dividing the project into smaller, manageable tasks or activities (often using a Work Breakdown Structure - WBS).
 - **Time & Resource Estimation:** Accurately estimating the duration required for each task and the resources (human, material, financial) needed.
 - **Concurrency:** Identifying tasks that can be performed in parallel to optimize workforce utilization and shorten project duration.

- **Dependency Management:** Recognizing and minimizing dependencies between tasks to avoid bottlenecks and delays where one task must wait for another.

-

b. Gantt Chart:

- **Definition:** A Gantt chart is a type of bar chart that visually represents a project schedule over time. It lists project tasks vertically on the left axis and time intervals horizontally on the top axis. Each task is represented by a horizontal bar, the position and length of which indicate the task's start date, duration, and end date.
- **History:** Developed by Henry Gantt around 1910-1915, it was a revolutionary tool for visualizing project schedules and progress.
- **Purpose:**
 - **Planning:** To lay out all the tasks required to complete a project.
 - **Scheduling:** To sequence tasks and set start/end dates and durations.
 - **Monitoring:** To track progress against the planned schedule (often shown by shading the bars or adding a progress line).
 - **Communication:** To provide a clear, visual overview of the project timeline to stakeholders.
-
- **Key Components:**
 - **Task List:** A breakdown of all tasks/activities required (often derived from a WBS). Found on the vertical axis.
 - **Timeline:** The time scale for the project (days, weeks, months). Found on the horizontal axis.
 - **Bars:** Horizontal bars representing individual tasks. The length of the bar corresponds to the task's duration. The position corresponds to its start and end dates.
 - **Milestones:** Significant points or achievements in the project, often represented by a specific shape (like a diamond) on the timeline. They have zero duration. (e.g., "Project Kick-off", "Requirements Approved", "Beta Release").
 - **Dependencies:** Lines or arrows linking tasks to show relationships (e.g., Task B cannot start until Task A is finished). Common types: Finish-to-Start (FS), Start-to-Start (SS), Finish-to-Finish (FF), Start-to-Finish (SF).
 - **Resources:** Indication of who or what is assigned to each task (often shown next to the task name or bar).
 - **Progress Indicator:** Shows how much of a task or the overall project is complete (e.g., shading within the bars, a vertical line indicating the current date).

- **Baseline:** The original planned schedule, often shown as lighter bars or outlines, against which actual progress is tracked.
-
- **Benefits (as per document & expanded):**
 - **Clarity:** Clearly shows tasks, durations, start/end dates.
 - **Simplicity:** Relatively easy to understand, even for non-project managers.
 - **Progress Tracking:** Visualizes progress against the plan.
 - **Resource Management:** Can help visualize resource allocation (though complex allocation is better handled by other tools).
 - **Dependency Visualization:** Shows basic overlaps and sequential dependencies.
-
- **Limitations:**
 - **Complexity:** Can become cluttered and hard to read for very large projects with many tasks.
 - **Dependency Clarity:** While dependencies can be shown, complex interrelationships are not as clear as in network diagrams (like PERT).
 - **Critical Path:** Doesn't automatically highlight the critical path (the sequence of tasks determining the shortest possible project duration) as clearly as PERT charts. It *can* be shown, but requires calculation and specific highlighting.
 - **Uncertainty:** Doesn't inherently handle uncertainty in task durations well (uses fixed estimates).
 - **Static:** Requires manual updates to reflect progress or changes (though software tools automate this).
-

c. PERT (Program Evaluation and Review Technique):

- **Definition:** PERT is a project management technique that uses a network diagram to represent tasks and their dependencies. It's particularly useful for complex projects with uncertainty in task durations, employing probabilistic time estimates.
- **Purpose:** To schedule and coordinate tasks within a project, especially when time is a critical factor and activity durations are uncertain. It helps identify the critical path.
- **Key Components:**
 - **Events/Nodes:** Represented typically by circles or rectangles, marking the start or completion of one or more activities.
 - **Activities/Arcs:** Represented by arrows connecting the nodes, indicating the tasks that need to be performed. The arrow shows the dependency.
 - **Time Estimates:** PERT uses three time estimates for each activity:

- Optimistic Time (O): Minimum possible time.
 - Pessimistic Time (P): Maximum possible time.
 - Most Likely Time (M): Best estimate of the time required.
-
- **Expected Time (TE):** Calculated using the formula: $TE = (O + 4M + P) / 6$. This weighted average is used in the analysis.
- **Critical Path:** The longest path through the network diagram, determining the minimum project duration. Activities on the critical path have zero slack/float (delaying them delays the project).
-
- **Benefits:**
 - Handles uncertainty in duration estimates.
 - Clearly identifies the critical path and task dependencies.
 - Useful for large, complex, non-routine projects.
 - Helps in risk assessment related to time.
-
- **Limitations:**
 - Can be complex to create and maintain.
 - Time estimates are subjective and can be inaccurate.
 - Focuses primarily on time, less on cost/resources.
 - Doesn't provide the intuitive time-based visualization of a Gantt chart.
-

d. Gantt vs. PERT:

Feature	Gantt Chart	PERT Chart
Visualization	Bar chart vs. Time	Network diagram (Nodes & Arcs)
Focus	Time, Schedule, Progress	Time, Dependencies, Critical Path
Complexity	Simpler, good for smaller projects	More complex, good for large ones
Time Estimate	Single point estimate (duration)	Probabilistic (O, M, P -> TE)
Dependencies	Shown, but can be less clear	Clearly shown via network
Critical Path	Not inherently shown (can add)	Primary output

Uncertainty	Poor handling	Good handling
Best Use	Communication, tracking progress	Planning complex, uncertain tasks

2. Step-by-Step Procedure (Creating a Gantt Chart):

1. **Define the Project Scope:** Clearly understand the project objectives and deliverables.
2. **Create a Work Breakdown Structure (WBS):** Decompose the project into major phases, then into smaller tasks, and potentially sub-tasks. This forms the basis of the task list.
Example: For "Develop Login Module": Tasks could be "Design UI", "Develop Frontend", "Develop Backend API", "Integrate Frontend/Backend", "Write Unit Tests", "Perform Integration Testing".
3. **Estimate Task Durations:** For each lowest-level task, estimate how long it will take (e.g., in hours, days, weeks). Be realistic.
4. **Identify Task Dependencies:** Determine the relationships between tasks. Which tasks must finish before others can start (FS)? Which can start together (SS)? etc. *Example: "Develop Frontend" might depend on "Design UI" being finished (FS).*
5. **Identify Milestones:** Mark key checkpoints or deliverables that signify progress but have no duration. *Example: "UI Design Approved", "Login Feature Complete".*
6. **Assign Resources (Optional but Recommended):** Assign personnel or other resources to each task. This helps in workload management and identifying potential bottlenecks.
7. **Choose a Tool:** Select software (like MS Project, Jira, Excel) or decide to draw manually (less common for complex projects).
8. **Input Data into Tool / Draw Chart:**
 - List tasks vertically.
 - Set up the timeline horizontally (choose appropriate units).
 - Plot each task as a bar according to its estimated start date and duration.
 - Draw lines/arrows between tasks to represent dependencies.
 - Mark milestones on the timeline (often using diamonds).
 - Add resource names next to tasks if applicable.
- 9.
10. **Review and Refine:** Check the chart for logical consistency, overloaded resources, and realistic timelines. Adjust as necessary.
11. **Set the Baseline:** Once the plan is finalized, save it as the baseline. This is the benchmark against which progress will be measured.
12. **Track Progress:** As the project progresses, update the chart:
 - Mark tasks as complete (or percentage complete, e.g., by shading the bars).

- Adjust timelines if delays occur.
- Compare actual progress against the baseline.
- Communicate status using the updated chart.

13.



3. Required Diagrams:

a. Example Gantt Chart (Financial Management System - Simplified):

(Imagine a visual chart here. I will describe it)

- **Vertical Axis (Tasks):**

- Project Initiation
 - Define Scope
 - Secure Funding
-
- Planning
 - Create Project Plan (Milestone: Plan Approved 💎)
 - Develop WBS
-
- Requirement Analysis
 - Gather Requirements
 - Analyze Requirements (Milestone: Requirements Finalized 💎)
-
- Design & Development
 - Design System Architecture
 - Develop Database (Depends on Arch. Design - FS)
 - Develop UI (Depends on Arch. Design - FS)
 - Develop Backend Logic (Depends on DB & Arch. - FS)
 - Integrate Modules (Depends on DB, UI, Backend - FS)
-
- Testing
 - Unit Testing (Parallel with Dev)
 - Integration Testing (After Integration)
 - User Acceptance Testing (UAT) (Milestone: UAT Sign-off 💎)
-
- Deployment
 - Prepare Production Env.

- Deploy Application (Milestone: Go Live )
 -
 - Post-Deployment
 - Monitor System
 - User Training
 -
-
- **Horizontal Axis (Timeline):** Weeks (Week 1, Week 2, ... Week 16)
- **Bars:** Each task has a bar showing its start week and duration (e.g., "Define Scope" might be a 1-week bar in Week 1; "Develop Database" might be a 3-week bar starting in Week 5).
- **Dependencies:** Arrows shown (e.g., arrow from the end of "Design System Architecture" bar to the start of "Develop Database" bar).
- **Milestones:** Diamonds placed on the timeline at the end of the relevant week (e.g., "Plan Approved  " at the end of Week 3).
- **Progress:** (Could be shown with shading) e.g., If we are in Week 7, bars for completed tasks are fully shaded, and ongoing tasks are partially shaded. A vertical line might indicate "Today" (Week 7).

(Self-note: The image provided in the OCR document for the Financial Management system is a good example, though it lacks explicit dependency lines and distinct milestone markers. The one in the Theory section is simpler but clearly shows the bar concept).

b. Simple PERT Chart Example (for comparison):

(Imagine a network diagram)

graph LR

```

A[Start] --> B(Define Scope - 2w);
B --> C(Secure Funding - 1w);
C --> D(Create Plan - 1w);
D --> E{Plan Approved};
E --> F(Gather Req - 2w);
E --> G(Design Arch - 3w);
F --> H(Analyze Req - 1w);
H --> I{Req Finalized};
G --> J(Develop DB - 3w);
G --> K(Develop UI - 4w);
I --> L(Develop Backend - 5w); // Assuming Req Finalized needed for Backend
J --> M(Integrate - 2w);
K --> M;
L --> M;
M --> N(Testing - 3w);

```

```
N --> O{UAT Sign-off};
O --> P(Deploy - 1w);
P --> Q[End];
```

```
style E fill:#f9f,stroke:#333,stroke-width:2px; // Milestone Style
style I fill:#f9f,stroke:#333,stroke-width:2px; // Milestone Style
style O fill:#f9f,stroke:#333,stroke-width:2px; // Milestone Style
```

content_copy

download

Use code [with caution](#). Mermaid

- **Explanation:** This shows tasks (like "Define Scope") as nodes/boxes with estimated durations (e.g., 2w = 2 weeks). Arrows show dependencies. Nodes like "Plan Approved" are milestones (events). This format clearly shows the flow and dependencies but not against a calendar timeline like Gantt. Calculating TE and finding the critical path would be the next step in a full PERT analysis.

4. Examples and Real-World Applications:

- **Software Development:** Planning sprints, tracking feature development, managing release cycles.
- **Construction:** Scheduling foundation laying, framing, plumbing, electrical work, finishing.
- **Event Planning:** Organizing venue booking, invitations, catering, speaker confirmations, setup.
- **Marketing Campaigns:** Planning ad creation, media buying, launch dates, performance tracking.
- **Research Projects:** Scheduling literature review, experiments, data analysis, report writing.
- **Manufacturing:** Planning production runs, material procurement, assembly line setup.

5. Viva Questions (Basic to Toughest):

1. **What is a Gantt chart?**
 - *Answer:* It's a horizontal bar chart used in project management to visually represent a project schedule over time, showing tasks, their durations, start/end dates, and dependencies.
- 2.
3. **What do the horizontal and vertical axes represent?**

- *Answer:* The vertical axis lists the project tasks or activities. The horizontal axis represents the time intervals (days, weeks, months).
- 4.
- 5. **What does the length of a bar on a Gantt chart signify?**
 - *Answer:* The length of the bar signifies the duration of the task.
- 6.
- 7. **What is a milestone in a Gantt chart? How is it represented?**
 - *Answer:* A milestone is a significant event or checkpoint in the project timeline that has zero duration. It often marks the completion of a major phase or deliverable. It's typically represented by a symbol like a diamond.
- 8.
- 9. **What are dependencies? Give an example.**
 - *Answer:* Dependencies show the relationships between tasks, indicating the order in which they must be performed. A common type is Finish-to-Start (FS), meaning one task must finish before the next can begin. Example: Coding must finish before Testing can begin.
- 10.
- 11. **What are the main benefits of using a Gantt chart?**
 - *Answer:* Clarity in visualizing the schedule, simplicity in understanding, ease of tracking progress, helps in communication, and shows task overlaps and basic dependencies.
- 12.
- 13. **What is the difference between a Gantt chart and a PERT chart?**
 - *Answer:* Gantt charts are bar charts visualizing tasks against time, good for communication and progress tracking using single time estimates. PERT charts are network diagrams focusing on dependencies and the critical path, using probabilistic time estimates to handle uncertainty.
- 14.
- 15. **When would you choose to use PERT over Gantt?**
 - *Answer:* You'd choose PERT for large, complex projects where task durations are uncertain, and identifying the critical path and understanding complex dependencies is paramount. It's often used in R&D or pioneering projects.
- 16.
- 17. **How do you create a Gantt chart? (Outline steps)**
 - *Answer:* Identify tasks (WBS), estimate durations, identify dependencies, identify milestones, choose a tool, plot tasks/dependencies/milestones on the timeline, review, set baseline, and track progress.

18.

19. Can a Gantt chart show the critical path? How?

- *Answer:* A standard Gantt chart doesn't automatically calculate or display the critical path like PERT does. However, most project management software *can* calculate it (using CPM algorithms based on the entered tasks, durations, and dependencies) and then highlight the critical path tasks (e.g., by coloring their bars red) on the Gantt view. So, yes, it *can* show it, but it's usually a calculated overlay.

20.

21. What are the limitations of Gantt charts?

- *Answer:* Can become cluttered for large projects, complex dependencies aren't always clear, doesn't inherently handle uncertainty well, doesn't automatically show the critical path, and requires regular updates to remain useful.

22.

23. What is a Work Breakdown Structure (WBS) and how does it relate to Gantt charts?

- *Answer:* A WBS is a hierarchical decomposition of the total scope of work to be carried out by the project team. It breaks down the project into smaller, more manageable components (phases, deliverables, tasks). The lowest level tasks from the WBS typically form the task list used on the vertical axis of the Gantt chart.

24.

25. How is progress typically shown on a Gantt chart?

- *Answer:* Progress can be shown in several ways: by shading the portion of the task bar that is complete, by having a separate percentage complete column, or by a vertical "Today" line that intersects the bars showing where progress *should* be versus where it *is*.

26.

27. Explain different types of task dependencies (FS, SS, FF, SF).

- *Answer:*
 - **Finish-to-Start (FS):** Task B cannot start until Task A finishes (Most common). *Ex: Testing starts after Coding finishes.*
 - **Start-to-Start (SS):** Task B cannot start until Task A starts. *Ex: Writing documentation starts when coding starts.*
 - **Finish-to-Finish (FF):** Task B cannot finish until Task A finishes. *Ex: Final inspection cannot finish until all painting finishes.*
 - **Start-to-Finish (SF):** Task B cannot finish until Task A starts (Least common). *Ex: A legacy system must keep running until the new system starts deployment.*

○

28.

29. **What is a baseline schedule and why is it important?**

- *Answer:* The baseline is the initial, approved project schedule (tasks, dates, durations, costs). It serves as a benchmark. It's important because it allows project managers to track actual performance against the original plan, measure variances (delays, scope changes), and make informed decisions to bring the project back on track if needed.

30.

6. Common Mistakes & How to Avoid Them:

- **Mistake:** Tasks too large / Not using WBS.
 - **Avoidance:** Break down the project using a WBS to get manageable tasks (typically tasks that take between 8-80 hours of effort, or fit within a reporting period).
-
- **Mistake:** Forgetting or incorrectly identifying dependencies.
 - **Avoidance:** Carefully analyze the workflow. Ask "What needs to be done *before* this task can start/finish?". Use software features to link tasks explicitly.
-
- **Mistake:** Unrealistic duration estimates.
 - **Avoidance:** Use historical data, expert judgment, or techniques like PERT's three-point estimation. Add buffer time cautiously, but don't pad excessively.
-
- **Mistake:** Not including milestones.
 - **Avoidance:** Identify key project checkpoints and deliverables and mark them clearly as milestones. This helps in tracking major progress points.
-
- **Mistake:** Ignoring resource allocation/constraints.
 - **Avoidance:** Assign resources to tasks and check for overallocation. If someone is assigned 16 hours of work in an 8-hour day, the schedule is unrealistic. Adjust task timing or resource assignments.
-
- **Mistake:** Not setting a baseline.
 - **Avoidance:** Once the initial plan is agreed upon, formally save it as the baseline before tracking begins.
-
- **Mistake:** Not updating the chart regularly.

- **Avoidance:** Schedule regular updates (e.g., daily or weekly) to reflect actual start/finish dates and percentage complete. An outdated chart is useless or misleading.
-
- **Mistake:** Making the chart too complex visually.
 - **Avoidance:** Use software features to filter or collapse tasks. Focus on showing the necessary level of detail for the audience. Use color-coding meaningfully, not excessively.
-

7. Documentation:

- The Gantt chart itself is a key piece of project documentation, usually forming a central part of the **Project Schedule** within the overall **Project Management Plan**.
- It's not a standalone document type like an SRS, but rather a visual tool used *within* planning and tracking documentation.
- When included in reports, ensure it's accompanied by explanatory text covering assumptions, progress status, identified issues, and next step

Experiment No. 02: Draw DFD (Data Flow Diagram)

Learning Objective:

Students will be able to identify the data flows, processes, sources, and destinations for their mini-project and analyze and design the DFD up to 2 levels (Context Level, Level 0, Level 1, potentially Level 2 as mentioned in the theory).

Tools Used:

- **Conceptual Tools:** Systems Analysis, Data Flow Concepts (Processes, Data Stores, External Entities, Data Flows), Context Diagram, Leveling/Decomposition.
- **Software Tools:** Draw.io (recommended in document), StarUML (mentioned in document, though primarily for UML), Microsoft Visio, Lucidchart, Pencil Project, or any diagramming tool.

1. Full Theory:

a. Data Flow Diagram (DFD):

- **Definition:** A DFD is a graphical representation of the "flow" of data through an information system, modeling its process aspects. DFDs show *what* a system does, not *how* it does it (they depict the logical flow of data, not the physical implementation or control flow). They map out the inputs, outputs, processes, and data storage within a system.
- **Purpose:**
 - To understand and document the scope and boundaries of a system (Context Diagram).
 - To visualize the major functions/processes within the system and how they interact through data exchange.
 - To communicate the system's data flow requirements clearly to both technical (developers) and non-technical (clients, users) stakeholders.
 - To provide a basis for further system design and analysis.
-
- **Key Components (Gane & Sarson notation is common, Yourdon/DeMarco is also used):**
 - **Process:** Represents an activity or function that transforms incoming data flow(s) into outgoing data flow(s). It signifies *work being done*.
 - **Representation:** Typically a circle (Yourdon/DeMarco) or a rectangle with rounded corners (Gane & Sarson). (The document shows a circle).
 - **Naming:** Uses a verb-noun phrase (e.g., "Process Payment", "Validate Order", "Generate Report"). Processes should have at least one input and one output data flow. Numbered for reference (e.g., 1.0, 2.0 at Level 0; 1.1, 1.2 if Process 1.0 is decomposed).
 -
 - **External Entity (Source/Sink/Terminator):** Represents an originator or receiver of data *outside* the boundary of the system being modeled. It interacts with the system by sending or receiving data.
 - **Representation:** Typically a rectangle. (The document shows a rectangle).
 - **Naming:** Uses a noun or noun phrase (e.g., "Customer", "Bank", "Tax Authority", "Manager"). They are outside the system, so we don't model their internal workings.
 -
 - **Data Store:** Represents a place where data is held or stored within the system for later use. It signifies *data at rest*.
 - **Representation:** Two parallel lines, or a rectangle open on one side (usually the right). (The document shows the open rectangle).

- **Naming:** Uses a noun or noun phrase (e.g., "Customer Records", "Orders", "Product Inventory"). Processes can read from and write to data stores. Data flows *to* a data store mean update (write, delete, add). Data flows *from* a data store mean read or retrieval. Direct flow between data stores or between external entities and data stores is generally disallowed (must go via a process).
 -
 - **Data Flow:** Represents the movement of data between processes, data stores, and external entities. It signifies *data in motion*.
 - **Representation:** A directed arrow.
 - **Naming:** Uses a noun or noun phrase describing the data being moved (e.g., "Payment Details", "Validated Order", "Customer Information", "Tax Report"). Arrows should not represent control flow, only data. Data flows should originate or terminate at a process.
 -
-

b. Levels of DFDs (Leveling/Decomposition):

DFDs are typically drawn in layers or levels, providing increasing detail. This technique is called leveling or decomposition.

- **Context Diagram (Level -1 or Level 0 - terminology varies):** The highest-level DFD. It shows the entire system as a single process bubble, along with the external entities that interact with it and the main data flows between the system and these entities. It defines the system's boundary and scope. *There is only ONE process bubble and NO data stores shown at this level.* (The document sample shows a Level 0 with multiple processes, which is less standard; typically the first diagram with one process is the Context Diagram). Let's assume the document's "Level 0" is actually the Context Diagram concept.
- **Level 0 DFD (or Level 1 if Context is 0):** This diagram decomposes the single process from the Context Diagram into the major high-level processes within the system. It shows the main data stores and how data flows between these processes, data stores, and the external entities already identified in the Context Diagram. It should be balanced with the Context Diagram (data flows entering/leaving the Level 0 diagram boundary must match those in the Context Diagram). (The document example labeled "Level 0" fits this description).
- **Level 1 DFD:** Decomposes one of the processes shown in the Level 0 DFD into more detailed sub-processes. It shows the data flows and data stores relevant to that specific parent process. Must be balanced with the parent process in the Level 0 DFD (inputs/outputs match). (The document example labeled "Level 1" fits this).

- **Level 2 DFD (and lower):** Further decomposes processes from Level 1 (or higher levels). Each level provides more detail about a specific part of the system. Decomposition continues until processes are simple enough to be clearly understood (primitive processes). (The document example labeled "Level 2" fits this).
- **Balancing:** A crucial concept. Data flows entering or leaving a parent process bubble in a higher-level DFD must match the data flows entering or leaving the boundary of the corresponding lower-level DFD that decomposes that parent process. This ensures consistency across levels.

c. DFD Drawing Guidelines & Rules (Essential Observations from Document & Standard Practice):

- **Unique Naming:** All components (processes, entities, stores, flows) should have unique and descriptive names.
- **Focus on Data, Not Control:** Arrows show data movement, not sequence or control flow (unlike flowcharts). Avoid logical decision representation (e.g., no diamond shapes for decisions).
- **Process Rules:**
 - Must have at least one input and one output data flow.
 - Named with Verb-Noun.
 - Numbered hierarchically (e.g., Process 3 -> 3.1, 3.2, 3.3).
-
- **Data Store Rules:**
 - Data must move through a process to get into or out of a store. No direct Entity-to-Store or Store-to-Store flows.
 - Named with Noun/Noun Phrase.
-
- **External Entity Rules:**
 - Interact only with processes. No direct Entity-to-Entity, Entity-to-Store flows.
 - Lie outside the system boundary.
 - Named with Noun/Noun Phrase.
-
- **Data Flow Rules:**
 - Must originate or terminate at a process.
 - Arrows indicate direction.
 - Named with Noun/Noun Phrase describing the data.
 - Data cannot flow directly between two Entities or two Data Stores.
-

- **Leveling & Balancing:** Maintain consistency of data flows across levels.
- **Clarity:** Avoid crossing lines as much as possible. Keep diagrams neat. Start with the Context Diagram and work downwards. Limit the number of processes per diagram (typically 5-9) to maintain readability.

2. Step-by-Step Procedure (Creating DFDs):

1. **Identify External Entities:** Who or what interacts with the system by providing input or receiving output? (e.g., User, Admin, Bank, Payment Gateway).
2. **Identify System Inputs & Outputs:** What data does the system receive from each entity, and what data does it send back?
3. **Draw the Context Diagram (Level -1/0):**
 - Draw a single circle/rounded rectangle representing the entire system (e.g., "Financial Management System").
 - Draw the identified external entities around the process.
 - Draw and label the major data flows between the entities and the system process based on the inputs/outputs identified in step 2.
- 4.
5. **Identify Major System Processes:** What are the main functions or transformations the system performs on the input data to produce the output data? (e.g., Manage Users, Process Transactions, Generate Reports).
6. **Identify Major Data Stores:** What information does the system need to store internally to perform its functions? (e.g., User Accounts, Transactions Log, Financial Reports).
7. **Draw the Level 0 DFD (Decomposing the Context Diagram):**
 - Draw the processes identified in step 4 (number them 1.0, 2.0, etc.).
 - Draw the data stores identified in step 5.
 - Include the external entities from the Context Diagram.
 - Draw the data flows:
 - Connect entities to processes (must match Context Diagram flows at the boundary).
 - Connect processes to other processes.
 - Connect processes to data stores (read/write).
 -
 - **Balance:** Ensure all data flows entering/leaving the Context Diagram are present at the boundary of the Level 0 DFD.
- 8.
9. **Decompose to Level 1 (and lower if needed):**

- Choose *one* complex process from the Level 0 DFD (e.g., Process 2.0 "Process Transactions").
- Identify the sub-processes required to perform that function (e.g., 2.1 Validate Payment, 2.2 Record Transaction, 2.3 Update Account Balance).
- Identify any data stores used *only* within this parent process (or those it interacts with from Level 0).
- Draw the Level 1 DFD for that parent process, showing the sub-processes, relevant data stores, and data flows between them.
- Include any data flows from the parent process diagram (Level 0) that enter or leave this specific process.
- **Balance:** Ensure the inputs/outputs of the Level 1 diagram match the inputs/outputs of the parent process bubble (e.g., Process 2.0) in the Level 0 diagram.

10.

11. **Repeat Step 7:** Continue decomposing complex processes to Level 2, Level 3, etc., until each process is simple enough (a "primitive process").

12. **Review and Refine:** Check all diagrams for consistency, balancing, clarity, and adherence to DFD rules. Ensure names are meaningful.

3. Required Diagrams (Examples for Financial Management System):

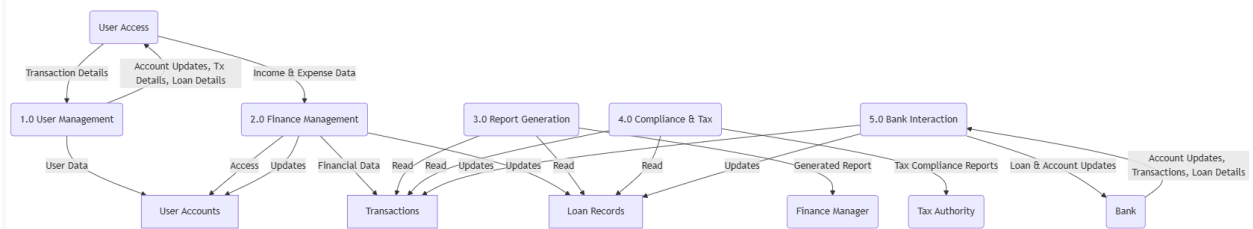
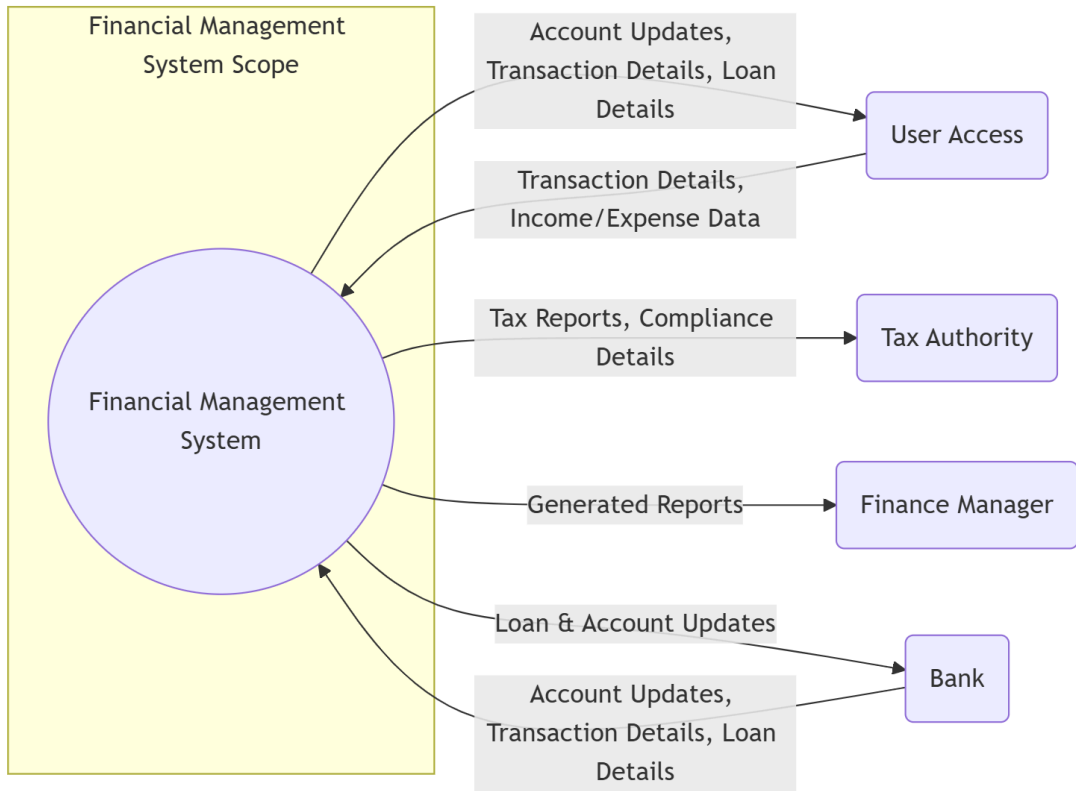
(Based on the diagrams in the provided OCR document)

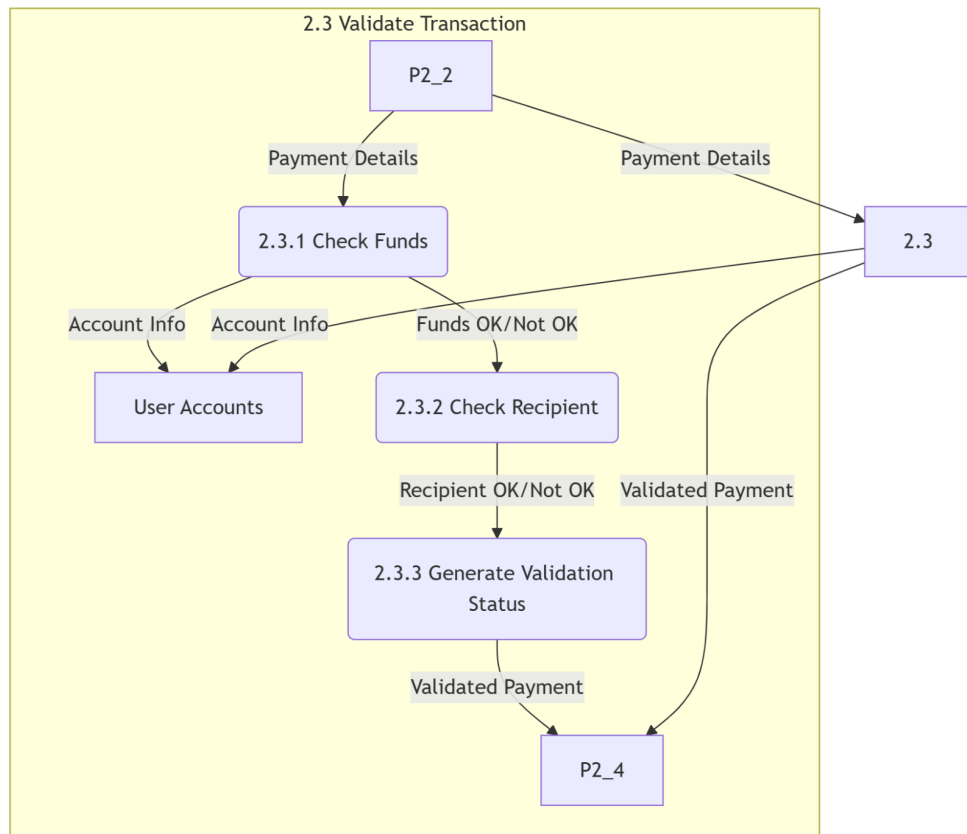
a. Context Diagram (Conceptual - might be considered the document's "Level 0")

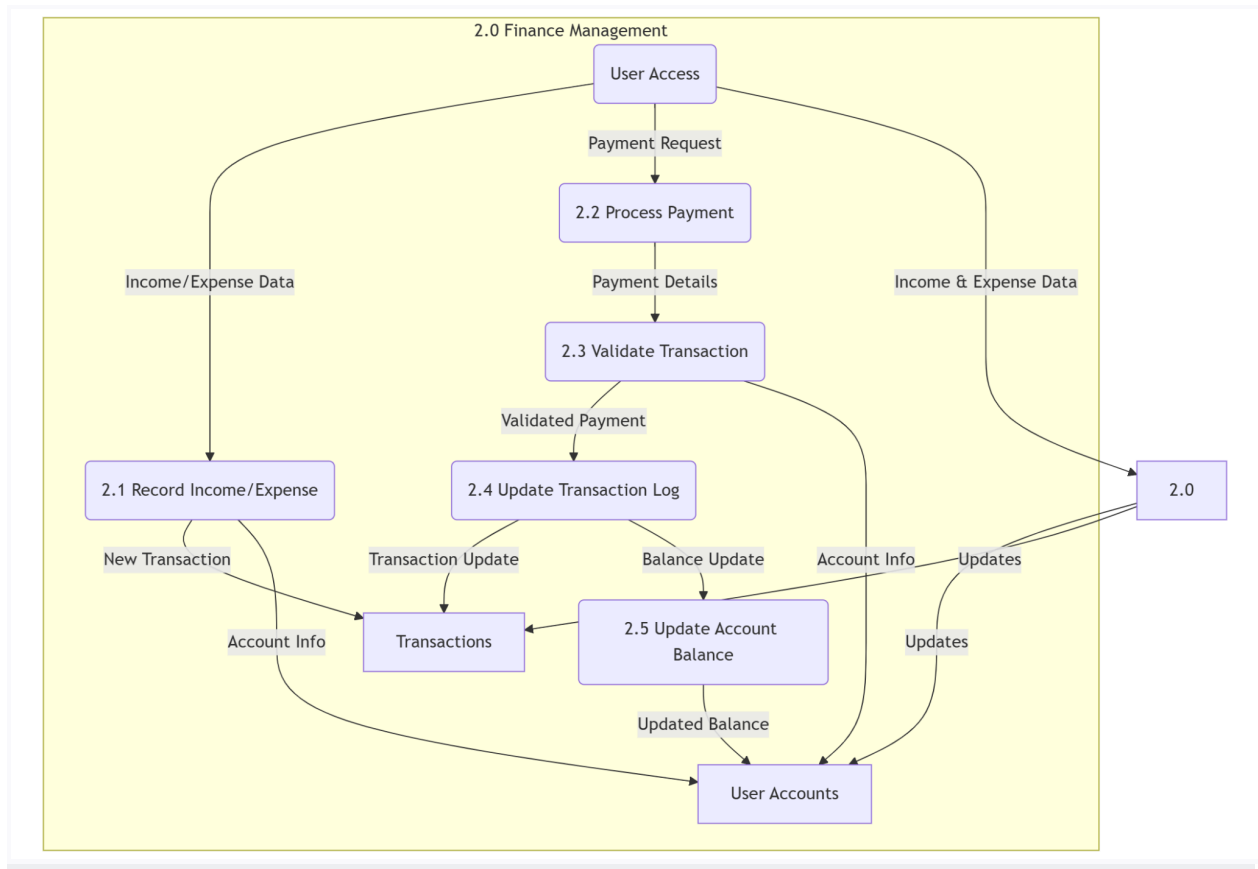
graph LR

```
subgraph Financial Management System Scope
    P((Financial Management System))
end
```

```
User(User Access) -->|Transaction Details, Income/Expense Data| P;
P -->|Account Updates, Transaction Details, Loan Details| User;
P -->|Tax Reports, Compliance Details| TaxAuth(Tax Authority);
P -->|Generated Reports| FinMgr(Finance Manager);
P -->|Loan & Account Updates| Bank(Bank);
Bank -->|Account Updates, Transaction Details, Loan Details| P;
```







- **Explanation:** Shows the FMS as one process. Identifies external entities (User, Tax Authority, Finance Manager, Bank) and the main data flowing between them and the system.

b. Level 0 DFD (Conceptual - represents the decomposition of the Context Diagram)

(This corresponds closely to the document's "Level 0" diagram)

graph TD

User(**User** Access) -- Transaction Details --> P1(1.0 User Management);

User -- Income & Expense Data --> P2(2.0 Finance Management);

P1 -- User Data --> DS1[User Accounts];

P2 -- Financial Data --> DS2[Transactions];

P2 -- Access --> DS1;

P2 -- Updates --> DS1;

P2 -- Updates --> DS3[Loan Records];

P3(**3.0** Report Generation) -- Read --> DS2;

P3 -- Read --> DS3;

P3 -- Generated Report --> FinMgr(Finance Manager);

P4(**4.0** Compliance & Tax) -- Read --> DS2;

P4 -- Read --> DS3;

P4 -- Tax Compliance Reports --> TaxAuth(Tax Authority);

P5(**5.0** Bank Interaction) -- Loan & Account Updates --> Bank(Bank);

Bank -- Account Updates, Transactions, Loan Details --> P5;
P5 -- Updates --> DS3;
P5 -- Updates --> DS2;
P1 -- Account Updates, Tx Details, Loan Details --> User; % Simplified output flow

content_copy

download

Use code with caution. Mermaid

- **Explanation:** Decomposes the FMS into major processes (User Management, Finance Management, Report Generation, Compliance, Bank Interaction - numbers are conceptual). Shows interaction with external entities and internal data stores (User Accounts, Transactions, Loan Records). Data flows connect these components. *Self-correction: The OCR diagram is slightly different but follows this principle. Balancing Check:* Ensure flows like "Transaction Details" from User connect to the appropriate process (Finance Management), and "Generated Report" to Finance Manager originates from the "Report Generation" process, matching the Context Diagram boundary flows.

c. Level 1 DFD (Example: Decomposing 2.0 Finance Management)

(This corresponds to the document's "Level 1" decomposition idea, focusing on Finance Management)

graph TD

subgraph 2.0 Finance Management

User(User Access) -- Income/Expense Data --> P2_1(2.1 Record Income/Expense);

User -- Payment Request --> P2_2(2.2 Process Payment);

P2_1 -- New Transaction --> DS_Tx[Transactions];

P2_1 -- Account Info --> DS_Acc[User Accounts]; % Read user info

P2_2 -- Payment Details --> P2_3(2.3 Validate Transaction);

P2_3 -- Validated Payment --> P2_4(2.4 Update Transaction Log);

P2_3 -- Account Info --> DS_Acc; % Read balance

P2_4 -- Transaction Update --> DS_Tx;

P2_4 -- Balance Update --> P2_5(2.5 Update Account Balance);

P2_5 -- Updated Balance --> DS_Acc;

end

% Inputs/Outputs matching 2.0 in Level 0

User -- Income & Expense Data --> 2.0;

2.0 -- Updates --> DS_Acc; % Access to User Accounts store from Level 0

2.0 -- Updates --> DS_Tx; % Access to Transactions store from Level 0

content_copy

download

Use code [with caution](#).Mermaid

- **Explanation:** Shows the sub-processes within "Finance Management". Includes interactions with data stores (possibly inherited from Level 0 or local temporary stores). Inputs (Income/Expense Data) and outputs (Updates to Stores) must balance with the "2.0 Finance Management" process in the Level 0 DFD. *The document's Level 1 diagram shows different processes like 'Analyze and Report' and 'Update Transaction Data Store' emerging from 'Finance Manager', which seems more like decomposing the reporting/admin functions, but the principle is the same.*

d. Level 2 DFD (Example: Decomposing 2.3 Validate Transaction)

(Further decomposition, not explicitly shown in the same way in the document's Level 2, but follows the principle)

graph TD

subgraph 2.3 Validate Transaction

P2_2 -- Payment Details --> P2_3_1(2.3.1 Check Funds);

P2_3_1 -- Account Info --> DS_Acc[User Accounts]; % Read balance

P2_3_1 -- Funds OK/Not OK --> P2_3_2(2.3.2 Check Recipient);

P2_3_2 -- Recipient OK/Not OK --> P2_3_3(2.3.3 Generate Validation Status);

P2_3_3 -- Validated Payment --> P2_4; % Output to next process

end

% Balancing Inputs/Outputs

P2_2 -- Payment Details --> 2.3;

2.3 -- Account Info --> DS_Acc;

2.3 -- Validated Payment --> P2_4;

content_copy

download

Use code [with caution](#).Mermaid

- **Explanation:** Breaks down "Validate Transaction" into finer steps. Balancing ensures the inputs/outputs match the parent process (2.3) in the Level 1 DFD. *The document's Level 2 example focuses on decomposing the 'User Management' and 'Finance Manager' interactions.*

4. Examples and Real-World Applications:

- **Any Information System:** Order Processing, Library Management, Banking Systems, University Enrollment, Inventory Control, Hospital Management. DFDs are fundamental in structured analysis and design methodologies.
- **Business Process Modeling:** Can be adapted to show how data moves through business processes, even if not fully automated.

5. Viva Questions (Basic to Toughest):

1. What is a DFD? What does it represent?

- *Answer:* A Data Flow Diagram is a graphical tool that shows how data moves through an information system. It represents processes that transform data, data stores where data is held, external entities that interact with the system, and the data flows connecting them.

2.

3. What are the four main symbols used in a DFD?

- *Answer:* Process (circle/rounded rectangle), External Entity (rectangle), Data Store (parallel lines/open rectangle), and Data Flow (arrow).

4.

5. What is the difference between a DFD and a Flowchart?

- *Answer:* DFDs show the flow of *data*, focusing on *what* the system does. Flowcharts show the flow of *control* (sequence of operations, decisions), focusing on *how* a process is done. Flowcharts use decision symbols (diamonds), DFDs do not.

6.

7. What is a Context Diagram? What does it show?

- *Answer:* It's the highest level DFD, showing the entire system as a single process. It depicts the system boundary, the external entities interacting with it, and the major data inputs and outputs. It does not show any internal data stores.

8.

9. What is Leveling or Decomposition in DFDs?

- *Answer:* It's the process of breaking down complex processes from a higher-level DFD into more detailed sub-processes in a lower-level DFD. This creates a hierarchy of diagrams (Context -> Level 0 -> Level 1, etc.).

10.

11. What does "Balancing" mean in the context of DFDs?

- *Answer:* Balancing means that the data flows entering and leaving a parent process bubble in a higher-level DFD must exactly match the data flows entering and leaving the boundary of the corresponding lower-level DFD that decomposes that process.

12.

13. **Can two external entities communicate directly in a DFD? Why or why not?**

- *Answer:* No. DFDs model the flow of data *in relation to the system*. Communication between external entities happens outside the system's scope and is not shown.

14.

15. **Can data flow directly between two data stores? Why or why not?**

- *Answer:* No. Data must be moved or processed by a process to go from one store to another. A process must read from one store and write to the other.

16.

17. **What is a "primitive process" in a DFD?**

- *Answer:* A primitive process is a process at the lowest level of decomposition that is considered simple enough to not require further breakdown. Its logic can typically be described easily (e.g., via pseudocode or structured English).

18.

19. **How do you name the different components in a DFD?**

- *Answer:* Processes use Verb-Noun phrases (e.g., "Calculate Tax"). External Entities and Data Stores use Noun/Noun phrases (e.g., "Customer", "Product File"). Data Flows use Noun/Noun phrases describing the data (e.g., "Order Details").

20.

21. **Where would you typically stop decomposing processes (i.e., how many levels deep)?**

- *Answer:* Decomposition stops when a process is primitive – meaning it performs a single, well-defined function that can be easily understood and specified without further breakdown. There's no fixed number of levels; it depends on the system's complexity.

22.

23. **What are the differences between Yourdon/DeMarco and Gane & Sarson DFD notations?**

- *Answer:* The main visual difference is the symbol for a process: Yourdon/DeMarco uses a circle, while Gane & Sarson uses a rectangle with rounded corners (often called a "bubble" anyway). Gane & Sarson sometimes also number data stores and place the process number in a specific location on the symbol. Functionally, they represent the same concepts.

24.

25. **How does a DFD help in system analysis and design?**

- *Answer:* In analysis, it helps understand and document the existing system or requirements for a new one by clarifying data movement and processing. In design, it provides a logical model that guides the development of physical system

components (databases, program modules). It helps identify necessary functions and data storage needs.

26.

27. What are some common errors made when drawing DFDs?

- *Answer:* Unbalanced diagrams across levels, incorrect data flow directions (e.g., entity-to-entity), including control flow or loops, vague naming, missing data flows, overly complex diagrams (too many processes on one level).

28.

29. If a process reads data from a data store, modifies it, and writes it back, how would you show this on a DFD?

- *Answer:* You would show two separate data flows: one arrow from the data store *to* the process (representing the read), and another arrow from the process *back to* the data store (representing the update/write). You wouldn't use a double-headed arrow typically, as they represent distinct actions on the data.

30.

6. Common Mistakes & How to Avoid Them:

- **Mistake:** Including Control Flow (Sequence, Decisions).
 - **Avoidance:** Remember DFDs are about *data* movement. Use arrows only for data. Do not use decision symbols. Sequence is implied by data availability, not explicit control lines.
-
- **Mistake:** Connecting Entities or Stores Directly.
 - **Avoidance:** All data must pass through a process if it comes from/goes to an entity or moves between stores. Redraw flows to include an intermediary process.
-
- **Mistake:** Forgetting Balancing.
 - **Avoidance:** Systematically check each decomposed diagram against its parent process. List the inputs/outputs of the parent bubble and ensure they match exactly the inputs/outputs crossing the boundary of the child diagram.
-
- **Mistake:** Vague Names (e.g., "Process Data", "Information").
 - **Avoidance:** Use specific names. Processes: Verb-Noun (Calculate Total, Validate User). Flows/Stores: Specific Noun (Validated Order, Customer Record, Payment Amount).
-
- **Mistake:** Black Holes, Gray Holes, Miracles.

- **Avoidance:** Check processes: A "black hole" has inputs but no outputs. A "gray hole" has outputs insufficient to generate from the inputs. A "miracle" has outputs but no inputs. Every process must transform data logically.
-
- **Mistake:** Overly Complex Diagrams.
 - **Avoidance:** Limit processes per diagram (5-9 rule). If a diagram gets too busy, check if processes can be grouped logically and decomposed further on the next level.
-

7. Documentation:

- DFDs (Context, Level 0, Level 1, etc.) are key components of the **System Analysis** documentation.
- They are often included within or referenced by a **Software Requirements Specification (SRS)** document, specifically in the sections describing functional requirements or system models.
- Each diagram should ideally be accompanied by:
 - A brief description of the scope it covers.
 - A **Data Dictionary:** A separate document or section defining every data flow and data store element in detail (e.g., Data Flow Name: "Customer Order", Description: Contains customer ID, items ordered, quantities, shipping address. Composition: Customer ID + Order Items + Shipping Address). This is crucial for complex systems.
-

Experiment No. 03: Implement UML Use-case

Learning Objective:

To understand and implement the dynamic behavioral view of a system using UML Use Case diagrams. Students will be able to identify actors, use cases, and their relationships for a given project/scenario.

Tools Used:

- **Conceptual Tools:** UML (Unified Modeling Language), Use Case Modeling, Actor Identification, Use Case Identification, Relationship Types (Include, Extend, Generalization).
- **Software Tools:** MS Word (for documentation), Draw.io (as mentioned), StarUML (as mentioned), Lucidchart, Visual Paradigm, Enterprise Architect, or any UML modeling tool.

1. Full Theory:

a. UML and Behavioral Diagrams:

- **UML (Unified Modeling Language):** A standardized graphical language used in software engineering for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.
- **Behavioral Diagrams:** A category of UML diagrams that depict the dynamic behavior of a system or its components. They show how the system interacts, responds to events, and changes over time. Use case diagrams are a key type of behavioral diagram.

b. Use Case Diagram:

- **Definition:** A Use Case Diagram provides a high-level, graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies or relationships between those use cases. It captures the system's functional requirements from the user's perspective.
- **Purpose:**
 - To model the intended functions (requirements) of a system.
 - To visualize the interaction between users (actors) and the system.
 - To define the system's boundary and scope.
 - To provide a context for more detailed requirement specifications (like Use Case descriptions).
 - To facilitate communication among stakeholders (users, developers, analysts).
- **Key Components:**
 - **Actor:** Represents a role played by a user or another system that interacts with the subject (system). Actors have goals they want to achieve by using the system.
 - *Representation:* Stick figure (even for non-human systems). (As shown in document).
 - *Naming:* Noun or noun phrase describing the role (e.g., "Customer", "Administrator", "Payment Gateway", "Reporting System").
 - *Types (as per document & standard practice):*
 - **Primary Actor:** Has user goals fulfilled through using services of the system. They initiate interaction. Usually drawn on the left. (e.g., Customer withdrawing cash).
 - **Supporting (or Secondary) Actor:** Provides a service (e.g., information) to the system being built. The system initiates interaction

with them. Usually drawn on the right. (e.g., Bank system providing authentication, Tax calculation service). The document example "Bank representatives" replenishing cash is a supporting actor for the ATM system's overall operation, though perhaps less directly involved in a typical *user* transaction use case.

-
- **Generalization:** Actors can have generalization relationships (e.g., "Authenticated User" might be a generalization of "Customer" and "Administrator").
-
- **Use Case:** Represents a specific, discrete functionality or service that the system provides to one or more actors to achieve their goals. It yields an observable result of value to a particular actor.
 - **Representation:** Ellipse/Oval. (As shown in document).
 - **Naming:** Verb-noun phrase describing the goal from the actor's perspective (e.g., "Withdraw Cash", "Login", "Generate Report", "Submit Order").
-
- **System Boundary:** Represents the scope of the system being modeled. It separates the internal components (use cases) from the external interactors (actors).
 - **Representation:** A rectangle drawn around the use cases. (As shown in document - "Book store", "Financial Management System (FMS)").
 - **Naming:** The name of the system is often placed inside or above the rectangle.
-
- **Relationships:** Connections between actors and use cases, or between use cases themselves.
 - **Association:** Links an Actor to a Use Case, indicating that the actor participates in that use case.
 - **Representation:** Solid line between actor and use case. Arrows are sometimes used to indicate the initiator (primary actor) but are often omitted for simplicity if navigability isn't critical to show.
 -
 - **Include («include»):** A relationship from a base use case to an included (or shared) use case, specifying that the behavior of the included use case is *always* inserted into the behavior of the base use case. Used to extract common behavior into a separate use case to avoid repetition.

- **Representation:** Dashed arrow from the base use case to the included use case, labeled with the stereotype «include». (As shown in document Figure-02).
- **Example:** "Login" might be included by "View Account Balance" and "Transfer Funds" because login is required for both. The base use cases are incomplete without the included one.

■

- **Extend («extend»):** A relationship from an extending use case to a base use case, specifying that the extending use case *conditionally* adds behavior to the base use case at a specific point (called an extension point). Used for optional or exceptional behavior.

- **Representation:** Dashed arrow from the extending use case to the base use case, labeled with the stereotype «extend». (As shown in document Figure-03).
- **Example:** "Calculate Bonus Points" might extend "Process Sale" but only if the customer is a loyalty program member. The base use case is complete even without the extending one.

■

- **Generalization:** A relationship between a general use case (parent) and a more specific use case (child) that inherits and adds to or overrides the behavior of the parent. Also used between actors.

- **Representation:** Solid line with a hollow triangle arrowhead pointing from the child (specific) to the parent (general). (As shown in document Figure-04 between Draw Rectangle/Square and Draw Polygon).
- **Example:** "Pay by Credit Card" and "Pay by PayPal" could be specializations of a general "Make Payment" use case.

■

○

●

2. Step-by-Step Procedure (Creating Use Case Diagrams):

1. **Identify the System Boundary:** Define what is considered *part* of the system and what is *external* to it. Draw the boundary rectangle and name the system.
2. **Identify Actors:** Who or what interacts with the system? Consider users, other systems, hardware devices, time-based events. (Ask: Who gets value? Who provides services? Who maintains the system?). List both primary and supporting actors.

3. **Identify Use Cases:** What are the main goals that each actor wants to achieve by using the system? Each goal typically corresponds to a use case. Phrase use cases as Verb-Noun. (Ask: What does each actor *do* with the system?).
4. **Draw Actors and Use Cases:** Place actors outside the system boundary rectangle and use cases inside. Place primary actors typically on the left, supporting actors on the right.
5. **Establish Associations:** Draw solid lines connecting actors to the use cases they interact with. An actor must be associated with at least one use case, and a use case must be associated with at least one actor.
6. **Identify Relationships Between Use Cases (Include, Extend):**
 - Look for common functionality required by multiple use cases. Extract this into a separate use case and connect it using «include» relationships from the base use cases.
 - Look for optional or conditional behavior related to a base use case. Model this as a separate use case connected via an «extend» relationship to the base use case. Define extension points in the base use case if necessary (often done in text descriptions).
- 7.
8. **Identify Generalization Relationships (Actors and Use Cases):**
 - Look for actors that share common roles or use cases that represent variations of a more general function. Use generalization arrows to show these relationships (child points to parent).
- 9.
10. **Review and Refine:** Check the diagram for clarity, completeness, and correctness. Ensure names are meaningful and follow conventions. Is the diagram easy to understand? Does it accurately reflect the system's intended functionality from the user's perspective?

3. Required Diagrams (Example for Financial Management System):

(Based on the diagram in the provided OCR document)

graph TD

subgraph Financial Management **System** (FMS)

UC1(Manage **User** Roles)

UC2(Login/Logout)

UC3(Export Reports)

UC4(Generate Financial Reports)

UC5(Track Audit Logs)

UC6(Generate Compliance Reports)

UC7(**View** Financial Transactions)

UC8(Generate Invoices)

UC9(Email Notification)

UC10(Send Invoices **to** Clients)

UC11(Validate Transactions)

UC12(Update Payment Status)

UC13(**Add** Payment Details)

UC14(Process Payment via **External** Gateway)

UC15(Verify Payment Details)

UC16(Restrict Unauthorized Access)

UC17(**User** Authentication **&** Access Control)

UC18(Report Generation)

UC19(Audit **&** Compliance)

UC20(Invoice Management)

UC21(Error Handling **for** Invalid Transactions)

UC22(Payment Management)

end

Admin(Administrator) --> UC1

Admin --> UC16

Admin --> UC17

Auditor(Auditor) --> UC19

Auditor --> UC5

Auditor --> UC6

FinMgr(Finance Manager) --> UC20

FinMgr --> UC18

AccStaff(Accounting Staff) --> UC22

AccStaff --> UC11

ExtPayGW(External Payment Gateway) -- Association --- UC14

% Relationships from OCR diagram

UC17 -- include --> UC2; % User Auth includes Login/logout conceptually

UC18 -- include --> UC4; % Report Gen includes Generate Fin Reports

UC18 -- include --> UC3; % Report Gen includes Export Reports

UC19 -- include --> UC5; % Audit & Comp includes Track Audit Logs

UC19 -- include --> UC6; % Audit & Comp includes Generate Comp Reports

UC20 -- include --> UC8; % Invoice Mgmt includes Generate Invoices

UC20 -- include --> UC10; % Invoice Mgmt includes Send Invoices

UC10 -- include --> UC9; % Send Invoices includes Email Notification

UC22 -- include --> UC11; % Payment Mgmt includes Validate Transactions

UC22 -- include --> UC13; % Payment Mgmt includes Add Payment Details

UC22 -- include --> UC12; % Payment Mgmt includes Update Payment Status

UC14 -- include --> UC15; % Process Payment includes Verify Payment Details

UC11 -- extend --> UC21; % Error Handling extends Validate Transactions

UC16 -- extend --> UC17; % Restrict Access extends User Authentication

% Connections from Actors to Higher-Level Use Cases (based on functionality)

Admin --> UC18 % Admin likely involved in overall Report Generation setup/access

FinMgr --> UC7 % Fin Mgr Views Transactions

FinMgr --> UC18 % Fin Mgr uses Report Generation

AccStaff --> UC7 % Acc Staff Views Transactions

AccStaff --> UC20 % Acc Staff involved in Invoice Management

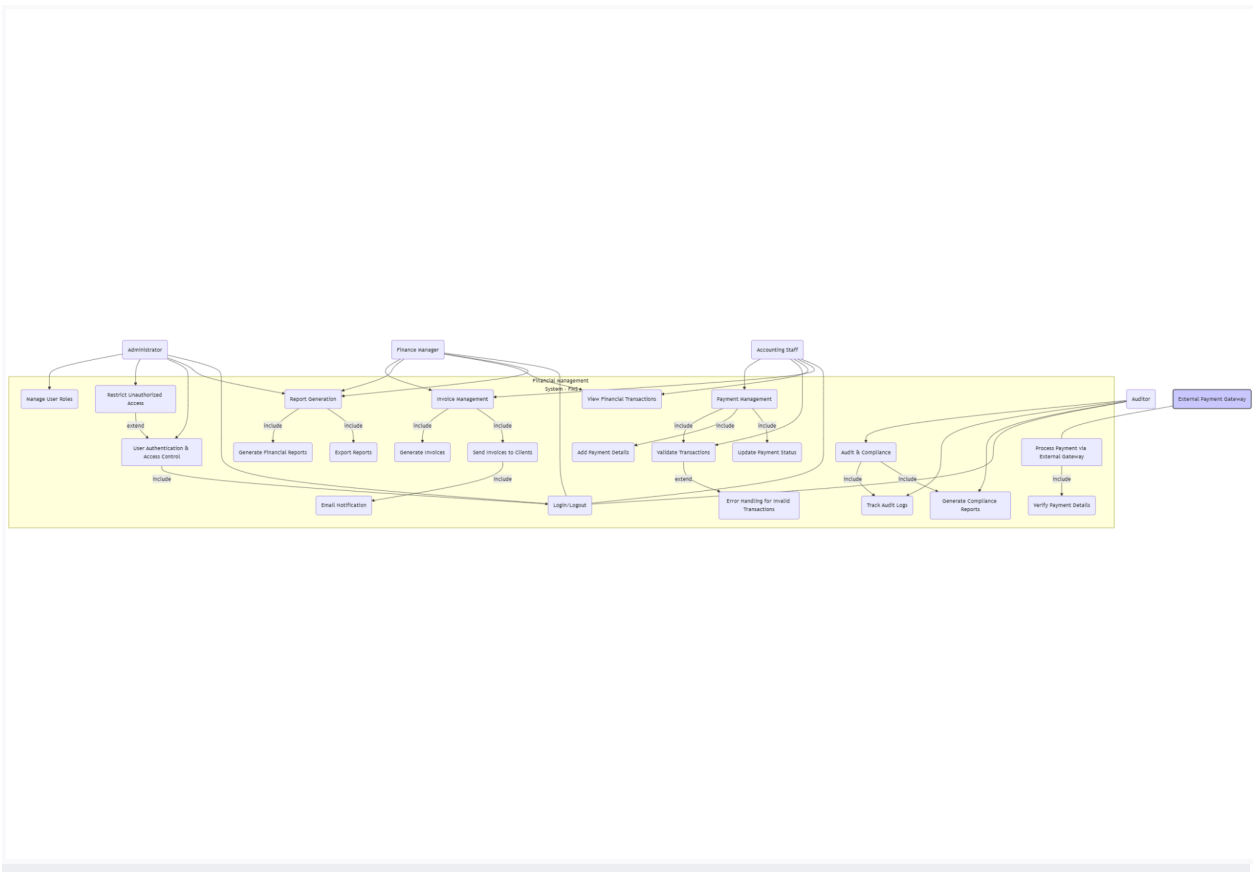
% Connecting relevant primary actors

Admin --- UC2 % Admin needs to login

Auditor --- UC2 % Auditor needs to login

FinMgr --- UC2 % Fin Mgr needs to login

AccStaff --- UC2 % Acc Staff needs to login



- Explanation:** This diagram captures the actors (Administrator, Auditor, Finance Manager, Accounting Staff, External Payment Gateway) interacting with the Financial Management System. Various functionalities are shown as use cases (ellipses) inside the system boundary. Lines show which actors participate in which use cases. Dashed arrows show «include» (for mandatory shared behavior like Login/Logout being part of User Authentication) and «extend» (for optional behavior like Error Handling extending Transaction Validation). *Note: The OCR diagram grouped related use cases visually (e.g., Report Generation, Audit & Compliance); I've represented these as higher-level conceptual use cases with include relationships to their components for clarity.*

4. Examples and Real-World Applications:

- Online Shopping:** Actors: Customer, Seller, Admin, Payment Gateway. Use Cases: Browse Products, Add to Cart, Checkout, Process Payment, Manage Inventory, Generate Sales Report.
- ATM System:** Actors: Customer, Bank System, Technician. Use Cases: Withdraw Cash, Check Balance, Deposit Funds, Transfer Funds, Perform Maintenance.

- **Library System:** Actors: Member, Librarian, System Admin. Use Cases: Search Book, Borrow Book, Return Book, Manage Members, Add Book Catalog.
- **Any system where user interaction defines core functionality.**

5. Viva Questions (Basic to Toughest):

1. What is a Use Case Diagram?

- *Answer:* It's a UML diagram that shows the interactions between actors (users or external systems) and a system, illustrating the system's functionalities (use cases) from the user's perspective.

2.

3. What are the main components of a Use Case Diagram?

- *Answer:* Actor, Use Case, System Boundary, and Relationships (Association, Include, Extend, Generalization).

4.

5. What is an Actor? Give examples.

- *Answer:* An Actor represents a role interacting with the system to achieve a goal. Examples: Customer, Administrator, Bank System, Timer (for time-triggered events).

6.

7. What is a Use Case? How should it be named?

- *Answer:* A Use Case represents a specific functionality the system provides to an actor, resulting in value. It should be named using a Verb-Noun phrase (e.g., "Place Order").

8.

9. What does the System Boundary represent?

- *Answer:* It represents the scope of the system being modeled, separating internal use cases from external actors.

10.

11. What is the difference between Primary and Supporting Actors?

- *Answer:* Primary actors initiate interaction with the system to achieve their goals (e.g., user withdrawing cash). Supporting actors provide services to the system, often initiated by the system (e.g., bank system providing authentication).

12.

13. Explain the «include» relationship with an example.

- *Answer:* An «include» relationship signifies that one use case (the included one) is *always* part of another (the base one). It's used for common/shared behavior. Example: "Login" is included by "Check Balance" and "Transfer Funds".

14.

15. **Explain the «extend» relationship with an example.**

- *Answer:* An «extend» relationship signifies that one use case (the extending one) *conditionally* adds behavior to another (the base one) at an extension point. It's used for optional or exceptional behavior. Example: "Apply Discount Code" extends "Checkout" only if the user provides a valid code.

16.

17. **Explain the Generalization relationship between Actors or Use Cases.**

- *Answer:* Generalization shows an "is-a-kind-of" relationship. A specific actor/use case (child) inherits properties/behavior from a general one (parent) and may add its own specifics. Example: "Registered User" and "Guest User" could be specializations of "User" actor.

18.

19. **What is the purpose of drawing Use Case diagrams?**

- *Answer:* To capture functional requirements, define system scope, understand user interactions, facilitate communication, and provide a basis for further design and testing.

20.

21. **Can an Actor interact directly with another Actor in a Use Case diagram?**

- *Answer:* No. Use Case diagrams model interactions *with the system*. Actor-to-actor communication happens outside the system boundary.

22.

23. **Can one Use Case interact directly with another Use Case without an Actor?**

- *Answer:* Only through «include», «extend», or generalization relationships. Direct association (solid line) typically links actors to use cases.

24.

25. **How detailed should Use Cases be on the diagram?**

- *Answer:* The diagram provides a high-level overview. The name should be concise (Verb-Noun). Detailed steps, pre-conditions, post-conditions, and alternative flows are documented separately in Use Case Specifications or Descriptions.

26.

27. **How do Use Case diagrams relate to functional requirements?**

- *Answer:* Use Case diagrams are a primary way to visualize and organize functional requirements. Each use case represents a specific function the system must perform for an actor. The collection of use cases largely defines the functional scope.

28.

29. **When modeling a login process, is "Login" best represented as its own use case included by others, or just as a pre-condition mentioned in text?**

- *Answer:* It's commonly modeled as a separate "Login" use case that is «include»d by other use cases requiring authentication. This promotes reuse and clearly shows the dependency. Listing it only as a pre-condition is less explicit on the diagram, though valid in the textual description. Modeling it as an included use case is generally preferred for clarity on the diagram.

30.

6. Common Mistakes & How to Avoid Them:

- **Mistake:** Use Cases too granular (e.g., "Enter Username", "Enter Password" instead of "Login").
 - **Avoidance:** Focus on the actor's *goal* that provides value. Combine trivial steps into a single, goal-oriented use case.
-
- **Mistake:** Confusing «include» and «extend».
 - **Avoidance:** Remember: «include» is for *mandatory*, shared behavior (base needs included). «extend» is for *optional*, conditional behavior (base is complete without extension). Arrow direction is crucial: Base -> Include; Extend -> Base.
-
- **Mistake:** Actors representing specific people instead of roles (e.g., "John Smith" instead of "Customer").
 - **Avoidance:** Actors define roles. Use role names.
-
- **Mistake:** Use Cases named as nouns or vague phrases (e.g., "Reports", "Manage").
 - **Avoidance:** Use strong Verb-Noun phrases describing the goal (e.g., "Generate Sales Report", "Manage User Accounts").
-
- **Mistake:** Drawing flows/sequences inside the diagram.
 - **Avoidance:** Use Case diagrams show *what* actors can do, not the *order* or *how*. Sequence details belong in Activity diagrams, Sequence diagrams, or textual descriptions.
-
- **Mistake:** Overuse of «include»/«extend», making the diagram cluttered.
 - **Avoidance:** Use these relationships judiciously to clarify structure, not for every minor variation. Sometimes generalization or simple association is sufficient. Prioritize readability.
-

7. Documentation:

- **Use Case Diagram:** The graphical representation itself.
- **Use Case Specification / Description:** A detailed textual document for each use case shown on the diagram. It typically includes:
 - **Use Case Name:** (Matching the diagram)
 - **Use Case ID:** Unique identifier.
 - **Actors:** Primary and supporting actors involved.
 - **Description:** Brief summary of the use case goal.
 - **Trigger:** Event that initiates the use case.
 - **Preconditions:** Conditions that must be true before the use case can start.
 - **Postconditions:** Conditions that must be true after the use case completes successfully.
 - **Normal Flow (Basic Path):** Step-by-step description of the primary successful scenario.
 - **Alternative Flows:** Steps for other valid scenarios or deviations.
 - **Exception Flows:** Steps for handling errors or conditions preventing success.
 - **Includes:** Reference to included use cases.
 - **Extends:** Reference to extending use cases.
 - **Extension Points:** (For base use cases being extended) Specific points where behavior can be added.
 - **Non-Functional Requirements:** (Optional, related performance, security, etc.)
 - **Notes/Issues:** Other relevant information.

-

Okay, let's proceed to the next experiment.

Experiment No. 05: Implement State/Activity diagrams

Learning Objective:

To implement the dynamic view of a system using UML Activity diagrams and State Machine diagrams (also known as Statechart diagrams). Students will be able to model workflows and object lifecycles.

Tools Used:

- **Conceptual Tools:** UML Dynamic Modeling, State Machine Concepts (State, Transition, Event, Guard, Action), Activity Diagram Concepts (Action, Flow, Decision Node, Merge Node, Fork Node, Join Node, Object Node, Partition/Swimlane).
- **Software Tools:** MS Word (as mentioned), Draw.io (as mentioned), StarUML, Lucidchart, Visual Paradigm, Enterprise Architect, or any UML modeling tool supporting these diagrams.

1. Full Theory:

A. State Machine Diagrams (Statechart Diagrams):

- **Definition:** A State Machine diagram (often called a Statechart diagram) is a UML behavioral diagram that models the behavior of a *single object*, specifying the sequence of states an object goes through during its lifetime in response to events, along with its responses and actions. It describes *how an object changes state*.
- **Purpose:**
 - To model the lifecycle of an object, from creation to destruction.
 - To describe how an object responds differently to the same event depending on its current state.
 - To model reactive systems (systems that respond to external or internal events).
 - Useful for modeling objects with complex, event-driven behavior (e.g., UI elements, device controllers, transaction processes).
-
- **Key Components:**
 - **State:** A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
 - **Representation:** Rectangle with rounded corners. (As shown in document Figure-01).
 - **Naming:** Descriptive name indicating the condition (e.g., Idle, Processing, Waiting for Input, Order Shipped).
 - **Types:**
 - **Initial State:** The state an object is in when created. **Representation:** Solid filled circle. (Figure-01). Only one per diagram region.
 - **Final State:** Indicates the object has completed its lifecycle. **Representation:** Circle surrounding a smaller solid filled circle (bull's eye). (Figure-01). Can be multiple.
 - **Simple State:** A state with no nested regions or substates.

- **Composite State:** A state that contains nested states (substates). Useful for modeling complex states with internal structure. (Not detailed in the provided text but important).
 -
 - *State Compartments (for Intermediate States - Figure-02):*
 - **Name Compartment:** Contains the state name.
 - **Internal Activities/Transitions Compartment:** Contains activities performed while the object is in this state. Common labels include:
 - `entry / action:` Action performed upon entering the state.
 - `exit / action:` Action performed upon exiting the state.
 - `do / activity:` Ongoing activity performed while in the state.
 - `event / action:` Action performed in response to an internal event without changing state.
 -
-
- **Transition:** A relationship between two states indicating that an object in the first state (source) will perform certain actions and enter the second state (target) when a specified event occurs and specified guard conditions are satisfied.
 - *Representation:* Solid arrow from the source state to the target state.
 - *Syntax (Label on arrow):* `event [guardCondition] / actionExpression`
 - `event:` The trigger that causes the transition (e.g., `buttonClicked`, `dataReceived`, `timeout`). Mandatory for most transitions (except completion transitions).
 - `guardCondition:` (Optional) A boolean expression that must be true for the transition to fire when the event occurs. Enclosed in square brackets `[]` (e.g., `[order valid]`, `[attempts < 3]`).
 - `actionExpression:` (Optional) An action or behavior executed *during* the transition. Preceded by a forward slash `/` (e.g., `/ incrementCounter()`, `/ logError()`).
 -
 - *Self-Transition:* A transition where the source and target state are the same. Useful for handling an event that doesn't cause a state change but triggers an action.
-

- **Event:** An occurrence that may trigger a state transition (e.g., receiving a signal, a method call, passage of time).
- **Action:** An executable, atomic computation that results in a change in state of the model or the return of a value. Actions are associated with transitions (/ action) or states (entry, exit, do). They run to completion without interruption.
- **Activity:** A non-atomic behavior that executes while an object is in a state (do / activity). Can be interrupted by events.

•

B. Activity Diagrams:

- **Definition:** An Activity Diagram is a UML behavioral diagram that depicts high-level business workflows or operational workflows, showing the flow of control or data between various actions or activities. It's essentially an advanced flowchart suitable for modeling processes.
- **Purpose:**
 - To model business processes or workflows.
 - To describe the steps in a use case.
 - To model complex sequential algorithms or parallel operations.
 - To show how different activities coordinate to provide a service.
-
- **Key Components:**
 - **Action/Activity:** Represents a single step or task within the workflow. A fundamental unit of behavior.
 - *Representation:* Rectangle with rounded corners (same symbol as state, but used differently). (Table-01).
 - *Naming:* Verb or verb phrase describing the task (e.g., Submit Order, Calculate Total, Validate Input).
 -
 - **Control Flow (Edge):** An arrow showing the sequence between actions. Control moves from one action to the next when the first action completes.
 - *Representation:* Solid arrow. (Table-01).
 - *Guard Condition:* (Optional) A boolean condition in square brackets [] placed on a control flow leaving a decision node, indicating that the flow is taken only if the condition is true (e.g., [valid data], [error detected]).
 -
 - **Initial Node:** Marks the starting point of the activity flow.

- **Representation:** Solid filled circle (same as state initial node). (Table-01). An activity can have multiple initial nodes.
-
- **Final Node:** Marks the end point(s) of the activity flow.
 - **Activity Final Node:** Stops the specific flow reaching it. **Representation:** Circle surrounding a solid filled circle (bull's eye, same as state final node). (Table-01).
 - **Flow Final Node:** (Less common) Terminates a single flow, doesn't stop the whole activity if parallel flows exist. **Representation:** Circle with an 'X' inside.
-
- **Decision Node:** Represents a conditional branch point. It has one incoming flow and multiple outgoing flows, each with a guard condition (though guards can sometimes be implicit or described in a note).
 - **Representation:** Diamond shape. (Table-01).
-
- **Merge Node:** Brings together multiple alternative flows originating from a decision node. It has multiple incoming flows and one outgoing flow. Control passes to the outgoing flow whenever *any* incoming flow arrives.
 - **Representation:** Diamond shape (same as decision node, but context differs: multiple inputs, one output). (Table-01).
-
- **Fork Node:** Splits a single incoming flow into multiple concurrent (parallel) flows.
 - **Representation:** Solid black bar with one incoming arrow and multiple outgoing arrows. (Table-01).
-
- **Join Node:** Synchronizes multiple concurrent flows. Control passes to the single outgoing flow only when *all* incoming flows have arrived at the join node.
 - **Representation:** Solid black bar with multiple incoming arrows and one outgoing arrow. (Table-01).
-
- **Partition (Swimlane):** A visual grouping mechanism (vertical or horizontal lanes) used to organize actions based on the responsible party (e.g., actor, department, system component). Helps clarify who does what.
 - **Representation:** Rectangular regions dividing the diagram. (Table-01, implicitly used in document Figure-05 User/System lanes).
-

- **Object Node:** Represents an object or data item flowing through the activity. Shows how data is passed between actions.
 - *Representation:* Rectangle.
 - *Object Flow:* Dashed arrow showing the flow of an object/data between an action and an object node, or between two object nodes.
-
-

C. State Machine Diagram vs. Activity Diagram:

Feature	State Machine Diagram	Activity Diagram
Focus	Lifecycle/states of a <i>single object</i>	<i>Workflow/process</i> involving multiple actions/objects
Flow Driven By	Events	Activity completion (control flow) / Data flow
Concurrency	Can be modeled (composite states)	Explicitly modeled (Fork/Join)
Main Use	Modeling reactive behavior, object state	Modeling business processes, use cases, algorithms
Resembles	State Transition Diagram	Flowchart (advanced)

2. Step-by-Step Procedure:

A. Creating State Machine Diagrams:

1. **Identify the Object:** Choose the object whose lifecycle you want to model (e.g., `Order`, `UserSession`, `ATM_Machine`).
2. **Identify States:** Determine the significant, distinct states the object can be in during its lifetime (e.g., `Order: Pending, Confirmed, Shipped, Delivered, Cancelled`).
3. **Identify Initial and Final States:** Determine the starting state upon creation and any states indicating completion/termination.

4. **Identify Events:** What internal or external events can cause the object to change state? (e.g., `confirmOrder`, `shipItems`, `paymentReceived`, `cancelButton`).
5. **Identify Transitions:** For each state, determine which events trigger transitions *out* of that state and what the target state is.
6. **Identify Guard Conditions:** For each transition, determine if there are conditions that must be met for it to fire (e.g., `[payment successful]` for transition from `Pending` to `Confirmed`).
7. **Identify Actions:** Determine actions performed *during* transitions (`/ action`) or upon state entry/exit or do activities within states.
8. **Draw the Diagram:**
 - Place the initial state.
 - Draw the main states.
 - Draw transitions as arrows between states, labeling them with `event [guard] / action`.
 - Add entry/exit/do actions inside state symbols if needed.
 - Add final states where appropriate.
- 9.
10. **Review and Refine:** Trace the object's lifecycle through the diagram for different event sequences. Ensure all significant states and transitions are captured.

B. Creating Activity Diagrams:

1. **Define Scope:** Determine the start and end points of the workflow/process being modeled (e.g., from `User submits request` to `Request fulfilled`).
2. **Identify Activities/Actions:** List the major steps or tasks involved in the workflow.
3. **Identify Sequence:** Determine the order in which actions typically occur. Connect them with control flow arrows.
4. **Identify Decisions:** Where does the flow branch based on conditions? Add Decision nodes and label outgoing flows with guard conditions `[]`. Add Merge nodes where alternative paths reconverge.
5. **Identify Parallelism:** Are there steps that can happen concurrently? Use Fork nodes to split the flow and Join nodes to synchronize them back together.
6. **Identify Actors/Responsibilities (for Swimlanes):** Determine who or what performs each action. Create partitions (swimlanes) for each responsible party and place actions within the correct lane.

7. **Identify Object Flows (Optional):** Determine what data/objects are produced or consumed by actions. Add Object nodes and connect them using object flow arrows (dashed) to show data movement.
8. **Add Initial and Final Nodes:** Mark the start(s) and end(s) of the activity.
9. **Draw the Diagram:** Assemble the components using standard UML notation. Arrange elements for clarity, minimizing crossed lines.
10. **Review and Refine:** Trace the workflow through the diagram. Is the logic correct? Are conditions clear? Is parallelism/synchronization correct? Is it easy to understand?

3. Required Diagrams (Examples for Financial Management / Fitness System):

a. State Machine Diagram (Example: User Session State)

stateDiagram-v2

```
[*] --> NotLoggedIn : Application Starts

NotLoggedIn --> Authenticating : login(credentials)

Authenticating --> LoggedIn : [credentials valid] / createSession()

Authenticating --> NotLoggedIn : [credentials invalid] / showError()

Authenticating --> NotLoggedIn : cancelLogin()

LoggedIn --> NotLoggedIn : logout() / destroySession()

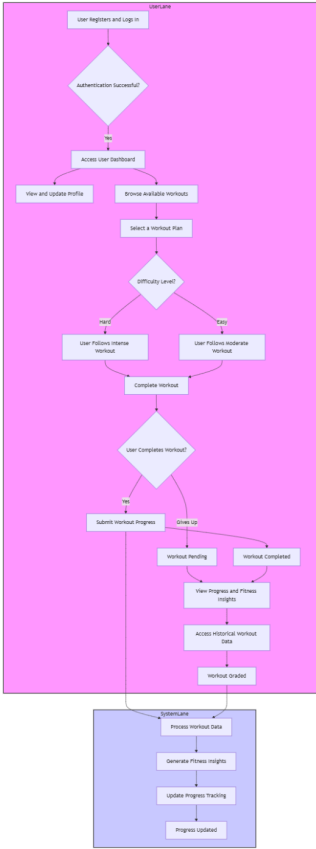
LoggedIn --> ProcessingRequest : submitRequest()

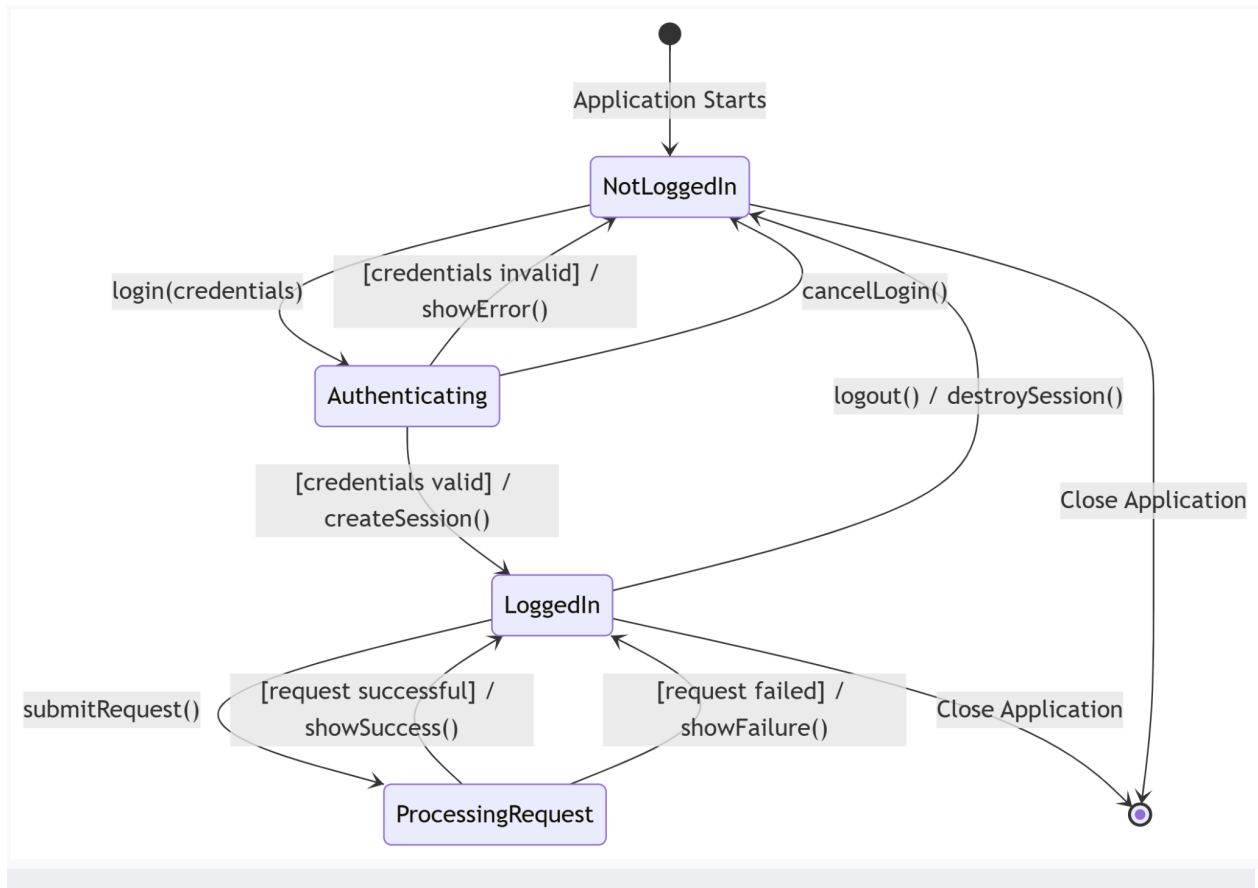
ProcessingRequest --> LoggedIn : [request successful] / showSuccess()

ProcessingRequest --> LoggedIn : [request failed] / showFailure()

LoggedIn --> [*] : Close Application

NotLoggedIn --> [*] : Close Application
```





- Explanation:** Models the state of a user's session. Starts ([*]) in NotLoggedIn. Entering credentials triggers a transition to Authenticating. Based on validation ([credentials valid]), it moves to LoggedIn (performing createSession) or back to NotLoggedIn. From LoggedIn, users can logout or submitRequest, moving temporarily to ProcessingRequest before returning to LoggedIn. Closing the application terminates the session ([*]).

b. Activity Diagram (Example: Workout Submission - from document Figure-05)

graph TD

subgraph User Lane

A[User Registers and Logs In] --> B{Authentication Successful?};

B -- Yes --> C[Access User Dashboard];

C --> D[View and Update Profile];

C --> E[Browse Available Workouts];

```
E --> F[Select a Workout Plan];  
  
F --> G{Difficulty Level?};  
  
G -- Hard --> H[User Follows Intense Workout];  
  
G -- Easy --> I[User Follows Moderate Workout];  
  
H --> J[Complete Workout];  
  
I --> J;  
  
J --> K{User Completes Workout?};  
  
K -- Yes --> L[Submit Workout Progress];  
  
K -- Gives Up --> M[Workout [Pending]];  
  
L --> N[Workout [Completed]];  
  
M --> O[View Progress and Fitness Insights];  
  
N --> O;  
  
O --> P[Access Historical Workout Data];  
  
P --> Q[Workout [Graded]];
```

end

subgraph **System Lane**

```
L --> S1[Process Workout Data];  
  
S1 --> S2[Generate Fitness Insights];  
  
S2 --> S3[Update Progress Tracking];  
  
S3 --> S4[Progress [Updated]];
```

end

% Connections **between** lanes

Q --> S1; % User submitting graded workout seems to trigger processing? The diagram is a bit ambiguous here. It might be L->S1 is the main trigger. Let's stick to L->S1 based on typical flow.

```
style User Lane fill:#f9f,stroke:#333,stroke-width:2px
```

```
style System Lane fill:#ccf,stroke:#333,stroke-width:2px
```

- **Explanation:** This diagram (based on the OCR'd Figure-05) models the workflow for a user selecting and completing a workout.
 - **Swimlanes:** Clearly separates actions performed by the User and the System.
 - **Initial Node:** Implied start at User Registers and Logs In.
 - **Actions:** Rounded rectangles like Access User Dashboard, Process Workout Data.
 - **Decision Nodes:** Diamonds like Authentication Successful?, Difficulty Level?, User Completes Workout?.
 - **Control Flows:** Arrows showing the sequence. Guard conditions (Yes, No, Hard, Easy) on flows leaving decision nodes.
 - **Final Node:** Implied end after Workout [Graded] or Progress [Updated].
 - *(Self-correction: The OCR diagram uses rectangles for states like*
-

4. Examples and Real-World Applications:

- **State Machines:**
 - UI element behavior (button states: Enabled, Disabled, Pressed).
 - Device controllers (Traffic light states: Red, Yellow, Green).
 - Parsing text or code (States: ReadingIdentifier, ReadingOperator).
 - Network protocols (TCP connection states: LISTEN, SYN_SENT, ESTABLISHED, CLOSED).
 - Transaction processing (Order states: Pending, Paid, Shipped).
-
- **Activity Diagrams:**
 - Modeling business processes (loan application, patient admission).
 - Detailing steps within a use case (how "Place Order" actually works).
 - Visualizing complex algorithms involving conditions and parallelism.

- Defining system workflows (document approval process).

•

5. Viva Questions (Basic to Toughest):

1. What is a State Machine diagram used for?

- *Answer:* To model the lifecycle of a single object, showing the sequence of states it passes through in response to events.

2.

3. What is an Activity diagram used for?

- *Answer:* To model workflows or processes, showing the flow of control and/or data between activities or actions.

4.

5. What is a 'State' in a State Machine diagram?

- *Answer:* A condition or situation in the life of an object where it exhibits certain behavior or waits for an event.

6.

7. What is an 'Action' or 'Activity' in an Activity diagram?

- *Answer:* A step or task performed within a workflow.

8.

9. What triggers a change from one state to another in a State Machine diagram?

- *Answer:* An Event, potentially qualified by a Guard condition.

10.

11. What do Fork and Join nodes represent in an Activity diagram?

- *Answer:* Fork splits one flow into multiple parallel flows. Join synchronizes multiple parallel flows back into one, waiting for all incoming flows to arrive.

12.

13. What do Decision and Merge nodes represent in an Activity diagram?

- *Answer:* Decision splits one flow into multiple alternative flows based on guard conditions. Merge brings multiple alternative flows back into a single flow.

14.

15. What is the difference between an 'Action' (in State Machines) and an 'Activity' (in State Machines or Activity Diagrams)?

- *Answer:* In State Machines, an Action (entry/exit/transition) is atomic and runs to completion. A `do / activity` within a state is non-atomic and can be interrupted. In Activity Diagrams, the terms Action and Activity are often used interchangeably for the steps in the workflow, though technically an Activity might be decomposable into Actions.

16.

17. **What are swimlanes (Partitions) used for in Activity diagrams?**

- *Answer:* To group actions according to the responsible party (actor, department, system), clarifying who performs which part of the workflow.

18.

19. **Can an object be in multiple states simultaneously in a standard UML State Machine?**

- *Answer:* In a basic state machine, no. However, UML supports composite states with *orthogonal regions*, where the object can be in one substate within each concurrent region simultaneously.

20.

21. **What is the purpose of the**

- *Answer:* `entry / action:` Performed once upon entering the state. `exit / action:` Performed once upon exiting the state. `do / activity:` Ongoing behavior performed while the object remains in the state.

22.

23. **How do object flows differ from control flows in Activity diagrams?**

- *Answer:* Control flows (solid arrows) show the sequence of execution between actions. Object flows (dashed arrows, often involving object nodes/rectangles) show the movement of data or objects being passed between actions.

24.

25. **When would you choose a State Machine diagram over an Activity diagram?**

- *Answer:* Choose a State Machine diagram when you need to model the event-driven behavior and changing states of a *single specific object* throughout its lifetime, especially if its response to events depends heavily on its current state. Choose an Activity diagram to model a *process or workflow* involving multiple steps, potentially multiple objects or actors, focusing on the sequence, conditions, and parallelism of actions.

26.

27. **Explain guard conditions and their role in both diagram types.**

- *Answer:* Guard conditions (`[condition]`) are boolean expressions. In State Machines, they qualify a transition – the transition only fires if the event occurs *and* the guard is true. In Activity Diagrams, they are placed on control flows leaving decision nodes, determining which path is taken based on whether the condition is true.

28.

29. **How can Activity diagrams represent the flow detailed within a single Use Case?**

- *Answer:* An Activity diagram can elaborate the "Normal Flow", "Alternative Flows", and "Exception Flows" described textually in a Use Case specification. The actions in the diagram correspond to the steps, decision nodes represent branching points based on conditions described in the flows, and swimlanes can show interactions between the initiating actor and the system.

30.

6. Common Mistakes & How to Avoid Them:

- **State Machines:**

- **Mistake:** Modeling a process instead of an object's state.
- **Avoidance:** Always ask "What object's lifecycle am I modeling?". If it's about workflow, use an Activity diagram.
- **Mistake:** Missing initial/final states.
- **Avoidance:** Ensure every state machine region has one initial state and appropriate final states.
- **Mistake:** Ambiguous transitions (same event triggering multiple transitions from one state without mutually exclusive guards).
- **Avoidance:** Ensure guards on competing transitions are mutually exclusive, or that the choice is deterministic.

-

- **Activity Diagrams:**

- **Mistake:** Confusing Decision/Merge and Fork/Join nodes.
- **Avoidance:** Remember Decision/Merge handle *alternative* paths (only one path taken). Fork/Join handle *parallel* paths (all paths run concurrently). Use the correct symbols (diamond vs. bar).
- **Mistake:** Unbalanced Fork/Join nodes (not all paths from a fork eventually reach the corresponding join).
- **Avoidance:** Carefully trace all parallel paths to ensure they synchronize correctly at a join.
- **Mistake:** Missing guard conditions after a decision node.
- **Avoidance:** Every path leaving a decision node (except possibly one 'else' path) should have a condition [].
- **Mistake:** Using Activity diagram syntax for state concepts (e.g., trying to show events triggering flow directly).
- **Avoidance:** Stick to control flow/object flow logic. Flow progresses when an action completes.

-

7. Documentation:

- **State Machine Diagrams & Activity Diagrams:** These are key behavioral models often included in the **System Design Document (SDD)** or detailed requirements specifications.
 - **Context:** They help explain complex logic described in Use Case specifications or functional requirements.
 - **Accompanying Text:** Should explain the scope of the diagram (which object's state, which workflow), definitions of non-obvious states/actions, and clarify complex conditions or actions.
-

This covers Experiment 05 on State Machine and Activity diagrams. Please indicate when you're ready for the next experiment using **"Next"**.

Experiment No. 06: Sketch Sequence diagram for the project

Learning Objective:

Students will be able to draw Sequence diagrams for the project, illustrating object interactions arranged in time sequence for specific scenarios.

Tools Used:

- **Conceptual Tools:** UML Interaction Diagrams, Object Lifelines, Messages (Synchronous, Asynchronous, Reply, Create, Destroy), Activation Bars, Combined Fragments (Optional but powerful: loops, alternatives, parallel).
- **Software Tools:** Dia (mentioned), StarUML (mentioned), Draw.io, PlantUML, Lucidchart, Visual Paradigm, Enterprise Architect, or any UML modeling tool supporting Sequence diagrams.

1. Full Theory:

a. UML Interaction Diagrams:

- **Definition:** A category of UML behavioral diagrams used to model the interactions between objects in a system. They show how objects collaborate to perform a specific task or achieve a goal. Sequence diagrams are the most common type of interaction diagram.
- **Purpose:** To visualize the flow of messages exchanged between objects over time for a particular scenario or use case realization.

b. Sequence Diagram:

- **Definition:** A Sequence diagram shows object interactions arranged in time sequence. It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. It emphasizes the time ordering of messages.
- **Purpose:**
 - To model the logic of a use case or a specific scenario.
 - To understand how objects collaborate to achieve a goal.
 - To visualize the flow of control and message passing between objects.
 - To identify dependencies and potential bottlenecks in interactions.
 - To help design object methods and responsibilities.
-
- **Key Components:**
 - **Lifeline:** Represents an individual participant (an object or an instance of a class, or even an actor) in the interaction.
 - *Representation:* A vertical dashed line extending downwards. At the top is a rectangle containing the object's name and optionally its class (e.g., `myOrder:Order` or `:Order` for an anonymous object, or `Administrator` for an actor participant). (Document shows `OperatorPanel`, `ATM`, etc.).
 -
 - **Message:** Represents communication between objects (lifelines). Messages are sent from one lifeline to another (or to itself).
 - *Representation:* Horizontal arrow from the sender lifeline to the receiver lifeline. Time progresses downwards on the diagram.
 - *Types:*
 - **Synchronous Message (Call Message):** Sender waits for the receiver to process the message and return control (or a reply).
Representation: Solid line with a solid (filled) arrowhead. (e.g., `getInitialCash()`). Used for standard method calls.
 - **Asynchronous Message:** Sender sends the message and continues its execution without waiting for the receiver. *Representation:* Solid line with an open stick arrowhead (`->`). Used for triggering parallel processes, event notifications.
 - **Reply Message (Return Message):** Receiver returns a value or control back to the sender after processing a synchronous message.
Representation: Dashed line with an open stick arrowhead (`-->`)

pointing back from the receiver to the sender. Often optional if the return is obvious or implied by the activation ending. (e.g., `initialCash` reply in the document example).

- **Create Message («create»):** Sender creates an instance of another class. *Representation:* Dashed arrow pointing to the rectangle (head) of the newly created object's lifeline. Often labeled «create».
- **Destroy Message («destroy»):** Sender terminates the existence of the receiver object. *Representation:* Arrow pointing to the receiver lifeline, ending with a large 'X' on the lifeline. Often labeled «destroy». (Document mentions this).

■

○

- **Activation Bar (Execution Specification):** Represents the period during which an object is performing an action, either directly or through a subordinate procedure. It shows when the object's method is active (on the call stack).

- *Representation:* Thin rectangle drawn on top of a lifeline. The message arrow typically starts/ends at the top of an activation bar. A reply message often originates from the bottom of the activation bar.

○

- **Self-Message (Recursive Message):** A message an object sends to itself, indicating invocation of its own method.

- *Representation:* Message arrow starting and ending on the same lifeline, usually creating a nested activation bar. (Document mentions this).

○

- **Combined Fragment:** Rectangular frames used to group sets of messages to show conditional or iterative execution. (Not explicitly in the document example, but essential for complex scenarios).

- *Types:*

- `opt`: Optional fragment (executes only if a guard condition is true).
- `alt`: Alternative fragment (like if-else; only one operand executes based on guard conditions).
- `loop`: Loop fragment (executes multiple times based on a condition).
- `par`: Parallel fragment (operands execute in parallel).
- `ref`: Reference to another existing sequence diagram.

■

- *Representation:* Large rectangle enclosing a portion of the diagram, with an operator label (e.g., `alt`, `loop`) in the top-left corner. Operands are separated

by dashed horizontal lines. Guard conditions `[]` are used within `opt`, `alt`, `loop`.

○

●

2. Step-by-Step Procedure (Creating Sequence Diagrams):

1. **Identify the Scenario:** Choose a specific use case or scenario to model (e.g., "Successful User Login", "Withdraw Cash Fails due to Insufficient Funds", "Generate Monthly Sales Report").
2. **Identify Participants:** Determine which objects or actors are involved in this specific scenario. These will become the lifelines.
3. **Establish Lifelines:** Draw a lifeline (rectangle head, dashed vertical line) for each participant across the top of the diagram. Place the primary actor/initiator usually on the left.
4. **Map Message Sequence:** Starting from the initiating event/message:
 - Draw the first message arrow from the sender lifeline to the receiver lifeline. Label the message appropriately (e.g., method call `login(user, pass)`).
 - Draw an activation bar on the receiver's lifeline to show it's processing the message.
 - If the receiver calls another object's method, draw a subsequent message arrow from the receiver's activation bar to the next object's lifeline, starting a nested activation bar there.
 - If a message is a reply, draw a dashed arrow back to the original caller, often coinciding with the end of the receiver's activation bar.
 - Continue drawing messages sequentially down the page, reflecting the time order of interactions.
- 5.
6. **Add Activation Bars:** Ensure activation bars clearly show the duration of execution for each message receiver. Nested calls result in nested activation bars.
7. **Use Combined Fragments (if needed):** If the scenario involves conditions (`if-else`), optional steps, or loops, enclose the relevant messages within `alt`, `opt`, or `loop` combined fragments, respectively. Add guard conditions `[]` where necessary.
8. **Show Object Creation/Destruction (if applicable):** Use create/destroy messages and symbols if objects are created or destroyed during the scenario.
9. **Review and Refine:** Trace the message flow for the scenario. Does it accurately represent the interactions? Are messages labeled clearly? Is the time sequence correct? Is it easy to understand?

3. Required Diagrams (Example for Financial Management System):

(Based on the diagram in the provided OCR document, Figure for "Financial Management System (FMS) Sequence Diagram")

sequenceDiagram

participant Admin as Administrator

participant Auditor

participant FinMgr as FinanceManager

participant AccStaff as AccountingStaff

participant ExtPayGW as ExternalPaymentGateway

participant FMS as FinancialManagementSystem (FMS)

%% Login Scenario (Example - simplified from OCR which combines many scenarios)

Admin ->> FMS: Login(username, password)

activate FMS

FMS ->> FMS: Verify Credentials

activate FMS #DarkGray

FMS -->> Admin: Login Success / Failure

deactivate FMS #DarkGray

deactivate FMS

%% Generate Reports Scenario (Example)

FinMgr ->> FMS: Generate Reports(type='Sales')

activate FMS

FMS ->> FMS: Fetch Report Data

activate FMS #DarkGray

FMS -->> FMS: Report Data

deactivate FMS #DarkGray

FMS -->> FinMgr: Display Reports(Report Data)

activate FinMgr

FinMgr -->> FMS: Export Reports(format='PDF')

FMS -->> FMS: Format Data for Export

activate FMS #DarkGray

FMS -->> FinMgr: Formatted Report File

deactivate FMS #DarkGray

deactivate FMS

deactivate FinMgr

%% Add Payment Details Scenario (Example - simplified from OCR)

AccStaff -->> FMS: Add Payment Details(details)

activate FMS

FMS -->> FMS: Validate Transactions(details)

activate FMS #DarkGray

alt [Validation OK]

FMS -->> FMS: Update Payment Status(status='Pending')

activate FMS #LightGrey

FMS -->> AccStaff: Confirmation (Details Added)

deactivate FMS #LightGrey

else [Validation Failed]

FMS -->> AccStaff: Error Message (Invalid Details)

end

deactivate FMS #DarkGray

deactivate FMS

%% Process Payment Scenario (Example - simplified from OCR)

AccStaff ->> FMS: Process Payment(paymentId)

activate FMS

FMS ->> ExtPayGW: Process Payment(paymentDetails)

activate ExtPayGW

ExtPayGW -->> FMS: Payment Result(success/fail)

deactivate ExtPayGW

FMS ->> FMS: Confirm Payment(result)

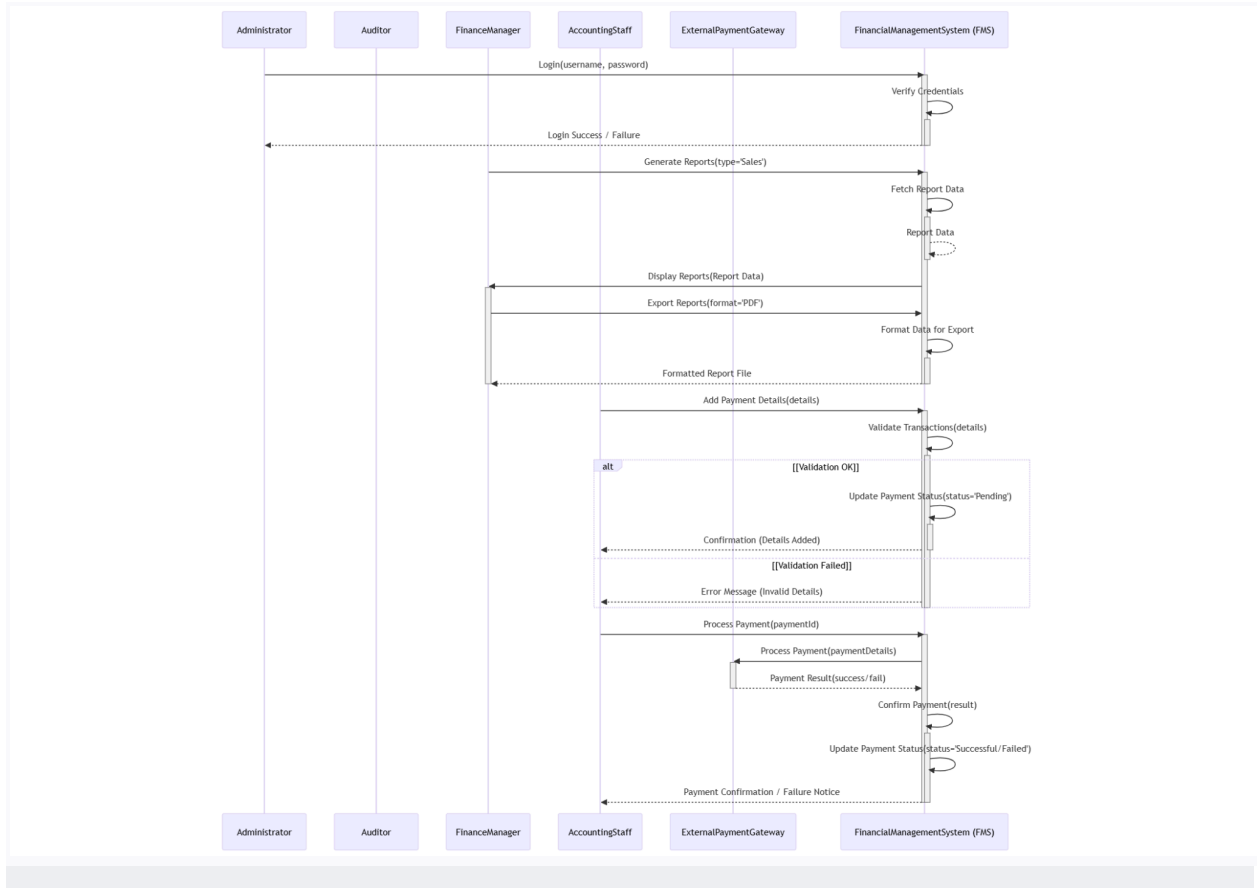
activate FMS #DarkGray

FMS ->> FMS: Update Payment Status(status='Successful/Failed')

FMS -->> AccStaff: Payment Confirmation / Failure Notice

deactivate FMS #DarkGray

deactivate FMS



- Explanation:** This diagram shows interactions for several simplified scenarios based on the OCR diagram:
 - Lifelines:** Represent the actors (Admin, Auditor, FinMgr, AccStaff, ExtPayGW) and the system (FMS).
 - Messages:** Arrows show calls like Login, Generate Reports, Add Payment Details. Synchronous calls have solid arrowheads. Replies (implicit or shown dashed) return results.
 - Activations:** Rectangles on lifelines show when an object/system is busy processing a message.
 - Combined Fragment** (Used in the "Add Payment Details" scenario to show the alternative paths based on validation outcome ([Validation OK] vs. [Validation Failed])).
 - (Self-correction: The OCR diagram attempts to show many scenarios on one diagram, which can become complex. Breaking it down into separate diagrams per key scenario is often clearer. My example above breaks it down).*

4. Examples and Real-World Applications:

- **Use Case Realization:** Detailing the step-by-step object interactions needed to fulfill a use case (e.g., How does the `Checkout` use case work? Show interactions between `ShoppingCart`, `Order`, `PaymentProcessor`, `Customer`).
- **Method Logic Design:** Visualizing the calls made within a complex method to other objects.
- **API Interaction:** Modeling the sequence of calls between a client application and a server API.
- **Distributed Systems:** Showing communication patterns between different services or components.
- **Troubleshooting:** Mapping out message flows to understand where a process might be failing.

5. Viva Questions (Basic to Toughest):

1. **What is a Sequence Diagram?**
 - *Answer:* A UML interaction diagram that shows how objects interact with each other by exchanging messages, arranged in a time sequence.
- 2.
3. **What does a Lifeline represent?**
 - *Answer:* An individual participant (object instance or actor) in the interaction, shown as a vertical dashed line.
- 4.
5. **What is a Message? Name different types.**
 - *Answer:* Communication between lifelines. Types include Synchronous (call), Asynchronous, Reply (return), Create, and Destroy messages.
- 6.
7. **How is time represented on a Sequence Diagram?**
 - *Answer:* Time progresses vertically downwards along the lifelines. Messages sent earlier appear higher up on the diagram.
- 8.
9. **What is an Activation Bar (Execution Specification)?**
 - *Answer:* A thin rectangle on a lifeline indicating the period during which an object is actively processing a message (its method is executing).
- 10.
11. **What is the difference between a Synchronous and an Asynchronous message? How are they shown?**
 - *Answer:* Synchronous: Sender waits for a reply. Shown with a solid line, filled arrowhead. Asynchronous: Sender does not wait. Shown with a solid line, open stick arrowhead (\rightarrow).

12.

13. What is the purpose of drawing Sequence Diagrams?

- *Answer:* To visualize object interactions for a specific scenario, understand message flow over time, detail use case logic, and aid in designing object responsibilities and methods.

14.

15. What is a Combined Fragment? Give examples of operators (

- *Answer:* A frame used to group messages for conditional or iterative execution. `alt` for alternatives (if-else), `loop` for repetition, `opt` for optional sequences (if).

16.

17. How would you model object creation and destruction?

- *Answer:* Creation: Use a «create» message (dashed arrow) pointing to the head/rectangle of the new object's lifeline. Destruction: Use a «destroy» message pointing to the lifeline, ending with a large 'X'.

18.

19. Can an actor lifeline receive messages?

- *Answer:* Yes. While actors typically initiate interactions, the system can send messages back to an actor lifeline (e.g., displaying information, asking for confirmation).

20.

21. How does a Sequence Diagram relate to a Use Case Diagram?

- *Answer:* A Sequence Diagram typically models *one specific scenario* or flow (e.g., the success path, or a specific error path) within a single Use Case. Multiple sequence diagrams might be needed to detail all important scenarios of one Use Case.

22.

23. How does a Sequence Diagram relate to a Class Diagram?

- *Answer:* A Class Diagram shows the static structure (classes, methods). A Sequence Diagram shows the dynamic interaction *using* instances of those classes and invoking the methods defined in the Class Diagram. Sequence diagrams help validate and refine the methods defined in the Class Diagram.

24.

25. What is the difference between a Sequence Diagram and a Communication Diagram (Collaboration Diagram)?

- *Answer:* Both are interaction diagrams. Sequence diagrams emphasize the *time ordering* of messages (vertical axis is time). Communication diagrams emphasize the

structural relationships or links between objects (layout is more flexible, sequence is shown by numbering messages).

26.

27. When should you use Combined Fragments versus drawing separate diagrams for different scenarios?

- *Answer:* Use Combined Fragments (`alt`, `opt`) for relatively simple conditional logic within a single scenario. If the alternative paths are very complex or represent fundamentally different scenarios (e.g., success vs. major failure path), drawing separate sequence diagrams often leads to clearer, less cluttered results.

28.

29. How can Sequence Diagrams help identify design problems?

- *Answer:* They can reveal issues like: an object doing too much work (many complex messages originating from it), excessive coupling (too many messages back and forth between two objects), performance bottlenecks (long sequences of synchronous calls), or incorrect object responsibility assignments.

30.

6. Common Mistakes & How to Avoid Them:

- **Mistake:** Trying to model complex logic/algorithms directly on the diagram.
 - **Avoidance:** Focus on *interactions* between objects. Use notes or refer to Activity diagrams for detailed algorithmic logic within a method.
-
- **Mistake:** Making diagrams too crowded by modeling every single getter/setter or trivial call.
 - **Avoidance:** Focus on the significant interactions that clarify the scenario's flow. Abstract away minor details unless they are crucial to understanding the sequence.
-
- **Mistake:** Incorrect arrow types (e.g., using synchronous for asynchronous calls or vice-versa).
 - **Avoidance:** Understand the semantics: Does the sender wait? Use solid arrowhead. Does it continue? Use open arrowhead. Is it a reply? Use dashed line.
-
- **Mistake:** Activation bars drawn incorrectly (not starting/ending with messages, overlapping incorrectly in nested calls).
 - **Avoidance:** Activation starts when a message arrives and ends when processing is done (often when a reply is sent or the last nested call returns). Nested calls have nested activation bars originating from within the caller's activation.
-

- **Mistake:** Trying to show too many scenarios on one diagram (common with the OCR example).
 - **Avoidance:** Create separate diagrams for distinct scenarios (e.g., successful login, failed login, report generation). Use `ref` fragments if needed to link them.
-
- **Mistake:** Lifelines representing classes instead of objects/instances.
 - **Avoidance:** Remember lifelines represent participants in a *specific interaction*. Use naming convention `instanceName:ClassName`. If the instance name isn't important, use `:ClassName`.
-

7. Documentation:

- Sequence Diagrams are behavioral models typically included in the **System Design Document (SDD)** or detailed Use Case realization documentation.
 - Each diagram should clearly state the **scenario** it represents.
 - They complement Class diagrams (static structure) and Use Case diagrams (functional overview) by showing dynamic behavior for specific instances.
-

Experiment No. 08: Change specification and use any SCM Tool to make different versions

Learning Objective:

Students will be able to understand the concepts of Software Configuration Management (SCM), handle change specifications, and use an SCM tool (like Git and GitHub) to manage different versions of project artifacts (e.g., source code).

Tools Used:

- **Conceptual Tools:** Software Configuration Management (SCM) principles, Configuration Items (CIs), Baseline, Version Control, Change Control Process, Branching, Merging.
- **Software Tools:** Git (Distributed Version Control System), GitHub / GitLab / Bitbucket (Hosting platforms for Git repositories), potentially a simple text editor or IDE to modify files.

1. Full Theory:

a. Software Configuration Management (SCM):

- **Definition:** SCM is a discipline in software engineering for systematically controlling the changes in a system and maintaining the integrity and traceability of the configuration items throughout the system's life cycle. It's about managing *evolution* and *change*.
- **Purpose:**
 1. To establish and maintain the integrity of software products.
 2. To manage changes effectively and prevent chaos.
 3. To ensure traceability between requirements, design, code, and tests.
 4. To support parallel development and teamwork.
 5. To enable reliable build and release management.
-
- **Key Activities/Procedures (as per document):**
 1. **Configuration Identification:** The process of identifying the components of the software system that need to be controlled (Configuration Items - CIs), assigning unique identifiers, and establishing relationships between them. CIs can be source code files, design documents, test cases, requirements specifications, build scripts, libraries, etc. Establishing a **baseline** (a formally agreed-upon version of one or more CIs) is a key part of identification.
 2. **Configuration Control:** The process of managing changes to the baselined CIs. This involves evaluating proposed changes (change requests), approving or rejecting them, and ensuring that approved changes are implemented correctly and documented. It prevents unauthorized or uncontrolled modifications.
 3. **Configuration Status Accounting:** The process of recording and reporting information about the CIs, baselines, approved changes, and the status of change requests throughout the project lifecycle. It answers questions like "What version is deployed?", "Which changes are in this build?".
 4. **Configuration Audits:** The process of verifying that the software product matches the configuration information recorded about it. Audits ensure that the procedures are being followed and that the delivered product contains all approved CIs and changes.
 - **Functional Configuration Audit (FCA):** Verifies that the CI's actual functionality and performance match the requirements specified in its documentation.
 - **Physical Configuration Audit (PCA):** Verifies that the CI, as built, corresponds to the technical documentation that defines it (e.g., correct files, versions).
 - 5.
-

b. Baseline:

- **Definition:** A baseline is a stable, formally reviewed, and agreed-upon version of a configuration item (or a set of CIs) that serves as a point of reference for further development. Once baselined, changes must go through the formal change control process.
- **Purpose:** Creates a known, stable state from which to manage future changes, ensuring consistency and control.

c. Version Control Systems (VCS):

- **Definition:** VCS (also known as Revision Control or Source Code Management - SCM tools) are software tools that help teams manage changes to source code (or other collections of files) over time. They track modifications, allowing users to revert to previous versions, compare changes, branch, and merge work.
- **Types:**
 - **Local VCS:** Keeps track of versions on a developer's local machine (e.g., RCS). Very limited for team collaboration.
 - **Centralized VCS (CVCS):** Uses a single central server to store all versions. Developers "check out" files from the server and "check in" changes. Examples: Subversion (SVN), CVS. *Weakness:* Single point of failure, requires network connection for most operations.
 - **Distributed VCS (DVCS):** Each developer has a full copy (clone) of the entire repository, including its history. Developers can commit changes locally and later synchronize with others. Examples: Git, Mercurial. *Strength:* No single point of failure, most operations are fast (local), better support for branching and merging, offline work possible.
-

d. Git:

- **Definition:** Git is a free and open-source distributed version control system designed for handling everything from small to very large projects with speed and efficiency. Created by Linus Torvalds.
- **Core Concepts:**
 - **Repository (Repo):** A database containing all the files, history, and configuration for a project. Can be local or remote.
 - **Working Directory:** The actual directory on your filesystem containing the project files you are currently working on.
 - **Staging Area (Index):** An intermediate area where you prepare changes before committing them. It allows you to select which modifications go into the next commit.

- **Commit:** A snapshot of the state of the repository at a specific point in time. Each commit has a unique ID (hash), an author, a timestamp, and a commit message describing the changes. Commits form the project history.
- **Branch:** A movable pointer to a specific commit, representing an independent line of development. Allows developers to work on features or fixes without affecting the main codebase (often the `main` or `master` branch).
- **Merge:** The process of combining changes from different branches back into one branch.
- **Clone:** Creating a full local copy of a remote repository.
- **Pull:** Fetching changes from a remote repository and merging them into the current local branch.
- **Push:** Sending local commits to a remote repository to share changes with others.

●

e. GitHub (and similar platforms like GitLab, Bitbucket):

- **Definition:** GitHub is a web-based hosting service for Git repositories. It provides a graphical interface, access control, and several collaboration features on top of the core Git functionality.
- **Key Features (beyond Git hosting):**
 - **Pull Requests (PRs):** A mechanism for proposing changes from one branch to another (often from a feature branch to `main`). Allows for code review, discussion, and automated checks before merging.
 - **Issue Tracking:** Managing tasks, bugs, feature requests.
 - **Collaboration:** Wikis, project boards (Kanban style), team management.
 - **Code Review:** Commenting on code changes within Pull Requests.
 - **CI/CD Integration (GitHub Actions):** Automating build, test, and deployment pipelines.
-
- **Git vs. GitHub:** Git is the underlying DVCS tool (command-line or GUI). GitHub is a platform that hosts Git repositories and provides collaboration tools around them. You use Git *with* GitHub.

f. Change Specification & Control Process (in context of SCM):

1. **Need for Change Identified:** A bug is found, a new feature is requested, or an improvement is suggested.

2. **Change Request (CR):** The proposed change is formally documented (e.g., in an issue tracker). It describes the change needed, the reason, the affected CIs, and priority.
3. **Evaluation:** The CR is reviewed (e.g., by a Change Control Board - CCB or project manager) for technical feasibility, impact on schedule/cost, alignment with project goals.
4. **Approval/Rejection:** The CR is approved or rejected.
5. **Implementation (using SCM tool):**
 - Developer typically creates a new branch in Git (e.g., `feature/add-login` or `bugfix/fix-calc-error`).
 - Makes the necessary code changes on this branch.
 - Commits changes frequently with clear messages related to the CR.
 - Tests the changes locally.
- 6.
7. **Verification/Review:**
 - Changes are often reviewed (e.g., via a Pull Request on GitHub).
 - Automated tests (CI) may run.
 - QA team may test the changes on the branch or a staging environment.
- 8.
9. **Merge:** Once approved and verified, the changes from the branch are merged back into the main development line (e.g., `main` or `develop` branch).
10. **Baseline Update:** The new version of the main branch may become part of a new baseline.
11. **Status Update:** The CR status is updated (e.g., "Closed", "Implemented"). Configuration Status Accounting records the change.

2. Step-by-Step Procedure (Using Git/GitHub for Versioning):

(Illustrates creating Version 1 and Version 2 of the BMS example from OCR)

1. **Setup:**
 - Install Git on your local machine.
 - Create a GitHub account (if using GitHub).
 - Create a new repository on GitHub (e.g., `BankingManagementSystem`).
 - Clone the empty repository to your local machine: `git clone <repository_url>`
 - Navigate into the repository directory: `cd BankingManagementSystem`
- 2.
3. **Create Version 1:**
 - Create the initial HTML file (e.g., `index.html`) with the Version 1 code (as shown in OCR Exp 8, Page 2).
 - **Stage the file:** `git add index.html` (or `git add .` to add all new/modified files).

- **Commit the file:** `git commit -m "Initial commit: Add Version 1 of BMS HTML and basic JS"` (Clear commit message).
- **Push to remote:** `git push origin main` (or `master`, depending on your default branch name). *Version 1 is now saved.*

4.

5. **Prepare for Version 2 (Start new work):**

- **Create a new branch:** `git branch enhance-validation-ui` (Create a branch named 'enhance-validation-ui').
- **Switch to the new branch:** `git checkout enhance-validation-ui` (Or use `git checkout -b enhance-validation-ui` to create and switch in one step).

6.

7. **Implement Version 2 Changes:**

- Modify `index.html` with the Version 2 code (enhanced validation, transactions, button animation, local storage - based on OCR Exp 8, Pages 3-6 description).
- Make several small, logical changes and commit them incrementally.
- **Stage changes:** `git add index.html`
- **Commit changes:** `git commit -m "Feat: Add enhanced form validation"`
- *Make more changes (e.g., add transaction history)*
- **Stage:** `git add index.html`
- **Commit:** `git commit -m "Feat: Implement transaction history log with timestamp"`
- *Make more changes (e.g., add local storage)*
- **Stage:** `git add index.html`
- **Commit:** `git commit -m "Feat: Add local storage integration"`
- *Make more changes (e.g., add button animation)*
- **Stage:** `git add index.html`
- **Commit:** `git commit -m "Style: Add button animation for better UX"`

8.

9. **Merge Version 2 into Mainline:**

- **Switch back to the main branch:** `git checkout main`
- **Merge the feature branch:** `git merge enhance-validation-ui` (This brings all commits from the feature branch into `main`).
- *(Handle merge conflicts if they occur - Git will mark conflicting sections in files; you need to manually resolve them, then .*
- **Push the merged main branch:** `git push origin main`. *Version 2 changes are now integrated.*

10.

11. View History:

- Use `git log` to see the commit history.
- Use `git log --oneline --graph --all` for a compact graphical view of branches and merges.
- Use GitHub's web interface to view the commit history, branches, and file changes visually. (The OCR Exp 8, Page 6 shows a GitHub commit history screenshot).

12.

3. Required Diagrams:

a. Simple Git Branching/Merging Diagram:

graph LR

```
A[C1: Init V1] --> B(C2: Add V1 JS);
subgraph main Branch
  B --> E(C5: Merge V2);
end
subgraph feature Branch (enhance-validation-ui)
  B --> C(C3: Add Validation);
  C --> D(C4: Add LocalStorage);
end
D --> E;
```

```
style A fill:#eee,stroke:#333,stroke-width:2px
style B fill:#eee,stroke:#333,stroke-width:2px
style E fill:#eee,stroke:#333,stroke-width:2px
style C fill:#ccf,stroke:#333,stroke-width:2px
style D fill:#ccf,stroke:#333,stroke-width:2px
```

content_copy

download

Use code with caution. Mermaid

- **Explanation:** Shows the `main` branch starting with commits C1, C2 (Version 1). A `feature` branch is created from C2. Commits C3, C4 happen on the feature branch (Version 2 development). Finally, the feature branch is merged back into `main` at commit C5.

b. GitHub Commit History Screenshot (Similar to OCR Exp 8, Page 6):

- This would visually show a list of commits, typically in reverse chronological order. Each entry would display:
 - Commit message (e.g., "Version 2: Enhanced validation...", "Add files via upload", "Initial commit").
 - Author name (e.g., KrutikSinghYadav).
 - Time of commit (e.g., "1 minute ago", "24 minutes ago").
 - Commit hash (unique ID, usually shortened).
 - Indication of the branch the commit belongs to (if viewing history for all branches).
-

4. Examples and Real-World Applications:

- **Software Development:** Managing code changes for features, bug fixes, releases. Collaboration among developers.
- **Web Development:** Tracking changes to HTML, CSS, JavaScript, backend code.
- **Document Management:** Versioning requirement specifications, design documents, user manuals.
- **Infrastructure as Code (IaC):** Managing configuration files for servers, networks using tools like Terraform or Ansible with Git.
- **Scientific Research:** Tracking changes in datasets, analysis scripts, research papers.

5. Viva Questions (Basic to Toughest):

1. **What is Software Configuration Management (SCM)?**
 - *Answer:* A discipline to systematically control changes to software artifacts, maintain integrity, and ensure traceability throughout the lifecycle.
- 2.
3. **What are the four main activities of SCM?**
 - *Answer:* Configuration Identification, Configuration Control, Configuration Status Accounting, and Configuration Audits.
- 4.
5. **What is a Baseline?**
 - *Answer:* A formally reviewed and agreed-upon version of configuration item(s) serving as a reference point for further development. Changes require formal control after baselining.
- 6.
7. **What is a Version Control System (VCS)? Why use it?**

- *Answer:* A tool to track and manage changes to files over time. Used for history tracking, reverting changes, collaboration, branching, and merging.

8.

9. **What is the difference between Centralized (CVCS) and Distributed (DVCS) version control? Give examples.**

- *Answer:* CVCS (e.g., SVN) has one central server; developers check out/in. DVCS (e.g., Git) gives each developer a full repository clone; commits are local first. DVCS is more robust, faster for local operations, and better for branching/merging.

10.

11. **What is Git?**

- *Answer:* A free, open-source, distributed version control system.

12.

13. **What is GitHub?**

- *Answer:* A web-based hosting service for Git repositories, providing collaboration features like Pull Requests and Issue Tracking.

14.

15. **Explain the difference between Git and GitHub.**

- *Answer:* Git is the underlying VCS tool. GitHub is a platform *hosting* Git repositories and adding features around it. You don't need GitHub to use Git, but GitHub relies on Git.

16.

17. **What is a Git repository (repo)?**

- *Answer:* A database storing the entire history and files for a project.

18.

19. **What is a 'commit' in Git? What makes a good commit message?**

- *Answer:* A commit is a snapshot of the repository's state at a point in time. A good commit message is concise, describes *what* changed and *why*, and often follows a standard format (e.g., imperative mood: "Fix login bug" not "Fixed login bug").

20.

21. **What is 'branching' in Git? Why is it useful?**

- *Answer:* Creating an independent line of development. Useful for working on new features or bug fixes without disrupting the main codebase, allowing parallel development and experimentation.

22.

23. **What is 'merging' in Git? What is a merge conflict?**

- *Answer:* Combining changes from different branches. A merge conflict occurs when Git cannot automatically combine changes because the same lines of code were modified differently on the branches being merged. It requires manual resolution.

24.

25. What is the purpose of

- *Answer:* `git add`: Stages changes (adds them to the staging area) preparing them for the next commit. `git commit`: Records the staged changes as a new snapshot (commit) in the local repository history. `git push`: Sends local commits to a remote repository (like GitHub).

26.

27. Explain the typical Change Control Process.

- *Answer:* Change Request -> Evaluation -> Approval/Rejection -> Implementation (often on a branch) -> Verification/Review -> Merge -> Baseline Update -> Status Update.

28.

29. How does SCM support traceability?

- *Answer:* By tracking changes (commits) linked to specific requirements or change requests (e.g., via commit messages referencing issue IDs), SCM tools allow you to trace why a change was made, who made it, when, and what requirement it fulfills. Version history provides lineage.

30.

6. Common Mistakes & How to Avoid Them:

- **Mistake:** Not committing frequently enough.
 - **Avoidance:** Commit small, logical units of work often. This makes history easier to understand and reduces the impact if you need to revert changes.
-
- **Mistake:** Writing meaningless commit messages (e.g., "stuff", "changes", "fix").
 - **Avoidance:** Write clear, concise messages explaining *what* was changed and ideally *why*. Follow conventions (e.g., imperative mood, reference issue numbers).
-
- **Mistake:** Committing large, unrelated changes together.
 - **Avoidance:** Use the staging area (`git add`) selectively to group related changes into separate commits.
-
- **Mistake:** Working directly on the `main/master` branch for significant changes.

- **Avoidance:** Use feature or bugfix branches for new development or fixes to keep the main branch stable.
-
- **Mistake:** Poor handling of merge conflicts (e.g., blindly accepting one version).
 - **Avoidance:** Carefully examine the conflict markers inserted by Git. Understand both sets of changes and manually edit the file to integrate them correctly. Test thoroughly after resolving.
-
- **Mistake:** Forgetting to `git pull` before starting work or `git push` after committing.
 - **Avoidance:** Develop habits: Pull changes from the remote before starting new work on a shared branch. Push your local commits regularly to share them and back them up.
-

7. Documentation:

- **Commit History:** The primary SCM documentation, showing the evolution of the project.
- **Change Log:** Often generated from commit messages or maintained separately, summarizing changes between releases.
- **Change Request Forms/Issue Tracker:** Documents the initiation, evaluation, and approval of changes (part of Configuration Control).
- **Release Notes:** Summarize key changes, features, and fixes included in a specific software release.
- **Configuration Management Plan:** A formal document outlining the SCM procedures, tools, roles, and responsibilities for a project.

Experiment No. 09: Design test cases and generate test scripts in Selenium

Learning Objective:

Students will be able to design unit test cases for a software component and generate automated test scripts using a tool like Selenium IDE.

Tools Used:

- **Conceptual Tools:** Software Testing Fundamentals, Test Case Design Techniques (Equivalence Partitioning, Boundary Value Analysis), Unit Testing, Black-Box Testing, Test Scripting, Test Automation Concepts.

- **Software Tools:** Selenium IDE (Browser extension for record and playback), A simple program/web page to test (e.g., the square calculation program described, or the Banking App from Exp 8), Web Browser (Firefox, Chrome).

1. Full Theory:

a. Software Testing:

- **Definition:** Software testing is the process of evaluating a software item to detect differences between given input and expected output. It involves executing a program or system with the intent of finding errors. It's an investigation conducted to provide stakeholders with information about the quality of the product or service under test.
- **Purpose:**
 - To find defects/bugs.
 - To verify that the software meets the specified requirements (Verification).
 - To validate that the software meets the user's needs (Validation).
 - To build confidence in the software's quality.
 - To provide information for decision-making (e.g., release readiness).
 - To prevent defects by identifying issues early in the lifecycle.

●

b. Test Case:

- **Definition:** A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly. It typically includes inputs, execution preconditions, expected results, and postconditions.
- **Components of a Test Case:**
 - **Test Case ID:** Unique identifier.
 - **Description/Purpose:** What requirement or function is being tested.
 - **Preconditions:** Conditions that must be met before the test can be executed (e.g., user logged in, specific data exists).
 - **Test Steps:** Sequence of actions to perform.
 - **Test Data (Input):** Specific input values required for the test steps.
 - **Expected Result:** The outcome anticipated if the software functions correctly according to the requirements.
 - **Actual Result:** The outcome observed during test execution (filled in after running the test).
 - **Status:** Pass/Fail (based on comparing Expected and Actual results).
 - **(Optional):** Priority, Environment, Postconditions, etc.

-
- **Example Format (from document):** Test case X: {Input Set, Expected Output Set} e.g., Test case 1 : {I1 , O1} where I1 is the input data and O1 is the expected output.

c. Test Case Design Techniques: Techniques used to select effective test cases.

- **Black-Box Testing:** Testing without knowledge of the internal structure or code of the software. Focuses on testing functionality against requirements by providing inputs and examining outputs. (Mentioned in the document).
 - **Equivalence Partitioning (EP):** Divides input data into partitions (classes) from which test cases can be derived. Assumes that all values within a partition will be processed similarly. Select one representative value from each partition.
 - *Example (Square program, range 1-100):*
 - Partition 1: Valid numbers (1 to 100). Test Case: Input 50.
 - Partition 2: Numbers below range (< 1). Test Case: Input 0, Input -2.
 - Partition 3: Numbers above range (> 100). Test Case: Input 101.
 - Partition 4: Non-numeric input. Test Case: Input "abc".
 - Partition 5: Empty input. Test Case: Input "".
 -
 -
 - **Boundary Value Analysis (BVA):** Focuses on testing values at the boundaries (edges) of equivalence partitions, as errors often occur here. Tests the minimum value, just above minimum, nominal value, just below maximum, and maximum value.
 - *Example (Square program, range 1-100):*
 - Boundaries of valid range: 1, 100.
 - Test Cases: 0 (below min), 1 (min), 2 (above min), 99 (below max), 100 (max), 101 (above max).
 -
 -
-
- **White-Box Testing:** Testing based on knowledge of the internal structure, design, and code of the software. Focuses on testing paths, conditions, and statements within the code. (Not the focus of this experiment, but good to know the contrast).

d. Unit Testing:

- **Definition:** A level of software testing where individual units or components of a software are tested in isolation. A unit is the smallest testable part of any software (e.g., a function, method, procedure, module, class).
- **Purpose:** To validate that each unit of the software performs as designed. It helps find bugs early in the development cycle. Typically performed by developers.
- **Example (Square program):** Testing the `main` function or a hypothetical `calculateSquare(n)` function would be unit testing.

e. Test Script:

- **Definition:** A set of instructions (often automated) that is performed on a system under test to verify that the system performs as expected. Automated test scripts are written in a scripting or programming language and executed by testing tools.
- **Manual vs. Automated:** Test cases can be executed manually by a tester following steps, or automatically via a test script run by a tool.

f. Selenium:

- **Definition:** Selenium is a popular open-source framework for automating web browsers. It provides tools and libraries to interact with web elements, simulate user actions (clicking, typing), and verify application behavior.
- **Components:**
 - **Selenium WebDriver:** Provides APIs in various programming languages (Java, Python, C#, etc.) to create robust, browser-based automation scripts that interact directly with the browser.
 - **Selenium IDE (Integrated Development Environment):** A browser extension (primarily Chrome/Firefox) that allows testers to record user interactions with the browser, edit the recorded steps, and play them back for simple test automation. It's good for beginners or creating quick scripts. (This is the tool mentioned in the experiment).
 - **Selenium Grid:** Allows running tests in parallel on multiple machines and browsers simultaneously, speeding up test execution.
-
- **Selenium IDE Features:**
 - **Record and Playback:** Records user actions on a web page and generates corresponding commands (test script).
 - **Command Editing:** Allows modification of recorded commands, targets, and values.

- **Target Selection:** Helps identify web elements (using ID, name, XPath, CSS selectors).
- **Control Flow:** Basic control flow commands (if/else, loops - though often limited compared to WebDriver).
- **Debugging:** Step-through execution, setting breakpoints.
- **Export:** Can sometimes export recorded scripts to WebDriver code formats (though often needs significant refinement).

•

2. Step-by-Step Procedure:

A. Designing Test Cases (for the Square Program Example):

1. **Identify Functionality:** The program calculates the square of a number if it's within the range [1, 100]. Otherwise, it prints "Beyond the range".
2. **Identify Inputs:** User-provided number n .
3. **Identify Outputs:** Calculated square or "Beyond the range" message.
4. **Apply Equivalence Partitioning:**
 - Valid Range [1, 100]: Partition $V = \{n \mid 1 \leq n \leq 100\}$.
 - Invalid Range (Below): Partition $I_Below = \{n \mid n < 1\}$.
 - Invalid Range (Above): Partition $I_Above = \{n \mid n > 100\}$.
 - (Optional: Consider non-integer types if applicable, though `scanf("%d")` handles this).
- 5.
6. **Apply Boundary Value Analysis (on the valid partition):**
 - Min: 1
 - Min+1: 2
 - Max-1: 99
 - Max: 100
 - Boundaries for invalid partitions: 0 (edge of I_Below), 101 (edge of I_Above).
- 7.
8. **Create Test Cases:** Combine EP and BVA results.

TC ID	Description	Input (n)	Expected Output	Test Technique
TC1	Test value below min boundary	0	Beyond the range	BVA

TC2	Test min boundary value	1	Square of 1 is 1	BVA
TC3	Test value just above min	2	Square of 2 is 4	BVA
TC4	Test nominal valid value	62	Square of 62 is 3844	EP (Valid)
TC5	Test value just below max	99	Square of 99 is 9801	BVA
TC6	Test max boundary value	100	Square of 100 is 10000	BVA
TC7	Test value above max boundary	101	Beyond the range	BVA
TC8	Test negative value	-2	Beyond the range	EP (Invalid)

9.

(Note: These match closely with the inputs/outputs listed in the document, demonstrating BVA and EP implicitly).

B. Generating Test Scripts using Selenium IDE (for the Finance Tracker Login Example):

(Based on the screenshots and description in the document)

1. **Install Selenium IDE:** Add the Selenium IDE extension to your Chrome or Firefox browser.
2. **Open Selenium IDE:** Click the extension icon. Choose "Record a new test in a new project".
3. **Enter Project Name:** e.g., "finance-tracker".
4. **Enter Base URL:** The URL of the application login page (e.g., `https://finance-tracker-rosy-ten.vercel.app/login` from the document). Click "Start Recording".
5. **Interact with the Web Page:** Selenium IDE opens the URL in a new browser window. Perform the login actions manually:
 - Click in the User ID/Email field.
 - Type the username (e.g., `testbot@gmail.com`).
 - Click in the Password field.
 - Type the password (e.g., `test12345`).
 - Click the "Login" button.

- (Optionally, add an assertion: Right-click on an element expected after login, like the "Logout" button or username display, go to Selenium IDE -> Assert -> text/presence).
- 6.
- 7. **Stop Recording:** Go back to the Selenium IDE window and click the "Stop Recording" button (red circle).
- 8. **Name the Test:** Give the recorded test a name (e.g., "Login Test").
- 9. **Review Recorded Script:** Selenium IDE will show the recorded commands:
 - open: Navigates to the base URL + /login.
 - set window size: Records browser size (optional).
 - click: Records clicks on elements (identified by name, ID, CSS, XPath). Target might be name=email.
 - type: Records typing into input fields. Target might be name=email, Value is testbot@gmail.com. Target might be name=password, Value is test12345.
 - click: Records click on the Login button (Target might be css=.bg-blue-500 or similar).
- 10.
- 11. **Edit Script (Optional):** Modify commands, targets, or values if needed. Add assertions to verify outcomes (e.g., assertText, assertElementPresent).
- 12. **Playback the Test:** Click the "Run current test" button (play icon). Selenium IDE will execute the commands in the browser.
- 13. **Observe Results:** The IDE log will show each step being executed and its outcome (OK/Failed). Green checkmarks indicate success. A summary shows total runs and failures. (As seen in document screenshots Step 2 & 3).

3. Required Diagrams:

- **Test Case Table:** As shown in section 2.A above. This is the primary "diagram" for test case design.
- **Selenium IDE Screenshots:** As shown in the document, illustrating the recording, script view, and playback results.

4. Examples and Real-World Applications:

- **Test Case Design:** Applied to any software feature (login, search, calculation, form submission) to ensure thorough testing of inputs and boundary conditions.
- **Selenium IDE:**
 - Quickly automating simple, repetitive web tests (smoke tests, basic regression tests).
 - Assisting manual testers by automating parts of their workflow.

- Learning test automation concepts and element locators before moving to WebDriver.
- Creating simple demo scripts.
-
- **Limitations of Selenium IDE:** Not suitable for very complex test logic, extensive data-driven testing, or robust error handling compared to WebDriver + programming language scripts. Maintenance can be harder for large test suites.

5. Viva Questions (Basic to Toughest):

1. What is software testing?

- *Answer:* Executing software with the intent of finding defects and verifying it meets requirements.

2.

3. What is a test case?

- *Answer:* A set of inputs, preconditions, steps, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

4.

5. What is the difference between black-box and white-box testing?

- *Answer:* Black-box testing focuses on *what* the software does (inputs/outputs) without knowing the internal code. White-box testing focuses on *how* the software does it, testing internal code paths and structures.

6.

7. Explain Equivalence Partitioning.

- *Answer:* Dividing input data into partitions where all members are expected to be processed similarly. Test cases are designed using one representative value from each partition.

8.

9. Explain Boundary Value Analysis.

- *Answer:* Testing values at the edges (boundaries) of equivalence partitions, as errors often occur there.

10.

11. What is Unit Testing?

- *Answer:* Testing individual, isolated software components or units (like functions or methods).

12.

13. What is Selenium?

- *Answer:* An open-source framework for automating web browser interactions for testing purposes.

14.

15. What is Selenium IDE?

- *Answer:* A browser extension that allows recording user interactions with a web page and playing them back as automated tests.

16.

17. What does 'Record and Playback' mean in Selenium IDE?

- *Answer:* The IDE records user actions (clicks, types) as commands and can then automatically execute (playback) those commands in the browser.

18.

19. What are 'Targets' and 'Values' in Selenium IDE commands?

- *Answer:* 'Target' identifies the web element the command interacts with (e.g., using its ID, name, XPath). 'Value' is the data used (e.g., the text to type in a field).

20.

21. What is an Assertion in testing? How might you add one in Selenium IDE?

- *Answer:* An assertion is a check within a test script to verify if a certain condition is true (e.g., if specific text is present, if an element exists). It determines if the test step passes or fails. In Selenium IDE, you can often right-click on an element and select "Assert" options, or manually add commands like `assertText`, `assertElementPresent`.

22.

23. Can Selenium IDE handle complex logic like loops or conditional execution?

- *Answer:* Selenium IDE has some basic control flow commands (`if`, `while`, `times`), but they are generally less flexible and powerful than using a full programming language with Selenium WebDriver.

24.

25. What are web element locators? Give examples used by Selenium.

- *Answer:* Locators are strategies used to find HTML elements on a web page. Examples include ID, Name, Class Name, Tag Name, Link Text, Partial Link Text, CSS Selector, and XPath. Selenium IDE tries to automatically select one during recording.

26.

27. What are the limitations of using only Record and Playback tools like Selenium IDE?

- *Answer:* Scripts can be brittle (break easily with UI changes), limited support for complex logic/error handling/data-driven testing, harder maintenance for large suites, may encourage less thoughtful test design compared to manual coding.

28.

29. How would you design test cases for the 'Password' field on a login form using EP and BVA? (Assume rules: 8-16 characters, must include uppercase, lowercase, number).

- Answer:

- **EP (Valid):** One valid password (e.g., `ValidPass1`).
- **EP (Invalid Length):** Too short (e.g., `Pass1`), Too long (e.g., `VeryLongValidPassword1`).
- **EP (Invalid Content):** Missing uppercase (e.g., `validpass1`), Missing lowercase (e.g., `VALIDPASS1`), Missing number (e.g., `ValidPass`).
- **EP (Empty):** `""`.
- **BVA (Length):** 7 chars (invalid), 8 chars (valid min), 16 chars (valid max), 17 chars (invalid).
- *(Combine these to create specific test cases).*

-

30.

6. Common Mistakes & How to Avoid Them:

- **Test Case Design:**

- **Mistake:** Only testing the "happy path" (successful scenario).
- **Avoidance:** Use EP and BVA to systematically identify negative test cases (invalid inputs, boundary conditions, error scenarios).
- **Mistake:** Test cases are not specific enough (vague steps or expected results).
- **Avoidance:** Write clear, unambiguous steps and define precise expected outcomes.

-

- **Selenium IDE:**

- **Mistake:** Relying solely on automatically recorded locators (which might be unstable, like complex XPath's).
- **Avoidance:** Learn to identify more robust locators (like unique IDs or Names) and manually edit the 'Target' field in the script if needed.
- **Mistake:** Not adding Assertions.
- **Avoidance:** A script without checks only tests if the application *runs* without crashing, not if it produces the *correct* results. Add assertions (`assertText`, `assertElementPresent`, `assertValue`, etc.) to verify outcomes.
- **Mistake:** Creating overly long, monolithic recordings.
- **Avoidance:** Record smaller, focused tests for specific functionalities.

- **Mistake:** Assuming recorded scripts are robust and maintainable for complex applications.
- **Avoidance:** Recognize IDE is best for simple tasks/learning. For complex, long-term automation, invest in learning WebDriver with a programming language.

-

7. Documentation:

- **Test Plan:** Outlines the overall testing strategy, scope, resources, schedule, and deliverables.
- **Test Case Specification:** Detailed documentation of individual test cases (using a format like the table in 2.A).
- **Test Script:** The automated script generated by Selenium IDE (usually saved as a `.side` file containing the project, tests, commands, targets, values).
- **Test Report:** Summarizes the results of test execution (number of tests run, passed, failed, defects found). Selenium IDE provides a basic execution log.

Task: Understand and Prepare a Software Requirements Specification (SRS)

Learning Objective:

Students will be able to understand the purpose, structure, and content of an SRS document and be able to identify and document functional and non-functional requirements for a software system.

Tools Used:

- **Conceptual Tools:** Requirements Engineering, Requirement Elicitation Techniques, Functional Requirements, Non-Functional Requirements, IEEE 830-1998 Standard (or similar standards).
- **Software Tools:** Word Processor (MS Word, Google Docs), Requirements Management Tools (e.g., Jira with specific configurations, Jama Connect, DOORS - though typically overkill for student projects).

1. Full Theory:

a. Software Requirements Specification (SRS):

- **Definition:** An SRS is a formal document that provides a complete description of the behavior of a software system to be developed. It includes a set of use cases describing interactions users will have with the software, as well as non-functional requirements

(performance, quality standards, constraints). The SRS establishes the agreement between the customer/stakeholders and the developers on what the software product will do.

- **Purpose:**

- **Communication:** Provides a clear and unambiguous description of the system for all stakeholders (customers, users, developers, testers, project managers).
- **Agreement:** Forms the basis of the contract between the client and the supplier.
- **Foundation for Design:** Guides the system architects and designers in creating the system structure.
- **Basis for Testing:** Provides the criteria against which the final product will be verified and validated. Testers create test cases based on the SRS.
- **Foundation for Estimation:** Helps in estimating project cost, resources, and timelines.
- **System Evolution:** Serves as a reference for future maintenance and enhancement.

-

- **Audience:** As mentioned in the FMS SRS (Section 1.3), the audience is diverse: developers, project managers, marketing staff, testers, documentation writers. The document needs to be understandable by different groups, potentially requiring different levels of detail or focus in various sections.

- **Characteristics of a Good SRS:**

- **Correct:** Accurately reflects the requirements.
- **Unambiguous:** Every statement has only one interpretation.
- **Complete:** Includes all significant requirements (functional, non-functional, constraints). Defines responses to all valid and invalid inputs.
- **Consistent:** Requirements do not contradict each other.
- **Ranked for Importance/Stability:** Indicates which requirements are essential (high priority) vs. desirable, and how likely they are to change.
- **Verifiable/Testable:** There must be a finite, cost-effective process by which a person or machine can check that the software product meets the requirement.
- **Modifiable:** Structure and style should allow changes to be made easily, completely, and consistently while retaining history.
- **Traceable:** The origin of each requirement is clear, and it facilitates referencing each requirement in future development or documentation.

-

b. IEEE 830-1998 Standard:

- A widely recognized standard providing guidelines for writing good SRS documents. It suggests a structure (though adaptable) to ensure completeness and clarity. The FMS SRS document explicitly states it follows this standard (Section 1.2).

c. Functional vs. Non-Functional Requirements:

- **Functional Requirements:** Define *what* the system should do. They describe the specific services, functions, calculations, data processing, and user interactions the system must provide. Often derived from Use Cases.
 - *Example (FMS SRS Section 3):* "REQ-1: The system must allow authorized users to enter payment details..." (Payment Tracking), "REQ-1: The system must allow users to generate customizable reports..." (Report Generation).
-
- **Non-Functional Requirements (NFRs):** Define *how* the system should perform its functions. They specify the quality attributes, constraints, and characteristics of the system. Often called "ilities" (reliability, usability, performance, security, maintainability, portability).
 - *Example (FMS SRS Section 5):* Performance Requirements (e.g., "Payment transactions should be processed within 2 seconds..."), Safety Requirements (e.g., "automated data backups every 24 hours..."), Security Requirements (e.g., "enforce multi-factor authentication..."), Software Quality Attributes (e.g., "ensure 99.9% availability...").
-

2. Standard SRS Structure (based on IEEE 830-1998 and the FMS example):

(This outlines the typical sections and their content, using the FMS SRS as a guide)

Table of Contents (Essential for navigation)

Revision History (Tracks changes to the SRS document itself)

1. Introduction

- * **1.1 Purpose:** Briefly state the purpose of the SRS document and the intended audience. (FMS SRS 1.1: Describes FMS, focuses on payment tracking, invoice generation, record management).
- * **1.2 Document Conventions:** Describe any standards or typographical conventions used (e.g., requirement identifiers like REQ-X). (FMS SRS 1.2: Mentions IEEE 830-1998).
- * **1.3 Intended Audience and Reading Suggestions:** Guide different types of readers on how to approach the document. (FMS SRS 1.3: Lists developers, PMs, marketing, testers, writers and suggests reading order).
- * **1.4 Project Scope:** Define the scope of the product – what it will and will *not* do. Relate it to

business goals. (FMS SRS 1.4: Streamline payments/transactions, track payments, generate invoices, manage records, improve efficiency/transparency).

* **1.5 References:** List any other documents or resources referenced (e.g., project plan, related standards, style guides). (FMS SRS 1.5: Lists Vision/Scope doc, IEEE standard, UI Style Guide, System Design Spec).

2. Overall Description

* **2.1 Product Perspective:** State if the product is standalone or related to other systems. Describe its relationship to system context (hardware, software, users). (FMS SRS 2.1: New, self-contained system, may integrate later, aims to improve efficiency/reduce errors).

* **2.2 Product Features:** Summarize the major functions the product will perform (high-level overview). Often corresponds to major sections in Specific Requirements. (FMS SRS 2.2: Lists payment tracking, invoice generation, record management, reporting).

* **2.3 User Classes and Characteristics:** Identify different types of users, their expected skills, and activities. (FMS SRS 2.3: Administrators, Finance Managers, Accounting Staff, Auditors - describes their roles and access levels).

* **2.4 Operating Environment:** Describe the environment in which the software will operate (hardware platforms, OS versions, integration with other applications). (FMS SRS 2.4: Cross-platform - Windows, Linux, macOS, Android, iOS; requires internet).

* **2.5 Design and Implementation Constraints:** List any factors that will restrict design options (e.g., regulatory policies, required technologies, hardware limitations, security requirements, specific language). (FMS SRS 2.5: Compliance (GST, GDPR/HIPAA), use of Flutter, Firebase for security, external API constraints, performance/scalability reqs, internal design conventions).

* **2.6 User Documentation:** List user documentation components (user manual, online help, tutorials) that will be delivered. (FMS SRS 2.6: Shows 'nil', indicating perhaps none planned or TBD).

* **2.7 Assumptions and Dependencies:** State any assumptions (factors believed to be true but not guaranteed) and dependencies (external factors the project relies on). (FMS SRS 2.7: Assumes secure internet, valid credentials; depends on 3rd-party gateways, Firebase Auth, Flutter compatibility, libraries for reports).

3. System Features (Specific Requirements - Functional)

* This section details the functional requirements. It can be organized by feature, use case, mode, or object. Each functional requirement should describe inputs, processing, and outputs. Use unique identifiers (REQ-X).

*** 3.x Feature/Use Case Name**

* **3.x.1 Description and Priority:** Overview of the feature/use case and its importance (High, Medium, Low). (FMS SRS uses this structure for 3.1 Access Control, 3.2 Payment Tracking, 3.3 Report Generation).

* **3.x.2 Stimulus/Response Sequences:** Describes the sequence of inputs (stimuli) and outputs (responses) for this feature. (FMS SRS details this for each feature).

* **3.x.3 Functional Requirements:** List the specific, detailed requirements (REQ-1, REQ-2...). These should be verifiable. (FMS SRS lists REQ-1 to REQ-5 for each feature).

4. External Interface Requirements

* **4.1 User Interfaces:** Describe the logical characteristics of user interfaces (e.g., GUI standards, screen layout constraints, standard buttons/functions, accessibility requirements, responsiveness). (FMS SRS 4.1: Simplicity, consistency, login screen, navigation menus, standard buttons, clear error messages, logical layout, keyboard shortcuts, responsive design).

* **4.2 Hardware Interfaces:** Specify logical/physical characteristics of interfaces to hardware components (supported device types, communication protocols). (FMS SRS 4.2: Interfaces with desktops, mobiles, POS systems via HTTP/HTTPS; handles user input, screen output, no specialized hardware beyond standard devices).

* **4.3 Software Interfaces:** Specify interfaces to other software (OS, databases, libraries, external components). Define purpose, data exchange, protocols. (FMS SRS 4.3: Integration with OS, cloud DBs, external libs via HTTP/HTTPS; interacts with DB for records/logs/auth; uses APIs for data sharing).

* **4.4 Communications Interfaces:** Specify requirements for communication functions (e.g., email, network protocols, security considerations, data transfer rates). (FMS SRS 4.4: Email notifications, web browser access via HTTP/HTTPS (encrypted), real-time sync, FTP for large files (secure), SSL/TLS, data validation before transfer).

5. Other Nonfunctional Requirements

* **5.1 Performance Requirements:** Specify static and dynamic numerical requirements (e.g., response time, throughput, capacity, resource utilization). Should be quantitative and testable. (FMS SRS 5.1: Specific timings for transactions, reports; TPS targets; query response times; sync time; UI responsiveness; uptime; failover time; security operation time).

* **5.2 Safety Requirements:** Requirements to prevent loss, damage, or harm that could result from system use (e.g., data backup/recovery, fault tolerance, handling hazardous conditions). (FMS SRS 5.2: Data backups, MFA, encryption, failover mechanisms, compliance (GDPR/HIPAA/GST), RBAC, real-time alerts).

* **5.3 Security Requirements:** Requirements concerning unauthorized access, use, modification, destruction, or disclosure (e.g., authentication, authorization, data encryption, audit trails, compliance). (FMS SRS 5.3: MFA, RBAC, AES-256 encryption, TLS 1.3, compliance (GDPR/HIPAA/PCI-DSS), session timeouts, account lockouts, audit logs, ISO 27001/SOC 2 compliance, anonymization/masking).

* **5.4 Software Quality Attributes:** Specify desired quality characteristics (e.g., Reliability,

Availability, Maintainability, Portability, Usability, Testability, Robustness). Define how they will be measured. (FMS SRS 5.4: Quantifies availability (99.9%), reliability (99.99% accuracy), portability (OS support, REST APIs), maintainability (modular code), robustness (TPS handling), testability (automation), usability (minimal training), reusability (shared components)).

* **5.x Other Requirements:** Any requirement not covered elsewhere (e.g., Database, Internationalization, Legal, Operations, Installation). (FMS SRS Section 6 is a placeholder for this).

Appendices

* **Appendix A: Glossary:** Definitions of terms, acronyms, abbreviations used in the SRS. (FMS SRS Appendix A provides examples like FMS, SRS, RBAC, MFA...).

* **Appendix B: Analysis Models:** Optional references or diagrams like DFDs, Class diagrams, State/Activity diagrams. (FMS SRS Appendix B is used as an Issues List instead).

* **Appendix C: Issues List:** A list of requirements that are TBD (To Be Determined). (FMS SRS Appendix B uses this format, listing unresolved issues like performance metrics, localization, external integration details...).

3. Requirements Elicitation Techniques (How to gather requirements):

- **Interviews:** Questioning stakeholders (users, clients, domain experts).
- **Workshops/Brainstorming:** Group sessions to generate ideas and reach consensus.
- **Observation:** Watching users perform their current tasks.
- **Questionnaires/Surveys:** Gathering information from a large number of people.
- **Document Analysis:** Studying existing system documentation, business process descriptions, standards, regulations.
- **Prototyping:** Creating a preliminary version or mock-up of the system to get user feedback.
- **Use Case Analysis:** Identifying actors and their goals to derive functional requirements.

4. Viva Questions (Basic to Toughest):

1. What is an SRS document?

- *Answer:* A Software Requirements Specification is a formal document describing the complete behavior, functionality, performance, and constraints of a software system to be developed.

2.

3. What is the main purpose of an SRS?

- *Answer:* To establish a clear agreement between stakeholders and developers on what the system will do, serving as a foundation for design, testing, and project management.

4.

5. **Who are the typical audiences for an SRS?**

- *Answer:* Customers, users, developers, testers, project managers, marketing, documentation writers.

6.

7. **What are the characteristics of a good SRS?**

- *Answer:* Correct, Unambiguous, Complete, Consistent, Verifiable, Modifiable, Traceable, Ranked for importance.

8.

9. **What is the difference between Functional and Non-Functional requirements? Give examples.**

- *Answer:* Functional requirements define *what* the system does (e.g., "User can submit payment"). Non-functional requirements define *how* well it does it (e.g., "Payment must be processed within 2 seconds", "System must use AES-256 encryption").

10.

11. **What is the purpose of the 'Scope' section in the Introduction?**

- *Answer:* To define the boundaries of the product – what features are included and what are explicitly excluded, often linking it to business objectives.

12.

13. **Why are 'Design and Implementation Constraints' important?**

- *Answer:* They limit the choices available to the designers and developers (e.g., requiring a specific database, programming language, or compliance with a regulation), impacting the final solution.

14.

15. **What kind of information goes into the 'External Interface Requirements' section?**

- *Answer:* Descriptions of how the system interacts with users (UI), hardware, other software systems, and communication networks.

16.

17. **Give examples of different Software Quality Attributes (Non-Functional Requirements).**

- *Answer:* Performance (speed, throughput), Security (access control, encryption), Reliability (uptime, failure rate), Usability (ease of use), Maintainability (ease of modification), Portability (running on different platforms).

18.

19. **What is the IEEE 830 standard?**

- *Answer:* It's a guideline published by IEEE providing a recommended structure and best practices for writing SRS documents.

20.

21. How are requirements typically identified (elicitation techniques)?

- *Answer:* Through interviews, workshops, observation, document analysis, prototyping, questionnaires, use case analysis.

22.

23. Why is it important for requirements to be 'Verifiable' or 'Testable'?

- *Answer:* If a requirement isn't verifiable, there's no objective way to determine if the developed system actually meets it. Testability allows testers to create specific tests to confirm compliance. Vague requirements (e.g., "user-friendly") are hard to verify.

24.

25. What is the 'Issues List' or 'TBD' section used for?

- *Answer:* To track requirements or details that are not yet finalized or decided upon, ensuring they aren't forgotten during development.

26.

27. How does the SRS relate to Use Cases?

- *Answer:* Use Cases describe user interactions and goals, which are a primary source for deriving functional requirements detailed in the SRS. The SRS might reference Use Case diagrams or specifications.

28.

29. How can ambiguity in requirements lead to problems? Provide an example.

- *Answer:* Ambiguity leads to different interpretations by stakeholders, designers, developers, and testers, resulting in a system that doesn't meet expectations. Example: Requirement "System should provide quick response". "Quick" is ambiguous. A better requirement: "System response time for login must be less than 1 second under peak load".

30.

5. Common Mistakes & How to Avoid Them:

- **Mistake:** Requirements are ambiguous or vague (using terms like "fast", "easy", "flexible").
 - **Avoidance:** Use quantitative and precise language. Define measurable criteria for non-functional requirements. Use glossaries to define terms.
- **Mistake:** Mixing requirements with design/implementation details.
 - **Avoidance:** The SRS should focus on *what* the system must do, not *how* it will be built (unless it's a specific constraint). Design details belong in the SDD.

-
- **Mistake:** Incompleteness (missing requirements, forgetting error handling, not considering all user classes).
 - **Avoidance:** Use systematic elicitation techniques. Review checklists based on standards (like IEEE 830). Use techniques like Use Case analysis to cover different scenarios. Explicitly consider invalid inputs and error conditions.
-
- **Mistake:** Inconsistency (requirements contradicting each other).
 - **Avoidance:** Review requirements carefully as a whole. Use modeling techniques (like DFDs, Class diagrams) to help identify conflicting logic or data needs.
-
- **Mistake:** Not prioritizing requirements.
 - **Avoidance:** Use methods like MoSCoW (Must have, Should have, Could have, Won't have) or numerical ranking to indicate importance, helping guide development effort.
-
- **Mistake:** Writing requirements that are not testable.
 - **Avoidance:** For each requirement, ask "How would I test this?". If you can't define a clear test, the requirement needs refinement.
-

6. Role in Software Lifecycle:

- The SRS is created during the **Requirements Engineering** phase, which typically follows initial feasibility studies and planning.
 - It serves as the primary input for the **System Design** phase.
 - It's used during **Implementation** to ensure developers build the correct features.
 - It forms the basis for **Testing** (Verification and Validation).
 - It's referenced during **Maintenance** and **Evolution** to understand the original intent and manage changes.
-