Fsd (DI02016021) - Problem Solver

Antigravity AI

June 4, 2026

Fsd

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Contents

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

CHAPTER 1

Introduction to Software Development Process

1.1 Software - Definition and Characteristics

Software is more than just code; it includes executable programs, associated data structures, and documentation. While traditional manufacturing evaluates physical wear and tear, software engineering evaluates logical characteristics such as reliability. Since software does not "wear out" like hardware, its reliability is measured mathematically using failure rates and time intervals.

Methodology:

Software Reliability and Failure Rate Calculation To evaluate the reliability characteristics of software over time, the following computational steps are applied:

1. Identify the Mean Time To Failure (MTTF), which represents the average time a software runs without failing.
2. Identify the Mean Time To Repair (MTTR), which represents the average time taken to fix the software after a failure.
3. Calculate the Mean Time Between Failures (MTBF) using the formula:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

4. Calculate the failure rate (λ), defined as the reciprocal of the MTTF:

$$\lambda = \frac{1}{\text{MTTF}}$$

5. Compute the Software Reliability $R(t)$ for a specified continuous operation time t using the exponential reliability model:

$$R(t) = e^{-\lambda t}$$

Worked Example:

Calculating MTBF and Reliability A financial software module is characterized by an MTTF of 200 hours and an MTTR of 10 hours.
Task: Calculate the MTBF, the failure rate (λ), and the reliability of the software for a continuous operation of 50 hours.
Solution:

1. **Calculate MTBF:**

$$\text{MTBF} = \text{MTTF} + \text{MTTR} = 200 + 10 = 210 \text{ hours}$$

2. **Calculate Failure Rate (λ):**

$$\lambda = \frac{1}{\text{MTTF}} = \frac{1}{200} = 0.005 \text{ failures/hour}$$

3. **Calculate Reliability $R(50)$:** Substitute $\lambda = 0.005$ and $t = 50$ into the reliability function:

$$R(50) = e^{-0.005 \times 50} = e^{-0.25}$$

Using the value $e \approx 2.71828$:

$$R(50) \approx 0.7788$$

Conclusion: The software has an MTBF of 210 hours, a failure rate of 0.005 per hour, and an approximate 77.88% chance of running without failure for 50 continuous hours.

1.2 Characteristics of Web-based Applications

Web-based applications exhibit distinct characteristics compared to conventional software, including network intensity, concurrency, and high performance demands. Because they are delivered over a network, performance metrics such as response time and throughput become critical computational factors for evaluating a web application's effectiveness.

Methodology:

Web Application Performance Evaluation To analyze the network and performance characteristics of a web application, follow these steps:

1. Measure the individual time components of a request:
   - Server Processing Time (T_{server})
   - Network Latency (T_{network})
   - Client Rendering Time (T_{client})
2. Calculate the Total Response Time (T_{total}) using the formula:

$$T_{\text{total}} = T_{\text{server}} + T_{\text{network}} + T_{\text{client}}$$

3. Record the Total Number of Requests (N) handled over a specific Time Period (P in seconds).
4. Calculate the Throughput (Th) using the formula:

$$Th = \frac{N}{P} \text{ (requests per second)}$$

Worked Example:

Computing Response Time and Throughput A web-based student portal application processes 4500 requests during a one-hour period. For a standard user request, the server takes 250 milliseconds to process the data, the network latency is measured at 120 milliseconds, and the browser rendering takes 150 milliseconds.
Task: Calculate the total response time per request in seconds and the overall throughput of the web application.
Solution:

1. **Calculate Total Response Time (T_{total}):**

$$T_{\text{server}} = 250 \text{ ms}, \quad T_{\text{network}} = 120 \text{ ms}, \quad T_{\text{client}} = 150 \text{ ms}$$

$$T_{\text{total}} = 250 + 120 + 150 = 520 \text{ ms}$$

Convert to seconds: 520 ms = 0.52 seconds.

2. **Calculate Throughput (Th):** Identify the number of requests $N = 4500$. Convert the time period

to seconds: $P = 1$ hour $= 3600$ seconds.

$$Th = \frac{4500}{3600} = 1.25 \text{ requests/second}$$

Conclusion: The application has a response time of 0.52 seconds per request and handles a throughput of 1.25 requests per second.

1.3 Software Engineering - A Layered Technology

Software engineering is built upon a layered technology comprising tools, methods, process models, and a bedrock focus on quality. The quality layer serves as the foundation. We can quantify the effectiveness of the quality assurance processes by evaluating defect density across different modules built using layered methods.

Methodology:

Defect Density and Quality Assessment To measure the impact of the quality focus layer, defect density is calculated:

1. Determine the total number of defects (D) discovered during the quality assurance testing of a module.
2. Measure the software size (S), typically quantified in Kilo Lines of Code (KLOC).
3. Calculate the Defect Density (DD) using the formula:

$$DD = \frac{D}{S}$$

4. Compute the overall project defect density by summing all defects across all modules and dividing by the total KLOC size of the project.

Worked Example:

Evaluating Defect Density across Layers A software system developed using a layered methodology consists of three distinct modules:

- Module A: Size = 12 KLOC, Defects = 42
- Module B: Size = 8 KLOC, Defects = 15
- Module C: Size = 20 KLOC, Defects = 60

Task: Calculate the defect density for each module and the overall defect density for the entire project.
Solution:

1. **Calculate individual defect densities:**

$$DD_A = \frac{42}{12} = 3.50 \text{ defects/KLOC}$$

$$DD_B = \frac{15}{8} = 1.875 \text{ defects/KLOC}$$

$$DD_C = \frac{60}{20} = 3.00 \text{ defects/KLOC}$$

2. **Calculate overall defect density:**

$$\text{Total Defects} = 42 + 15 + 60 = 117$$

$$\text{Total Size} = 12 + 8 + 20 = 40 \text{ KLOC}$$

$$DD_{\text{overall}} = \frac{117}{40} = 2.925 \text{ defects/KLOC}$$

Conclusion: Module A has the highest defect density (3.50), signaling that the quality assurance layer should enforce stricter reviews on that specific module. The overall project defect density is 2.925 defects/KLOC.

1.4 Software Myths

Software myths are erroneous beliefs about software and the processes used to build it. A prevalent management myth is that adding more programmers to a project that is already running late will catch it up to schedule. In reality, adding manpower introduces significant communication overhead, which can be computationally modeled using Brooks's Law.

Methodology:

Evaluating Communication Overhead Using Brooks Law The computational impact of adding personnel is analyzed by calculating communication channels:

1. Identify the initial number of developers on the project (n_1).
2. Calculate the initial number of communication channels (C_1) required:

$$C_1 = \frac{n_1(n_1 - 1)}{2}$$

3. Identify the new total number of developers (n_2) after new members are added.
4. Calculate the new number of communication channels (C_2):

$$C_2 = \frac{n_2(n_2 - 1)}{2}$$

5. Compute the increase in communication overhead (ΔC):

$$\Delta C = C_2 - C_1$$

Worked Example:

Impact of Adding Manpower to a Delayed Project A critical software project is behind schedule. The current development team has 5 members. Believing a common software myth, the project manager decides to add 4 new developers to speed up the work.
Task: Calculate the number of communication channels before and after the change, and determine the added overhead.
Solution:

1. **Initial State:** Given $n_1 = 5$.

$$C_1 = \frac{5(5 - 1)}{2} = \frac{5 \times 4}{2} = 10 \text{ channels}$$

2. **New State:** The new team size is $n_2 = 5 + 4 = 9$.

$$C_2 = \frac{9(9 - 1)}{2} = \frac{9 \times 8}{2} = 36 \text{ channels}$$

3. **Calculate the Overhead:**

$$\Delta C = 36 - 10 = 26 \text{ channels}$$

Conclusion: Adding 4 developers increased the communication complexity from 10 to 36 channels. The massive overhead (an addition of 26 new communication paths) severely delays the project further, effectively proving the myth false.

1.5 Software Process Framework and Umbrella Activities

A software process framework establishes the foundation for a complete software engineering process. Alongside the core framework activities (like communication, planning, and modeling), umbrella activities are applied throughout the software project. Risk management is a primary umbrella activity, which relies on the mathematical evaluation of risk exposure.

Methodology:

Risk Exposure Calculation for Project Management Risk management involves evaluating threats continuously across all framework phases. Calculate Risk Exposure (RE) to prioritize these threats:

1. Identify potential risks that could disrupt the software process framework.
2. Estimate the Probability of Occurrence (P) for each risk, expressed as a decimal between 0 and 1.
3. Estimate the Cost Impact (C) of the risk, typically measured in financial terms or working hours.
4. Calculate the Risk Exposure (RE) using the equation:

$$RE = P \times C$$

5. Prioritize the risks by ranking them from the highest RE value to the lowest. Focus the umbrella activities on mitigating the highest exposure first.

Worked Example:

Prioritizing Risks in Umbrella Activities During the continuous monitoring umbrella activity, a project manager identifies three major risks threatening the process framework:

- **Risk 1 (Requirement Changes):** 40% probability of occurring, estimated impact of $25,000.
- **Risk 2 (Key Staff Turnover):** 15% probability of occurring, estimated impact of $40,000.
- **Risk 3 (Development Tool Failure):** 5% probability of occurring, estimated impact of $100,000.

Task: Calculate the Risk Exposure for each risk and determine the priority order for mitigation.
Solution:

1. **Evaluate Risk 1:** Probability $P_1 = 0.40$, Impact $C_1 = 25000$.

$$RE_1 = 0.40 \times 25000 = \$10,000$$

2. **Evaluate Risk 2:** Probability $P_2 = 0.15$, Impact $C_2 = 40000$.

$$RE_2 = 0.15 \times 40000 = \$6,000$$

3. **Evaluate Risk 3:** Probability $P_3 = 0.05$, Impact $C_3 = 100000$.

$$RE_3 = 0.05 \times 100000 = \$5,000$$

Conclusion: The priority ranking based on Risk Exposure is: 1. Risk 1 ($10,000) 2. Risk 2 ($6,000) 3. Risk 3 ($5,000)
The project manager should direct the umbrella activities primarily toward mitigating Requirement Changes, despite it having the lowest absolute impact, because its high probability makes it the largest overall threat.

CHAPTER 2

Software Life Cycle Models

2.1 Select Software Process Model for Project Development

A software process model is an abstract representation of the architecture of the software process. Choosing the right model depends on project requirements, risk, timeline, and team size.

2.1.1 Waterfall Model

Methodology:

Waterfall Model Execution Steps The Waterfall model is a linear sequential approach.

1. **Requirements Analysis:** Capture and document all system requirements.
2. **System Design:** Define the overall system architecture and hardware requirements.
3. **Implementation:** Write the actual code in small units (programs).
4. **Integration and Testing:** Integrate the units and test the complete system for faults.
5. **Deployment:** Release the tested software into the production environment.
6. **Maintenance:** Fix issues that arise post-deployment and apply enhancements.

Worked Example:

Applying Waterfall Model to a Fixed-Requirement Library System **Problem:** Map the development of a university library management system with completely static and known requirements using the Waterfall model.
Solution:

1. **Requirements Analysis:** Document the need for book tracking, student ID validation, and fine calculation. Ensure no requirements will change.
2. **System Design:** Design the database schema for Books and Students and select the technology stack (e.g. Java and MySQL).
3. **Implementation:** Developers write the code for user login, issue/return logic, and admin dashboard independently.
4. **Integration and Testing:** Combine the modules and perform system testing to ensure book fines are calculated accurately.
5. **Deployment:** Install the application on the university's central server.
6. **Maintenance:** Patch any bugs found by librarians during the academic year.

2.1.2 Incremental Model

Methodology:

Incremental Model Execution Steps

1. **Partitioning:** Break the complete software requirements into multiple independent, manageable modules or iterations.
2. **Design Increment:** Design the architecture for the first core module.
3. **Develop and Test Increment:** Code and test the current module.
4. **Deploy Increment:** Release the increment to the customer for immediate use.
5. **Repeat:** Integrate the next module with the existing system and repeat steps 2-4 until the full product is complete.

Worked Example:

Incremental Delivery of a Word Processor **Problem:** Develop a word processing application using the Incremental Model.
Solution:

1. **Increment 1 (Core):** Implement basic file management (Create/Open/Save) and basic text entry. Deploy to users.
2. **Increment 2:** Implement text formatting (Bold/Italic/Underline) and font selection. Integrate and deploy.
3. **Increment 3:** Add spell checking and grammar checking features. Integrate and deploy.
4. **Increment 4:** Add advanced page layout features like margins and tables. Integrate and deploy.

2.1.3 Prototyping Model

Methodology:

Prototyping Model Iteration

1. **Initial Requirement Gathering:** Understand the fundamental project objectives and user needs.
2. **Quick Design:** Create a basic design focusing on the visible user interface.
3. **Build Prototype:** Develop a working model with minimal functionality to demonstrate the UI.
4. **Customer Evaluation:** Let the customer interact with the prototype and gather feedback.
5. **Refine Prototype:** Incorporate changes requested by the customer. Repeat steps 3-5 until approved.
6. **Final Development:** Use the approved prototype as the specification to build the actual, robust system.

Worked Example:

Prototyping a Custom Dashboard **Problem:** A client wants a complex data visualization dashboard but is unsure of exactly how it should look. Use the prototyping method.
Solution:

1. **Initial Requirement:** Client needs to view sales data and user metrics.

2. **Quick Design:** Sketch a dashboard layout with charts and tables.
3. **Build Prototype:** Create a static HTML/CSS mockup with hardcoded dummy data.
4. **Evaluation:** Client reviews and states the charts need to be interactive and positioned differently.
5. **Refine:** Adjust the layout and add a JavaScript charting library to the prototype.
6. **Final Development:** Once the client approves the interactive mockup, the team builds the backend database connection and actual logic.

2.1.4 Spiral Model

Methodology:

Spiral Model Iterations The model focuses heavily on continuous risk assessment through repeated loops (spirals). Each loop represents a phase and consists of four quadrants:

1. **Objective Setting:** Identify objectives for the phase and alternative approaches.
2. **Risk Assessment and Reduction:** Evaluate alternatives and identify technical risks. Build prototypes to mitigate these risks.
3. **Development and Validation:** Develop the next version of the software based on resolved risks and test it.
4. **Review and Planning:** Review results with the client and plan the next spiral phase.

Worked Example:

Spiral Model for a New Banking Application **Problem:** Develop a high-security banking application where financial data loss is a critical risk.
Solution:

1. **Spiral 1 (Concept):**
   - Objective: Define system feasibility.
   - Risk: Regulatory compliance unknown.
   - Dev: Build proof-of-concept for encryption.
   - Plan: Proceed to requirement phase.
2. **Spiral 2 (Requirements):**
   - Objective: Specify requirements.
   - Risk: Integration with legacy bank systems.
   - Dev: Build prototype of API bridge.
   - Plan: Proceed to design phase.
3. **Spiral 3 (Implementation):**
   - Objective: Build final system.
   - Risk: System performance under high load.
   - Dev: Implement full code and run stress tests.
   - Plan: Proceed to deployment.

2.1.5 Rapid Application Development (RAD)

Methodology:

Rapid Application Development Steps RAD emphasizes rapid prototyping over strict planning.

1. **Business Modeling:** Identify information flow between business functions.
2. **Data Modeling:** Define data objects needed to support the business.
3. **Process Modeling:** Transform data objects to achieve business objectives (CRUD operations).
4. **Application Generation:** Use automated tools to convert process models into actual code and interfaces quickly.
5. **Testing and Turnover:** Test new components (reused components are already tested) and deploy the system rapidly.

Methodology:

Algorithm for Selecting a Process Model

1. **Are requirements perfectly clear and fixed?** If Yes → Use **Waterfall Model**.
2. **Is rapid delivery of core features needed?** If Yes → Use **Incremental Model** or **RAD**.
3. **Are requirements unclear and UI-heavy?** If Yes → Use **Prototyping Model**.
4. **Is the project very large and high-risk?** If Yes → Use **Spiral Model**.

Worked Example:

Choosing the Right Model **Problem:** Select and justify the appropriate software process model for the following two projects: Project A: A flight control system for a new commercial aircraft. Project B: A straightforward attendance tracking system for a school.
Solution:

1. **Project A Selection: Spiral Model.** *Justification:* A flight control system is extremely high-risk. Failure can cost lives. The Spiral model's emphasis on continuous risk analysis and prototyping ensures safety and reliability at every phase.
2. **Project B Selection: Waterfall Model.** *Justification:* An attendance tracking system is simple and its requirements are completely understood and unlikely to change. The straightforward, linear approach of Waterfall is most efficient here.

2.2 Agile Development

2.2.1 Agile Process Framework

Methodology:

Agile Scrum Implementation Steps

1. **Product Backlog Creation:** The Product Owner lists and prioritizes all required features (user stories).
2. **Sprint Planning:** The team selects top-priority items from the backlog to complete in a short iteration (Sprint) usually lasting 2-4 weeks.
3. **Sprint Execution and Daily Stand-ups:** The team works on sprint tasks. A brief daily meeting is

held to track progress and identify blockers.

4. **Sprint Review:** At the end of the sprint the team demonstrates a working, shippable product increment to stakeholders.
5. **Sprint Retrospective:** The team reflects on the sprint process to improve efficiency for the next sprint.

2.2.2 Agile Principles

Methodology:

Core Agile Principles Application

1. **Early Delivery:** Deliver working software frequently (weeks rather than months).
2. **Embrace Change:** Accept changing requirements even late in development.
3. **Collaboration:** Business people and developers must work together daily.
4. **Motivation:** Build projects around motivated individuals and give them the environment they need.
5. **Simplicity:** Maximize the amount of work not done. Keep designs simple.

Worked Example:

Executing an Agile Sprint for a Mobile App **Problem:** Plan and execute a 2-week Agile sprint for a new food delivery app.
Solution:

1. **Product Backlog:** Includes User Registration, Restaurant Search, Cart Management, and Payment Gateway.
2. **Sprint Planning:** The team decides they can complete User Registration and Restaurant Search in the next 2 weeks. These form the Sprint Backlog.
3. **Execution:** Developers build the registration database and search UI. Daily stand-ups ensure no one is blocked by server issues.
4. **Sprint Review:** After 2 weeks the team demonstrates the working registration and search functionality to the client.
5. **Retrospective:** The team notes that testing took longer than expected and decides to write automated tests earlier in the next sprint.

2.2.3 Comparison of Agile Development with Traditional Models

Methodology:

Evaluation Criteria for Comparison To compare Agile with Traditional (like Waterfall) apply the following evaluation metrics:

1. **Requirement Flexibility:** Check if the model can absorb requirement changes mid-development.
2. **Delivery Frequency:** Measure how often the client receives working software.
3. **Customer Involvement:** Assess how often the customer provides feedback.
4. **Risk Management:** Evaluate when risks are identified and mitigated.

Worked Example:

Comparative Analysis Scenario **Problem:** Compare the Agile methodology with the traditional Waterfall model using a structured evaluation.
Solution:

Parameter	**Agile Development**	**Traditional Waterfall**
Flexibility	Highly flexible. Adapts to changing requirements easily.	Rigid. Difficult to change requirements once a phase is complete.
Delivery	Continuous delivery of working software increments.	Single delivery of the complete software at the very end.
Customer Role	Continuous involvement and feedback throughout the project.	High involvement at the beginning (requirements) and end (testing).
Risk Management	Risks are identified and addressed early in every sprint.	Risks are often discovered late during the testing phase.
Project Size	Best suited for complex unpredictable projects.	Best suited for simple predictable projects with clear requirements.

CHAPTER 3

Software Requirement Analysis

3.1 Identify Software Requirements

Methodology:

Requirement Gathering and Analysis Algorithm The process of identifying software requirements can be structured as an algorithm to systematically filter raw needs into clear specifications.

1. **Step 1: Stakeholder Identification.** List all entities $E = \{e_1, e_2, \ldots, e_n\}$ that will interact with the system (e.g. clients, end-users, admins).
2. **Step 2: Elicitation.** For each entity e_i, collect raw requirements R_i using interviews or surveys. Combine them into a universal set of raw requirements $R_{raw} = \bigcup R_i$.
3. **Step 3: Conflict Resolution.** Scan R_{raw} for contradictory requirements r_a and r_b. If r_a and r_b are logically incompatible, consult stakeholders to drop or modify one.
4. **Step 4: Feasibility Check.** For each requirement $r \in R_{raw}$, evaluate technical and economic feasibility. If unfeasible, remove r from the set.
5. **Step 5: Documentation.** Output the finalized set R_{final} into formal models such as use-case diagrams or feature lists.

Worked Example:

Apply Requirement Gathering Algorithm to a Library System **Problem Statement:** Apply the requirement gathering algorithm to develop a Library Management System. Raw inputs from the librarian are: 1. Add new books. 2. Automatically ban users who are late by 1 day permanently. 3. Issue books to students. Raw inputs from students are: 4. Search books. 5. Keep books indefinitely without fines. Resolve conflicts and finalize the requirements.
Solution:

1. **Step 1: Stakeholder Identification.**
   $E = \{\text{Librarian}, \text{Students}\}$.
2. **Step 2: Elicitation.**
   $R_{librarian} = \{\text{Add books}, \text{Permanent ban for 1-day late}, \text{Issue books}\}$.
   $R_{student} = \{\text{Search books}, \text{Keep indefinitely without fines}\}$.
   $R_{raw} = \{\text{Add books}, \text{Permanent ban}, \text{Issue books}, \text{Search books}, \text{No fines}\}$.
3. **Step 3: Conflict Resolution.**
   Conflict detected between "Permanent ban for 1-day late" (r_2) and "Keep indefinitely without fines" (r_5). Both represent extreme edge cases.
   Resolution: Introduce a moderate policy: "Impose a daily fine for late returns and ban after 30 days of non-payment."
4. **Step 4: Feasibility Check.**

"Add books", "Issue books", "Search books", and the new "Fine system" are all technically and logically feasible.

5. **Step 5: Documentation.**
   $R_{final} = \{\text{Add books}, \text{Issue books}, \text{Search books}, \text{Calculate late fines}, \text{Suspend users after 30 days}\}$.

3.2 Prepare Software Requirement Specifications

Methodology:

SRS Preparation and Evaluation Framework A Software Requirement Specification (SRS) is critically important as it acts as a technical blueprint. It serves various users: **Customers** (for validation), **Developers** (for coding), **Testers** (for test cases) and **Managers** (for scheduling). Evaluate an SRS using these quality metrics:

1. **Step 1: Check Completeness.** Ensure no edge cases are missing. If input X is defined, verify if system behavior for invalid X is also defined.
2. **Step 2: Check Unambiguity.** Scan for subjective adjectives (e.g. "fast", "secure", "reliable"). Replace them with quantifiable thresholds (e.g. "$<$ 200 ms", "AES-256 encryption").
3. **Step 3: Check Verifiability.** For each requirement R, ask: "Can a tester write a precise pass/fail test script for this?" If no, rewrite R.
4. **Step 4: Check Consistency.** Ensure no requirement R_x logically contradicts requirement R_y.

Worked Example:

Refactoring Bad SRS Statements **Problem Statement:** An initial SRS draft contains the following requirements. Identify their bad characteristics and refactor them into good SRS statements.

- R_1: "The software shall be user-friendly."
- R_2: "The system will generate reports."
- R_3: "The login page should load quickly."

Solution:

1. **Analyze R_1:**
   - **Defect**: Ambiguous and unverifiable. "User-friendly" is highly subjective.
   - **Refactored**: "A new user shall be able to complete a primary transaction within 3 minutes of opening the app without external assistance."
2. **Analyze R_2:**
   - **Defect**: Incomplete. It does not specify what kind of reports are generated or for whom.
   - **Refactored**: "The system shall generate a monthly PDF sales report for the Administrator on the 1st of every month."
3. **Analyze R_3:**
   - **Defect**: Ambiguous. "Quickly" cannot be tested as it lacks a numerical threshold.
   - **Refactored**: "The login page shall render within 1.5 seconds over a standard 4G network connection."

3.3 Types of Requirements in SRS

Methodology:

Requirement Classification Procedure Given a list of system requirements, use the following logical conditions to classify them into Functional Requirements (FR) and Non-Functional Requirements (NFR).

1. **Step 1: Evaluate Action.** Does the requirement specify a direct input, process, or output? If yes, it is an FR.
   - Rule: If $\exists$ function $f(x) \to y \implies$ FR.
2. **Step 2: Evaluate Quality Constraint.** Does the requirement specify performance, security, reliability, or physical constraints? If yes, it is an NFR.
   - Rule: If it describes properties of $f(x) \implies$ NFR.
3. **Step 3: Document.** Label each evaluated item accordingly.

Worked Example:

Classifying FR and NFR for a Banking System **Problem Statement:** Apply the classification procedure to the following requirements for a banking application:

- Req_A: The system must allow users to transfer funds between accounts.
- Req_B: The database must encrypt all user passwords using Bcrypt.
- Req_C: The system must deduct a 2% fee for international transfers.
- Req_D: The system must be available 99.99% of the time.

Solution:

1. **Evaluate Req_A:** Specifies an action (transfer funds). Maps to a process.
   - **Result**: Functional Requirement (FR).
2. **Evaluate Req_B:** Specifies a security constraint on how data is stored rather than a direct user action.
   - **Result**: Non-Functional Requirement (NFR).
3. **Evaluate Req_C:** Specifies a business rule/calculation (deduct 2% fee). Maps to an internal mathematical process.
   - **Result**: Functional Requirement (FR).
4. **Evaluate Req_D:** Specifies a reliability/availability metric.
   - **Result**: Non-Functional Requirement (NFR).

CHAPTER 4

Software Project Management

4.1 Responsibility of Software Project Manager

Software project management is the art and discipline of planning and leading software projects to ensure they are completed on time, within budget, and to the required specifications. While this involves significant human and organizational elements, modern project management relies heavily on quantitative estimation and tracking methods.

Job Responsibility

The key responsibilities of a software project manager include:

1. **Project Planning and Scheduling:** Defining the scope and breaking down the project into manageable tasks.
2. **Cost and Effort Estimation:** Calculating the budget and resources required using empirical models.
3. **Risk Management:** Identifying potential pitfalls and computing their probable impact.
4. **Resource Allocation:** Assigning tasks to personnel while balancing workloads.
5. **Progress Tracking:** Using metrics to monitor schedule adherence.

Necessary Skill

To achieve these responsibilities, a manager requires:

- **Analytical Skills:** Ability to compute estimates, variances, and probabilities.
- **Technical Literacy:** Understanding the Software Development Life Cycle (SDLC) to properly sequence technical dependencies.
- **Problem Solving:** Algorithmic thinking to resolve scheduling conflicts and critical path delays.
- **Leadership and Communication:** Directing teams effectively and managing stakeholder expectations.

4.2 Scheduling

Scheduling involves deciding which tasks must be performed and determining the optimal timeline for their execution based on their dependencies and durations.

4.2.1 Work Breakdown Structure

A Work Breakdown Structure (WBS) is a hierarchical decomposition of the total scope of work to be carried out. By breaking a large project into smaller, granular activities, it becomes possible to mathematically estimate time and cost.

4.2.2 Activity Network and CPM

An Activity Network visually represents the sequence of project tasks. The Critical Path Method (CPM) is a mathematical algorithm used to predict project duration by identifying the longest sequence of dependent activities.

Methodology:

Critical Path Method Algorithm The Critical Path Method evaluates an activity network to find the minimum project duration and the slack of each task.

1. **Identify Activities and Dependencies:** List all tasks, their expected duration, and immediate predecessors.
2. **Forward Pass (Compute Earliest Times):** Calculate Earliest Start (ES) and Earliest Finish (EF).
   - $ES = \max(EF \text{ of all immediate predecessors})$ (For starting nodes, $ES = 0$)
   - $EF = ES + \text{Duration}$
3. **Backward Pass (Compute Latest Times):** Calculate Latest Finish (LF) and Latest Start (LS).
   - $LF = \min(LS \text{ of all immediate successors})$ (For ending nodes, $LF =$ Total Project Duration)
   - $LS = LF - \text{Duration}$
4. **Calculate Float (Slack):** Compute the flexibility in the start time of an activity without delaying the project.
   - Float $= LS - ES$ (which is strictly equal to $LF - EF$)
5. **Identify Critical Path:** Any activity with a Float of exactly 0 is on the Critical Path. A delay in any of these activities will delay the entire project.

Worked Example:

Finding Critical Path and Project Duration **Problem:** Given the following activities with their durations (in weeks) and predecessors, determine the critical path and total project duration.

Activity	Predecessor	Duration
A	-	3
B	A	4
C	A	2
D	B	5
E	C	1
F	C	2
G	D	4
H	E and F	3

Solution:
Step 1: Forward Pass (Calculate ES and EF)

- Activity A: ES $= 0$, EF $= 0 + 3 = 3$
- Activity B: ES $= 3$, EF $= 3 + 4 = 7$
- Activity C: ES $= 3$, EF $= 3 + 2 = 5$
- Activity D: ES $= 7$, EF $= 7 + 5 = 12$
- Activity E: ES $= 5$, EF $= 5 + 1 = 6$
- Activity F: ES $= 5$, EF $= 5 + 2 = 7$

- Activity G: ES = 12, EF = $12 + 4 = 16$
- Activity H: ES = $\max(EF_E, EF_F) = \max(6, 7) = 7$, EF = $7 + 3 = 10$

The maximum EF across all end tasks (G and H) is 16. Therefore, the total project duration is 16 weeks. **Step 2: Backward Pass (Calculate LF and LS)** Start from the project end duration (16 weeks). For tasks with no successors (G and H), LF = 16.

- Activity G: LF = 16, LS = $16 - 4 = 12$
- Activity H: LF = 16, LS = $16 - 3 = 13$
- Activity D (Predecessor to G): LF = $LS_G = 12$, LS = $12 - 5 = 7$
- Activity E (Predecessor to H): LF = $LS_H = 13$, LS = $13 - 1 = 12$
- Activity F (Predecessor to H): LF = $LS_H = 13$, LS = $13 - 2 = 11$
- Activity B (Predecessor to D): LF = $LS_D = 7$, LS = $7 - 4 = 3$
- Activity C (Predecessor to E and F): LF = $\min(LS_E, LS_F) = \min(12, 11) = 11$, LS = $11 - 2 = 9$
- Activity A (Predecessor to B and C): LF = $\min(LS_B, LS_C) = \min(3, 9) = 3$, LS = $3 - 3 = 0$

Step 3: Calculate Float (LS - ES)

- A: $0 - 0 = 0$ (Critical)
- B: $3 - 3 = 0$ (Critical)
- C: $9 - 3 = 6$
- D: $7 - 7 = 0$ (Critical)
- E: $12 - 5 = 7$
- F: $11 - 5 = 6$
- G: $12 - 12 = 0$ (Critical)
- H: $13 - 7 = 6$

Conclusion: Activities with a float of 0 form the critical path. The critical path is **A** $\rightarrow$ **B** $\rightarrow$ **D** $\rightarrow$ **G** with a total minimum completion duration of **16 weeks**.

4.2.3 PERT Chart

Program Evaluation and Review Technique (PERT) is a probabilistic method used when individual task durations are highly uncertain. Instead of a single duration, PERT uses three estimates to compute a weighted expected time and the variance for each activity.

Methodology:

PERT Expected Time and Variance Calculation To handle uncertainty in scheduling, follow these statistical estimation steps:

1. **Estimate Three Time Values:** For each activity, determine:
   - Optimistic Time (T_o): Minimum time under ideal conditions.
   - Most Likely Time (T_m): Expected time under normal conditions.
   - Pessimistic Time (T_p): Maximum time assuming heavy delays.

2. **Calculate Expected Time (T_e):** The weighted average favoring the most likely time.

$$T_e = \frac{T_o + 4T_m + T_p}{6}$$

3. **Calculate Variance (σ^2):** The degree of uncertainty for the activity.

$$\sigma^2 = \left(\frac{T_p - T_o}{6}\right)^2$$

4. **Project Variance:** To find the total project uncertainty, sum the variances of only the activities on the critical path.

Worked Example:

PERT Estimation and Uncertainty Analysis **Problem:** An essential software module requires development. The manager gathers estimates from the engineering team: an optimistic time of 4 days, a most likely time of 7 days, and a pessimistic time of 16 days. Calculate the expected completion time and its variance.
Solution: Step 1: Identify Given Values

- $T_o = 4$
- $T_m = 7$
- $T_p = 16$

Step 2: Calculate Expected Time (T_e)

$$T_e = \frac{T_o + 4T_m + T_p}{6}$$

$$T_e = \frac{4 + 4(7) + 16}{6} = \frac{4 + 28 + 16}{6}$$

$$T_e = \frac{48}{6} = 8 \text{ days}$$

Step 3: Calculate Variance (σ^2)

$$\sigma^2 = \left(\frac{T_p - T_o}{6}\right)^2$$

$$\sigma^2 = \left(\frac{16 - 4}{6}\right)^2 = \left(\frac{12}{6}\right)^2$$

$$\sigma^2 = 2^2 = 4 \text{ days}^2$$

Conclusion: The weighted expected time for the module is 8 days, with a variance of 4.

4.2.4 Gantt Chart

A Gantt chart is a horizontal bar chart used to visually represent a project schedule. The X-axis represents the timeline, and the Y-axis lists the tasks. The length of each bar represents the duration (T_e) of the task, and its position along the timeline corresponds to its calculated Earliest Start (ES) and Earliest Finish (EF). Dependencies can be drawn as arrows linking the end of a predecessor bar to the start of a successor bar.

4.3 Risk Management

Risk management is the systematic process of anticipating, analyzing, and resolving risks before they derail a project schedule or budget.

4.3.1 Identification

The first step is itemizing potential threats. These may include staff turnover, requirement churn, hardware failures, or dependency on unproven third-party APIs.

4.3.2 Assessment

Not all risks require equal attention. Assessment involves quantifying risks to prioritize mitigation efforts effectively.

Methodology:

Risk Exposure Calculation Risk Exposure (RE), also known as expected loss, quantifies the magnitude of a risk by factoring its severity and probability.

1. **Determine Probability (P):** Estimate the likelihood of the risk occurring. Express this as a decimal from 0.0 (impossible) to 1.0 (certain).
2. **Determine Impact (C):** Estimate the cost of the risk if it occurs (measured in dollars, weeks of delay, or a severity scale).
3. **Calculate Risk Exposure (RE):**
$$RE = P \times C$$
4. **Prioritize:** Rank the identified risks in descending order based on their RE value. High RE risks are targeted for active mitigation.

Worked Example:

Calculating and Prioritizing Risk Exposure **Problem:** A software project is evaluated for threats, resulting in three major risks:

- **Risk A (API Change):** The external payment API provider might change their specification. Probability = 40%. Cost of refactoring = \$15,000.
- **Risk B (Hardware Failure):** The legacy build server might crash. Probability = 10%. Cost of replacement and lost productivity = \$50,000.
- **Risk C (Key Developer Leaves):** A specialized database expert might accept another job. Probability = 25%. Cost of hiring, onboarding, and delay = \$30,000.

Calculate the Risk Exposure for each risk and determine the priority order for the mitigation plan.
Solution:
Step 1: Calculate RE for Risk A (API Change)

- $P = 0.40$
- $C = \$15,000$
- $RE_A = 0.40 \times 15,000 = \$6,000$

Step 2: Calculate RE for Risk B (Hardware Failure)

- $P = 0.10$
- $C = \$50,000$
- $RE_B = 0.10 \times 50,000 = \$5,000$

Step 3: Calculate RE for Risk C (Key Developer Leaves)

- $P = 0.25$

- $C = \$30,000$
- $RE_C = 0.25 \times 30,000 = \$7,500$

Step 4: Prioritize based on RE Ordering the risks from highest to lowest expected loss:

1. Risk C: \$7,500
2. Risk A: \$6,000
3. Risk B: \$5,000

Conclusion: The manager must prioritize mitigating Risk C (Key Developer Leaves) as it carries the highest overall risk exposure, despite having a lower impact severity than Risk B and a lower probability than Risk A.

4.3.3 Mitigation

Mitigation strategies are action plans designed to modify the Risk Exposure equation by either reducing the Probability (P) or reducing the Impact (C). Standard strategies include:

- **Avoidance:** Adjust project scope or architecture to bypass the risk entirely (e.g., using an older, stable API instead of a beta version). Reduces P to 0.
- **Transfer:** Shift the financial or functional burden to a third party (e.g., buying insurance or outsourcing the high-risk module). Limits C.
- **Reduction (Mitigation):** Invest time early to reduce probability or severity (e.g., cross-training employees to reduce the impact if a key developer leaves). Lowers P or C.
- **Acceptance:** Recognize the risk but determine that the cost of mitigation outweighs the Risk Exposure. A contingency budget is established to cover the loss if it occurs.

CHAPTER 5
Software Design

5.1 Software Design Process

Methodology:

Software Design Process Steps The software design process translates user requirements into a suitable form for the programmer to start coding. The key computational and analytical activities follow a clear algorithm:

1. **Architectural Design:** Identify the main structural components of the system and establish their relationships.
2. **High-Level Design:** Decompose the architectural components into less complex modules. Define interfaces between these modules.
3. **Detailed Design:** Design the internal details of each module including data structures and algorithmic workflows.
4. **Database Design:** Formulate the logical and physical data models based on the required data structures.
5. **User Interface Design:** Design how the user will interact with the software applying interaction rules.

Worked Example:

Applying Design Methodologies Problem: Choose and apply a design methodology for a Simple Calculator application.
Solution:

1. **Identify Methodology:** Function-Oriented Design using a Top-Down Approach.
2. **Top-Level Function:** Main Calculator Controller.
3. **Sub-Functions Decomposition:**
   - Function 1: Read Input
   - Function 2: Perform Arithmetic (Addition Subtraction Multiplication Division)
   - Function 3: Display Output
4. **Execution Logic:** The controller reads input values executes a conditional statement to call the specific arithmetic function based on the operator and passes the result to the display module.

5.2 Introduction of Cohesion

Methodology:

Evaluating Cohesion Level Cohesion measures the strength of relationships between algorithmic elements within a single module. High cohesion is desirable for maintainable software. The types of cohesion ranked from best (highest) to worst (lowest) are:

1. **Functional Cohesion:** All elements contribute to a single well-defined computational task.
2. **Sequential Cohesion:** The output data of one element is the direct input to the next element.
3. **Communicational Cohesion:** Elements operate on the same input data set or produce the same output data.
4. **Procedural Cohesion:** Elements are grouped because they always follow a specific sequence of execution in an algorithm.
5. **Temporal Cohesion:** Elements are grouped because they are executed simultaneously (e.g. initialization routines).
6. **Logical Cohesion:** Elements are grouped logically even if they perform fundamentally different tasks (e.g. grouping all input/output routines together).
7. **Coincidental Cohesion:** Elements have no meaningful algorithmic relationship.

Worked Example:

Determining Cohesion Type Problem: A software module contains the following tasks: Initialize database connection open log file and reset user session variables. Identify its cohesion type.
Solution:

1. **Analyze tasks:** The module performs database initialization file handling and variable resetting.
2. **Identify relationship:** These tasks do not share data variables or contribute to a single mathematical or functional goal. However they are all executed precisely when the system starts up.
3. **Determine cohesion:** Since the elements are executed at the same temporal phase (during system startup) this is an example of **Temporal Cohesion**.

5.3 Introduction of Coupling

Methodology:

Evaluating Coupling Level Coupling measures the degree of interdependence between different software modules. Low coupling is desirable to prevent cascading errors. The types of coupling ranked from best (lowest) to worst (highest) are:

1. **Data Coupling:** Modules share discrete data elements through parameters (e.g. passing an integer or float value).
2. **Stamp Coupling:** Modules share a composite data structure (like a struct or object) but only utilize specific parts of it.
3. **Control Coupling:** One module controls the algorithmic execution flow of another by passing control information (e.g. passing a boolean flag to trigger an if-else block).
4. **External Coupling:** Modules are tied to a specific external environment or communication protocol.

5. **Common Coupling:** Modules share and modify global data variables.
6. **Content Coupling:** One module directly manipulates or relies on the internal local variables of another module.

Worked Example:

Determining Coupling Type Problem: Module A passes a boolean flag variable 'isError' to Module B. If 'isError' is true Module B branches to print an error message; otherwise it logs a success message. Identify the type of coupling between Module A and Module B.
Solution:

1. **Analyze interaction:** Module A explicitly passes a boolean variable to Module B.
2. **Determine purpose of variable:** The variable 'isError' acts as a control signal dictating the internal branching logic of Module B.
3. **Determine coupling:** Passing control information to manipulate a module's execution path is a direct example of **Control Coupling**.

5.4 Data Flow Diagram Model

Methodology:

Designing a Data Flow Diagram A Data Flow Diagram (DFD) maps out the flow of data elements through a computational system. **DFD Core Symbols:**

- **External Entity (Square):** Represents data sources or sinks outside the system boundary.
- **Process (Circle):** Represents a computational transformation of incoming data into outgoing data.
- **Data Store (Parallel Lines):** Represents a repository where data is held for future retrieval.
- **Data Flow (Arrow):** Represents the pipeline through which data packets travel between components.

DFD Architectural Levels:

- **Level-0 (Context Diagram):** Treats the entire software system as a single process node interacting with external entities.
- **Level-1 DFD:** Decomposes the primary process into major sub-processes and local data stores.

Design Validation Rules:

1. A process must possess at least one input data flow and at least one output data flow.
2. Data cannot flow directly from one external entity to another.
3. Data cannot flow directly between two data stores; it must be transformed by a process.

Worked Example:

Designing a Context DFD Problem: Outline the components and flows for a Level-0 DFD of an Automated Teller Machine (ATM) System.
Solution:

1. **Identify the Central Node:** The entire ATM System acts as Process 0.
2. **Identify External Entities:** The user (Customer) and the backend server (Bank Database).

3. **Map the Data Flows:**

Source Entity	Data Packet Flow	Destination Entity
Customer	PIN and Transaction Details	ATM System
ATM System	Authentication Request	Bank Database
Bank Database	Authorization Status	ATM System
ATM System	Dispensed Cash	Customer

4. **Validation:** All data flows interact with the central Process 0 ensuring no direct communication between Customer and Bank Database.

5.5 Introduction of Data Dictionary

Methodology:

Creating a Data Dictionary Entry A data dictionary acts as a centralized repository containing precise mathematical and structural definitions of data elements. It ensures semantic consistency. **Standard Algebraic Notation:**

- `=` Denotes "is defined as" or "is composed of".
- `+` Denotes concatenation or "and".
- `[x | y]` Denotes a logical OR (choose either x or y).
- `{x}` Denotes iteration or repetition of element x.
- `(x)` Denotes that element x is optional.
- `**` Encloses a comment or descriptive text.

Worked Example:

Defining a Data Dictionary Problem: Formulate a data dictionary definition for a "Student Record" consisting of a required Roll Number a Name an optional Email Address and multiple Phone Numbers.
Solution:

1. **Identify Structural Components:** Roll Number Name Email Phone Number.
2. **Apply Algebraic Notation:**
   - Student_Record = Roll_Number + Name + (Email) + {Phone_Number}
   - Roll_Number = 12 * [0-9] * 12 ** Exactly 12 digits **
   - Name = First_Name + (Middle_Name) + Last_Name
   - Phone_Number = [Home_Phone | Mobile_Phone]

5.6 User Interface Design

Methodology:

User Interface Design Guidelines User Interface (UI) design dictates the interaction algorithms between the human operator and the machine. **Primary UI Modalities:**

1. **Command Line Interface (CLI):** Text-based deterministic interaction requiring exact syntax.
2. **Graphical User Interface (GUI):** Visual event-driven interaction utilizing Windows Icons Menus

and Pointers (WIMP).

Algorithmic Characteristics of Good UI:

- **Consistency:** Visual elements and interaction states must remain uniform across the system state matrix.
- **Simplicity:** Minimize cognitive load by reducing extraneous visual noise.
- **Feedback:** The system must output observable state changes in response to every user input event.
- **Forgiveness:** Implement error-recovery algorithms (e.g. undo functions) to reverse unintended state mutations.

Worked Example:

Designing a GUI Component Problem: Specify the UI components required for a standard Authentication Screen and justify them using algorithmic UI principles.
Solution:

1. **Define Component Matrix:**
   - **Input Field 1:** Username text string.
   - **Input Field 2:** Password masked text string.
   - **Action Button:** 'Login' (Triggers authentication algorithm).
   - **Action Link:** 'Forgot Password' (Triggers recovery workflow).
2. **Apply UI Characteristics Justification:**
   - **Simplicity:** The screen contains strictly the 4 components necessary for state transition.
   - **Feedback Loop:** If the authentication algorithm returns false the UI dynamically renders a visual error string 'Invalid credentials'.
   - **Forgiveness:** The 'Forgot Password' link provides an algorithmic path to recover from a forgotten password state.

CHAPTER 6
Software Testing

6.1 Introduction of Testing

Software testing is the process of evaluating a system or its components to determine whether it satisfies specified requirements. In computational terms, it involves executing a program with test data to find errors, measure performance, and ensure reliability.

6.2 Test Cases and Test Suit

A **test case** is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly. A **test suite** is a structured collection of test cases.

Methodology:

Test Case Design Procedure

1. Identify the requirement or feature to be tested.
2. Define the input data (Test Data).
3. Determine the expected output.
4. Execute the test and record the actual output.
5. Compare actual and expected outputs to determine Pass or Fail.

Worked Example:

Design Test Cases for User Login Design a test suite for a login system that requires a username of exactly 8 characters and a numeric password.
Solution:

Test ID	Username	Password	Expected Output	Status
TC01	user1234	12345	Login Successful	Pass
TC02	user123	12345	Error: Username length invalid	Pass
TC03	user1234	abcde	Error: Password must be numeric	Pass
TC04	user12345	12345	Error: Username length invalid	Pass

6.3 Introduction to Verification and Validation

Verification and Validation (V&V) are independent procedures used together to check that a software system meets its specifications and fulfills its intended purpose.

- **Verification:** The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. It answers the question: *Are we building the product right?*

- **Validation:** The process of evaluating software at the end of the development process to determine whether it satisfies user requirements. It answers the question: *Are we building the right product?*

6.4 Unit Testing

Unit testing involves testing individual components or modules of a software application independently to ensure each part functions correctly.

Methodology:

Unit Testing Process

1. Isolate the unit of code to be tested.
2. Provide mock inputs or stubs if the unit depends on other modules.
3. Execute the unit with various test cases including normal, edge, and invalid inputs.
4. Verify that the unit returns the expected result.

Worked Example:

Unit Test for Triangle Validity A function takes three integer side lengths and returns whether they can form a valid triangle. Write unit tests for this function.
Solution: The mathematical condition for a valid triangle is that the sum of the lengths of any two sides must be strictly greater than the length of the remaining side.

Test Case	Inputs (a b c)	Expected Output	Reason
UT01	3 4 5	Valid	$3+4>5$, $4+5>3$, $3+5>4$
UT02	1 2 3	Invalid	$1+2$ is not >3
UT03	0 4 5	Invalid	Side length cannot be zero
UT04	-1 2 2	Invalid	Side length cannot be negative

6.5 Black-box Testing

Black-box testing treats the software as a black box without knowing its internal code structure. It focuses on inputs and outputs. Key computational methods for generating test data include Equivalence Partitioning and Boundary Value Analysis.

Methodology:

Boundary Value Analysis Boundary Value Analysis (BVA) focuses on the boundaries of input domains.

1. Identify the input variable and its valid range $[min, max]$.
2. Test at the exact boundaries: min and max.
3. Test just below the boundaries: $min - 1$ and $max - 1$.
4. Test just above the boundaries: $min + 1$ and $max + 1$.
5. Test a nominal (normal) value well within the range.

Methodology:

Equivalence Partitioning Equivalence Partitioning (EP) divides inputs into groups that are expected to exhibit similar behavior.

1. Divide the input domain into valid and invalid partitions.
2. Select one representative value from each partition for testing.

Worked Example:

BVA and EP for Age Input An application accepts an age parameter which must be an integer between 18 and 60 inclusive. Apply Boundary Value Analysis and Equivalence Partitioning to design test data.
Solution:
Step 1: Equivalence Partitioning

- **Invalid Partition 1:** Age < 18 (e.g., 15)
- **Valid Partition:** $18 \leq$ Age ≤ 60 (e.g., 30)
- **Invalid Partition 2:** Age > 60 (e.g., 65)

Step 2: Boundary Value Analysis The boundaries are $min = 18$ and $max = 60$. The test values are:

- $min - 1 = 17$ (Invalid)
- $min = 18$ (Valid)
- $min + 1 = 19$ (Valid)
- Nominal $= 35$ (Valid)
- $max - 1 = 59$ (Valid)
- $max = 60$ (Valid)
- $max + 1 = 61$ (Invalid)

Step 3: Combined Test Cases

Input Age	Expected Output	Testing Method
15	Error: Underage	EP
17	Error: Underage	BVA
18	Accepted	BVA
19	Accepted	BVA
30	Accepted	EP and BVA Nominal
59	Accepted	BVA
60	Accepted	BVA
61	Error: Overage	BVA
65	Error: Overage	EP

6.6 White-box Testing

White-box testing requires knowledge of the internal code structure. A fundamental computational technique is Basis Path Testing, which uses Cyclomatic Complexity to determine the number of independent paths through the code.

Methodology:

Cyclomatic Complexity and Basis Path Testing

1. Draw the control flow graph (CFG) from the source code.
2. Calculate Cyclomatic Complexity $V(G)$ using one of the formulas:

- $V(G) = E - N + 2$ (where E = number of edges, N = number of nodes)
- $V(G) = P + 1$ (where P = number of predicate or decision nodes)
- $V(G)$ = Number of enclosed regions + 1

3. Identify exactly $V(G)$ independent paths through the control flow graph.
4. Design test cases to execute each identified path.

Worked Example:

Calculate Cyclomatic Complexity for Conditionals Consider the following pseudocode:

```
1. function checkNumber(x)
2.   if x > 0 then
3.       print("Positive")
4.   else
5.       if x < 0 then
6.           print("Negative")
7.       else
8.           print("Zero")
9.       end if
10.  end if
11. end function
```

Solution:
Step 1: Identify Nodes and Edges

- **Node 1:** Start and `if x > 0` (Decision node)
- **Node 2:** `print("Positive")`
- **Node 3:** `if x < 0` (Decision node)
- **Node 4:** `print("Negative")`
- **Node 5:** `print("Zero")`
- **Node 6:** End function

Edges: $1 \to 2$ (True path for $x > 0$)
$1 \to 3$ (False path for $x > 0$)
$2 \to 6$
$3 \to 4$ (True path for $x < 0$)
$3 \to 5$ (False path for $x < 0$)
$4 \to 6$
$5 \to 6$
Edges $E = 7$, Nodes $N = 6$.
Step 2: Calculate Cyclomatic Complexity Using the formula $V(G) = E - N + 2$:

$$V(G) = 7 - 6 + 2 = 3$$

Alternatively, using predicate nodes P: Nodes 1 and 3 are predicate nodes.

$$V(G) = P + 1 = 2 + 1 = 3$$

Step 3: Identify Basis Paths Since $V(G) = 3$, there are 3 independent paths:

- **Path 1:** $1 \to 2 \to 6$ (Test case: $x = 5$)
- **Path 2:** $1 \to 3 \to 4 \to 6$ (Test case: $x = -3$)
- **Path 3:** $1 \to 3 \to 5 \to 6$ (Test case: $x = 0$)

Worked Example:

Cyclomatic Complexity for a Loop Determine the Cyclomatic Complexity and independent paths for the following loop logic.

```
1. i = 0
2. while (i < 10) do
3.     if (i % 2 == 0) then
4.         print("Even")
5.     end if
6.     i = i + 1
7. end while
8. print("Done")
```

Solution:
Step 1: Identify Nodes and Edges

- **Node 1:** Initialization `i = 0`
- **Node 2:** Loop condition `while (i < 10)` (Decision node)
- **Node 3:** `if (i % 2 == 0)` (Decision node)
- **Node 4:** `print("Even")`
- **Node 5:** `i = i + 1`
- **Node 6:** `print("Done")`

Edges: $1 \to 2$
$2 \to 3$ (True path for while)
$2 \to 6$ (False path for while)
$3 \to 4$ (True path for if)
$3 \to 5$ (False path for if)
$4 \to 5$
$5 \to 2$ (Loop back)
Total Edges $E = 7$. Total Nodes $N = 6$.
Step 2: Calculate Complexity Using the formula $V(G) = E - N + 2$:

$$V(G) = 7 - 6 + 2 = 3$$

Using predicate nodes P: The decision nodes are Node 2 and Node 3. Therefore, $P = 2$.

$$V(G) = P + 1 = 2 + 1 = 3$$

Step 3: Basis Paths

- **Path 1:** $1 \to 2 \to 6$ (Loop is bypassed)
- **Path 2:** $1 \to 2 \to 3 \to 5 \to 2 \to 6$ (Loop executes, if condition is false)
- **Path 3:** $1 \to 2 \to 3 \to 4 \to 5 \to 2 \to 6$ (Loop executes, if condition is true)

Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Unit 4: Software Project Management

Program Evaluation and Review Technique (PERT)

The Expected Time (T_e) for a task in PERT is calculated using a weighted average of three time estimates: optimistic, most likely, and pessimistic times. This formula places more weight on the most likely estimate to reflect a realistic expected duration.

$$T_e = \frac{T_o + 4T_m + T_p}{6}$$

Where:

- T_e = **Expected Time**: The calculated average duration of a task.
- T_o = **Optimistic Time**: The minimum possible time required to complete a task, assuming everything goes perfectly.
- T_m = **Most Likely Time**: The best estimate of the time required to complete a task, assuming everything proceeds normally.
- T_p = **Pessimistic Time**: The maximum possible time required to complete a task, assuming everything goes wrong.