Fsd (4331604) - 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.

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.

Contents

List of Figures

List of Tables

CHAPTER 1

About Software Development

1.1 Software Definition and Characteristics

Methodology:

Software Characteristic Evaluation To evaluate the characteristics of a software product mathematically or logically, follow these steps:

1. **Identify Software Type:** Determine if it is system software, application software, engineering or scientific software, embedded software or web application.
2. **Analyze Wear and Tear:** Confirm that software does not "wear out" like hardware. Plot the failure rate curve.
3. **Check Custom-built vs Assembled:** Determine if the software components are custom-built or assembled from reusable components.
4. **Calculate Defect Density:** Measure the number of defects found per KLOC (Kilo Lines of Code).

Worked Example:

Evaluate the characteristics of a newly developed Payroll System **Step 1: Identify Software Type**
The Payroll System falls under *Application Software* as it is designed to process business data.
Step 2: Analyze Wear and Tear
Unlike hardware which has a bathtub curve for failure rates, the payroll software does not wear out due to environmental factors. However failure rate may spike due to unhandled changes.
Step 3: Check Custom-built vs Assembled
The system uses pre-built database drivers (assembled) but the core business logic is custom-built.
Step 4: Calculate Defect Density
If the system has 50 KLOC and 100 defects were found during testing:

$$\text{Defect Density} = \frac{\text{Total Defects}}{\text{Total KLOC}} = \frac{100}{50} = 2 \text{ defects per KLOC}$$

The software is thus characterized by a relatively low defect density.

1.2 Characteristics of Web-based Application

Methodology:

Analyzing Web Application Characteristics A web-based application is evaluated using the following specific parameters:

1. **Network Intensiveness:** Measure the bandwidth required and request frequency.
2. **Concurrency:** Determine the number of concurrent users supported without performance degradation.

3. **Unpredictable Load:** Analyze how the application handles sudden spikes in traffic.
4. **Availability:** Calculate the required uptime percentage.
5. **Data Driven:** Assess the volume and complexity of data processed and presented.

Worked Example:

Calculate availability and concurrency metrics for an E-commerce WebApp **Step 1: Network Intensiveness**
The WebApp requires 500 KB per page load. For 1000 requests per minute, the required bandwidth is 500 KB $\times$ 1000 = 500 MB per minute.
Step 2: Concurrency
The server configuration allows a maximum of 5000 concurrent database connections. Thus maximum concurrency is 5000 users at any exact second.
Step 3: Unpredictable Load
During a flash sale, traffic spikes to 20000 concurrent users. The application utilizes a load balancer to distribute requests across 4 servers to handle this load.
Step 4: Availability
If the application must guarantee 99.9% uptime per year (365 days):

$$\text{Downtime Allowed} = 365 \times 24 \times 60 \times (1 - 0.999) = 525.6 \text{ minutes per year}$$

The system meets characteristics if unplanned downtime is less than 525.6 minutes annually.

1.3 Software Engineering - A Layered Technology

Methodology:

Layered Technology Model Application The layered technology approach to software engineering involves applying layers systematically:

1. **Quality Focus Layer:** Define the specific quality metrics the product must meet.
2. **Process Layer:** Select the framework that defines activities and tasks.
3. **Methods Layer:** Choose the technical "how-tos" for building software.
4. **Tools Layer:** Implement automated tools to support the process and methods.

Worked Example:

Apply the Layered Technology Model to a Library Management System **Step 1: Quality Focus Layer**
Goal: System must achieve a response time of < 2 seconds and maintain 0 data loss during power failures.
Step 2: Process Layer
Adopt the Agile process model. Define 2-week sprints for delivering continuous functional increments.
Step 3: Methods Layer
Apply Object-Oriented Analysis and Design (OOAD). Create use cases for "Issue Book" and "Return Book". Define classes such as `Member`, `Book` and `Transaction`.
Step 4: Tools Layer
Use Jira for process tracking, UML modeling tools for design, Git for version control and JUnit for automated testing.

1.4 Software Myths

Methodology:

Software Myth Analysis Algorithm To systematically debug and correct management or developer myths:

1. **Identify the Myth:** State the commonly held but false belief.
2. **Analyze the Impact:** Determine how this myth negatively affects project schedule or quality.
3. **Apply the Reality Check:** Provide the factual and empirical evidence that debunks the myth.
4. **Calculate the Correction Factor:** If applicable, calculate the true cost or time using empirical models.

Worked Example:

Debunking the Myth of Adding Manpower to a Late Project **Step 1: Identify the Myth**
Myth: "If we get behind schedule we can just add more programmers and catch up."
Step 2: Analyze the Impact
Adding 3 new developers to a project that is 2 months late. Management expects the timeline to shorten linearly.
Step 3: Apply the Reality Check
Reality: Adding manpower to a late software project makes it later (Brooks's Law). New team members require training and increase communication overhead.
Step 4: Calculate the Correction Factor
Original team: 4 members. Communication channels $= \frac{n(n-1)}{2} = \frac{4\times3}{2} = 6$.
New team: 7 members. Communication channels $= \frac{7\times6}{2} = 21$.
The communication complexity increases by $\frac{21}{6} = 3.5$ times. The time spent training and communicating outweighs the productive coding time of the new members further delaying the project.

1.5 Software Process Framework and Umbrella Activities

Methodology:

Implementing Process Framework and Umbrella Activities A process framework consists of core activities and umbrella activities. Execute them via these steps:

1. **Execute Core Framework Activities:** Run through Communication, Planning, Modeling, Construction and Deployment.
2. **Apply Software Project Tracking:** Monitor progress against the project plan.
3. **Perform Risk Management:** Calculate Risk Exposure (RE) for identified risks.
4. **Conduct Software Quality Assurance:** Execute technical reviews and calculate defect removal efficiency (DRE).
5. **Manage Configuration:** Track versions and baseline changes.

Worked Example:

Calculate Risk Exposure and Defect Removal Efficiency for a Project **Step 1: Execute Core Activities**
The team completes the modeling phase and moves to construction.
Step 2: Apply Project Tracking
Project is at week 4 out of 10. Completed 40% of tasks, on schedule.
Step 3: Perform Risk Management

Identify a risk: “Key developer might leave”.
Probability of occurrence (P) = 0.20 (20%).
Cost Impact (C) = \$15,000 in delays and hiring.

$$\text{Risk Exposure (RE)} = P \times C = 0.20 \times 15000 = \$3,000$$

Set aside a contingency reserve of \$3,000.

Step 4: Conduct Software Quality Assurance

During technical reviews before testing, the team finds 40 defects (E). During actual testing and after release, 10 defects are found (D).

$$\text{Defect Removal Efficiency (DRE)} = \frac{E}{E+D} \times 100 = \frac{40}{40+10} \times 100 = 80\%$$

A DRE of 80% indicates strong quality assurance mechanisms during umbrella activities.

CHAPTER 2

Software Life Cycle Models

2.1 Traditional Software Life Cycle Models

2.1.1 Waterfall Model

Methodology:

Waterfall Model Phases The Waterfall Model is a sequential linear approach to software development. It flows downwards through several phases where each phase must be completed before the next begins.

1. **Requirements Gathering and Analysis**: Collect all possible requirements of the system to be developed and document them in a requirement specification document.
2. **System Design**: Study the requirements from the first phase and prepare system and software design. Specify overall system architecture.
3. **Implementation**: Develop the system in small programs called units which are integrated in the next phase. Each unit is developed and tested for its functionality.
4. **Integration and Testing**: Integrate all the units developed in the implementation phase into a system after testing each unit. Test the entire system for any faults and failures.
5. **Deployment of System**: Once functional and non-functional testing is done deploy the product in the customer environment or release it into the market.
6. **Maintenance**: Fix any issues that come up in the client environment. Release patches or new versions.

Worked Example:

Applying Waterfall Model to a Payroll System **Problem:** Apply the Waterfall Model steps to develop a simple corporate Payroll Management System.
Solution:

1. **Requirements Gathering and Analysis**: Identify that the HR department needs a system to calculate monthly salaries based on attendance, tax deductions, and bonuses. Output: Software Requirements Specification (SRS) document outlining all calculation formulas.
2. **System Design**: Design the database schema for Employees, Attendance, and Salary. Create UI wireframes for HR login and report generation. Output: System Design Document.
3. **Implementation**: Programmers write the code in Python or Java. Create individual modules: Authentication module, Attendance module, Calculation module, and Report module.
4. **Integration and Testing**: Combine all modules. QA team tests the system by inputting dummy employee data and verifying if the calculated salary output matches manual calculations exactly.

5. **Deployment of System**: Install the software on the company servers and train the HR staff to use it.
6. **Maintenance**: Update the tax calculation logic next year when the government introduces new tax brackets.

2.1.2 Incremental Model

Methodology:

Incremental Model Process The Incremental Model divides the product into multiple standalone builds. The product is designed implemented and tested incrementally until the product is finished.

1. **Requirement Analysis**: Understand the overall system requirements and divide them into multiple prioritized functional increments.
2. **Design and Development (Increment 1)**: Design and code the first and most critical increment (core functionality).
3. **Testing and Delivery (Increment 1)**: Test the first increment and deliver it to the customer for immediate use and evaluation.
4. **Iteration**: For each subsequent increment (2 to n):
   (a) Design and develop the new increment.
   (b) Integrate the new increment with the previously delivered increments.
   (c) Test the integrated system.
   (d) Deliver the updated software to the customer.
5. **Finalization**: Continue the process until all planned increments are integrated and the final product is complete.

Worked Example:

Applying Incremental Model to an Ecommerce App **Problem:** Plan the development of an E-commerce Application using the Incremental Model.
Solution:

1. **Requirement Analysis**: Divide the app into three increments based on priority:
   - Increment 1: User Registration, Product Catalog, and Shopping Cart.
   - Increment 2: Online Payment Gateway Integration and Order Tracking.
   - Increment 3: User Reviews, Recommendations, and Wishlist.
2. **Increment 1 (Core Build)**: Design, implement, and test the product catalog and cart. Users can view products and add them to a cart but must use Cash on Delivery. Deploy this to users immediately.
3. **Increment 2 (Addition)**: Develop the secure payment gateway module. Integrate it with the existing catalog and cart from Increment 1. Test the payment flow. Deploy the update so users can now pay online.
4. **Increment 3 (Enhancement)**: Develop the recommendation engine and review system. Integrate with the existing codebase. Test thoroughly and deliver the final complete product.

2.1.3 Prototyping Model

Methodology:

Prototyping Model Steps The Prototyping Model involves building a working replica (prototype) of the system before developing the actual software. It is used when user requirements are not well understood.

1. **Initial Requirement Gathering**: Collect basic and known requirements from the customer. Detailed requirements are not needed at this stage.
2. **Quick Design**: Create a preliminary design focusing on the user interface and system inputs/outputs rather than internal logic.
3. **Build Prototype**: Develop the initial working prototype based on the quick design.
4. **Customer Evaluation**: Present the prototype to the customer. The customer evaluates the prototype and provides feedback identifying required changes and missing functionalities.
5. **Refine Prototype**: Modify and refine the prototype based on customer feedback. Repeat steps 3 through 5 until the customer is satisfied.
6. **Develop Final Product**: Once the prototype is approved use it as the basis to develop the actual software system (often following a waterfall approach).

Worked Example:

Applying Prototyping Model to a Custom ERP Interface **Problem:** A client needs a custom Enterprise Resource Planning (ERP) dashboard but cannot clearly articulate their workflow requirements. Apply the Prototyping Model.
Solution:

1. **Initial Requirement Gathering**: The client mentions they need a dashboard showing "daily sales" "inventory alerts" and "employee attendance".
2. **Quick Design and Build Prototype**: Create a mock HTML/CSS frontend with hardcoded data. It visually displays graphs for sales, a list for inventory, and a table for attendance. No real database is connected.
3. **Customer Evaluation**: The client uses the mock dashboard. They realize the sales graph should also show weekly comparisons and the inventory alerts need a "re-order" button right next to them.
4. **Refine Prototype**: Update the frontend code to include the weekly comparison toggle and the re-order buttons. Show it to the client again. The client approves the visual flow.
5. **Develop Final Product**: Use the approved prototype UI. Now write the backend logic, connect to the SQL database, and implement the actual functionality for the final system.

2.1.4 Spiral Model

Methodology:

Spiral Model Iterations The Spiral Model combines elements of both design and prototyping in stages in an effort to combine advantages of top-down and bottom-up concepts. It is highly risk-driven. Each loop in the spiral represents a phase.
Each iteration (loop) consists of four quadrants:

1. **Objective Setting (Quadrant 1)**: Determine objectives, alternatives, and constraints for the current iteration. Gather requirements.
2. **Risk Assessment and Reduction (Quadrant 2)**: Identify and evaluate risks. Develop strategies

to mitigate these risks (often by building a prototype).

3. **Development and Validation (Quadrant 3)**: Develop the next level of the product (e.g. initial prototype then subsequent versions). Test the developed software.

4. **Planning the Next Phase (Quadrant 4)**: Review the progress with the customer. Plan the next iteration based on the feedback and current project status.

Worked Example:

Applying Spiral Model to a Healthcare Data App **Problem:** Develop a high-risk mobile application that stores and transmits sensitive patient medical records. Outline the first two iterations of the Spiral Model. **Solution: Iteration 1: Feasibility and Security Proof of Concept**

1. **Objective Setting**: Goal is to determine if medical records can be securely transmitted from local clinics to a central cloud server.

2. **Risk Assessment**: Major risk is data interception and HIPAA compliance violation.

3. **Development**: Build a small prototype that encrypts dummy data, sends it over a network, and decrypts it. Test the encryption strength.

4. **Planning**: Review security test results. Plan the next iteration to add actual user authentication.

Iteration 2: Core Architecture Development

1. **Objective Setting**: Implement secure doctor and patient login modules integrated with the secure transmission prototype.

2. **Risk Assessment**: Risk of unauthorized access due to weak passwords. Mitigation: Implement Multi-Factor Authentication (MFA).

3. **Development**: Code the login UI, backend authentication logic, and MFA system. Integrate it.

4. **Planning**: Review the login system with stakeholders. Plan Iteration 3 to add patient record viewing features.

2.1.5 Rapid Application Development Model

Methodology:

RAD Model Phases Rapid Application Development (RAD) is an incremental model that emphasizes a short development cycle. It uses component-based construction and heavy reliance on rapid prototyping.

1. **Business Modeling**: Analyze the information flow between various business functions. Identify what information drives the business process.

2. **Data Modeling**: Refine the information gathered into a set of data objects needed to support the business. Define attributes and relationships.

3. **Process Modeling**: Transform the data objects into information flows necessary to implement a business function. Create processes to add, modify, delete, or retrieve data.

4. **Application Generation**: Use automated tools (like low-code platforms or code generators) to construct the software quickly from the process and data models.

5. **Testing and Turnover**: Test the new components and interfaces. Since RAD reuses existing components, overall testing time is reduced. Deliver the software to users quickly.

Worked Example:

Applying RAD to an Internal Survey Tool **Problem:** A company needs an internal employee satisfaction survey tool within 30 days. Plan the execution using RAD.
Solution:

1. **Business Modeling**: Determine that HR needs to send questions and employees need to submit answers. Output is an aggregated report for management.
2. **Data Modeling**: Define data objects: Employee ID, Question List, Responses, and Survey ID.
3. **Process Modeling**: Define operations: `CreateSurvey()`, `SubmitResponse()`, and `GenerateReport()`.
4. **Application Generation**: Instead of coding from scratch, developers use a rapid application framework (like Django or a low-code tool) with pre-built form modules and database bindings to instantly generate the survey UI and reporting dashboards.
5. **Testing and Turnover**: Test the specific custom logic added for `GenerateReport()`. The standard form submission components are already proven. Deploy the tool to employees on day 25.

2.2 Agile Process and Principles

Methodology:

Agile Principles and Process Algorithm Agile development is an iterative and incremental approach that focuses on flexibility, continuous improvement, and rapid delivery of working software. It is guided by the Agile Manifesto.
Core Values of Agile:

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

Key Algorithmic Steps in General Agile Process:

1. **Product Backlog Creation**: Maintain a prioritized list of all features, enhancements, and bug fixes required for the product.
2. **Sprint Planning**: Select a small subset of high-priority items from the backlog to complete in a short fixed timebox (Sprint usually 2-4 weeks).
3. **Sprint Execution**: The team works collaboratively to design, code, and test the selected items. Daily stand-up meetings track progress.
4. **Sprint Review and Retrospective**: At the end of the sprint demonstrate the working software to stakeholders. Reflect on the team process and identify improvements for the next sprint.
5. **Repeat**: Go back to Sprint Planning and repeat until the final product is delivered.

2.2.1 Comparison of Agile Development with Traditional Models

Feature	Traditional Models (e.g. Waterfall)	Agile Development
Approach	Sequential and linear.	Iterative and incremental.
Flexibility	Rigid and highly resistant to changes once development starts.	Highly flexible and welcomes changing requirements late in development.
Customer Involvement	High at the beginning (requirements) and end (delivery).	Continuous and active involvement throughout the project.
Delivery	Single final delivery at the end of the project lifecycle.	Frequent continuous delivery of small working software increments.
Documentation	Comprehensive and heavy documentation required upfront.	Minimal just enough documentation; focus is on working software.
Risk Management	High risk; errors are often discovered late in the testing phase.	Low risk; frequent testing and reviews catch issues early.

2.2.2 Extreme Programming Model

Methodology:

Extreme Programming Practices Extreme Programming (XP) is an Agile framework focused on improving software quality and responsiveness to changing customer requirements through strict engineering practices.
Core XP Practices Algorithm:

1. **Planning Game**: Business and development cooperate to produce the maximum business value as quickly as possible. Create User Stories.
2. **Small Releases**: Put a simple system into production quickly then release new versions on a very short cycle.
3. **Test-Driven Development (TDD)**: Write a failing automated unit test before writing the functional code. Then write just enough code to pass the test.
4. **Pair Programming**: Two programmers work together at one workstation. One types the code (Driver) while the other reviews each line as it is typed (Observer).
5. **Continuous Integration**: Integrate and build the system many times a day every time a task is completed.
6. **Refactoring**: Restructure the system without changing its behavior to remove duplication and improve communication.

Worked Example:

Applying Pair Programming and TDD in XP **Problem:** Using XP, implement a function `calculateDiscount(price)` that returns a 10% discount for prices over $100, and 0% otherwise.
Solution:

1. **Write the Test First (TDD)**: Programmer A (Driver) writes an automated test: `assert`

```
calculateDiscount(150) == 15
assert calculateDiscount(50) == 0
```

2. **Run the Test**: The test fails because `calculateDiscount` does not exist yet.
3. **Write the Code (Pairing)**: Programmer B takes the keyboard (becomes Driver) and Programmer A reviews. Programmer B writes minimal code to pass the test:

```
function calculateDiscount(price) {
    if (price > 100) return price * 0.10;
    return 0;
}
```

4. **Run Test Again**: The tests now pass.
5. **Continuous Integration**: They immediately push this new function to the shared repository triggering an automated system build to ensure it does not break other parts of the application.

2.2.3 Scrum Model

Methodology:

Scrum Framework Roles and Events Scrum is an Agile framework that helps teams work together to develop, deliver, and sustain complex products. It is structured around specific roles, events, and artifacts.
Scrum Roles:

1. **Product Owner**: Represents the stakeholders, defines requirements, and manages the Product Backlog.
2. **Scrum Master**: Facilitates the Scrum process, removes impediments, and ensures the team follows Scrum principles.
3. **Development Team**: Self-organizing cross-functional group of professionals who do the actual work of delivering a potentially releasable Increment of product.

Scrum Events (The Process Algorithm):

1. **Sprint**: A time-boxed iteration of 1 to 4 weeks during which a usable and potentially releasable product increment is created.
2. **Sprint Planning**: The entire team collaborates to define the work to be performed in the Sprint. Items are moved from the Product Backlog to the Sprint Backlog.
3. **Daily Scrum**: A 15-minute time-boxed daily event for the Development Team to synchronize activities and create a plan for the next 24 hours.
4. **Sprint Review**: Held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. Stakeholders are invited to provide feedback.
5. **Sprint Retrospective**: The team discusses what went well, what could be improved, and commits to improvements for the next Sprint.

Worked Example:

Executing a Scrum Sprint for a Mobile Game **Problem:** Track the execution of a 2-week Sprint for a team developing a 2D platformer mobile game using the Scrum framework.
Solution:

1. **Sprint Planning**: The Product Owner prioritizes "Implement Player Jump Mechanics" and "Add Level 1 Background". The Development Team estimates the effort and commits to these tasks, adding them to the Sprint Backlog.
2. **Daily Scrum (Day 3)**:
    - **Developer 1**: "Yesterday I coded the jump physics. Today I will add the jump animation. No blockers."
    - **Artist**: "Yesterday I finished the cloud sprites. Today I am working on the ground tiles. Blocker: I need the exact resolution dimensions from the developer."
    - **Scrum Master**: "I will ensure the developer provides the resolution dimensions immediately after this meeting."
3. **Development Execution**: The team spends the remaining 9 days completing the jump logic and integrating the background art.
4. **Sprint Review (Day 14)**: The team demonstrates a playable character jumping across the new Level 1 background to the stakeholders. Stakeholders approve the increment but suggest the jump feels too floaty. The Product Owner adds "Tweak Jump Gravity" to the Product Backlog.
5. **Sprint Retrospective (Day 14)**: The team realizes communication between art and code was slightly delayed. They agree to maintain a shared document for art asset specifications in the next Sprint.

CHAPTER 3

Software Requirement Analysis

Software Requirement Analysis is a critical phase in the software development life cycle where the needs and expectations of the stakeholders are identified, analyzed, and documented. This chapter focuses on the computational and systematic approach to gathering requirements, analyzing them for feasibility, and organizing them into a formal Software Requirement Specification (SRS) document.

3.1 Requirement Gathering

Requirement gathering involves systematic techniques to collect information about the intended software from stakeholders.

Methodology:

Requirement Gathering Process To systematically gather requirements for a software project, follow these ordered steps:

1. **Identify Stakeholders:** List all individuals or groups who have an interest in the system (e.g. end-users, managers, clients).
2. **Select Elicitation Techniques:** Choose appropriate techniques such as interviews, questionnaires, observation, or brainstorming sessions.
3. **Collect Raw Data:** Conduct the sessions and record all needs, expectations, and constraints expressed by the stakeholders.
4. **Categorize Data:** Group the collected raw data into logical categories such as user features, system performance, and security constraints.

Worked Example:

Gathering Requirements for a Library Management System **Problem Statement:** A university wants to digitize its library. Outline the application of the requirement gathering process to identify the core needs. **Solution:**

1. **Identify Stakeholders:** Students, Librarians, University Administrators, IT Support Staff.
2. **Select Elicitation Techniques:**
   - Interviews with the Chief Librarian.
   - Questionnaires distributed to students.
   - Observation of current manual book-issue processes.
3. **Collect Raw Data:**
   - Students need to search for books online.
   - Librarians want automated late-fee calculation.

- The system must run on existing university servers.

4. **Categorize Data:**
   - User Features: Online search, Book reservation.
   - Administrative Features: Late-fee calculation, Inventory management.
   - Technical Constraints: Compatibility with university servers.

3.2 Analyze the Requirements

Once requirements are gathered, they must be analyzed to resolve conflicts, ensure feasibility, and define clear system boundaries.

Methodology:

Requirement Analysis Framework Use the following steps to analyze raw requirements:

1. **Consistency Check:** Identify and resolve conflicting requirements provided by different stakeholders.
2. **Completeness Check:** Ensure no critical system functions or constraints are missing.
3. **Feasibility Analysis:** Evaluate if the requirement can be implemented within the given time, budget, and technological constraints.
4. **Prioritization:** Rank requirements based on their importance to the core business logic (e.g. Critical, High, Medium, Low).

Worked Example:

Analyzing Conflicting Requirements **Problem Statement:** During the analysis of a mobile application, the following requirements are identified:

- Requirement A (from Marketing): The app must load high-resolution promotional videos on the home screen.
- Requirement B (from IT Support): The app must use minimal mobile data and load within 2 seconds on a 3G network.

Apply the requirement analysis framework to resolve this issue.
Solution:

1. **Consistency Check:** Requirement A and Requirement B are in direct conflict. High-resolution videos consume large amounts of data and take longer to load on slow networks.
2. **Feasibility Analysis:** It is technically infeasible to download large video files over a 3G network within 2 seconds.
3. **Resolution Generation:** Propose a compromise.
4. **Updated Requirement:** The app will display high-quality compressed images by default and provide a "Click to Play Video" option for users on Wi-Fi networks.

3.3 Software Requirement Specification (SRS)

The Software Requirement Specification (SRS) is a formal document that acts as an agreement between the developers and the clients. It details what the software will do and the constraints under which it must operate.

3.3.1 Importance of SRS

- Establishes the basis for agreement between clients and suppliers.
- Reduces development effort by identifying omissions and misunderstandings early.
- Provides a baseline for validation and verification processes.
- Serves as a reference for future maintenance and enhancements.

3.3.2 Users of SRS

The SRS document is utilized by various stakeholders throughout the software lifecycle:

- **Clients and Users:** To verify that their needs are correctly understood.
- **System Designers:** As a blueprint to create the software architecture.
- **Developers:** To understand the exact features to be coded.
- **Testers:** To develop test cases and ensure the final product meets the specified criteria.

3.3.3 Characteristics of a Good SRS

A high-quality SRS document must possess the following characteristics:

- **Correct:** Every requirement stated must be one that the software shall meet.
- **Unambiguous:** Every requirement must have only one interpretation.
- **Complete:** All necessary requirements of the system must be included.
- **Consistent:** No two requirements should conflict with each other.
- **Verifiable:** There must be a cost-effective process to check if the software meets the requirement.

Methodology:

Structuring an SRS Document A standard SRS document should be structured using the following sections:

1. **Introduction:** Purpose, Scope, Definitions, and Overview.
2. **Overall Description:** Product perspective, User characteristics, General constraints.
3. **Specific Requirements:** Detailed functional and non-functional requirements.
4. **Appendices:** Additional models, diagrams, or supporting information.

3.4 Types of Requirements in SRS

Requirements in an SRS are broadly classified into Functional Requirements (FR) and Non-Functional Requirements (NFR).

3.4.1 Functional Requirements

Functional requirements specify specific behaviors or functions. They define what the system *must do*. Examples include data processing rules, specific inputs, and expected outputs.

3.4.2 Non-Functional Requirements

Non-functional requirements specify criteria that can be used to judge the operation of a system, rather than specific behaviors. They define *how well* the system must perform. Examples include performance, security, usability, and reliability.

Methodology:

Classifying System Requirements To classify a requirement, apply the following logical test:

1. Read the requirement statement.
2. Ask: "Does this describe a specific action, feature, or computation the system executes?" If yes, it is a **Functional Requirement**.
3. Ask: "Does this describe a quality attribute, performance metric, or environmental constraint?" If yes, it is a **Non-Functional Requirement**.

Worked Example:

Classifying Requirements for a Banking System **Problem Statement:** Given the following list of requirements for a new online banking portal, classify each as either Functional (FR) or Non-Functional (NFR).

1. The system must allow a user to check their account balance.
2. The system must encrypt all user passwords using SHA-256.
3. The system shall generate a monthly bank statement in PDF format.
4. The system must handle up to 10000 concurrent user sessions without crashing.
5. The application pages must load in under 3 seconds on a standard broadband connection.

Solution:

1. **Action/Feature:** "allow user to check account balance". This defines what the system does.
   Classification: Functional Requirement (FR)
2. **Quality/Constraint:** "encrypt user passwords using SHA-256". This describes a security constraint on how data is handled.
   Classification: Non-Functional Requirement (NFR)
3. **Action/Feature:** "generate monthly bank statement". This is a specific output feature.
   Classification: Functional Requirement (FR)
4. **Quality/Constraint:** "handle up to 10000 concurrent user sessions". This specifies a performance and reliability metric.
   Classification: Non-Functional Requirement (NFR)
5. **Quality/Constraint:** "load in under 3 seconds". This specifies a performance metric.
   Classification: Non-Functional Requirement (NFR)

CHAPTER 4

Software Project Management

4.1 Responsibility of Software Project Manager

While software project management encompasses many theoretical aspects, the primary focus of a project manager in computational terms involves quantifiable estimation, scheduling, and risk assessment. The key responsibilities include:

- **Project Planning and Scheduling**: Developing Work Breakdown Structures (WBS) and creating timelines using Critical Path Method (CPM) and Program Evaluation and Review Technique (PERT).
- **Project Monitoring and Control**: Tracking project progress using quantitative metrics such as Earned Value Management (EVM) and Gantt charts.
- **Risk Management**: Identifying risks, quantifying their impact mathematically, and planning mitigation strategies.

4.2 Scheduling Techniques

Scheduling determines the timeline of the project. It involves dividing the project into a Work Breakdown Structure (WBS) and calculating the overall duration using network diagramming methods.

4.2.1 Critical Path Method (CPM)

The Critical Path Method (CPM) calculates the longest sequence of dependent activities and determines the shortest possible project duration.

Methodology:

Critical Path Method Algorithm **Step 1: Activity Analysis** Identify all activities, their durations, and their immediate predecessors.
Step 2: Forward Pass (Earliest Times) Calculate Earliest Start (ES) and Earliest Finish (EF) for each activity.

- $ES = \max(EF \text{ of all immediate predecessors})$ (For starting activities, $ES = 0$)
- $EF = ES + \text{Duration}$

Step 3: Backward Pass (Latest Times) Calculate Latest Finish (LF) and Latest Start (LS) for each activity.

- $LF = \min(LS \text{ of all immediate successors})$ (For the last activity, $LF =$ Project's maximum EF)
- $LS = LF - \text{Duration}$

Step 4: Slack Calculation and Critical Path Calculate the slack (float) for each activity:

- Slack $= LS - ES$ or Slack $= LF - EF$

The critical path is the sequence of activities where Slack $= 0$.

Worked Example:

Computing the Critical Path **Problem:** Given the following project activities, determine the critical path and the project duration.

Activity	Predecessor	Duration (Days)
A	None	3
B	A	4
C	A	2
D	B	5
E	C	4
F	D and E	2

Solution:
Step 1 & 2: Forward Pass (Calculate ES and EF)

- Activity A (Start): $ES = 0$, $EF = 0 + 3 = 3$
- Activity B (Depends on A): $ES = 3$, $EF = 3 + 4 = 7$
- Activity C (Depends on A): $ES = 3$, $EF = 3 + 2 = 5$
- Activity D (Depends on B): $ES = 7$, $EF = 7 + 5 = 12$
- Activity E (Depends on C): $ES = 5$, $EF = 5 + 4 = 9$
- Activity F (Depends on D and E): $ES = \max(12, 9) = 12$, $EF = 12 + 2 = 14$

The total project duration is 14 days.
Step 3: Backward Pass (Calculate LF and LS)

- Activity F (End): $LF = 14$, $LS = 14 - 2 = 12$
- Activity D (Precedes F): $LF = 12$, $LS = 12 - 5 = 7$
- Activity E (Precedes F): $LF = 12$, $LS = 12 - 4 = 8$
- Activity B (Precedes D): $LF = 7$, $LS = 7 - 4 = 3$
- Activity C (Precedes E): $LF = 8$, $LS = 8 - 2 = 6$
- Activity A (Precedes B and C): $LF = \min(3, 6) = 3$, $LS = 3 - 3 = 0$

Step 4: Slack Calculation

Activity	ES	EF	LS	LF	Slack (LS - ES)	Critical?
A	0	3	0	3	0	Yes
B	3	7	3	7	0	Yes
C	3	5	6	8	3	No
D	7	12	7	12	0	Yes
E	5	9	8	12	3	No
F	12	14	12	14	0	Yes

Result: The critical path is **A** $\rightarrow$ **B** $\rightarrow$ **D** $\rightarrow$ **F** with a total duration of 14 days.

4.2.2 Program Evaluation and Review Technique (PERT)

PERT is used when activity durations are highly uncertain. It uses three time estimates to calculate an expected time and variance for each activity.

Methodology:

PERT Time and Variance Formulas For each activity, let:

- T_o = Optimistic time (best case scenario)
- T_m = Most likely time (normal scenario)
- T_p = Pessimistic time (worst case scenario)

Step 1: Calculate Expected Time (T_e)

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

Step 2: Calculate Activity Variance (V)

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

Step 3: Calculate Project Variance and Standard Deviation Project variance is the sum of the variances of the activities on the **critical path**. The standard deviation (σ) is the square root of the project variance.

Worked Example:

Calculating Expected Time and Variance **Problem:** An activity on the critical path has the following estimated durations (in weeks): Optimistic (T_o) = 2, Most Likely (T_m) = 5, Pessimistic (T_p) = 14. Calculate the expected duration and variance.
Solution:
Step 1: Compute Expected Time (T_e)

$$T_e = \frac{2 + 4(5) + 14}{6} = \frac{2 + 20 + 14}{6} = \frac{36}{6} = 6 \text{ weeks}$$

Step 2: Compute Activity Variance (V)

$$V = \left(\frac{14 - 2}{6}\right)^2 = \left(\frac{12}{6}\right)^2 = 2^2 = 4 \text{ weeks}^2$$

Result: The expected duration is 6 weeks with a variance of 4.

4.2.3 Project Monitoring and Control (EVM)

Earned Value Management (EVM) is a quantitative technique used to monitor project performance and track schedule and cost variances.

Methodology:

Earned Value Management Metrics **Basic Elements:**

- **PV (Planned Value):** The budgeted cost of the work scheduled to be done.
- **EV (Earned Value):** The budgeted cost of the work actually completed.
- **AC (Actual Cost):** The actual cost incurred for the work completed.

Performance Formulas:

- **Cost Variance (CV):** $CV = EV - AC$ (Negative indicates over budget)
- **Schedule Variance (SV):** $SV = EV - PV$ (Negative indicates behind schedule)

- **Cost Performance Index (CPI):** $CPI = \frac{EV}{AC}$ (Less than 1 indicates over budget)
- **Schedule Performance Index (SPI):** $SPI = \frac{EV}{PV}$ (Less than 1 indicates behind schedule)

Worked Example:

Calculating EVM Metrics **Problem:** A project has a total budget of $10,000 to be completed in 10 months. After 5 months, 40% of the work is completed, and the actual cost incurred is $4,500. Calculate CV, SV, CPI, and SPI. Determine the project's health.
Solution:
Step 1: Calculate PV EV and AC

- Total Budget (BAC) = $10,000. Month 5 represents 50% of the planned schedule.
- Planned Value (PV) $= 0.50 \times 10,000 = \$5,000$
- Earned Value (EV) $= 0.40 \times 10,000 = \$4,000$
- Actual Cost (AC) = $4,500

Step 2: Calculate Variances

- $CV = EV - AC = 4000 - 4500 = -\$500$
- $SV = EV - PV = 4000 - 5000 = -\$1000$

Step 3: Calculate Indices

- $CPI = \frac{EV}{AC} = \frac{4000}{4500} \approx 0.89$
- $SPI = \frac{EV}{PV} = \frac{4000}{5000} = 0.80$

Conclusion: Since CV and SV are negative, and both CPI and SPI are less than 1.0, the project is currently over budget and behind schedule.

4.3 Risk Management

Risk management involves identifying, assessing, and mitigating risks. The computational aspect of risk assessment centers on calculating the Risk Exposure. Mitigation involves developing a plan to lower the probability or impact of high-priority risks.

Methodology:

Calculating Risk Exposure Risk Exposure (RE) quantifies the magnitude of a risk based on its likelihood and potential loss.
Formula:

$$RE = P \times C$$

Where:

- P = Probability of the risk occurring (expressed as a decimal between 0 and 1).
- C = Cost or Impact if the risk materializes (usually in dollars or days).

Risks are then prioritized based on their RE value in descending order, guiding the focus of mitigation strategies.

Worked Example:

Risk Assessment and Prioritization **Problem:** A software project has identified three potential risks:

- Risk 1: Key developer leaves the project. Probability: 20%, Estimated impact: $15,000
- Risk 2: Server failure during testing. Probability: 10%, Estimated impact: $25,000
- Risk 3: Scope creep due to unclear requirements. Probability: 50%, Estimated impact: $8,000

Calculate the Risk Exposure for each and determine the highest priority risk for mitigation.
Solution:
Step 1: Calculate RE for each risk

- Risk 1: $RE = 0.20 \times 15,000 = \$3,000$
- Risk 2: $RE = 0.10 \times 25,000 = \$2,500$
- Risk 3: $RE = 0.50 \times 8,000 = \$4,000$

Step 2: Prioritize Risks

Risk	Probability	Impact	Risk Exposure	Priority
Risk 3	0.50	$8,000	$4,000	1
Risk 1	0.20	$15,000	$3,000	2
Risk 2	0.10	$25,000	$2,500	3

Result: Risk 3 (Scope creep) has the highest Risk Exposure ($4,000) and should be the primary focus for mitigation, despite having the lowest raw impact cost.

CHAPTER 5
Software Design

5.1 Software Design Process

The software design process involves transforming user requirements into a suitable form, which helps the programmer in software coding and implementation.

Methodology:

Software Design Methodology The process of software design typically follows these structured steps:

1. **Understand the Requirements:** Analyze the Software Requirement Specification (SRS) document to understand functionality, performance, and constraints.
2. **Identify the Sub-systems:** Decompose the entire system into smaller, manageable sub-systems or modules.
3. **Establish Architecture:** Define how different sub-systems interact and integrate with each other.
4. **Design Components:** Detail the internal algorithms, data structures, and interfaces for each sub-system.
5. **Review and Refine:** Evaluate the design against requirements and refine it to improve modularity.

Worked Example:

Design Process for a Simple Calculator **Problem:** Apply the software design process to design a basic calculator application that performs addition, subtraction, multiplication, and division.
Solution:

1. **Understand Requirements:** The system must accept two numbers and an operator, perform the calculation, and display the result. It must handle division by zero.
2. **Identify Sub-systems:**
   - Input Module: Read inputs from the user.
   - Processing Module: Perform arithmetic operations.
   - Output Module: Display the result or error message.
3. **Establish Architecture:** A central controller receives input, passes it to the processor, and sends the result to the output module.
4. **Design Components:**
   - `add(a b)` returns `a+b`
   - `divide(a b)` checks if `b==0`, if so, returns error, else `a/b`.
5. **Review and Refine:** Ensure error handling is robustly designed across the interface.

5.2 Cohesion and Coupling

Good software design strives for **High Cohesion** (strong intra-module focus) and **Low Coupling** (minimal inter-module dependency).

Methodology:

Analyzing Module Cohesion Cohesion measures the strength of the relationship between elements within a single module. To evaluate cohesion:

1. **Identify Module Tasks:** List all operations performed by the module.
2. **Determine Relationship:** Check how these tasks are related:
   - **Functional (Best):** All elements contribute to a single, well-defined task.
   - **Sequential:** Output of one element is the input to another.
   - **Communicational:** Elements operate on the same input data.
   - **Procedural:** Elements are grouped to ensure a specific order of execution.
   - **Temporal:** Elements are executed at the same time (e.g., initialization).
   - **Logical:** Elements perform similar logical functions (e.g., all input operations).
   - **Coincidental (Worst):** No meaningful relationship between elements.
3. **Classify:** Assign the cohesion level based on the strongest relationship.

Worked Example:

Evaluating Module Cohesion **Problem:** A module named `ProcessCustomer` performs the following tasks: reads customer data from a database, calculates the customer discount based on the data, and prints the customer receipt using the discount. Determine its cohesion.
Solution:

1. **Tasks:** Read Data → Calculate Discount → Print Receipt.
2. **Relationship:** The output of reading data is the input for calculating the discount. The output of the discount calculation is the input for printing the receipt.
3. **Classification:** Since the output of one part serves as the input to the next part, this module exhibits **Sequential Cohesion**.

Methodology:

Analyzing Module Coupling Coupling measures the degree of interdependence between different modules. To evaluate coupling:

1. **Examine Interfaces:** Look at how modules communicate (parameters, shared data).
2. **Classify the Dependency:**
   - **Data (Best):** Modules communicate by passing simple, basic data parameters.
   - **Stamp:** Modules pass composite data structures (like a full record or object) but use only a part of it.
   - **Control:** One module passes a control flag to dictate the execution logic of another.
   - **External:** Modules share a dependency on an external device or protocol.
   - **Common:** Multiple modules share the same global data space.
   - **Content (Worst):** One module directly modifies or relies on the internal workings of another.

3. **Determine Level:** Identify the worst coupling present between the modules.

Worked Example:

Evaluating Module Coupling **Problem:** Module A retrieves a complete `StudentProfile` record (containing ID, Name, Address, Blood Group, Grades) and passes the entire record to Module B. Module B only uses the `ID` to fetch library fines. Determine the coupling between Module A and Module B.
Solution:

1. **Interface:** Module A passes a complex data structure (`StudentProfile`) to Module B.
2. **Dependency Analysis:** Module B receives the entire structure but only operates on a single field (`ID`).
3. **Classification:** This is an example of **Stamp Coupling**, because a composite data structure is passed, but only a fraction of its contents is actually used by the receiving module.

5.3 Approach of Software Design

5.3.1 Function Oriented Design

Methodology:

Developing Data Flow Diagrams A Data Flow Diagram (DFD) illustrates how data flows through a system.

1. **Identify External Entities:** Determine sources and destinations of data outside the system (represented by rectangles).
2. **Draw Context Diagram (Level 0):** Represent the entire system as a single process (circle) interacting with external entities.
3. **Identify Major Processes (Level 1):** Break the main system down into major functional subprocesses.
4. **Identify Data Stores:** Define where data is stored temporarily or permanently (represented by open rectangles).
5. **Connect with Data Flows:** Use directed arrows to show the movement of data between entities, processes, and data stores. Label all data flows.

Worked Example:

Constructing a DFD for a Library System **Problem:** Draw a Level 0 Context Diagram and a Level 1 DFD for a simple Library Book Issue System.
Solution:
Level 0 Context Diagram:

External Entity: Student
System Process: Library System
Flows: Student sends "Book Request" to Library System. System returns "Book Issued" to Student.

Level 1 DFD:

Processes:
P1: Verify Student
P2: Check Book Availability
P3: Issue Book

Data Stores:
D1: Student Database
D2: Book Database

Data Flows:
1. Student $\xrightarrow{\text{Student ID}}$ P1
2. P1 $\xrightarrow{\text{ID Query}}$ D1 $\rightarrow$ P1 (Valid Student)
3. P1 $\xrightarrow{\text{Book Name}}$ P2
4. P2 $\xrightarrow{\text{Search Book}}$ D2 $\rightarrow$ P2 (Status)
5. P2 $\xrightarrow{\text{Book Available}}$ P3
6. P3 $\xrightarrow{\text{Update Record}}$ D2
7. P3 $\xrightarrow{\text{Book Issued}}$ Student

Methodology:

Creating a Data Dictionary A Data Dictionary is a centralized repository of information about data elements.

1. **Identify Data Item:** Pick a data flow or data store from the DFD.
2. **Define Composition:** Break it down using standard notation:
   - = denotes "is composed of"
   - + denotes "AND" (sequence)
   - [|] denotes "OR" (selection)
   - { } denotes "iteration" or repeating elements
   - ( ) denotes "optional" element
3. **Define Base Elements:** Specify data types and limits for atomic elements.

Worked Example:

Defining a Data Dictionary Entry **Problem:** Create a data dictionary entry for a "Student Registration Form" which includes Name, Roll Number, Optional Email, and a list of selected Subjects (minimum 1 subject, maximum 5).
Solution:

Element	Definition
Student_Registration	= Name + Roll_Number + (Email) + {Subject}
Name	= First_Name + (Middle_Name) + Last_Name
Roll_Number	= 12 numeric characters
Email	= String containing '@' and '.'
Subject	= ["Mathematics" | "Physics" | "Chemistry" | "Computer"]

Note: The iteration {Subject} implies 1 to 5 occurrences based on system constraints.

5.3.2 Object Oriented Design

Methodology:

Developing Use Case Diagrams Use case diagrams represent the dynamic behavior of a system from a user's perspective.

1. **Identify Actors:** Define who or what interacts with the system (stick figures).
2. **Identify Use Cases:** Determine the main functionalities the system provides to the actors (ovals).
3. **Define Relationships:**
   - **Association:** Connect actor to use case with a solid line.
   - **≪include≫:** A use case that is strictly required by another use case.
   - **≪extend≫:** A use case that adds optional behavior to a base use case.
4. **Draw System Boundary:** Enclose all use cases in a rectangle to represent the system scope.

Worked Example:

Use Case Diagram for an ATM System **Problem:** Model the use-case scenarios for a basic ATM System where a Customer can Withdraw Cash, Check Balance, and Change PIN. Authentication is required for all operations.
Solution:

System Boundary: ATM System
Actors: Customer, Bank Server (Secondary Actor)
Use Cases:
- UC1: Withdraw Cash
- UC2: Check Balance
- UC3: Change PIN
- UC4: Authenticate User

Relationships:
- Customer $\longleftrightarrow$ UC1, UC2, UC3
- UC1, UC2, UC3 $\xrightarrow{\text{≪include≫}}$ UC4 (Authentication is mandatory)
- UC4 $\longleftrightarrow$ Bank Server (Server verifies credentials)

Methodology:

Developing Activity Diagrams Activity diagrams describe the control flow from one activity to another, useful for modeling workflows.

1. **Start Node:** Represented by a solid filled circle.
2. **Identify Activities:** Represent steps or actions using rounded rectangles.
3. **Control Flows:** Connect activities using directional arrows.
4. **Decision Nodes:** Use diamonds for conditional branching. Label output edges with guard conditions (e.g., `[Valid]`, `[Invalid]`).
5. **Forks and Joins:** Use heavy solid lines to show concurrent activities splitting (fork) and merging (join).
6. **End Node:** Represented by a solid circle surrounded by a hollow circle.

Worked Example:

Activity Diagram for Online Login **Problem:** Create an activity sequence for a user logging into a portal. **Solution:**

Step 1: Start Node → [Enter Credentials]
Step 2: [Enter Credentials] → [Validate Credentials]
Step 3: [Validate Credentials] → Decision Diamond
Step 4 (Branch 1): Decision [Valid] → [Show Dashboard] → End Node
Step 5 (Branch 2): Decision [Invalid] → [Show Error Message] → [Enter Credentials]

5.4 User Interface Design

Methodology:

Evaluating User Interface Characteristics A good User Interface (UI) must adhere to specific characteristics to ensure usability. During design, verify the interface against these principles:

1. **Simplicity:** Are common tasks easy to perform without extensive training?
2. **Consistency:** Do similar actions across different screens use identical terminology and visual cues?
3. **Feedback:** Does the system acknowledge user actions (e.g., loading spinners, success messages)?
4. **Error Handling and Tolerance:** Are error messages clear, and can users easily undo mistakes?
5. **Familiarity:** Does the design utilize standard conventions (like a floppy disk icon for save) that users already understand?

Worked Example:

Identifying Types of User Interfaces **Problem:** Compare the command execution for creating a new folder in a Command Line Interface (CLI) versus a Graphical User Interface (GUI). Highlight the design approach. **Solution:**

1. **Command Line Interface (CLI):**
   - **Action:** User types `mkdir new_folder` and presses Enter.
   - **Design Focus:** Relies on user memory to recall commands and syntax. Highly efficient for expert users but has a steep learning curve. Text-based interaction.
2. **Graphical User Interface (GUI):**
   - **Action:** User right-clicks in a window → selects "New" → selects "Folder" → types "new_folder" → clicks away.
   - **Design Focus:** Relies on visual recognition. Uses WIMP (Windows, Icons, Menus, Pointer) paradigm. Easier for novices due to discoverability, though sometimes slower for repetitive tasks.

CHAPTER 6
Software Coding and Testing

Software coding and testing are critical phases in the software development lifecycle where the design is translated into executable code and subsequently verified for correctness. This chapter focuses on practical methods for applying coding standards, conducting code reviews, and designing effective test cases using black-box and white-box testing techniques.

6.1 Coding Standards and Guidelines

Coding standards ensure that software is readable, maintainable, and less prone to errors. Guidelines typically cover naming conventions, formatting rules, and structural principles.

Methodology:

Applying Coding Standards To ensure code meets standard guidelines, follow these systematic steps:

1. **Meaningful Naming**: Assign clear, descriptive names to variables, functions, and classes. Use camelCase or snake_case consistently.
2. **Proper Indentation**: Use standard spacing (e.g., 4 spaces) for code blocks to maintain structural clarity.
3. **Avoid Magic Numbers**: Replace hard-coded values with named constants to clarify their purpose.
4. **Modularity**: Break down complex functions into smaller, single-purpose modules.
5. **Comments and Documentation**: Add comments to explain *why* a piece of code exists, not just *what* it does.

Worked Example:

Refactoring Code to Meet Standards Consider the following poorly written code snippet. Apply standard coding guidelines to refactor it.

Original Code:

```
def calc(a, b):
  if a > 40:
    return a * b * 1.5
  else:
    return a * b
```

Solution:

1. **Meaningful Naming**: The function name `calc` is vague. It appears to calculate wages based on hours `a` and rate `b`. We rename it to `calculateWage`. Variables `a` and `b` become `hoursWorked` and `hourlyRate`.
2. **Avoid Magic Numbers**: `40` represents standard weekly hours, and `1.5` represents the overtime multiplier. We define these as constants.

3. **Formatting and Comments**: Add a docstring and proper indentation.

Refactored Code:

```
STANDARD_HOURS = 40
OVERTIME_MULTIPLIER = 1.5

def calculateWage(hoursWorked, hourlyRate):
    """Calculates total wage including overtime pay if applicable."""
    if hoursWorked > STANDARD_HOURS:
        overtimeHours = hoursWorked - STANDARD_HOURS
        regularPay = STANDARD_HOURS * hourlyRate
        overtimePay = overtimeHours * hourlyRate * OVERTIME_MULTIPLIER
        return regularPay + overtimePay
    else:
        return hoursWorked * hourlyRate
```

6.2 Code Review Techniques

Code review involves systematic examination of source code to find defects, improve quality, and share knowledge. The two primary methods are Walkthroughs and Inspections.

Methodology:

Conducting a Code Walkthrough A code walkthrough is an informal review process where the author leads the team through the code.

1. **Preparation**: The author prepares the code and necessary documentation.
2. **Presentation**: The author explains the logic and structure step-by-step.
3. **Evaluation**: Peers analyze the code for logical errors, readability, and adherence to standards.
4. **Action Items**: Reviewers suggest improvements which are noted for revision.

Worked Example:

Identifying Defects in a Walkthrough During a code walkthrough, the following snippet for reversing an array is presented. Identify the logical errors.

Code Snippet:

```
def reverseArray(arr):
    n = len(arr)
    for i in range(n):
        temp = arr[i]
        arr[i] = arr[n - i - 1]
        arr[n - i - 1] = temp
    return arr
```

Solution:

1. **Execution Trace**: Let `arr = [1, 2, 3, 4]`. Length `n = 4`.
2. **Iteration 1** ($i = 0$): Swaps `arr[0]` and `arr[3]`. Array becomes `[4, 2, 3, 1]`.
3. **Iteration 2** ($i = 1$): Swaps `arr[1]` and `arr[2]`. Array becomes `[4, 3, 2, 1]`.
4. **Iteration 3** ($i = 2$): Swaps `arr[2]` and `arr[1]`. Array becomes `[4, 2, 3, 1]`.

5. **Iteration 4** ($i = 3$): Swaps `arr[3]` and `arr[0]`. Array becomes `[1, 2, 3, 4]`.
6. **Defect Identified**: The loop runs for `n` iterations, meaning the array is reversed and then reversed back to its original state.
7. **Correction**: The loop should only iterate up to `n // 2`.

6.3 Software Testing Fundamentals

Testing is the process of evaluating a system to verify that it satisfies specified requirements and to identify errors.

Methodology:

Designing a Test Suite A test suite is a collection of test cases designed to validate a specific component.

1. **Understand the Requirement**: Clearly define what the function or module should achieve.
2. **Identify Test Scenarios**: Brainstorm positive, negative, and edge-case scenarios.
3. **Define Test Cases**: For each scenario, specify the inputs and the expected output.
4. **Format**: Document the test cases in a structured format (Test ID, Description, Input, Expected Output).

Worked Example:

Creating a Unit Test Suite Design a unit test suite for a function `divide(a, b)` that returns the quotient of a divided by b. If b is zero, the function should raise an error.
Solution:

Test ID	Description	Inputs	Expected Output
TC01	Positive numbers	$a = 10, b = 2$	5.0
TC02	Negative numbers	$a = -10, b = -2$	5.0
TC03	Mixed signs	$a = -10, b = 2$	-5.0
TC04	Zero numerator	$a = 0, b = 5$	0.0
TC05	Divide by zero (edge case)	$a = 10, b = 0$	Raise Error
TC06	Fractional result	$a = 5, b = 2$	2.5

6.4 Black-Box Testing

Black-box testing focuses on the functional requirements of the software without knowing its internal code structure.

Methodology:

Equivalence Partitioning and Boundary Value Analysis These techniques minimize test cases while maximizing coverage.

1. **Equivalence Partitioning (EP)**: Divide the input domain into classes of data where all values in one class are treated equivalently by the software. Select one representative value from each class.
2. **Boundary Value Analysis (BVA)**: Identify the boundaries between equivalence classes. Test the exact boundary values, and values just inside and outside the boundaries.

Worked Example:

Applying EP and BVA A software system accepts an age input for a user registration form. The valid age range is 18 to 60 inclusive. Apply EP and BVA to design test cases.

Solution:

Step 1: Equivalence Partitioning We identify three partitions based on the valid range:

- Invalid Partition 1 (Too young): Age < 18
- Valid Partition (Acceptable age): $18 \leq$ Age ≤ 60
- Invalid Partition 2 (Too old): Age > 60

Selecting one representative value per partition yields:

Partition	Test Input	Expected Output
Invalid (Age < 18)	15	Reject
Valid ($18 \leq$ Age ≤ 60)	35	Accept
Invalid (Age > 60)	70	Reject

Step 2: Boundary Value Analysis The boundaries are 18 and 60. We test the boundary itself and the values immediately adjacent.

- Lower boundary tests: 17 (invalid), 18 (valid), 19 (valid)
- Upper boundary tests: 59 (valid), 60 (valid), 61 (invalid)

Summary of BVA test cases:

Boundary	Test Input	Expected Output
Just below lower	17	Reject
Exact lower	18	Accept
Just above lower	19	Accept
Just below upper	59	Accept
Exact upper	60	Accept
Just above upper	61	Reject

6.5 White-Box Testing

White-box testing is a technique that tests internal structures or workings of an application. The tester requires knowledge of the internal code.

Methodology:

Basis Path Testing and Cyclomatic Complexity Basis path testing guarantees that every independent path through the code is executed at least once.

1. **Draw Control Flow Graph (CFG)**: Map the code statements to nodes and the flow of control to edges.
2. **Calculate Cyclomatic Complexity ($V(G)$)**:

$$V(G) = E - N + 2$$

   where E is the number of edges and N is the number of nodes. Alternatively $V(G) = P + 1$ where P is the number of predicate (decision) nodes.
3. **Identify Independent Paths**: List a set of paths equal to $V(G)$. An independent path must introduce at least one new set of processing statements or a new condition.
4. **Create Test Cases**: Design test cases to execute each path.

Worked Example:

Calculating Cyclomatic Complexity Consider the following pseudo-code. Calculate its cyclomatic complexity and identify the independent paths.

Pseudo-code:

```
1. start
2. input X, Y
3. if X > Y then
4.    Z = X
5. else
6.    Z = Y
7. end if
8. print Z
9. end
```

Solution:

Step 1: Build the Control Flow Graph

- Node 1: lines 1, 2, 3 (Input and condition evaluation)
- Node 2: line 4 (if branch)
- Node 3: line 6 (else branch)
- Node 4: lines 8, 9 (print and end)

Edges connecting the nodes:

- Edge 1 to 2 (True condition)
- Edge 1 to 3 (False condition)
- Edge 2 to 4 (After if block)
- Edge 3 to 4 (After else block)

Thus $N = 4$ nodes and $E = 4$ edges.

Step 2: Calculate Cyclomatic Complexity Using the formula $V(G) = E - N + 2$:

$$V(G) = 4 - 4 + 2 = 2$$

Alternatively using predicate nodes (P): Node 1 is the only decision node so $P = 1$.

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

Step 3: Identify Independent Paths Since $V(G) = 2$, we need to define 2 independent paths:

- **Path 1**: $1 \to 2 \to 4$ (Occurs when $X > Y$)
- **Path 2**: $1 \to 3 \to 4$ (Occurs when $X \leq Y$)

Step 4: Create Test Cases

Path	Input	Expected Output	Nodes Traversed
1	$X = 10, Y = 5$	$Z = 10$	1, 2, 4
2	$X = 3, Y = 8$	$Z = 8$	1, 3, 4

Appendix: Key Formulae

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

Unit 1: Software Quality and Metrics

System Availability

The allowable downtime for a system based on its required availability percentage.

$$\text{Downtime Allowed} = \text{Total Time} \times (1 - \text{Availability})$$

Defect Density

A measure of the number of defects found in a software component relative to its size, typically expressed in KLOC (Kilo Lines of Code).

$$\text{Defect Density} = \frac{\text{Total Defects}}{\text{Software Size (e.g., KLOC)}}$$

Defect Removal Efficiency (DRE)

A metric used to determine how effectively defects were removed during the development process before delivery.

$$\text{DRE} = \frac{E}{E + D} \times 100$$

Where:

- E = Number of defects found before delivery.
- D = Number of defects found after delivery.

Unit 4: Project Estimation and Risk Management

PERT Estimation

Used in Project Evaluation and Review Technique (PERT) to calculate expected time and variance.

Expected Time (T_e):

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

Variance (V):

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

Where:

- T_o = Optimistic time estimate
- T_m = Most likely time estimate
- T_p = Pessimistic time estimate

Risk Exposure (RE)

Measures the overall threat of a risk occurring.

$$\text{RE} = P \times C$$

Where:

- P = Probability of the risk occurring.
- C = Cost or impact of the risk.

Unit 6: Software Testing and Complexity

Cyclomatic Complexity ($V(G)$)

A software metric used to indicate the complexity of a program. It measures the number of linearly independent paths through a program's source code.

Based on Flow Graph:

$$V(G) = E - N + 2$$

Where:

- E = Number of edges in the flow graph.
- N = Number of nodes in the flow graph.

Based on Predicate Nodes:

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

Where:

- P = Number of predicate (decision) nodes in the flow graph.