Se (4353202) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Unit 1: Introduction to Software Engineering

Although the foundations of Software Engineering involve numerous conceptual frameworks, applying a computational and systematic approach to these concepts is essential for modern software development. This chapter focuses on quantitative methods for estimating effort differences between simple programs and software products, allocating project effort across the generic process framework, prioritizing risks within umbrella activities, and calculating the relative cost of defect correction as part of software engineering's quality focus.

1.1 Cost Multipliers: Programs vs Software Products

A simple program written for personal use requires significantly less effort than a robust, generalized software product. Fred Brooks established a heuristic to estimate the effort required to scale a program into a commercial product or system.

Methodology:

Brooks Rule for Cost Multipliers To estimate the effort or cost required to transform a basic program into a commercial software product or system, apply the following scaling factors:

1. **Software Product (E_{prod}):** A program intended to be run, tested, repaired, and extended by anybody. It requires generalized testing and documentation.

$$E_{prod} = 3 \times E_{prog}$$

2. **Programming System (E_{sys}):** A collection of interacting programs. It requires precise interfaces and system integration.

$$E_{sys} = 3 \times E_{prog}$$

3. **Programming Systems Product (E_{sys_prod}):** A generalized collection of interacting programs intended for widespread use.

$$E_{sys\_prod} = 9 \times E_{prog}$$

Where E_{prog} is the effort (or cost) required to develop a basic, standalone program.

Worked Example:

Estimating Product Development Effort A startup has developed a standalone scheduling program prototype which required 4 person-months of effort. The founders want to release this as a generalized Software Product for individual users, and later as a complete Programming Systems Product for enterprise integration. Calculate the estimated effort for both targets.

Solution:

1. **Identify the base effort:** The effort for the basic program is $E_{prog} = 4$ person-months.

2. **Calculate the effort for the Software Product:** Using the multiplier for a generalized product:

$$E_{prod} = 3 \times E_{prog} = 3 \times 4 = 12 \text{ person-months}$$

3. **Calculate the effort for the Programming Systems Product:** Using the multiplier for an enterprise system product:

$$E_{sys\_prod} = 9 \times E_{prog} = 9 \times 4 = 36 \text{ person-months}$$

The team will need approximately 12 person-months to release the basic product and 36 person-months to build the enterprise-level systems product.

1.2 Generic Process Framework and Effort Allocation

Software engineering incorporates a generic process framework comprising five core activities: Communication, Planning, Modeling, Construction, and Deployment. To manage these effectively, practitioners use heuristic rules to allocate effort across these phases.

Methodology:

The 40-20-40 Effort Distribution Rule A widely accepted heuristic for allocating total project effort (E_{total}) across the generic process framework dictates:

1. **Front-end Activities (40%):** Includes Communication, Planning, and Modeling (Requirements and Design).

$$E_{front} = 0.40 \times E_{total}$$

2. **Construction Activity (20%):** Involves the actual Coding of the software.

$$E_{coding} = 0.20 \times E_{total}$$

3. **Back-end Activities (40%):** Includes Testing and Deployment.

$$E_{back} = 0.40 \times E_{total}$$

This highlights that software engineering is heavily weighted toward analysis, design, and testing rather than just coding.

Worked Example:

Allocating Framework Activity Effort A software project is estimated to require a total effort of 85 person-days. Apply the 40-20-40 rule to determine how many person-days should be budgeted for Front-end activities, Construction, and Back-end activities.
Solution:

1. **Identify the total estimated effort:**

$$E_{total} = 85 \text{ person-days}$$

2. **Calculate effort for Front-end Activities:**

$$E_{front} = 0.40 \times 85 = 34 \text{ person-days}$$

3. **Calculate effort for Construction (Coding):**

$$E_{coding} = 0.20 \times 85 = 17 \text{ person-days}$$

4. **Calculate effort for Back-end Activities:**

$$E_{back} = 0.40 \times 85 = 34 \text{ person-days}$$

The project manager should allocate 34 days for planning and modeling, 17 days for coding, and 34 days for testing and deployment.

1.3 Umbrella Activities: Risk Exposure Calculation

Umbrella activities, such as software project tracking and risk management, are applied continuously throughout the software process rather than being restricted to a single phase. A primary quantitative technique in risk management is calculating the Risk Exposure for identified potential problems.

Methodology:

Calculating Risk Exposure Risk Exposure (RE) quantifies the overall threat of a specific risk by combining the probability of the risk occurring with the financial or time impact if it does occur.

1. **Determine the Probability (P):** Estimate the likelihood of the risk occurring, expressed as a decimal value between 0.0 (impossible) and 1.0 (certain).
2. **Determine the Impact (C):** Estimate the cost, effort, or time lost if the risk materializes.
3. **Calculate the Risk Exposure (RE):**

$$RE = P \times C$$

Risks with the highest RE values must be prioritized for mitigation during umbrella activity reviews.

Worked Example:

Prioritizing Project Risks A software project team identifies two potential risks during their continuous umbrella tracking activities. Risk A: A key developer leaves the team. Probability is estimated at 30%, and the impact would cost \$15,000 to hire and train a replacement. Risk B: The main database server crashes. Probability is estimated at 5%, and the impact would cost \$40,000 in lost data and downtime. Determine which risk has a higher Risk Exposure.
Solution:

1. **Calculate Risk Exposure for Risk A:** Probability $P_A = 0.30$ Impact $C_A = \$15,000$

$$RE_A = 0.30 \times 15,000 = \$4,500$$

2. **Calculate Risk Exposure for Risk B:** Probability $P_B = 0.05$ Impact $C_B = \$40,000$

$$RE_B = 0.05 \times 40,000 = \$2,000$$

3. **Compare the values:** Although Risk B has a significantly higher absolute impact cost, Risk A presents a higher Risk Exposure (\$4,500 vs \$2,000) due to its higher probability. The team should prioritize mitigation strategies for Risk A.

1.4 Layered Technology and Quality Costs

Software engineering is fundamentally a layered technology relying on a bedrock of quality focus. Process models, methods, and tools all rest upon this foundation. One of the most important computational models regarding software quality is the escalating cost of fixing defects over the project lifecycle.

Methodology:

Relative Cost of Defect Correction The cost to fix a software defect increases exponentially the later it is found in the software development life cycle (SDLC). A common logarithmic heuristic is the Factor of 10 Rule.

1. Assign a phase index k to the stages of development:

SDLC Phase	Index (k)
Requirements and Planning	1
Design	2
Coding	3
Testing	4
Release and Maintenance	5

2. Calculate the cost of fixing a defect at phase k (C_k) given the baseline cost to fix it during Phase 1 (C_1):

$$C_k = C_1 \times 10^{(k-1)}$$

Worked Example:

Calculating Exponential Defect Costs During a project audit, it is determined that fixing a logical error during the Requirements phase ($k = 1$) would have cost \$25 in analyst time. However, the defect went unnoticed and was only discovered after the software was released to customers. Calculate the cost to fix the defect during the Release phase ($k = 5$) using the Factor of 10 Rule.

Solution:

1. **Identify the baseline cost and target phase:** The baseline cost $C_1 = \$25$. The target phase is Release and Maintenance, so $k = 5$.

2. **Apply the cost escalation formula:**

$$C_5 = C_1 \times 10^{(5-1)}$$

$$C_5 = 25 \times 10^4$$

$$C_5 = 25 \times 10,000 = \$250,000$$

By failing to detect the defect during the initial phase, the cost of correction amplified from \$25 to \$250,000, underscoring the critical necessity of early quality assurance activities in the software engineering layers.

CHAPTER 2

Software Development Life Cycle

2.1 Introduction to SDLC

The Software Development Life Cycle (SDLC) is a structured process used by software engineering teams to design, develop, and test high-quality software. A typical SDLC includes phases such as Requirement Analysis, System Design, Coding, Testing, Deployment, and Maintenance.

While the SDLC is often viewed as a theoretical framework, selecting the appropriate model and managing its parameters (such as project duration, risk prioritization, and iteration velocity) involve direct computational problem-solving and quantitative decision making.

2.2 Software Development Models

Different SDLC models handle project phases differently. Selecting the optimal model based on requirement stability and project constraints is a critical first step.

Methodology:

Selection Criteria for SDLC Models To systematically select an appropriate SDLC model for a given problem statement, follow these steps:

1. **Analyze Requirements:** Determine if requirements are strictly defined or expected to change frequently.
2. **Assess Timeline and Deliverables:** Identify if core functionality is needed early or if the entire system can be delivered at once.
3. **Evaluate Risk:** Determine the level of technical or business risk involved.
4. **Select the Optimal Model:**
   - **Waterfall:** Use when requirements are completely clear and stable.
   - **Incremental:** Use when core features are needed early and modules can be developed sequentially.
   - **Prototype:** Use when UI/UX is critical but requirements are ambiguous.
   - **Spiral:** Use for complex projects with high risk and uncertainty.
   - **RAD:** Use when development time is strictly limited (e.g. 2 to 3 months) and components are reusable.
   - **Agile:** Use for dynamic projects requiring continuous customer collaboration and rapid releases.

Worked Example:

Selecting an SDLC Model for a Banking System **Problem Statement:** A national bank wants to develop a core banking system. The requirements are strictly defined by regulatory authorities and are highly unlikely to change during development. High security and extensive documentation are strictly required. Which

model should be used? Justify your choice.
Solution:

1. **Analyze Requirements:** The requirements are strictly defined and stable due to banking regulations.
2. **Evaluate Risk and Documentation:** Security is critical and exhaustive documentation is mandated.
3. **Select Model:** The **Waterfall Model** is the most appropriate.
4. **Justification:** The Waterfall model excels when requirements are well-understood and do not change. It inherently demands extensive documentation at each phase. The linear sequential approach ensures that critical phases like security design are thoroughly completed and verified before coding begins.

2.2.1 Waterfall and Incremental Models

The Waterfall model executes phases strictly one after another. The Incremental process model divides the whole project into multiple standalone modules (increments) which are developed and delivered sequentially or in parallel.

Methodology:

Effort and Duration Calculation for Incremental Model When dividing a project into N increments, scheduling depends on whether development is sequential or parallel.

- **Sequential Development:** A single team develops one increment after another.

$$\text{Total Time} = \sum_{i=1}^{N} \text{Time of Increment}_i$$

- **Parallel Development:** Multiple autonomous teams work on different increments simultaneously.

$$\text{Total Time} = \max(\text{Time of Increment}_1, \text{Time of Increment}_2, \ldots, \text{Time of Increment}_N)$$

Worked Example:

Incremental Model Scheduling **Problem Statement:** A university management software is divided into 3 increments: Admissions (requires 4 weeks), Academics (requires 6 weeks), and Examinations (requires 5 weeks). Calculate the total completion time if the increments are developed sequentially by one team versus parallelly by three distinct teams.
Solution: Step 1: Sequential Development

$$\text{Total Time} = \text{Time(Admissions)} + \text{Time(Academics)} + \text{Time(Examinations)}$$

$$\text{Total Time} = 4 + 6 + 5 = 15 \text{ weeks}$$

Step 2: Parallel Development

$$\text{Total Time} = \max(4, 6, 5)$$

$$\text{Total Time} = 6 \text{ weeks}$$

Conclusion: Parallel development reduces the schedule from 15 weeks to 6 weeks, assuming no inter-dependencies block the teams.

2.2.2 Prototype and Spiral Models

The Prototype model focuses on building a working approximation of the system early. The Spiral model integrates prototyping with risk management and is structured in iterative loops (spirals).

Methodology:

Calculating Risk Exposure in Spiral Model In the Spiral model's risk analysis phase, prioritizing risks is critical. This is done using Risk Exposure (RE). **Formula:**

$$RE = P \times C$$

Where:

- P = Probability of risk occurrence (expressed as a decimal between 0 and 1)
- C = Cost or Impact of the risk (in monetary value or delay duration)

Steps:

1. Identify risks and estimate their probability (P) and impact (C).
2. Calculate RE for each risk.
3. Prioritize risks with the highest RE for immediate mitigation.

Worked Example:

Risk Analysis Calculation **Problem Statement:** During a Spiral model iteration for a cloud migration project, two major risks are identified:

- Risk A (Data Loss): Probability is 5% (0.05) and the cost of recovery is $50,000.
- Risk B (Schedule Delay): Probability is 30% (0.30) and the penalty cost is $10,000.

Calculate the Risk Exposure for both and determine which risk should be prioritized.
Solution: Step 1: Calculate RE for Risk A (Data Loss)

$$RE_A = 0.05 \times 50,000 = 2,500$$

Step 2: Calculate RE for Risk B (Schedule Delay)

$$RE_B = 0.30 \times 10,000 = 3,000$$

Step 3: Compare and Prioritize Since RE_B ($3,000) > RE_A ($2,500), **Risk B (Schedule Delay)** has a higher overall risk exposure and must be prioritized for mitigation during this spiral.

2.2.3 Rapid Application Development Model

RAD is an incremental model that emphasizes a very short development cycle using component-based construction.

Methodology:

Timeboxing in Rapid Application Development RAD heavily utilizes strict timeboxes (e.g. 60 to 90 days). To meet these constraints, teams must be sized appropriately. **Formula:**

$$\text{Required Team Size} = \frac{\text{Total Estimated Effort (Person-Months)}}{\text{Timebox Duration (Months)}}$$

Worked Example:

Team Size Calculation in RAD **Problem Statement:** An enterprise portal project requires an estimated total effort of 36 person-months. The client mandates the use of the RAD model with a strict delivery timebox of 4 months. How many developers need to be assigned?

Solution: Step 1: Identify Given Values

- Total Estimated Effort = 36 person-months
- Timebox Duration = 4 months

Step 2: Apply Formula

$$\text{Required Team Size} = \frac{36}{4} = 9 \text{ developers}$$

Conclusion: A team of 9 developers must be deployed to complete the project within the required 4-month timebox.

2.3 Agile Development Model

Agile development relies on the principles of continuous iteration, constant user feedback, and adapting to changing requirements even late in the process. Popular Agile frameworks include Scrum and Extreme Programming (XP).

2.3.1 Agile Scrum Sprint Metrics

In Scrum, development is divided into fixed-length iterations called Sprints (typically 1 to 4 weeks). Team progress is measured using quantitative metrics like Velocity and Burndown Rates.

Methodology:

Agile Scrum Sprint Planning and Velocity Velocity measures the volume of work (in Story Points) a team successfully completes in a single Sprint. **Formulas:**

$$\text{Average Velocity} = \frac{\sum \text{Completed Story Points across } K \text{ Sprints}}{K}$$

$$\text{Estimated Sprints Remaining} = \frac{\text{Remaining Product Backlog Points}}{\text{Average Velocity}}$$

Worked Example:

Calculating Scrum Velocity and Project Duration **Problem Statement:** An Agile team is developing a mobile application. The total estimated size of the product backlog is 140 Story Points. Over the first three sprints, the team completed 21, 25, and 20 Story Points respectively. Calculate the team's average velocity and estimate how many more sprints are required to finish the remaining backlog.

Solution: Step 1: Calculate Average Velocity

$$\text{Average Velocity} = \frac{21 + 25 + 20}{3} = \frac{66}{3} = 22 \text{ points/sprint}$$

Step 2: Calculate Remaining Backlog Total Backlog = 140 points
Points completed = 66 points

$$\text{Remaining Backlog} = 140 - 66 = 74 \text{ points}$$

Step 3: Estimate Remaining Sprints

$$\text{Remaining Sprints} = \frac{74}{22} \approx 3.36 \text{ Sprints}$$

Since sprints are discrete timeboxes, the team will require **4 more sprints** to complete the remaining backlog.

Methodology:

Burndown Rate Calculation A Sprint Burndown chart visually tracks the remaining work over the sprint duration. Tracking numerical rates helps predict if the sprint goal will be met. **Formulas:**

$$\text{Ideal Burndown Rate} = \frac{\text{Total Sprint Work}}{\text{Total Days in Sprint}}$$

$$\text{Actual Burndown Rate} = \frac{\text{Work Completed So Far}}{\text{Days Elapsed}}$$

If Actual Rate $\geq$ Ideal Rate, the team is ahead of or on schedule.

Worked Example:

Sprint Burndown Analysis **Problem Statement:** A Scrum team commits to 120 hours of task work for a 10-day sprint. After 5 days, they have completed 50 hours of work. Calculate the Ideal and Actual Burndown Rates. Is the team on track to complete the sprint?
Solution: Step 1: Calculate Ideal Burndown Rate

$$\text{Ideal Rate} = \frac{120}{10} = 12 \text{ hours/day}$$

Step 2: Calculate Actual Burndown Rate

$$\text{Actual Rate} = \frac{50}{5} = 10 \text{ hours/day}$$

Step 3: Compare Rates The actual burndown rate (10 hours/day) is less than the ideal rate (12 hours/day). The team is **behind schedule** and needs to increase their pace to finish the remaining 70 hours in the last 5 days (which would require 14 hours/day).

2.3.2 Extreme Programming Estimation

Extreme Programming (XP) mandates technical practices like Test-Driven Development and Pair Programming. Pair programming requires two developers at a single workstation.

Methodology:

Pair Programming Effort Estimation in XP While pair programming uses two developers simultaneously, it often reduces the number of initial defects, vastly cutting down debugging time. **Formulas for Total Person-Hours:**

$$\text{Effort}_{\text{solo}} = \text{Coding Time}_{\text{solo}} + \text{Debugging Time}_{\text{solo}}$$

$$\text{Effort}_{\text{pair}} = 2 \times (\text{Coding Time}_{\text{pair}} + \text{Debugging Time}_{\text{pair}})$$

Worked Example:

Evaluating Pair Programming Efficiency **Problem Statement:** A complex database module requires development. Solo programming is estimated to take 30 hours of coding and 20 hours of debugging. Pair programming the same module is estimated to take 16 hours of coding and 4 hours of debugging. Calculate the total person-hours for both approaches and determine which is more cost-effective.
Solution: Step 1: Calculate Solo Effort 1 developer is involved.

$$\text{Effort}_{\text{solo}} = 30 + 20 = 50 \text{ person-hours}$$

Step 2: Calculate Pair Effort 2 developers are involved simultaneously.

$$\text{Effort}_{\text{pair}} = 2 \times (16 + 4) = 2 \times 20 = 40 \text{ person-hours}$$

Conclusion: Pair programming requires **40 person-hours** compared to 50 person-hours for solo programming. For this complex module, pair programming is highly cost-effective and saves 10 person-hours overall.

CHAPTER 3

Software Requirement Analysis and Design

3.1 Software Requirement Specification (SRS)

Methodology:

Identifying Requirements for SRS Software Requirement Specification (SRS) involves gathering and analyzing requirements to formalize what the software must do. The process focuses on categorizing requirements into two main types:

1. **Identify Functional Requirements (FR):** These describe the specific behavior or functions of the system. Ask: "What should the system do?"
2. **Identify Non-Functional Requirements (NFR):** These specify the quality attributes and constraints of the system. Ask: "How well should the system perform its functions?" (e.g. performance, security, usability, reliability).
3. **Document the Requirements:** List them clearly using unambiguous language.

Worked Example:

Identify Functional and Non-Functional Requirements for an Online Book Store **Problem:** A client wants an online bookstore where users can search for books, add them to a cart, and make payments. The system must load pages within 2 seconds, be highly secure for payments, and handle up to 10000 concurrent users. Identify the functional and non-functional requirements.
Solution:
Step 1: Extract Functional Requirements (What the system does)

- Users shall be able to search for books by title, author, or ISBN.
- Users shall be able to add books to a shopping cart.
- Users shall be able to proceed to checkout and make a payment.
- The system shall generate a digital receipt after a successful transaction.

Step 2: Extract Non-Functional Requirements (System constraints and qualities)

- **Performance:** The system must load web pages within 2 seconds.
- **Security:** All payment transactions must be securely encrypted (e.g. using SSL/TLS).
- **Scalability:** The system must be capable of handling up to 10000 concurrent users without performance degradation.
- **Availability:** The system should be available 24/7 with minimal downtime.

3.2 Software Design Fundamentals

Methodology:

Evaluating Cohesion and Coupling Software design aims for **High Cohesion** and **Low Coupling**.

1. **Evaluate Cohesion (Module Strength):** Determine how closely the tasks within a single module are related.
    - *Best to Worst:* Functional → Sequential → Communicational → Procedural → Temporal → Logical → Coincidental.
2. **Evaluate Coupling (Module Interdependence):** Determine the degree of interdependence between different modules.
    - *Best to Worst:* Data → Stamp → Control → External → Common → Content.

Worked Example:

Identify the Type of Cohesion and Coupling **Problem:** 1. Module A passes an entire student record to Module B, but Module B only uses the student's age to calculate eligibility. What type of coupling is this? 2. A module contains functions to initialize the database, print a startup message, and open a log file, all executed at system startup. What type of cohesion is this?
Solution:
Part 1: Identifying Coupling

- **Analysis:** Module A passes a composite data structure (student record) to Module B, but Module B only uses a part of it (age).
- **Conclusion:** This is **Stamp Coupling**. To improve design to **Data Coupling**, Module A should only pass the specific data required (student age).

Part 2: Identifying Cohesion

- **Analysis:** The functions in the module are grouped together simply because they must all be executed at the same time (during system startup). They do not share data or perform a single unified task.
- **Conclusion:** This is **Temporal Cohesion**.

3.3 Data Flow Diagrams (DFD)

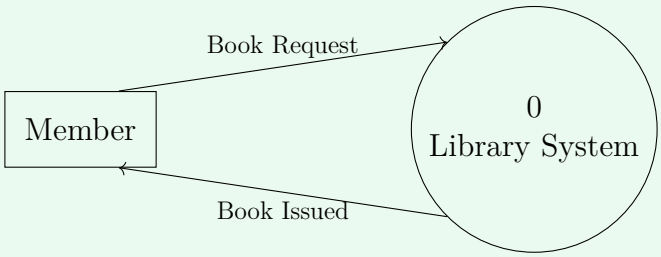
Methodology:

Constructing Data Flow Diagrams A DFD represents the flow of data through a system.

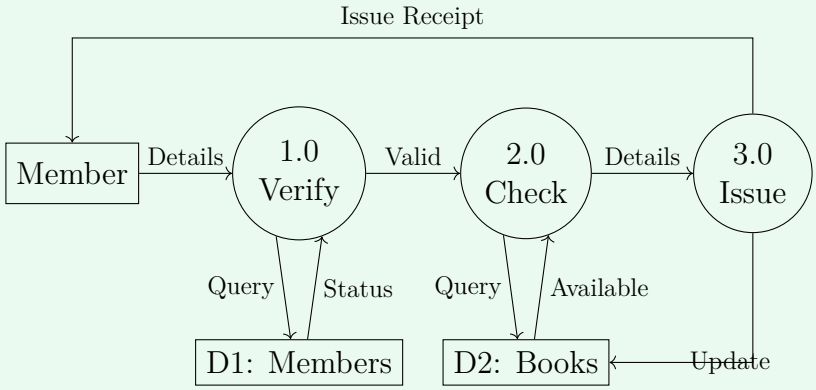
1. **Identify External Entities:** Determine the sources and destinations of data outside the system (represented by rectangles).
2. **Identify Data Flows:** Determine the data moving between entities, processes, and data stores (represented by arrows).
3. **Construct Context Diagram (Level 0):** Represent the entire system as a single process (circle) interacting with external entities. No data stores are shown at this level.
4. **Construct Level 1 DFD:** Decompose the single system process into major sub-processes. Add data stores where data is kept at rest. Ensure data flow consistency (balancing) with the Context Diagram.

Worked Example:

Draw Context and Level 1 DFD for a Library Management System **Problem:** Design the Level 0 (Context) and Level 1 Data Flow Diagrams for a Library Management System where a Member requests books, the system verifies the member and book availability from the database, and issues the book.
Solution:
Step 1: Context Diagram (Level 0)



Step 2: Level 1 DFD



3.4 Object Modeling with UML

Methodology:

Designing Use Case and Class Diagrams **Use Case Diagram:** Captures system functionality from the user's perspective.

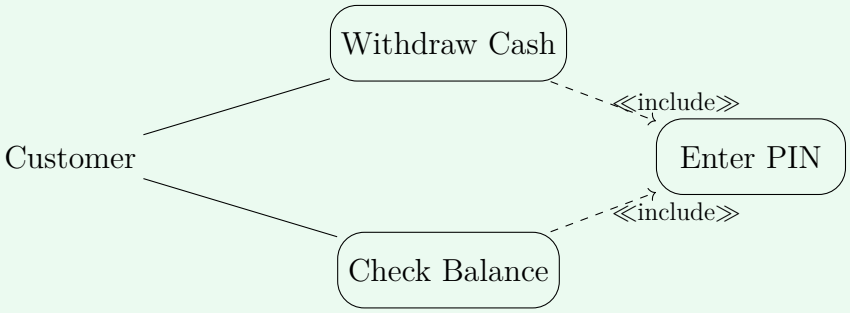
1. Identify Actors (users or external systems).
2. Identify Use Cases (actions performed by the system).
3. Draw relationships (association, ≪include≫, ≪extend≫).

Class Diagram: Captures the static structure of the system.

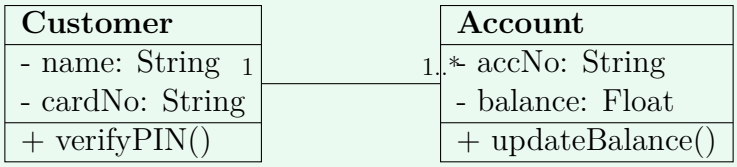
1. Identify Classes (entities like Customer, Account).
2. Add Attributes (data) and Methods (operations) to each class.
3. Define relationships (association, inheritance, aggregation).

Worked Example:

Use Case and Class Diagrams for an ATM System **Problem:** Create a Use Case Diagram and a Class Diagram for an ATM System where a Customer can withdraw cash and check balance.
Solution:
Part 1: Use Case Diagram



Part 2: Class Diagram



Methodology:

Designing Sequence and Activity Diagrams **Sequence Diagram:** Captures the dynamic behavior and message flow over time.

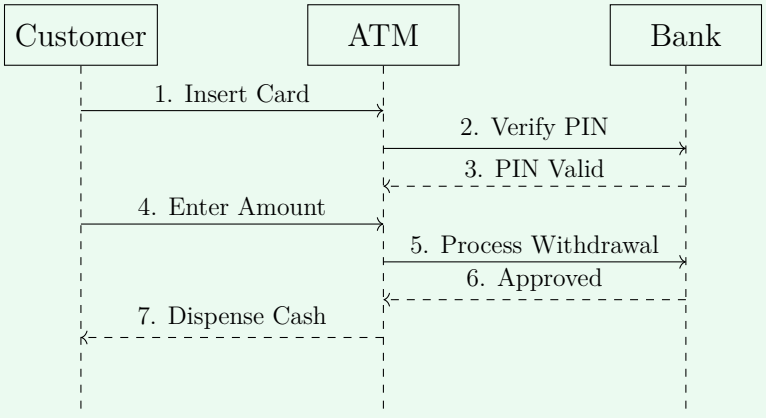
1. Draw objects/actors as rectangles across the top.
2. Draw vertical dashed lines (lifelines) for each object.
3. Draw horizontal arrows for messages exchanged, ordered top to bottom.

Activity Diagram: Captures the flow of control from activity to activity.

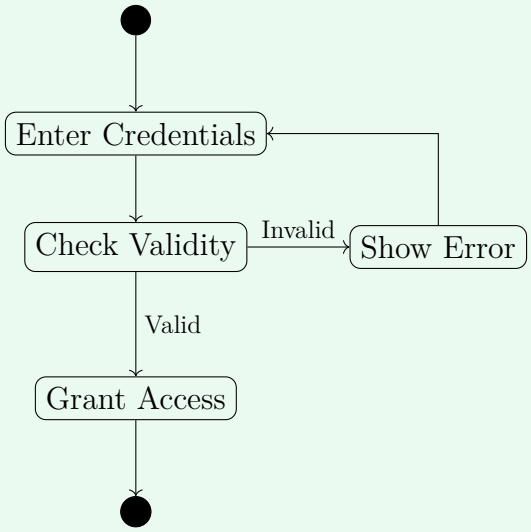
1. Use a solid circle for the initial state.
2. Use rounded rectangles for activities.
3. Use text conditions on arrows for decision points.
4. Use a concentric circle for the final state.

Worked Example:

Sequence and Activity Diagrams **Problem 1: Sequence Diagram for ATM Cash Withdrawal**



Problem 2: Activity Diagram for User Login Process


Enter Credentials
Check Validity
Invalid
Show Error
Valid
Grant Access

CHAPTER 4

Software Coding and Testing

Software coding and testing form a critical phase where the design is translated into executable code and systematically verified for correctness. This unit focuses on the computational and analytical techniques used to measure code complexity, derive independent execution paths, and systematically generate test cases using both white-box and black-box testing strategies.

4.1 White-Box Testing and Complexity Metrics

White-box testing examines the internal logical structure of the software. A primary metric used to evaluate this complexity is Cyclomatic Complexity, which determines the number of independent paths through a program.

Methodology:

Drawing a Control Flow Graph A Control Flow Graph (CFG) visually represents the sequence of execution and decision points in a program.

1. **Identify Nodes:** Each sequential statement or block of sequential statements is represented by a node (a circle).
2. **Identify Edges:** The flow of control between statements is represented by directed edges (arrows).
3. **Map Decision Points:** Conditional statements (if, while, for) are represented as predicate nodes with two or more outgoing edges (e.g. True and False paths).
4. **Combine Sequential Blocks:** Consecutive statements without any branching can be merged into a single node.
5. **Connect to Exit:** All paths must eventually merge and lead to a final exit node.

Methodology:

Calculating Cyclomatic Complexity Cyclomatic Complexity $V(G)$ is a software metric used to measure the logical complexity of a program. It defines the number of independent paths in the basis set. It can be calculated using three formulas:

1. **Edge-Node Formula:** $V(G) = E - N + 2$, where E is the number of edges and N is the number of nodes.
2. **Predicate Node Formula:** $V(G) = P + 1$, where P is the number of predicate nodes (nodes with more than one outgoing edge).
3. **Region Formula:** $V(G) = R$, where R is the total number of regions in the flow graph, including the one outer (unbounded) region.

Worked Example:

Calculate Cyclomatic Complexity of a Code Segment Consider the following pseudo-code:

```
1. int i = 1;
2. while (i <= 10) {
3.     if (i % 2 == 0) {
4.         print("Even");
5.     } else {
6.         print("Odd");
7.     }
8.     i++;
9. }
10. print("Done");
```

Calculate its cyclomatic complexity using all three methods.

Step 1: Identify Nodes and Edges Let's assign node numbers to statements:

- Node 1: Statement 1 (Initialization)
- Node 2: Statement 2 (while condition)
- Node 3: Statement 3 (if condition)
- Node 4: Statement 4 (print "Even")
- Node 5: Statement 6 (print "Odd")
- Node 6: Statement 8 (i++)
- Node 7: Statement 10 (print "Done")

Step 2: Trace Edges Edges exist between:

- $1 \to 2$
- $2 \to 3$ (Loop entry)
- $2 \to 7$ (Loop exit)
- $3 \to 4$ (If True)
- $3 \to 5$ (If False)
- $4 \to 6$ (Path merges)
- $5 \to 6$ (Path merges)
- $6 \to 2$ (Loop back)

Step 3: Count Components for Formulas

- Nodes $N = 7$
- Edges $E = 8$
- Predicate Nodes $P = 2$ (Node 2 for `while` and Node 3 for `if`)
- Regions $R = 3$ (Two bounded regions formed by the loop and the if-else branch plus one outer unbounded region).

Step 4: Calculate $V(G)$

- Using Edge-Node: $V(G) = E - N + 2 = 8 - 7 + 2 = 3$
- Using Predicate: $V(G) = P + 1 = 2 + 1 = 3$
- Using Regions: $V(G) = R = 3$

The cyclomatic complexity is 3.

Methodology:

Deriving the Basis Set of Independent Paths Basis Path Testing ensures that every independent path through a program is executed at least once.

1. **Draw CFG:** Create the Control Flow Graph from the source code.
2. **Calculate $V(G)$:** Determine the cyclomatic complexity to know the exact number of independent paths needed.
3. **Identify Baseline Path:** Choose a main or most typical path from the start node to the end node.
4. **Flip Decisions:** Generate subsequent paths by changing the outcome of one predicate (decision) node at a time in the baseline path, keeping other decisions the same if possible.
5. **Verify:** Ensure you have identified exactly $V(G)$ paths and that each path introduces at least one new set of processing statements or a new condition edge.

Worked Example:

Find Independent Paths for a CFG Given the CFG from the previous example with $N = 7$, $E = 8$, and $V(G) = 3$.

- Nodes: 1, 2, 3, 4, 5, 6, 7.
- Edges: (1,2), (2,3), (2,7), (3,4), (3,5), (4,6), (5,6), (6,2).

Step 1: Identify Cyclomatic Complexity As calculated, $V(G) = 3$. We need to find exactly 3 independent paths.
Step 2: Identify Path 1 (Baseline) Let's choose the path that bypasses the loop entirely (condition at Node 2 is false). Path 1: $1 \to 2 \to 7$
Step 3: Identify Path 2 (Flip first condition) Assume the loop executes once and the `if` condition is true. Path 2: $1 \to 2 \to 3 \to 4 \to 6 \to 2 \to 7$
Step 4: Identify Path 3 (Flip second condition) Assume the loop executes once and the `if` condition is false. Path 3: $1 \to 2 \to 3 \to 5 \to 6 \to 2 \to 7$
Step 5: Verification We have exactly 3 paths.

- Path 1 covers edges: (1,2), (2,7).
- Path 2 introduces new edges: (2,3), (3,4), (4,6), (6,2).
- Path 3 introduces new edges: (3,5), (5,6).

Since each path introduces at least one new edge, they form a valid basis set of independent paths.

4.2 Halstead Complexity Metrics

Halstead metrics provide another quantitative way to evaluate code complexity based on the operators and operands present in the source code.

Methodology:

Calculating Halstead Complexity Metrics Halstead's metrics measure the complexity of a program directly from source code by counting operators and operands.

1. **Identify Operators and Operands:**
   - n_1: Number of distinct operators (keywords, math operators, logical operators).
   - n_2: Number of distinct operands (variables, constants).

- N_1: Total number of occurrences of operators.
- N_2: Total number of occurrences of operands.

2. **Calculate Vocabulary (n) and Length (N):**
   - Program Vocabulary: $n = n_1 + n_2$
   - Program Length: $N = N_1 + N_2$
3. **Calculate Estimated Length ($\hat{N}$):** $\hat{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2)$
4. **Calculate Volume (V):** $V = N \times \log_2(n)$
5. **Calculate Difficulty (D) and Effort (E):**
   - Difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
   - Effort: $E = D \times V$

Worked Example:

Calculate Halstead Metrics for a Code Segment Consider the following C code segment:

```
int a = 5;
int b = 10;
int c = a + b;
```

Step 1: Identify Operators and Operands Operators:

- `int` (type declaration) - 3 occurrences
- `=` (assignment) - 3 occurrences
- `+` (addition) - 1 occurrence
- `;` (terminator) - 3 occurrences

Distinct operators (n_1) = 4. Total operator occurrences (N_1) = $3 + 3 + 1 + 3 = 10$.
Operands:

- Variables: `a`, `b`, `c` (occurrences: `a`:2, `b`:2, `c`:1)
- Constants: `5`, `10` (occurrences: `5`:1, `10`:1)

Distinct operands (n_2) = 5 (`a`, `b`, `c`, `5`, `10`). Total operand occurrences (N_2) = $2 + 2 + 1 + 1 + 1 = 7$.
Step 2: Calculate Vocabulary and Length

- Program Vocabulary (n) = $n_1 + n_2 = 4 + 5 = 9$.
- Program Length (N) = $N_1 + N_2 = 10 + 7 = 17$.

Step 3: Calculate Volume (V) $V = N \times \log_2(n) = 17 \times \log_2(9) \approx 17 \times 3.17 = 53.89$ bits.
Step 4: Calculate Difficulty (D) and Effort (E)

- Difficulty $D = \frac{n_1}{2} \times \frac{N_2}{n_2} = \frac{4}{2} \times \frac{7}{5} = 2 \times 1.4 = 2.8$.
- Effort $E = D \times V = 2.8 \times 53.89 = 150.89$.

The computed volume is 53.89 and the required effort is 150.89.

4.3 Black-Box Testing Techniques

Black-box testing focuses on validating the software against its functional requirements without looking at the internal code structure.

Methodology:

Equivalence Class Partitioning Equivalence Partitioning divides the input domain into classes of data from which test cases can be derived.

1. **Identify Input Conditions:** Read the requirement specification to find input variables and their constraints (ranges, specific values, sets).
2. **Define Valid Classes:** Identify subsets of inputs that should be accepted by the system.
3. **Define Invalid Classes:** Identify subsets of inputs that violate conditions and should be rejected.
4. **Design Test Cases:** Select one representative value from each equivalence class (both valid and invalid) as a test case.

Worked Example:

Design Test Cases using ECP A program requires a user to enter a valid month number (1 to 12). Design test cases using Equivalence Class Partitioning.

Step 1: Identify Input Conditions

- Input: `MonthNumber` (Integer).
- Condition: Range $1 \leq$ MonthNumber ≤ 12.

Step 2: Define Valid Equivalence Classes

- Valid Class 1 (VC1): Integers between 1 and 12, inclusive. $(1 \leq M \leq 12)$

Step 3: Define Invalid Equivalence Classes

- Invalid Class 1 (IC1): Integers less than 1. $(M < 1)$
- Invalid Class 2 (IC2): Integers greater than 12. $(M > 12)$

Step 4: Select Representative Test Cases

- From VC1, select a valid value, e.g. $M = 5$.
- From IC1, select an invalid value below range, e.g. $M = -3$.
- From IC2, select an invalid value above range, e.g. $M = 15$.

Final Test Cases:

Test Case ID	Input (M)	Expected Output
TC1	5	Valid
TC2	-3	Invalid
TC3	15	Invalid

Methodology:

Boundary Value Analysis Errors often occur at the extremes of input domains. Boundary Value Analysis (BVA) focuses on values at the boundaries of equivalence classes.

1. **Identify Input Range:** Determine the minimum (min) and maximum (max) valid values for an

input.

2. **Identify Boundary Values:** For a closed range $[min, max]$, identify the following test inputs:
   - Minimum value: min
   - Just above minimum: $min + 1$ (or smallest possible increment)
   - Nominal value: A value well within the range
   - Just below maximum: $max - 1$
   - Maximum value: max
3. **Identify Invalid Boundaries (Robustness):** Select values just outside the boundaries: $min - 1$ and $max + 1$.

Worked Example:

Design Test Cases using BVA A shopping website offers free shipping for order totals between $50.00 and $500.00. Design test cases using Boundary Value Analysis.

Step 1: Identify Input Range

- Input: `OrderTotal` (Float/Decimal).
- Valid Range: $[50.00, 500.00]$.
- Since it is currency, the minimum increment is 0.01.

Step 2: Identify Valid Boundary Values

- Minimum (min): 50.00
- Just above minimum ($min + 0.01$): 50.01
- Nominal value: 200.00
- Just below maximum ($max - 0.01$): 499.99
- Maximum (max): 500.00

Step 3: Identify Invalid Boundaries

- Just below minimum ($min - 0.01$): 49.99
- Just above maximum ($max + 0.01$): 500.01

Final BVA Test Cases:

Test Case ID	Input Value	Expected Output	Boundary Type
TC1	49.99	Invalid	Below lower boundary
TC2	50.00	Valid	Lower boundary
TC3	50.01	Valid	Just above lower
TC4	200.00	Valid	Nominal value
TC5	499.99	Valid	Just below upper
TC6	500.00	Valid	Upper boundary
TC7	500.01	Invalid	Above upper boundary

CHAPTER 5

Software Coding and Testing

5.1 Code Review and Software Documentation

Methodology:

Code Review Techniques Code review ensures code quality by identifying defects early. The process typically follows these structured steps:

1. **Code Walkthrough:** An informal review process.
   - **Step 1:** The author of the code organizes a meeting and distributes the code to the team.
   - **Step 2:** The author presents the code step-by-step, explaining the logic.
   - **Step 3:** Reviewers ask questions, trace the logic, and suggest improvements.
2. **Code Inspection:** A formal, checklist-driven review.
   - **Step 1:** Roles are assigned (Moderator, Author, Reader, Recorder).
   - **Step 2:** Reviewers independently examine the code against a predefined checklist.
   - **Step 3:** The team holds an inspection meeting to log defects (without debating solutions).

Methodology:

Software Documentation Practices Proper documentation requires structured formatting for both internal and external audiences:

1. **Internal Documentation (Code-level):**
   - **Header Blocks:** Include file name, author, creation date, and overall purpose at the top of the file.
   - **Function Comments:** Detail input parameters, expected return values, and possible exceptions.
   - **Inline Comments:** Explain complex algorithms focusing on the "why" rather than the "what".
2. **External Documentation (User-level):**
   - **User Manual:** Step-by-step instructions on how end-users can operate the software features.
   - **Installation Guide:** Technical steps required to install, configure, and deploy the system.

5.2 White-Box Testing and Basis Path Testing

White-box testing (or glass-box testing) involves testing the internal structure and logic of the code. A key computational technique in white-box testing is determining the Cyclomatic Complexity to design Basis Path test cases.

Methodology:

Basis Path Testing Method **Step 1: Construct the Flow Graph** Convert the source code into a directed graph where nodes represent statements (or blocks of sequential statements) and edges represent control flow paths.
Step 2: Calculate Cyclomatic Complexity $V(G)$ Compute the complexity using any of the following mathematical formulas:

1. $V(G) = E - N + 2$ (where E = number of edges, N = number of nodes)
2. $V(G) = P + 1$ (where P = number of predicate/decision nodes)
3. $V(G)$ = Total number of enclosed regions + 1

Step 3: Identify Independent Paths Trace exactly $V(G)$ distinct paths from the start node to the end node. Each independent path must traverse at least one new edge not covered by previous paths.
Step 4: Design Test Cases Derive specific input values that guarantee the execution of each identified independent path.

Worked Example:

Calculating Cyclomatic Complexity and Paths Consider the following pseudocode designed to find the maximum of two numbers:

```
1. int max = 0;
2. if (A > B) {
3.     max = A;
4. } else {
5.     max = B;
6. }
7. print(max);
```

Step 1: Construct the Flow Graph

- Node 1: Statements 1 and 2 (Initialization and Condition check)
- Node 2: Statement 3 (True branch execution)
- Node 3: Statement 5 (False branch execution)
- Node 4: Statement 7 (Print output and exit)

Edges connecting the nodes: $1 \to 2$, $1 \to 3$, $2 \to 4$, $3 \to 4$.
Step 2: Calculate Cyclomatic Complexity

- Using $E - N + 2$: $E = 4$, $N = 4 \implies V(G) = 4 - 4 + 2 = 2$.
- Using $P + 1$: Node 1 is the only predicate node, so $P = 1 \implies V(G) = 1 + 1 = 2$.

Step 3: Identify Independent Paths We need exactly 2 independent paths based on $V(G)$:

- **Path 1:** $1 \to 2 \to 4$
- **Path 2:** $1 \to 3 \to 4$

Step 4: Design Test Cases

- **Test Case for Path 1:** Set A = 10, B = 5. (Condition A > B is True). Expected Output: 10.
- **Test Case for Path 2:** Set A = 5, B = 10. (Condition A > B is False). Expected Output: 10.

5.3 Black-Box Testing Techniques

Black-box testing verifies functional requirements without examining the internal source code. The primary computational and analytical methods used are Equivalence Partitioning (EP) and Boundary Value Analysis (BVA).

Methodology:

Equivalence Partitioning Method **Step 1: Identify the Input Domain** Determine the input conditions and ranges specified by the software requirements.
Step 2: Create Equivalence Classes Divide the input domain into mathematical sets (classes) where the software is expected to behave identically:

- **Valid Classes:** Subsets of inputs that should be accepted by the system.
- **Invalid Classes:** Subsets of inputs that should trigger error handling or be rejected.

Step 3: Select Representative Values Pick exactly one test value from each equivalence class to formulate your test cases.

Worked Example:

Applying Equivalence Partitioning A university admission system accepts student percentage scores between 50 and 100 inclusive.
Step 1 & 2: Identify Domain and Create Classes

- **Valid Class:** $50 \leq \text{Percentage} \leq 100$
- **Invalid Class 1:** $\text{Percentage} < 50$
- **Invalid Class 2:** $\text{Percentage} > 100$

Step 3: Select Representative Values

Equivalence Class	Test Input Value	Expected Result
Valid Class	75	Accepted
Invalid Class 1	40	Rejected
Invalid Class 2	105	Rejected

Methodology:

Boundary Value Analysis Method **Step 1: Identify Boundaries** Find the minimum (min) and maximum (max) strict numerical limits of the input domain.
Step 2: Generate Test Cases Create test inputs exactly at, just below, and just above the boundaries. The standard values to test are:

- Lower boundary: $min - 1$, min, $min + 1$
- Upper boundary: $max - 1$, max, $max + 1$

Worked Example:

Applying Boundary Value Analysis An e-commerce discount code is valid only for cart totals ranging from \$100 to \$500.
Step 1: Identify Boundaries $min = 100$, $max = 500$.
Step 2: Generate Test Cases

Boundary Condition	Mathematical Calculation	Test Input	Expected Result
$min - 1$	$100 - 1$	\$99	Invalid
min	100	\$100	Valid
$min + 1$	$100 + 1$	\$101	Valid
$max - 1$	$500 - 1$	\$499	Valid
max	500	\$500	Valid
$max + 1$	$500 + 1$	\$501	Invalid

5.4 Unit Testing and Test Case Templates

Unit testing involves verifying individual modules or functions in complete isolation. To standardize this process, test cases must be documented using predefined templates that capture all necessary execution details.

Methodology:

Creating a Test Case Template A standard test case template ensures all testing scenarios are documented clearly for reproducibility.

1. **Test Case ID:** A unique alphanumeric identifier (e.g., TC_001).
2. **Test Description:** A brief, clear summary of what specific functionality is being tested.
3. **Preconditions:** The required system state or environment setup before executing the test.
4. **Test Steps:** Sequentially numbered actions the tester must perform on the system.
5. **Test Data:** The specific input values required to execute the test steps.
6. **Expected Result:** The exact behavior, UI change, or output expected from the system.
7. **Actual Result:** The actual behavior observed (left blank until test execution).
8. **Pass or Fail Status:** The final outcome of the test (left blank until test execution).

Worked Example:

Writing a Test Case for User Login Draft a formal test case template for testing an unsuccessful login attempt due to an incorrect password.

Test Case ID	TC_LOGIN_002
Test Description	Verify that the system rejects a login attempt with an invalid password.
Preconditions	1. The application server is running. 2. The user is on the login page. 3. The user has a registered, active account.
Test Steps	1. Enter a valid registered username in the username field. 2. Enter an invalid/incorrect password in the password field. 3. Click the "Login" button.
Test Data	Username: `admin_user` Password: `wrongpass123`
Expected Result	The system displays a red error message stating "Invalid credentials" and the user remains on the login page.
Actual Result	*(To be filled during test execution)*
Pass / Fail Status	*(To be filled during test execution)*

Appendix: Key Formulae

This appendix provides a quick reference to the general formulae and mathematical relationships discussed in this book, categorized by unit and topic.

Unit 1: Effort, Cost, and Risk Estimation

Brooks's Law Estimations

Used to estimate the relative effort required for different types of software products based on a basic program's effort (E_{prog}).

$$E_{prod} = 3 \times E_{prog} \quad \text{(Programming Product)}$$
$$E_{sys} = 3 \times E_{prog} \quad \text{(Programming System)}$$
$$E_{sys\_prod} = 9 \times E_{prog} \quad \text{(System Product)}$$

40-20-40 Rule for Effort Distribution

A general rule of thumb for distributing total effort (E_{total}) across different phases of the software development life cycle.

$$E_{front} = 0.40 \times E_{total} \quad \text{(Front-end activities: Requirements \& Design)}$$
$$E_{coding} = 0.20 \times E_{total} \quad \text{(Coding activities)}$$
$$E_{back} = 0.40 \times E_{total} \quad \text{(Back-end activities: Testing \& Integration)}$$

Cost of Defect Removal

Estimates the increasing cost of fixing a defect at later phases of development, assuming an exponential growth.

$$C_k = C_1 \times 10^{(k-1)}$$

Where C_k is the cost at phase k, and C_1 is the cost at the initial phase.

Risk Exposure

Calculates the expected impact of a risk.

$$RE = P \times C$$

Where P is the probability of the risk occurring, and C is the cost or impact if it occurs.

Unit 2: Agile and Project Tracking

Burndown Rates

Used to track progress in Agile sprints.

$$\text{Ideal Burndown Rate} = \frac{\text{Total Sprint Work}}{\text{Total Days in Sprint}}$$

$$\text{Actual Burndown Rate} = \frac{\text{Work Completed So Far}}{\text{Days Elapsed}}$$

Team and Timeline Estimation

Formulas for calculating team size and total project time for sequential or parallel delivery increments.

$$\text{Required Team Size} = \frac{\text{Total Estimated Effort (Person-Months)}}{\text{Timebox Duration (Months)}}$$

$$\text{Total Time (Sequential)} = \sum_{i=1}^{N} \text{Time of Increment}_i$$

$$\text{Total Time (Parallel)} = \max(\text{Time of Increment}_1, \text{Time of Increment}_2, \ldots, \text{Time of Increment}_N)$$

Velocity and Sprint Planning

Used to estimate future progress based on past performance in Agile teams.

$$\text{Average Velocity} = \frac{\sum \text{Completed Story Points across } K \text{ Sprints}}{K}$$

$$\text{Estimated Sprints Remaining} = \frac{\text{Remaining Product Backlog Points}}{\text{Average Velocity}}$$

Programming Effort Estimation

Comparing the effort of solo programming versus pair programming.

$$\text{Effort}_{\text{solo}} = \text{Coding Time}_{\text{solo}} + \text{Debugging Time}_{\text{solo}}$$
$$\text{Effort}_{\text{pair}} = 2 \times (\text{Coding Time}_{\text{pair}} + \text{Debugging Time}_{\text{pair}})$$

Unit 4: Halstead's Software Metrics

Halstead's Base Metrics

Fundamental metrics based on the counting of operators and operands in a program.

$$n = n_1 + n_2 \qquad \text{(Program Vocabulary)}$$
$$N = N_1 + N_2 \qquad \text{(Program Length)}$$

Where n_1 and n_2 are the number of unique operators and operands, respectively. N_1 and N_2 are the total number of operators and operands.

Estimated Program Length

An approximation of the program length based solely on the program vocabulary.

$$\hat{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2)$$

Halstead Effort

Estimates the mental effort required to develop or maintain the software.

$$E = D \times V$$

Where D is the Difficulty and V is the Volume of the program.

Unit 5: Cyclomatic Complexity

McCabe's Cyclomatic Complexity, $V(G)$

A metric used to indicate the complexity of a program based on its control flow graph. It can be calculated using three different approaches:

$$V(G) = E - N + 2 \qquad \text{(Based on Edges } (E) \text{ and Nodes } (N))$$

$$V(G) = P + 1 \qquad \text{(Based on Predicate/Decision Nodes } (P))$$

$$V(G) = \text{Total number of enclosed regions} + 1 \qquad \text{(Based on Regions)}$$

Note: When counting regions, always include the single outer (unbounded) region.