UNIT-2

Software Project Management: Software project management complexities, Responsibilities of a software project manager, Metrics for project size estimation, Project estimation techniques, Empirical Estimation techniques, COCOMO, Halstead's software science, Risk management.

Requirements Analysis and Specification: Requirements gathering and analysis, Software Requirements Specification (SRS), Formal system specification, Axiomatic specification, Algebraic specification, Executable specification and 4GL.

REQUIREMENTS ANALYSIS AND SPECIFICATION

1.1 REQUIREMENTS GATHERING AND ANALYSIS INTRODUCTION:

- The complete set of requirements are never available in the form of a single document from the customer. Customers provide a comprehensive document containing a precise description of what he wants. So, the requirements have to be gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources.
- We can conceptually divide the requirements gathering and analysis activity into two separate tasks:
    1. Requirements gathering
    2. Requirements analysis

1. Requirements Gathering

- Requirements gathering is also popularly known as *requirements elicitation.* The primary objective of the requirements gathering task is to collect the requirements from the stakeholders or user. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents.
- Gathering requirements turns out to be especially challenging. Good analysts share their experience and expertise with the customer. The analyst gathers requirements:
    - **Studying existing documentation:** The analyst usually studies all the available documents before visiting the customer site. These documents discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.
    - **Interview:**Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each from them by interviewing.
    - **Task analysis:** Consider the software that provides a set of services (functionalities). A service is also called a *task*. The analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate.

2.Requirements Analysis:

- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to

be developed. Identify and resolve the three problems that he detects in the gathered requirements are :

1. **Anomaly**: When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system.
2. **Inconsistency**: Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
3. **Incompleteness**: An experienced analyst can detect most of the missing, incomplete features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

1.2 SOFTWARE REQUIREMENTS SPECIFICATION (SRS):

The Software Requirements Specification (SRS) document is created after the analyst has gathered and refined all necessary information about the software to be developed. It ensures that the requirements are complete, consistent, and free of inconsistencies. The SRS organizes these user requirements in a structured format.

1.2.1 Users of SRS Document:

Users, customers, and marketing personnel: They use it to ensure the system meets their needs. Marketing personnel need it to explain requirements to customers.

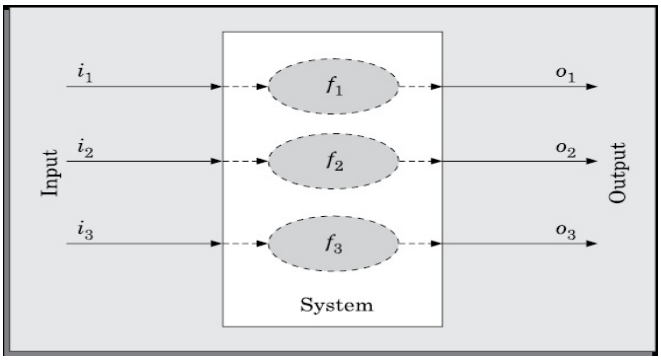
- **Software developers**: They refer to the SRS to develop the software as per the customer's specifications.
- **Test engineers**: They use the SRS to understand the required functionalities and create test cases for validation.
- **User documentation writers**: They rely on the SRS to understand the product features for writing user manuals.
- **Project managers**: They use the SRS for cost estimation and project planning.
- **Maintenance engineers**: They refer to the SRS to understand system functionalities and plan modifications
- Why Spend Time and Resource to Develop an SRS Document?
- **Forms an agreement between customers and developers**: It sets expectations for both the customer and the developer regarding the software requirements.
- **Reduces future reworks**: Preparing the SRS document forces stakeholders to thoroughly think through all requirements before development begins, reducing the need for redesign, recoding, and retesting.
- **Provides a basis for estimating costs and schedules**: Project managers use the SRS to estimate the size of the software and determine the effort and cost for development. It also helps in price negotiations and work scheduling.
- **Provides a baseline for validation and verification**: The SRS serves as a reference to check if the developed software meets the requirements. It is also used by test engineers to create test plans.
- **Facilitates future extensions**: The SRS acts as a foundation for planning future enhancements to the software..

1.2.2 Characteristics of a Good SRS Document:

A good SRS document should possess several desirable qualities, as outlined by the IEEE Recommended Practice for Software Requirements Specifications (IEEE 830). These qualities include:

- **Concise**: The SRS should be brief, clear, unambiguous, consistent, and complete. Avoid verbosity and irrelevant details, as they can reduce readability and increase the chance of errors.
- **Implementation-independent**: The SRS should focus only on what the system should do, not how it should do it. It should describe the externally visible behavior of the system (i.e., the "black box") and avoid design decisions.
- **Traceable**: Traceability helps verify the results of each phase of development and allows for analyzing the impact of changes to requirements on design and code.
- **Modifiable**: Scattering requirement descriptions across multiple locations should be avoided, as it complicates modifications.
- **Identification of responses to undesired events**: The SRS should specify how the system will respond to exceptional or undesired events that may arise during operation.
- **Verifiable**: All requirements should be verifiable, e.g: "the software should display book availability when a book name is entered" are verifiable. Non-verifiable features should be listed separately as goals in the SRS.

These qualities ensure the SRS document is effective in guiding development, accommodating changes, and facilitating verification.



1.2.3 Attributes of Bad SRS Documents:

- **Over-specification**: This occurs when the analyst goes into detail about the "how-to" aspects of the system, limiting the designer's flexibility. E.g: specifying how to store library membership records (e.g., by first name or ID number) restricts design freedom. The SRS should focus on "what" the system needs to do, not "how" it does it.
- **Forward references**: The SRS should avoid forward referencing to maintain clarity.
- **Wishful thinking**: Features that may be difficult or unrealistic to implement should be avoided, as they can lead to misunderstandings and unrealistic expectations.
- **Noise**: "Noise" refers to irrelevant information that does not contribute to the software development process. E.g: Customer registration department's working hours are not useful for software developers and should be excluded from the SRS, as they only clutter the document.

1.2.4 Important Categories of Customer Requirements:

The key categories of user requirements that should be clearly documented are:

- **Functional Requirements**: These capture the functionalities that users expect from the system. The functional requirements should clearly describe each functionality of the system, including the corresponding input and output data.
- **Non-functional Requirements:**Non-functional requirements capture aspects of the system that are not related to specific functions but are essential for overall performance and usability. These include:
  - **Design and Implementation Constraints**: These are limitations on how the system can be designed or built, such as regulatory policies, hardware limitations, required technologies, communication protocols, security concerns, or design conventions.
  - **External Interfaces Required**: This includes interfaces with other systems (hardware, software), user interfaces, and report formats. For user interfaces, specific design aspects like screen layouts, GUI standards, keyboard shortcuts, and error messages should be specified.
  - **Other Non-functional Requirements**: This covers like performance (e.g., transactions per second), reliability, accuracy, and security. These are critical for system acceptance, and failure to meet them can render the software unacceptable to the customer.
- **Goals of Implementation:** Provides general suggestions that are not mandatory but may help guide the developers. For instance, goals may include designing the system to be easily adaptable for future revisions or expansions.

1.2.5 Functional Requirements:

The Functional requirements include high-level functions, that can be broken down into smaller subrequirements. A high-level function represents a task the user can perform with the system to accomplish something useful. **e.g:** Printing a receipt during an ATM withdrawal might not be considered a high-level function because it's an automatic part of the withdrawal process, not something explicitly requested by the user. It could be better treated as part of the "withdraw money" function instead.

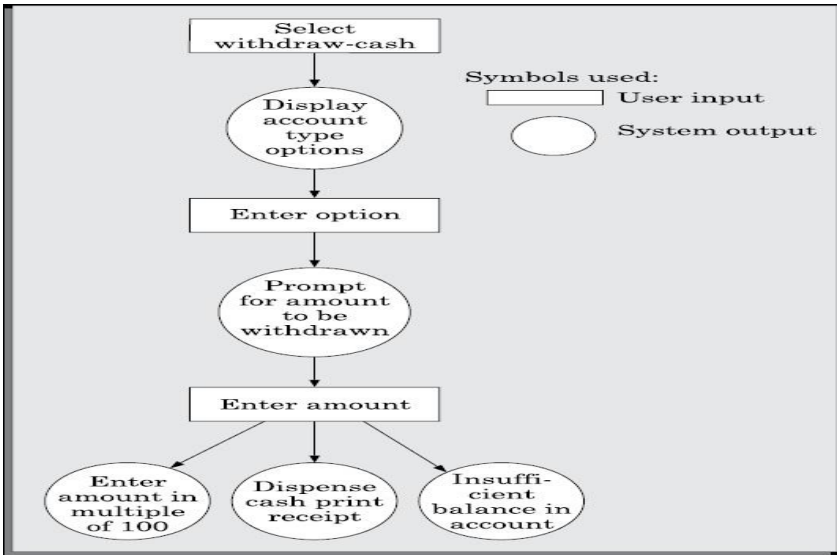
Each high-level function typically involves:

1. Accepting user input through an interface.
2. Processing that input to generate a response.
3. Displaying the result in a suitable format.

For instance, in a library automation system, a high-level function like "search-book" involves inputting a book name, searching the database, and showing the matched results.
Identifying high-level functions becomes easier with practice and experience.

Are high-level functions of a system similar to mathematical functions?

- High-level functions usually involve multiple interactions between the system and the user. A high-level function can consist of several smaller sub-requirements, each reflecting a part of the interaction.
- *E.g:* In a high-level function, the user may first input data, and the system then responds and the user may provide more input, and the system responds again. This interaction typically alternates between user inputs and system actions, forming a sequence of steps.



1.2.6 How to Identify the Functional Requirements?

High-level functional requirements are often identified from an informal problem description or a conceptual understanding of the problem. Each high-level requirement represents a way the system is used by a user to accomplish a meaningful task.

Identify the types of users and their expected services from the software.

Deciding whether a functionality should be considered a high-level requirement or a subfunction can sometimes be subjective. E.g: In a Library Automation System, when a user invokes the "issue-book" function, the system might ask for book details. This is a high-level function or part of the "issue-book" function. While some decisions are clear, others may require more thought and judgment.

1.2.7 How to Document the Functional Requirements?

Once all high-level functional requirements are identified, the next step is to document them. Each function is described by identifying the state of input data, the input data domain, the output data domain, and the type of processing required to produce the output.

E.g: , consider documenting the *withdraw-cash* function of an ATM system:

Example: Withdraw Cash from ATM

This function involves multiple user interactions and scenarios that vary depending on the conditions.

R.1: Withdraw Cash

Description: The function checks the user's account type, verifies the account balance, and then either dispenses cash or displays an error message if there are insufficient funds.

R.1.1: Select Withdraw Amount Option

- Input: The user selects the "Withdraw amount" option.
- Output: The system prompts the user to choose their account type.

R.1.2: Select Account Type

- Input: The user selects an account type from options like savings, checking, or deposit.
- Output: The system prompts the user to enter the withdrawal amount.

R.1.3: Get Required Amount

- Input: The user enters the amount to withdraw (between 100 and 10,000, in multiples of 100).

- Output: The system dispenses the requested cash and prints a transaction statement.
- Processing: The system checks if the account balance is sufficient to cover the requested amount. If the balance is enough, the money is debited from the account. If not, an error message is shown.

1.2.8 Organisation of the SRS Document:

Guidelines for structuring the document are based on the organization, preferences of the system analyst, company policies, product type, and the intended audience.
All SRS documents should address three core topics:

- Functional Requirements
- Non-functional Requirements
- System Implementation Guidelines

Introduction:

- Describes where and how the software will be used.
- Project Scope: It should mention areas that may need to be automated in the future.
- Environmental Characteristics: It can include system requirements and user skill levels.

Product Perspective:

- Product Perspective: Clarifies whether the software is a new system or a replacement for an existing one.
- Product Features: Summarizes the ways in which the software will be used.
- User Classes: Identifies and describes the different types of users and their roles, expertise, and expected interactions with the system.
- Operating Environment: Details the hardware platform, operating system, and other software that the system will interact with.

Design and Implementation Constraints:

- Discusses various constraints that may limit design and implementation, including:
  - Corporate or regulatory policies
  - Hardware limitations (e.g., memory, timing)
  - External application interfaces
  - Required technologies, tools, and databases
  - Specific communication protocols, and security considerations
  - Design conventions or programming standards

User Documentation:

- Lists the types of user documentation that will be provided, such as user manuals, online help, troubleshooting guides, etc.

External interface requirements:

1.*User Interfaces:*

Includes high-level description of the system's user interfaces. It includes:

- Sample screen images to illustrate the interface.
- GUI standards or style guides to be followed (e.g., button styles, fonts, colors).
- Screen layout constraints (e.g., position of buttons, fields, or menus).
- Standard push buttons (like "Help") that will appear on every screen.
- Keyboard shortcuts for efficient interaction.
- Error message display standards to ensure consistency and clarity.

2.*Hardware Interfaces:*

Explains the interaction between the software and hardware components of the system. It includes:

- Supported device types (e.g., printers, sensors).
- The nature of data and control interactions between the software and hardware.
- The communication protocols used to facilitate these interactions.

3.*Software Interfaces:*

Explains how the software connects with other software components, such as:

- Databases (e.g., types of databases, how the software interacts with them).
- Operating systems and their integration.
- Tools, libraries, and integrated commercial components used by the software.
- Data inputs to the software and data outputs, specifying their purposes.

4*.Communications Interfaces:*

This section covers the communication requirements for the software. It includes:

- Communication protocols (e.g., e-mail, web access, server communications).
- Message formatting used in the communication.
- Communication standards (e.g., TCP/IP, FTP, HTTP, SHTTP).
- Security measures for communication (e.g., encryption, authentication).
- Data transfer rates and synchronization mechanisms for smooth communication.

1.2.9 Techniques for Representing Complex Logic:

Representation of complex decision-making logic in SRS documents, especially when multiple alternatives and conditions are involved. The use of decision trees and decision tables to manage these complexities.

- **Decision Trees and Decision Tables**: Decision trees offer a graphical view of the logic and actions taken, where edges represent conditions, and leaf nodes represent the resulting actions. Decision tables list variables and actions based on outcomes, helping in structuring the decision process.
- **Use Cases**: Decision trees and Tables are helpful for mapping out complex decision logic. e.g : In a Library Membership Management Software (LMS), different options such as "new member," "renewal," and "cancel membership" lead to various conditions and actions based on user inputs.

DECISION TABLE:

The passage describes **decision tables**, which are a way of representing decision-making logic and corresponding actions in a tabular or matrix form. Here's a summary of the key points:

- **Structure**: The upper rows of the table represent the conditions or variables that need to be evaluated. The lower rows specify the actions that should be taken when a particular condition is satisfied.
- **Rules**: Each column in the table represents a rule, which is a combination of conditions. If a certain combination of conditions is true, the corresponding action is executed.

In LMS problem, a decision table would help clarify the decision-making process by showing all possible combinations of conditions (e.g., whether a member is valid or not) and the actions that should be taken for each condition combination.

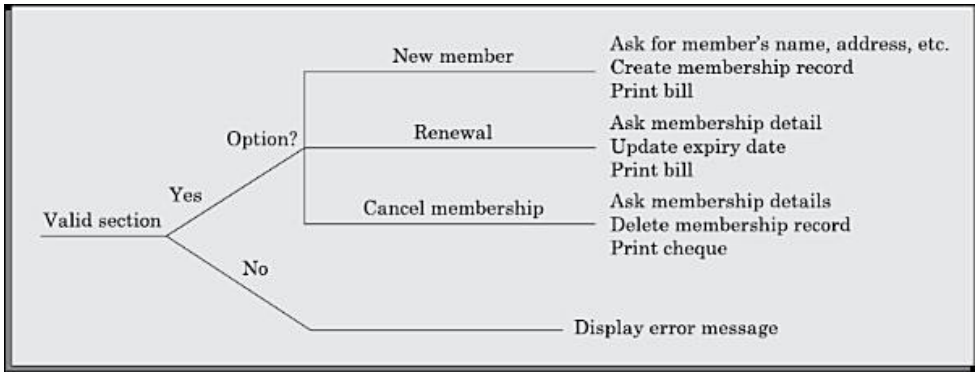


Table 4.1: Decision Table forthe LMS Problem

Conditions				
Valid selection	no	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message		*		
Ask member's name, etc.			*	
Build customer record			*	
Generate bill			*	*
Ask membership details				*
Update expiry date				*
Print cheque				*
Delete record				*

- **Decision table *versus* Decision tree:**

The differences between **decision trees** and **decision tables** based on three key considerations:

1. **Readability**:
   - **Decision Trees**: Easier to read and understand when the number of conditions is small, as they provide a visual, hierarchical representation.
   - **Decision Tables**: Require examining all possible combinations of conditions, which may be cumbersome and lead to missing out on certain combinations if not carefully reviewed.
2. **Explicit Representation of the Order of Decision Making**:
   - **Decision Trees**: Better suited for situations where multiple levels of decision-making are needed. They represent decisions hierarchically, making it easier to follow the order of decisions.
   - **Decision Tables**: Do not explicitly represent the order of decisions and are typically used for a single decision-making level.
3. **Representing Complex Decision Logic**:
   - **Decision Trees**: Become harder to understand and manage when there are many conditions and actions. The tree can become too large to fit on a single page, making it complex.
   - **Decision Tables**: Preferred for handling complex decision logic involving many decisions, as they are more compact and can represent large numbers of decisions systematically.

1.3 FORMAL SYSTEM SPECIFICATION:

These techniques are used to describe a system clearly and prove that it is correctly implemented, i.e., the system meets its given specification.

A system is said to be correctly implemented when it satisfies its specification, which can be described in two primary ways:

1. Property-Oriented Approach: Specifies the system by listing its desirable properties (e.g., the desired behaviors or outcomes).
2. Model-Oriented Approach: Specifies the system as an abstract model, often through mathematical structures, focusing on the system's design and functionality.

1.3.1 Formal Technique:

A formal technique is a mathematical method used in software and hardware systems to:

- Specify a system.
- Verify whether a specification is achievable.
- Ensure that an implementation meets its specification.
- Prove properties of the system without running it.

- **Syntactic and Semantic Domains :**
  1. Syntactic Domains:
     - An alphabet of symbols: The set of symbols used to construct formulas.
     - Formation rules: Rules that guide how to form well-structured formulas from these symbols.
     - Well-formed formulas: These are the correctly structured formulas used to specify a system. They play a key role in defining the system's behavior and properties.
  2. Semantic Domains:
     - Abstract Data Type Specification: Used to define algebras, theories, and programs.
     - Programming Languages: Used to specify the relationship between input values and their corresponding output values.
     - Concurrent and Distributed Systems used to define:
       - State sequences
       - Event sequences
       - State-transition sequences
       - Synchronization trees
       - Partial orders
       - State machines
- **Satisfaction relation:**
  - Satisfaction Relation: It's crucial to check if the semantic domain satisfies the given specifications. This is determined using a semantic abstraction function, which is a type of homomorphism.
- **Model vs Property-Oriented Methods:**

Model-Oriented Approach:

- The system's behavior is defined directly by constructing a mathematical model of the system, using structures like tuples, relations, sets, functions, etc.
- A typical model-oriented specification would define the basic and describe how these operations change the system state.

- Example: For a producer/consumer system, the operations produce (p) and consume (c) are defined, with transitions like S1 + p ⇒ S (where S is the state before and after the operation).
- Popular Techniques: Z, CSP, CCS.

Property-Oriented Approach:

- The system's behavior is defined indirectly by listing properties the system must satisfy, typically in the form of axioms.
- Example: For the producer/consumer system, properties might include conditions like "the consumer can start consuming only after the producer has produced an item."
- Techniques: Axiomatic specification and algebraic specification.
- Benefits: This is suitable for requirements specification, as they focus on what the system must not do (i.e., the allowed properties and constraints). They allow flexibility in implementation and make it easier to alter or augment specifications later.

Comparison:

- Property-Oriented:
  - Suitable for requirements specification.
  - Specifies the system by listing what it does not do, allowing multiple possible implementations.
  - More flexible and adaptable to changes, making it easier to alter specifications.
- Model-Oriented:
  - More suited for system design specification.
  - Defines the system by directly describing operations and their effects, resulting in a more rigid specification that requires substantial changes when modifications occur.

1.3.2 Operational Semantics: Operational Semantics refers to the way executions of a system are represented within formal methods.

Linear Semantics:

- Describes system behavior as a sequence of events or states.
- Concurrency is represented by non-deterministic interleavings of atomic actions (e.g., a||b is represented as a; b and b; a).
- It represents concurrency through sequential interleavings.
- Fairness and justice restrictions are often added to make the model more realistic by preventing unwanted interleavings.

Branching Semantics:

- The system's behavior is represented as a directed graph, where nodes represent possible states, and the descendants of each node represent states generated by atomic actions.
- This approach distinguishes branching points in computation but still represents concurrency through interleaving.

Maximally Parallel Semantics:

- Assumes that all concurrent actions enabled in a state occur simultaneously.
- This model is unnatural for real-world systems because it assumes that all computational resources are available, which may not be the case in actual execution.

Partial Order Semantics:

- Describes system behavior by a partial order among events or states.
- Events have a precedence order (some events must occur before others), while concurrent events are incomparable (they can happen independently).
- This approach offers a more realistic representation of concurrency, as it does not rely on interleaving and respects independent events.

Merits and limitations of formal methods:

Merits:

1. **Rigorous Specifications**: Formal methods help clarify system behavior, identify flaws early, and reduce costs in later development stages.
2. **Mathematical Soundness**: Formal methods are based on mathematics, allowing for reasoning about properties and ensuring that implementations meet specifications.
3. **Automation**: Verification techniques like tableau methods and automatic theorem proving help automate the verification process, making it faster and more reliable.
4. **Executable Specifications**: Formal specifications can be executed to get immediate feedback, useful for validating completeness and behavior.
5. **Systematic Development**: Formal methods provide a structured approach, which is particularly useful for large, complex systems.

Limitations:

1. **Difficult to Learn and Use**: Formal methods require specialized knowledge, making them challenging for developers to master.
2. **Incompleteness of Logic**: Some aspects of systems cannot be fully verified due to the limitations of formal logic.
3. **Complexity Issues**: As systems grow larger, formal methods become harder to manage and analyze.
4. **Complementary with Informal Methods**: Formal specifications should be used alongside informal descriptions to improve comprehensibility and address gaps in verification.

1.4 AXIOMATIC SPECIFICATION:

In axiomatic specification define the operations of a system in terms of pre-conditions and post-conditions. These conditions are expressed as axioms describing how the system should behave.

- **Pre-conditions**: These are the conditions that must be true before a function or operation can be successfully invoked. They specify the input requirements or constraints that must be satisfied.
- **Post-conditions**: These are the conditions that must be true after the function or operation has been executed. They define the constraints on the results produced by the function for it to be considered successful.

Key Points:

1. **Pre-conditions**: Describe the requirements on the input parameters that must be met before the operation can proceed.
2. **Post-conditions**: Define the constraints on the output or results after the operation is completed.
3. **Axioms**: Pre- and post-conditions are expressed as logical axioms, which provide a formal specification of the system's behavior.
4. **First-order Logic**: A formal language used to write the pre- and post-conditions in axiomatic specification.

How to develop an axiomatic specifications?

To develop axiomatic specifications for a function, follow these systematic steps:

1. Establish Input Range: Define the range of input values where the function is expected to behave correctly.
2. Define Input Constraints: Use a predicate to specify conditions that the input parameters must satisfy.
3. Define Output Constraints: Specify a predicate that defines the condition that must hold for the output if the function behaves correctly.
4. Specify Input Changes: Identify any changes made to the function's input parameters during execution (for pure functions, this step is unnecessary as inputs are not modified).
5. Combine Pre- and Post-Conditions: Finally, combine the pre-conditions and post-conditions to complete the axiomatic specification of the function.

Examples:

1. Example 1: A function f(x) that returns half the input value if $x \leq 100$, or double the value if $x > 100$:
    - Pre-condition: $x \in R$ (x is a real number).
    - Post-condition:
        - If $x \leq 100$, then $f(x) = x / 2$.
        - If $x > 100$, then $f(x) = 2 * x$.

1.5 ALGEBRAIC SPECIFICATION:

These technique defines an object class or type in terms of the relationships between the operations defined on that type. It was introduced by Guttag in the specification of abstract data types. This uses algebras to specify system behavior. Key elements of algebraic specification:

- Heterogeneous Algebra: Unlike traditional homogeneous algebras (single set with multiple operations), a heterogeneous algebra consists of several sets with different operations defined on them.
- Sorts: Each set in a heterogeneous algebra is called a "sort."
- Operations: These include both constructors (which create or modify data types) and inspectors (which evaluate data without modifying it).

Sections of an Algebraic Specification:

1. Types Section: Specifies the data types (or sorts) being used.
2. Exception Section: Lists possible exceptional conditions or errors during operations.

3. Syntax Section: Defines the signatures of the operations, which detail the sets of inputs and the resulting output.
4. Equations Section: Contains rewrite rules (equations) that define the meaning of operations based on the interface procedures. Equations may include conditional expressions.

Categories of Operators:

1. Basic Construction Operators: Essential for creating or modifying entities of a type (e.g., create and append).
2. Extra Construction Operators: Non-essential construction operators that aren't necessary to generate all values of a type (e.g., remove).
3. Basic Inspection Operators: Used to evaluate attributes without modifying the data type (e.g., eval, get).
4. Extra Inspection Operators: Additional inspection operators not needed to evaluate the basic attributes.

Steps for Developing Algebraic Specifications:

1. Identify Required Operations: Recognize which operations are needed for the data type.
2. Classify Operators: Classify them into basic and extra construction operators, and basic and extra inspection operators.
3. Write Axioms for Operators: Write axioms (equations) for composing the operations, ensuring they handle all meaningful compositions of operators.
4. Rewrite Rules: Use rewrite rules to simplify sequences of operations on the data type.

e.g : For a point data type:

- Operations: create, xcoord, ycoord, isequal.
- Types: point, boolean, integer.
- Syntax:
    - create: integer × integer → point
    - xcoord: point → integer
    - ycoord: point → integer
    - isequal: point × point → boolean
- Equations:
    - xcoord(create(x, y)) = x
    - ycoord(create(x, y)) = y
    - isequal(create(x1, y1), create(x2, y2)) = ((x1 = x2) ∧ (y1 = y2))

Properties of algebraic specifications:

1. Completeness:

- Definition: Completeness ensures that any arbitrary sequence of operations on the interface procedures can be reduced using the given equations. If the specification is incomplete, there might be a point where no equation can be applied to further reduce an expression, leading to an inability to simplify the system.
- Challenge: There is no simple method to guarantee that an algebraic specification is complete.

2. Finite Termination Property:

- Definition: This property ensures that the application of rewrite rules to any expression involving the interface procedures will eventually terminate. In other words, the sequence of reductions will not continue indefinitely.
- Challenge: In general, deciding whether an algebraic equation will terminate is undecidable. However, if each rewrite rule reduces the right-hand side to fewer terms than the left, the termination will be guaranteed.

3. Unique Termination Property:

- Definition: This property ensures that the application of rewrite rules in different orders will always result in the same final result. It ensures that no matter the sequence of choices during the reduction process, the final outcome will be consistent.
- Challenge: Verifying the unique termination property is a difficult problem and requires checking if all possible sequences of applying rewrite rules yield the same result.

Example of FIFO Queue Specification

- Operations: create, append, remove, first, isempty.
- Types: queue, boolean, element.
- Exception Conditions: underflow, novalue.
- Equations: Define the behavior of operations using rewrite rules.
    1. isempty(create()) = true
    2. isempty(append(q, e)) = false
    3. first(create()) = novalue
    4. first(append(q, e)) = if isempty(q) then e else first(q)
    5. remove(create()) = underflow
    6. remove(append(q, e)) = if isempty(q) then create() else append(remove(q), e)
- In this example:
    - Basic Constructors: create, append.
    - Extra Constructor: remove (because the queue can still be realized without remove).
    - Basic Inspectors: first, isempty.

The specification involves 6 axioms (2 basic constructors × 3 inspectors).

1.5.1 Auxiliary Functions :

one needs to introduce extra functions not part of the system to define the meaning of some interface procedures. These are called auxiliary functions.

e.g: Let us specify a bounded FIFO queue having a maximum size of MaxSize and supporting the operations create, append, remove, first, and isempty; where the operations have their usual meaning.

Types:

defines queue uses boolean, element, integer Exception: underflow, novalue, overflow

Syntax:

1. create : $\phi \rightarrow$ queue
2. append : queue × element → queue + {overf low}
3. size : queue → integer
4. remove : queue → queue + {underf low}
5. f irst : queue → element + {novalue}
6. isempty : queue → boolean

Equations:

1. f irst(create()) = novalue
2. first(append(q,e)) = if size(q)= MaxSize then overflow else if isempty(q) then e else first(q)
3. remove(create()) = underf low
4. remove(append(q, e)) = if isempty(q) then create() else . if size(q) = M axS ize then overf low else append(remove(q), e)
5. size(create()) = 0
6. size(append(q, e)) = if size(q) = M axS ize then overf low else size(q) + 1
7. isempty(q) = if (size(q) = 0) then true else false

1.5.2 Structured Specification:

The techniques that have successfully been used to reduce the effort in writing the specifications.

- *Incremental specification:* The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.
- *Specification instantiation*: This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Pros and Cons of algebraic specifications

Mathematical Basis:

- Algebraic specifications have a strong mathematical foundation and are based on heterogeneous algebras. This makes them precise and unambiguous.

Study of Operations:

- Algebraic specifications allow the automatic analysis of any arbitrary sequence of operations involving the interface procedures, which is useful for understanding the behavior of a system.

Shortcomings:

- Side Effects: Algebraic specifications cannot handle side effects. In programming, side effects are changes to the state or environment outside the function's scope (e.g., modifying global variables, changing memory, etc.). This limits their ability to model real-world programs accurately.
- Integration with Programming Languages: Since algebraic specifications can't deal with side effects, they are difficult to integrate with typical programming languages that rely on side effects (e.g., updating variables, printing output, etc.).

Complexity in Understanding:

- Algebraic specifications, while precise, can be hard to understand. Their abstract nature and reliance on mathematical concepts may not be easily accessible to all developers, especially when applied to large or complex systems.

1.6 EXECUTABLE SPECIFICATION AND 4th Generation Languages:

- Executable Specifications: Allow direct execution of system specifications, reducing the need for separate implementation.
- 4GLs: Used for creating executable specifications, especially effective in data processing applications due to high levels of abstraction and software reuse.

- Efficiency of 4GLs: While they simplify development, 4GLs are slower and less efficient than traditional programming languages (3GLs).
- Performance Comparison: Rewriting 4GL programs in 3GLs can lead to up to 50% lower memory usage and a tenfold reduction in execution time.
- Software Reuse: The power of 4GLs comes from the reuse of common abstractions across similar data processing applications.

SOFTWARE PROJECT MANAGEMENT

INTRODUCTION:

- Software project management is important for the success of a project. Many projects fail because of poor management, not a lack of skills or resources.
- The goal is to help a team work effectively to complete the project. The project manager leads the team, handling administrative tasks, while a technical leader may handle technical decisions in larger projects.

1. SOFTWARE PROJECT MANAGEMENT COMPLEXITIES:

Managing software projects is more complex than managing many other types of projects. *Several factors contribute to this complexity*:

1. **Invisibility**: Software cannot be physically seen until it's completed and operational, making it difficult to track progress. Unlike a building project where progress can be visually assessed, software progress is harder to measure and usually relies on milestones or document reviews.
2. **Changeability**: Software is easier to change compared to hardware, so changes to software requirements are common, especially later in the project. These changes can come from shifts in business needs, changes in hardware or underlying systems, or simply from client changes. This frequent need for changes adds to the complexity of managing software projects.
3. **Complexity**: Software often has millions of interacting parts, which creates many potential risks. The intricate relationships between functions, data, and system operations make it much harder to manage compared to simpler projects.
4. **Uniqueness**: Every software project is unique, presenting different challenges and situations. This is unlike more predictable projects like manufacturing, where the processes are more standardized. A software project manager often faces new, unanticipated problems.
5. **Exactness of the Solution**: Unlike mechanical components, which can function with slight tolerances, software requires exact precision. Even small errors can cause the system to fail. This need for strict conformity adds risk and complexity, especially when trying to reuse parts of existing software.
6. **Team-oriented, Intellect-intensive Work**: Software development is a team effort that requires high intellectual engagement. Unlike labor-intensive projects like construction, software projects involve a high level of collaboration, with team members frequently interacting and reviewing each other's work, further complicating management.

2. RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER:

- **The job of a software project manager** involves a wide range of responsibilities, from behind-the-scenes tasks like building team morale to visible activities like client presentations.
- Their duties generally include project proposal writing, cost estimation, scheduling, staffing, software process tailoring, monitoring and control, risk management, report writing, and client interaction.

These responsibilities can be divided into two main categories:

1. **Project Planning**: This begins after the feasibility study and before requirements analysis. It involves estimating project characteristics and planning activities based on those estimates. The project plan is adjusted as more data becomes available throughout the project.
2. **Project Monitoring and Control**: This happens once the development begins. The main focus here is ensuring the project stays on track and adjusting plans as needed to address unforeseen situations.

The Skills or knowledge of project management techniques, a successful project manager requires good judgment, decision-making skills, and experience. Communication skills, team building, and the ability to track progress and interact with customers are also crucial. Three key skills for success are:

- Knowledge of project management techniques
- Decision-making capabilities
- Experience managing similar projects

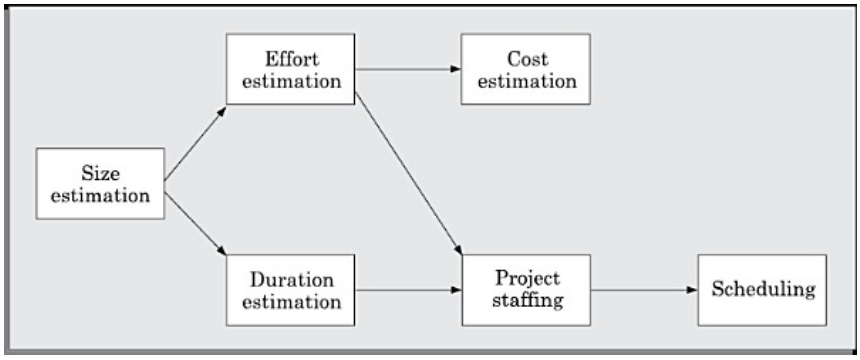
2.1 PROJECT PLANNING

- Once a software project is deemed feasible, the project manager begins the **project planning** phase, which must be completed before any development work starts.
- Effective planning is crucial, as unrealistic estimates for time and resources can lead to schedule delays, customer dissatisfaction, low team morale, and even project failure.
- Therefore, project planning requires careful attention and is heavily influenced by the project manager's knowledge of estimation techniques and past experience.

The main activities involved in project planning include:

1. **Estimation**:
    - **Cost**: Estimating how much it will cost to develop the software.
    - **Duration**: Estimating how long the development will take.
    - **Effort**: Estimating how much effort (in terms of person-hours) is needed to complete the project.

   Accurate estimations are vital, as they influence later activities like scheduling and staffing.
2. **Scheduling**: After estimating the project parameters, schedules for resources (e.g., manpower) are created.
3. **Staffing**: Plans for organizing and staffing the project are developed.
4. **Risk Management**: Identifying potential risks, analyzing them, and creating a plan for mitigating them.
5. **Miscellaneous Plans**: This includes creating other plans, like quality assurance and configuration management plans.



2.2 SLIDING WINDOW PLANNING

- It is usually very difficult to make accurate plans for large projects at project initiation.
- At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. The project parameters are re-estimated periodically. The project manager can plan the subsequent activities more accurately and with increasing levels of confidence.
- Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as sliding window planning.

2.3 Organisation of the software project management plan (SPMP) document

1. **Introduction**
   - (a) Objectives
   - (b) Major Functions
   - (c) Performance Issues
   - (d) Management and Technical Constraints
2. **Project estimates**
   - (a) Historical Data Used
   - (b) Estimation Techniques Used
   - (c) Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   - (a) Work Breakdown Structure
   - (b) Task Network Representation
   - (c) Gantt Chart Representation
   - (d) PERT Chart Representation
4. **Project resources**
   - (a) People
   - (b) Hardware and Software
   - (c) Special Resources
5. **Staff organisation**
   - (a) Team Structure
   - (b) Management Reporting
6. **Risk management plan**
   - (a) Risk Analysis
   - (b) Risk Identification
   - (c) Risk Estimation

(d) Risk Abatement Procedures

7. **Project tracking and control plan**
   (a) Metrics to be tracked
   (b) Tracking plan
   (c) Control plan
8. **Miscellaneous plans**
   (a) Process Tailoring
   (b) Quality Assurance Plan
   (c) Configuration Management Plan
   (d) Validation and Verification
   (e) System Testing Plan
   (f) Delivery, Installation, and Maintenance Plan

3. METRICS FOR PROJECT SIZE ESTIMATION:

- Accurate estimation of project size is crucial for estimating other project parameters such as effort, completion time, and cost. But what exactly does "project size" mean? It is not simply the number of bytes of source code or the size of the executable code.
- Instead, **project size** refers to the **complexity of the problem** in terms of the effort and time needed to develop the product.

There are two common metrics used to measure project size:

1. **Lines of Code (LOC)**: This metric measures the size based on the number of lines in the source code.
2. **Function Points (FP)**: This metric measures size based on the functionality provided to the user, regardless of the technology or programming language used.

3.1 LINES OF CODE (LOC):

- The **Lines of Code (LOC)** metric is one of the simplest and most popular ways to measure the size of a software project.
- It counts the number of source code instructions, excluding comments and header lines.
- While determining LOC at the end of a project is easy, **estimating LOC at the start of a project** is very challenging.

However, the LOC metric has several important **shortcomings**:

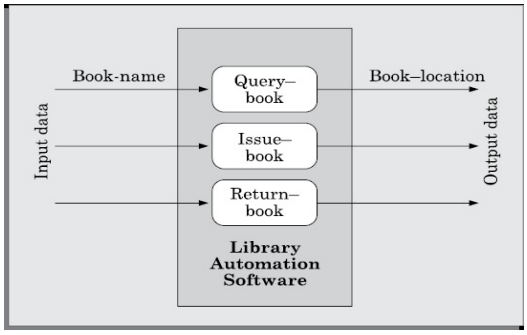
1. **Focuses on Coding Activity Alone**: LOC only measures coding effort and ignores other life cycle activities such as design, testing, and specification.
2. **Variability in Coding Style**: LOC counts can vary widely depending on individual coding styles. Different programmers might write the same logic using different approaches, leading to different LOC counts.
3. **Poor Correlation with Code Quality and Efficiency**: A larger LOC count doesn't necessarily indicate better code quality or higher efficiency. **Disadvantages for Reuse and High-Level Programming**: The LOC metric penalizes the use of higher-level programming languages and code reuse. If a programmer uses library routines or reusable components, the LOC count will be lower, which may inaccurately suggest that less effort was put into the project. This could discourage developers from reusing code, even though it's a more efficient approach.
4. **Does Not Address Logical Complexity**: LOC measures **lexical complexity**, which only looks at the number of lines, not the **logical or structural complexity** of the program.

5. **Difficult to Estimate Early On**: Accurately predicting LOC at the beginning of a project is extremely difficult. LOC can only be accurately determined after the code is written, making it less useful during the **project planning phase**, where estimates are needed before development begins.

3.2 FUNCTION POINT (FP) METRIC:

The **Function Point (FP)** metric, introduced by Albrecht in 1983, is a way to measure the size of a software project based on its features and functions, rather than just counting lines of code (LOC). Here's why it's useful:

1. **Early Estimation**: Unlike LOC, which you can only count after the software is built, you can calculate function points **early in the project**, just from the project's requirements or problem description.
2. **Focus on Features**: Function points measure the size of a software product based on how many **functions or features** it has (like input forms, reports, or queries). More functions mean more effort to develop.
3. **Accounts for Complexity**: Some features are more complex than others. For example, a simple "help" feature is easier to build than a "bank transaction" feature. The function point method considers **how many data items** a function handles and how many **files** it needs to access. More data or files means more complexity.



Function point (FP) metric computation:

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

- **Step 1:** Compute the unadjusted function point (UFP) using a heuristic expression.
- **Step 2:** Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.
- **Step 3:** Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

Step 1. The unadjusted function points (UFP) is **Computed** as the weighted sum of 5 characteristics of a product as shown:

- **UFP =** (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Step 2. Refine parameters

- UFP computed at the end of step 1 is a gross indicator of the problem size.This UFP needs to be refined. This is possible, since each parameter (input, output, etc.) has been

implicitly assumed to be of average complexity. E.g:, some input values may be extremely complex, some very simple, etc..

Table 3.1: Refinement of Function Point Entities

Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

- The overall project size are considered to refine the UFP computed in step2. 14 parameters that can influence the development effort from 0 (no influence) to 6 (strong influence). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP i.e. **FP=UFP*TCF.**

4. PROJECT ESTIMATION TECHNIQUES:

Estimating key project parameters like size, effort, duration, and cost is crucial for planning. Accurate estimates help set the right price, allocate resources, and schedule tasks. There are three main types of estimation techniques:

1. **Empirical Estimation Techniques**: Based on past data and experience.
2. **Heuristic Techniques**: Based on rules of thumb or general guidelines.
3. **Analytical Estimation Techniques**: Use mathematical models and formulas.

1.Empirical Estimation Techniques

- Empirical estimation techniques are essentially based on making an educated guess of the project parameters.
- While using this technique, prior experience with development of similar products is helpful.

2.Heuristic Techniques

- Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions.
- Estimated Parameter = $c_1 * e^{d_1}$
- Estimated Resource = $c_1 * e^{d_1} + c_2 * e^{d_2} + \ldots\ldots$
- c_1, c_2, d_1, d_2 are constants, e_1, e_2 are basic (Independent) characteristics of s/w.
- Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project.

5. EMPIRICAL ESTIMATION TECHNIQUES:

- **Empirical Estimation Techniques** are commonly used for project size estimation, but they still involve some guesswork.
- Two popular methods are
  - **Expert Judgement** and
  - **Delphi Estimation**.

1. **Expert Judgement**:
   - An expert provides an educated guess about the project size after analyzing it.
   - The expert estimates the cost of different components and combines them to create an overall estimate.
   - Shortcomings: Human errors, individual biases, lack of relevant experience, and oversight can affect the accuracy.
2. **Delphi Estimation**:
   - A group of experts anonymously provides their estimates after reviewing the project requirements.
   - The estimates are summarized by a coordinator and shared with the experts for re-estimation in multiple rounds.
   - This process reduces bias and the influence of dominant voices but requires more time and effort than expert judgement.

6. COCOMO—A HEURISTIC ESTIMATION TECHNIQUE:

- **Constructive Cost Estimation Model (COCOMO)**, proposed by Boehm in 1981, is a widely used method for software project estimation. It follows a three-stage process for estimating project size and effort. These stages allow for progressively more accurate estimates:
- **Basic COCOMO**: Provides an initial, rough estimate based on the size of the software (measured in lines of code).

Three basic classes of software development projects:

In Boehm's **COCOMO** model, projects are classified based on both the characteristics of the product and the development team. The classification considers three types of software development:

1. **Organic**: This is for well-understood applications developed by small, experienced teams. The team members are familiar with similar projects.
2. **Semidetached**: This involves a mix of experienced and inexperienced team members, where the team may have some experience with related systems but is unfamiliar with some aspects of the current project.
3. **Embedded**: This type deals with software tightly coupled to hardware or subject to strict operational regulations. The team may have limited experience in related systems.

The **complexity of the software** is categorized into three levels:

- **Application software** (least complex)
- **Utility software** (moderately complex)
- **System software** (most complex)

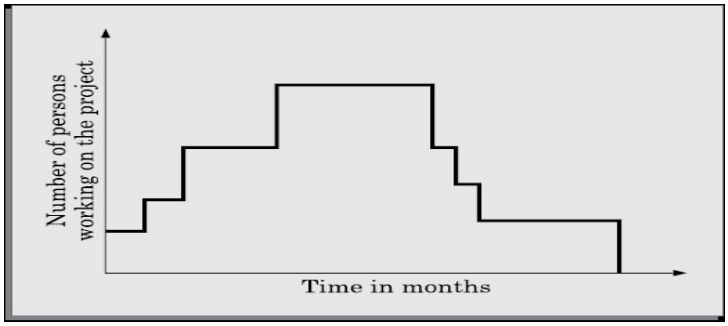


Figure 3.3: Person-month curve.

General form of the COCOMO expressions:

The basic COCOMO model is a single variable heuristic model that gives an approximate estimate of the project parameters.

- Effort = a1 × (KLOC)a2 PM
- Tdev = b1 × (Effort)b2 months

Where,

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- a1, a2, b1, b2 are constants for each category of software product.
- Tdev is the estimated time to develop the software.

ESTIMATION OF DEVELOPMENT EFFORT (IN PERSON-MONTHS):

- **Organic**: Effort=2.4×(KLOC)1.05 PM
  {Effort} = 2.4 *{KLOC})^{1.05}{PM}
  Effort=2.4×(KLOC)1.05PM
- **Semi-detached**:
  Effort=3.0×(KLOC)1.12 PM
  {Effort} = 3.0 {KLOC})^{1.12}PM
  Effort=3.0×(KLOC)1.12PM
- **Embedded**:
  Effort=3.6×(KLOC)1.20 PM
  {Effort} = 3.6 {KLOC})^{1.20}{PM}
  Effort=3.6×(KLOC)1.20PM

Estimation of Development Time (in Months):

- **Organic**: Tdev=2.5×(Effort)0.38 Months
  T{dev}} = 2.5{Effort})^{0.38}{Months}
  Tdev=2.5×(Effort)0.38Months
- **Semi-detached**:
  Tdev=2.5×(Effort)0.35 Months
  T{dev}} = 2.5 {Effort})^{0.35}{Months}
  Tdev=2.5×(Effort)0.35Months
- **Embedded**:
  Tdev=2.5×(Effort)0.32 Months
  Tdev = 2.5 Effort^{0.32} {Months}
  Tdev=2.5×(Effort)0.32Months

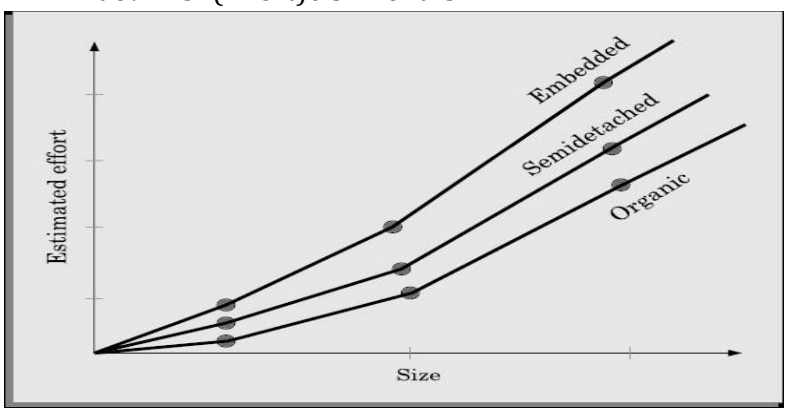
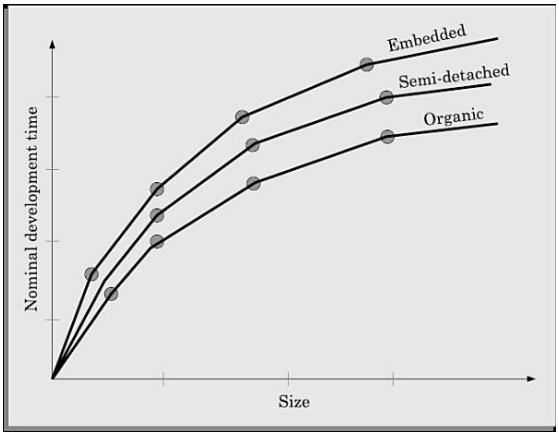


Figure 3.4: Effort versus product size.

1. **INTERMEDIATE COCOMO**:
   - Refines the estimate by considering factors like project complexity, team capability, and other project characteristics.

Observations from the development time—size plot

- The development time of a software project increases at a **slower rate** as the project size grows.
- This might seem surprising because we would expect the time to increase more.
- However, the reason is that **COCOMO** assumes the project is developed by a **team** of developers, so tasks can be done at the same time, which helps keep the development time increase moderate.



Cost Estimation:

- **To estimate the project cost, you multiply the** estimated effort **(in person-months) by the** manpower cost per month**. However, this only covers** manpower costs.

Implications of Effort and Duration Estimate:

- If you try to **complete the project faster** than the estimated time, the **cost will increase significantly**.
- If you take **longer** than the estimated duration, the **cost won't decrease much**.
- If you reduce the duration too much, some team members may **end up being idle**, which increases costs.
- An **optimal team size** ensures everyone is busy but not overwhelmed, and the project is completed in the shortest time without wasting resources.

Staff-Size Estimation:

- You can't simply divide effort by duration to determine the staffing level. This method could lead to delays and cost overruns.
- The staffing model needs to consider the optimal team size for the project's success, and this is more complex than just dividing effort by time.

Example: For an organic software **product with an estimated size of** 32,000 lines of code**:**

1. **Effort**:
   Using the COCOMO formula: Effort = 2.4 × (32)^1.05 = 91 person-months (PM)
2. **Nominal Development Time**:

Using the COCOMO formula: Development Time = 2.5 × (91)^0.38 = 14 months

3. **Cost**:
   If the average salary of a software developer is **Rs. 15,000 per month**, then: Cost = 91 PM × Rs. 15,000 = Rs. 1,465,000

2. COMPLETE COCOMO:

- Offers the most detailed estimate by incorporating all project attributes, including environmental and organizational factors, to generate a highly refined cost and schedule estimate.
- The **basic** and **intermediate COCOMO models** treat a software product as a single entity, which doesn't work well for large projects with different parts (sub-systems) that may have different complexities. For example, one part of the system might be simple, another might be complex, and some parts might be linked to hardware or have strict requirements.

Complete COCOMO Model:

- The **complete COCOMO model** solves this by estimating the cost and time for each part of the system separately. The estimates for all parts are then added together to get the total estimate. This approach makes the estimate more accurate.

Example: Distributed MIS System

For a distributed management information system **(MIS)**, the parts might be:

- **Database**: Estimated using the semi-detached model.
- **User Interface (GUI)**: Estimated using the organic model.
- **Communication**: Estimated using the embedded model.

Each part is estimated separately, and the total cost is the sum of those estimates.

Improving Accuracy:

- To make the estimates more accurate, companies can adjust the model using data from previous projects. This helps to fine-tune the estimates for future projects.

7. HALSTEAD'S SOFTWARE SCIENCE—AN ANALYTICAL TECHNIQUE:

7.1 Halstead's Software Science is a technique used to measure the size, development effort, and cost of software products. It relies on counting specific elements in the program's source code, called **operators** and **operands**.

Key Terms:

- **Operators**: These are elements that perform actions or operations in the program (e.g., arithmetic operators like +, logical operators like &&, or control statements like if, while).
- **Operands**: These are variables or values that operators act upon (e.g., numbers, variables, or function arguments).

Important Variables:

- **h1**: Number of unique operators in the program.
- **h2**: Number of unique operands in the program.
- **N1**: Total number of operators used in the program.
- **N2**: Total number of operands used in the program.

7.2 Length of a Program (N):

- The length of a program is defined as the total number of **operators** and **operands** used in the code.
- Formula: **N = N1 + N2**

  Where: **N1** = Total number of operators.
  N2 = Total number of operands.

7.3 Program Vocabulary (h):

- The vocabulary of a program is the number of **unique** operators and operands.
- Formula: **h = h1 + h2**

  Where: **h1** = Number of unique operators.
  h2 = Number of unique operands.

7.4 Program Volume (V):

- Volume quantifies the size of the program and compensates for the effect of the programming language used. It's the minimum number of bits needed to represent the program.
- Formula: **V = N log2 h**

  Where: **N** = Program length (total operators + operands).
  h = Program vocabulary (unique operators + operands).

7.5 Potential Minimum Volume (V):*

- This is the volume of the most concise program that can solve a problem. For a problem with n data items, it's the simplest program that uses minimal operators and operands.
- Formula: ***V = (2 + h2) log2 (2 + h2)****

  Where: **h2** = Number of operands.

7.6 Program Level (L):

- The program level measures the level of abstraction of the programming language. Higher-level languages are easier to write programs in because they require fewer mental efforts.
- Formula: ***L = V / V****
- A high-level language means a lower effort in developing a program.

7.7 Effort (E):

- The effort required to develop a program depends on its volume and level.
- Formula: **E = V / L** Or equivalently: $\mathbf{E = V^2 / V}$*
- This is proportional to the square of the volume and represents the mental effort needed to write and understand the program.

7.8 Programmer's Time (T):

- The time required for the programmer to develop the program is based on the effort and the programmer's speed of mental processing (*S*).
- Formula: **T = E / S**
- Where **S** is typically 18 for programming tasks.

7.9 Length Estimation (before coding starts):

- Halstead suggests that you can estimate the length of a program even before starting by using the number of unique operators and operands.
- This approach assumes that identical parts of a program will be refactored into functions or procedures, leading to a more efficient program structure.

7.2 STAFFING LEVEL ESTIMATION:

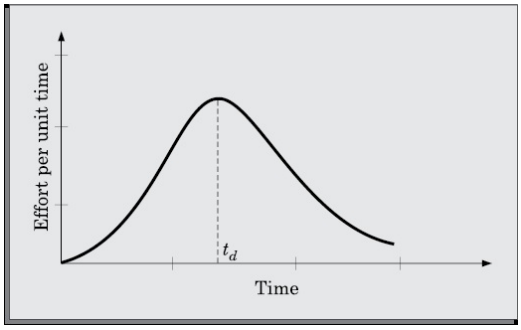
- Once the **effort** required to complete a software project is estimated, the **staffing requirement** for the project can be determined.
- This is crucial because the right staffing pattern affects the project's overall success and efficiency.

Norden's Work

- Norden studied staffing patterns for R&D projects and found they are different from those in sales or manufacturing.
- In sales, the number of staff stays roughly the same over time, like in a supermarket where it depends on the number of sales counters.

changing levels of staff:

- **Early stages**: The need for staff is low as planning and research are done.
- **Middle stages**: The need for staff increases and peaks as the project moves forward.
- **Final stages**: The number of staff needed decreases as the project is completed.



.

8. RISK MANAGEMENT:

- Every project faces potential risks, and without proper risk management, even well-planned projects can fail.
- A **risk** is an event or situation that could negatively affect the project in the future.
- Risk management focuses on preventing risks from happening and minimizing their impact if they do.

Risk management involves three main activities:

1. **Risk Identification** – Recognizing potential risks.
2. **Risk Assessment** – Evaluating the likelihood and impact of each risk.
3. **Risk Mitigation** – Creating strategies to reduce or manage the risks.

1.Risk Identification:

- The project manager must identify risks as early as possible.
- Once a risk is identified, effective risk management plans should be created to minimize the potential impact.

1. **Project Risks**: These involve issues related to budget, schedule, personnel, resources, and customer-related problems. A major project risk is schedule slippage, as software is intangible.
2. **Technical Risks**: These risks are related to design, implementation, interfacing, testing, and maintenance.

3. **Business Risks**: These involve risks like developing a product no one wants or losing financial support for the project.

2.Risk Assessment:

- The objective of risk assessment is to rank the risks in terms of their damage causing potential.
- The likelihood of a risk becoming real (r).
- The consequence of the problems associated with that risk (s).

3.Risk Mitigation:

- After assessing all the identified risks, the project manager prioritizes the most likely and damaging risks and develops plans to handle them. Different types of risks require different strategies.

1. **Avoid the Risk:**
   - **Process-related risks**: These arise from tight schedules, budgets, or resources.
   - **Product-related risks**: These are due to challenging product features or strict quality/reliability goals.
   - **Technology-related risks**: These stem from committing to use certain technologies.
   - #Risk leverage= (risk expose before reduction- risk expose after reduction)/cost of reduction.
2. **Transfer the risk:** This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.
3. **Risk reduction:** This involves planning ways to contain the damage due to a risk. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.