SOFTWARE ENGINEERING

Module – 3 (REQUIREMENT ANALYSIS AND SPECIFICATION)

CONTENTS

Importance of Requirements Analysis and Specification

- In **plan-driven life cycle models**, it's **essential** to **understand and document exact customer requirements** *before* starting development.
- In the past, many projects failed because developers started coding **without confirming** what the customer really wanted.
- This leads to:
  - Frequent changes in later phases
  - Increased **development cost**
  - **Customer dissatisfaction**
  - Possible **legal disputes**

❖✔️ Therefore, experienced developers treat **requirements analysis and specification** as a **critical phase** in software development.

Role of a Good Requirements Document

- A good **requirements document**:
  - Clarifies what features the software should have
  - Serves as a **foundation** for later phases like design, coding, testing, etc.
- In **outsourced projects**, it acts as a **contract** between the customer and developer.
- Even in **in-house product development**, the marketing team may act as the customer, so precise documentation is still important.
- In **small service-based projects**, **agile methods** may allow **incremental development of requirements**.

When Does Requirements Analysis Start?

- Begins **after the feasibility study**, when the project is confirmed to be **technically and financially viable**.

Who Performs This Phase?

- Conducted by **experienced team members**, often called **System Analysts**.
- Analysts may spend time at the **customer site**.
- Responsibilities:
  - Collect data about what the software must do
  - Understand the exact user requirements
  - Identify and remove inconsistencies, ambiguities, and incompleteness
  - Create the **Software Requirements Specification (SRS)** document

Validation of the SRS Document

- **Internal Review** by the project team to ensure:
  - Accuracy
  - Completeness
  - Consistency
  - Unambiguity
- **Customer Review**:
  - Customer reviews and approves the SRS.
  - Once approved, the SRS becomes the **basis for all future development**.
  - Also acts as a **legal agreement** between customer and developer.

Main Activities in Requirements Analysis & Specification

- **Requirements Gathering and Analysis**:
  - Collecting and analyzing customer needs.
- **Requirements Specification**:
  - Documenting the analyzed requirements in the form of the SRS.

- Requirements analysis and specification is vital to **project success**.
- It helps avoid **costly changes** and **disputes** later.
- The **SRS document** is the key outcome, serving as both a **blueprint** and a **contract**.
- Conducted by **system analysts**, reviewed by both developers and customers.
- Forms the **foundation** for all subsequent development activities.

3.1 Requirements Gathering and Analysis

Goal of Requirements Analysis and Specification Phase

- To **clearly understand customer requirements** and organize them into a formal document called the **Software Requirements Specification (SRS)**.
- The **SRS** is the final and crucial outcome of this phase.

Realistic Expectation

- Customers **do not provide all requirements** in a single document.
- Requirements are **scattered**, vague, and come from **multiple sources** (different stakeholders).
- These must be **gathered and then analyzed** to remove issues like **ambiguities, inconsistencies, and incompleteness**.

Two Main Tasks

- **Requirements Gathering (also called Elicitation)**
- **Requirements Analysis**

3.1.1 Requirements Gathering

- **Objective:** **Collect all software requirements from stakeholders.**
- **Challenges:**
  - Multiple stakeholders with partial views
  - Scattered and unstructured information
  - No existing system in case of new software
- **If a manual system exists (e.g., office record-keeping):**
  - Easier to gather input/output forms and process steps.
- **Typical Activities:**
  - **Study existing documents** like the Statement of Purpose (SoP)
  - **Visit customer site**
  - **Interview** users and representatives
  - Use methods like:
    - **Questionnaires**
    - **Task analysis**
    - **Scenario analysis**
    - **Form analysis**

Important Techniques Used in Requirements Gathering

- **Studying Existing Documentation**
  - Read SoP documents that outline context, goals, stakeholders, and expected features.
- **Interview**
  - Identify different user categories and their needs.
  - Use **Delphi technique** to circulate a draft, gather feedback, and iterate until agreement.
- **Task Analysis**
  - Break down software into user-performed **tasks**.
  - Define steps for each task (e.g., issuing a book in a library system).
- **Scenario Analysis**
  - Understand how tasks behave under different conditions (scenarios).
  - E.g., Book issue successful, book reserved, or member exceeded issue limit.
- **Form Analysis**
  - Analyze existing manual **input forms and output reports**.
  - Understand how outputs are generated from inputs

✅ Case Study: Office Automation in CSE Department

- A team of students automated grading, leave, and store tasks for a department.
- Analyst interviewed staff, collected forms, understood tasks, scenarios, and outputs.
- Input/output data formats and office procedures were accurately captured.

3.1.2 Requirements Analysis

- **Objective:**
  - Analyze the gathered data to:
  - Fully understand customer requirements
  - Eliminate **ambiguities, inconsistencies, and incompleteness**

➤ Pre-analysis Questions:

- What is the problem?
- Why is it important?
- What are the inputs/outputs?
- What processes are needed?
- What are the potential difficulties?
- What external interfaces exist?

Common Problems in Requirements

1. Anomaly (Ambiguity)

- Multiple meanings due to vague terms
- **Example**: “Switch off heater when temperature is high” (what is "high"?)

2. Inconsistency

- Conflicting requirements from different stakeholders
- **Example**: One says “switch off furnace at 500°C”; another says “keep it on and trigger water spray”.

3. Incompleteness

- Missing necessary features or data
- **Example**: Requirement to email parents about grades, but no provision to input parent email addresses.

➤ Can all problems be detected?

- Many issues are caught during careful review.
- Some are **subtle** and hard to detect.
- Use of **formal methods** can help systematically find these issues.
- Formal specification is especially used for **safety-critical systems**.

- Requirements must be **gathered methodically** from multiple sources.
- They are then **analyzed** to ensure clarity, consistency, and completeness.
- The outcome is a validated **SRS document** that forms the **foundation** for the rest of development.

3.2 Software Requirements Specification (SRS)

Once all requirements are gathered and cleaned (i.e., **no inconsistencies, ambiguities, or incompleteness**), the analyst organizes them into a **structured, informal document** called the **Software Requirements Specification (SRS)**.

Importance of the SRS

- One of the **most important and hardest** documents to write in the software development life cycle.
- Must satisfy the needs of **multiple users with different perspectives**.

3.2.1 Users of SRS Document

- Various stakeholders use the SRS for different purposes:

User	Purpose
Users, Customers, Marketers	Verify the system meets their needs.
Software Developers	Develop software based on clearly defined requirements.
Test Engineers	Design test cases and test plans from input/output details.
User Documentation Writers	Understand features to write user manuals.
Project Managers	Estimate cost, effort, and plan the schedule.
Maintenance Engineers	Understand system functions for debugging, updating, or enhancements.

📌 **The SRS acts like a contract between the developers and the customer, and can even be used in legal disputes.**

3.2.2 Why Create an SRS Document?

- Benefits of having a well-prepared SRS:
- **Contractual agreement**: Sets clear expectations between developers and customers.
- **Reduces rework**: Early clarity avoids redesign, recoding, and retesting later.
- **Cost and schedule estimation**: Helps project managers estimate effort, budget, and delivery time.
- **Baseline for validation & verification**: Used to check if the final product meets the documented requirements.
- **Supports future extensions**: Serves as a reference for planning enhancements.

3.2.3 Characteristics of a Good SRS Document

- As per **IEEE 830 Standard**, a good SRS must be:

- Figure 3.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software

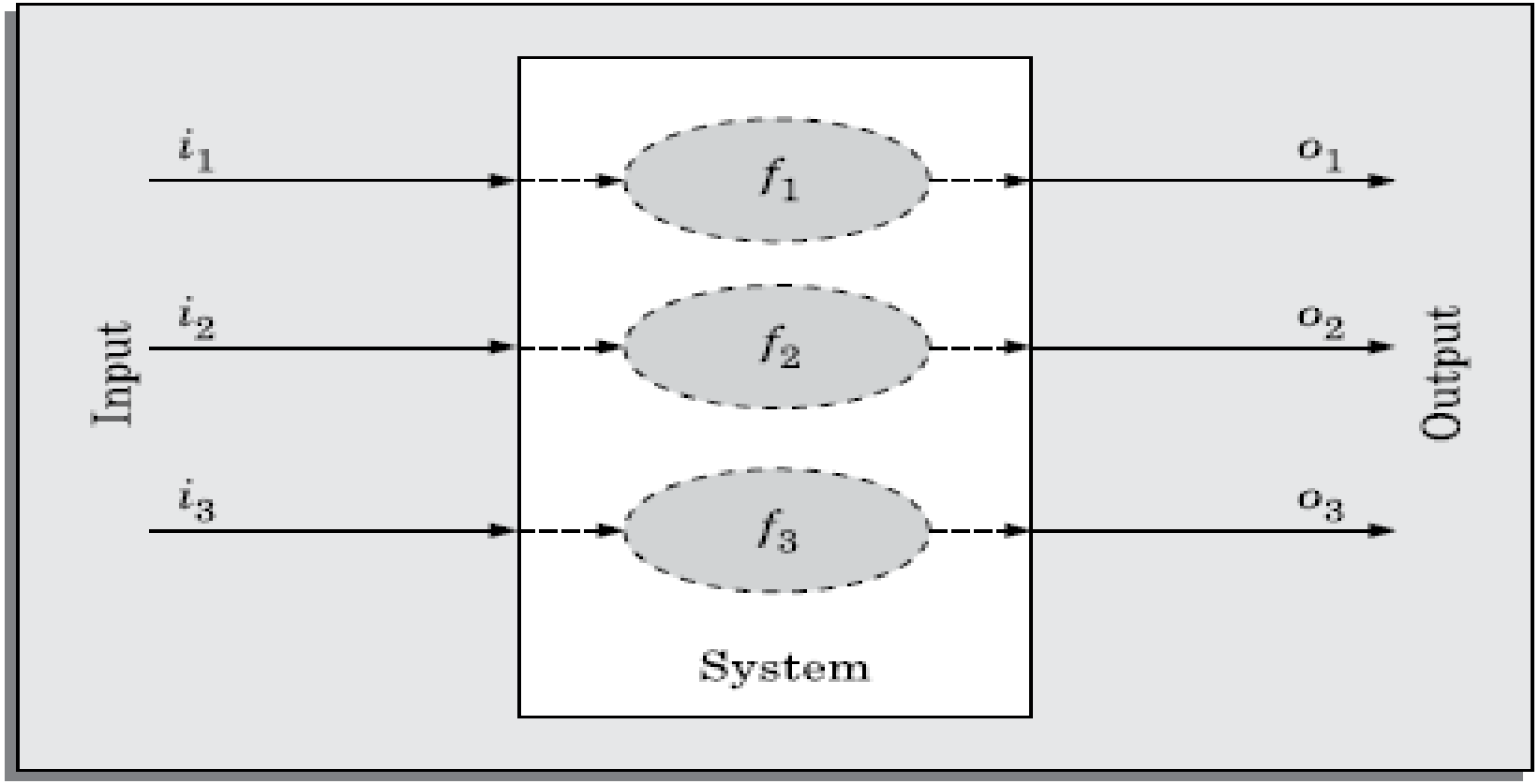


FIGURE 3.1 The black-box view of a system as performing a set of functions..

Characteristic	Description
Concise	Clear, brief, complete, and consistent; avoids unnecessary detail.
Implementation-independent	Focuses on what the system does, not how; avoids design decisions.
Traceable	Each requirement can be traced to its design, code, and test case, and vice versa.
Modifiable	Well-structured to accommodate changes without confusion.
Identifies responses to errors	Specifies how the system reacts to undesired or exceptional events.
Verifiable	Requirements must be testable. Vague terms like “user-friendly” are avoided.

✅ **The SRS provides a black-box view: it specifies the external behavior without discussing internal logic.**

3.2.4 Attributes of Bad SRS Documents

- Common problems found in poorly written SRS documents:

Problem	Explanation
Over-specification	Includes design-level decisions (e.g., internal storage mechanisms).
Forward References	Refers to content explained much later, reducing readability.
Wishful Thinking	Contains unrealistic or vague desires that are hard to implement.
Noise	Irrelevant details (e.g., staff working hours) that clutter the document.

Other common issues include:

- **Ambiguity**
- **Incompleteness**
- **Contradictions**

✅ **These issues must be avoided or used as a checklist to review and improve the quality of the SRS.**

3.2.5 Important Categories of Customer Requirements (IEEE 830)

A well-structured SRS document **categorizes requirements** into key sections:

1. Functional Requirements

- Describe **what the system should do**.
- Each function (fi) is like a mathematical transformation:

  f: I → O → input data (Ii) transforms into output data (oi).

- These include:
  - **User inputs**
  - **Processing logic**
  - **Expected outputs**

- Each function must clearly mention its inputs and corresponding outputs.

🧠 **Note: Functional requirements form the core of the SRS and are essential for system design and testing.**

2. Non-Functional Requirements

These are **mandatory quality attributes** and constraints **not related to specific functions**, but affect overall system behavior and quality.

They include:

- **a) Design and Implementation Constraints**

Restrictions on how the system must be built.

  - Examples:
    - Use Oracle DBMS
    - Follow specific security protocols
    - Use a particular programming language or platform

- **b) External Interfaces**
  - Requirements related to interaction with:
    - Hardware
    - Software
    - Communication channels
    - Users (GUI layout, screen design standards, error messages, etc.)

c) Other Non-Functional Requirements

- Performance (e.g., 100 transactions/sec)
- Reliability
- Security
- Usability
- Maintainability
- Portability

❖ 💡 IEEE 830 recommends documenting:

- **External interfaces**
- **Design & implementation constraints**

in separate sections, followed by **other non-functional requirements**.

3. Goals of Implementation

- These are **optional, non-binding suggestions** from the customer.
- Developers may consider them during design.
- Examples:
  - Easy support for future enhancements
  - Reusability
  - Flexible for adding new device support

- 📌 **Difference**:
  - **Non-functional requirement**: Must be tested for compliance.
  - **Goal**: Desirable, not tested during system acceptance.

4. **Classification Criteria**

Type	Definition
Functional Requirement	Input → Output transformation (functions of the system)
Non-Functional Requirement	Testable characteristics not expressible as functions (e.g., performance, security)
Goal of Implementation	Non-testable customer suggestions (e.g., design for future ease of change)

3.2.6 Functional Requirements

📌 **What Are They?**

- Functional requirements describe **high-level functions** the user **explicitly invokes** to get useful work done.
- Each high-level function:
  - Accepts input
  - Processes it
  - Produces output

Example:

In a **Library System**, a function like search-book:

- Takes keywords as input
- Searches the database
- Displays matched books

Important Clarification

- **Not every activity is a high-level function.**
- Example: Printing ATM receipts is **part of** withdraw money, not a separate high-level function.

🔄 High-Level Function = Multiple Sub-Requirements

- A function often involves **step-by-step interactions** between user and system.
- Each interaction step can be a **sub-requirement**.

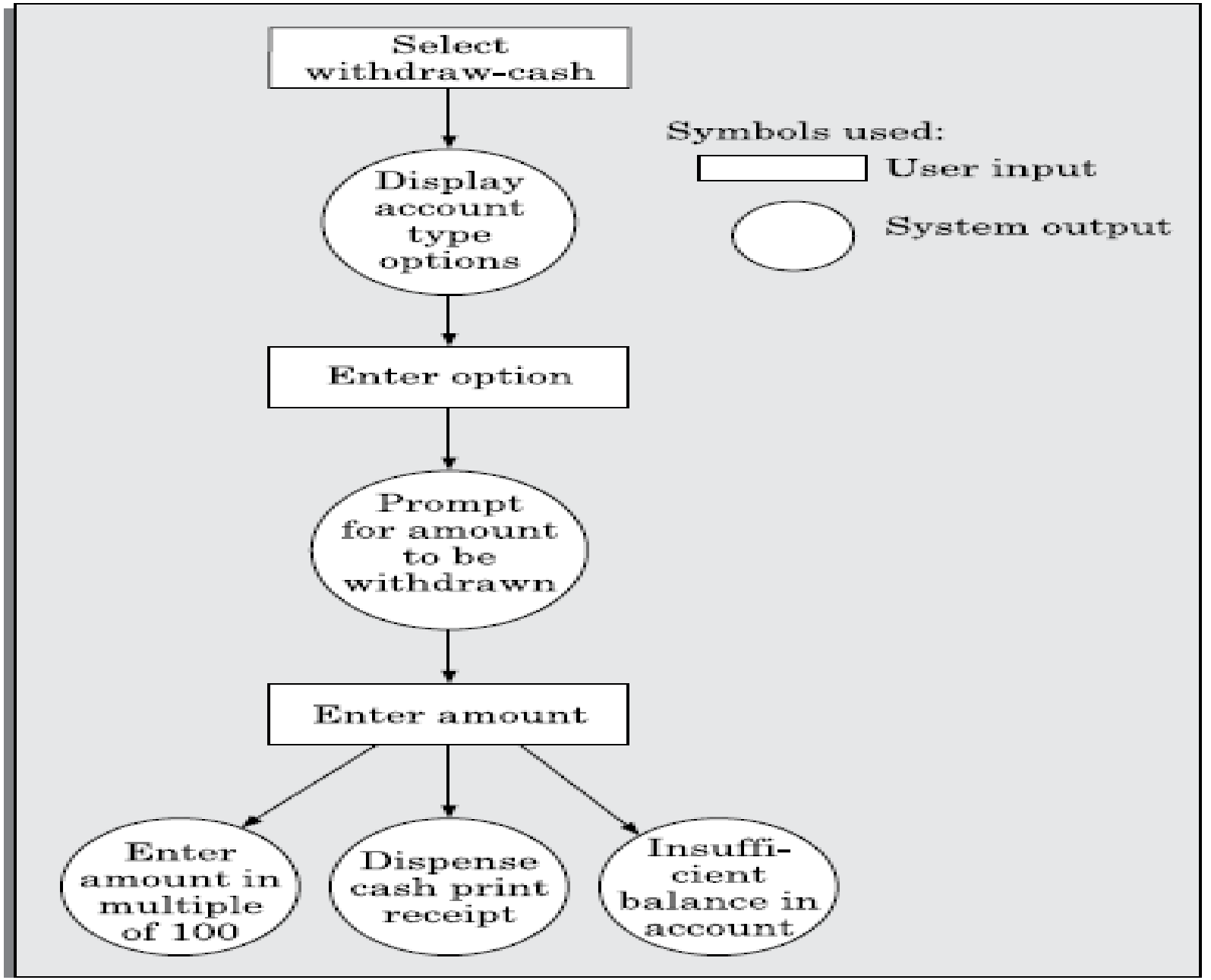


FIGURE 3.2 User and system interactions in high-level functional requirement.

Can You Identify All Inputs and Outputs Exactly?

- Ideally yes, but not always possible, especially **without a working prototype**.
- In such cases, use **high-level terms** to describe the data.
- Example: In ATM withdraw-cash:
  - User input is split across steps (e.g., account type, amount), not given all at once.

Aspect	Description
Functional Requirements	What the system should do (functions with input and output).
Non-Functional Requirements	Quality and technical constraints (performance, interfaces, security, etc.)
Goals of Implementation	Optional suggestions (e.g., future flexibility) — not tested during acceptance.
High-Level Functions	User-invoked useful services, often involving multiple interaction steps.

3.2.7 How to Identify the Functional Requirements?

✦ Steps to Identify Functional Requirements:

- **Start with informal problem description** or domain understanding.
- **Identify user types** (different users → different expectations).
- **List services** expected by each user.

✦ Subjectivity in Identification:

- Some decisions involve judgment:
  E.g., In a **Library System**, entering book details might be a sub-function of "Issue Book" instead of a separate high-level function.

- Use **common sense** and **visualize usage scenarios** to determine what constitutes a **meaningful unit of functionality** for the user.

3.2.8 How to Document Functional Requirements?

📌 What to Include for Each Functional Requirement:

- **State** when the function is invoked.
- **Input domain**
- **Output domain**
- **Processing logic**
- **Scenarios** (alternate paths depending on user input)

Example: Withdraw Cash from ATM

ID	Requirement	Input	Output	Processing
R.1	Withdraw Cash	-	Cash or Error Message	Check balance, dispense cash or error
R.1.1	Select withdraw amount option	Menu option	Prompt for account type	-
R.1.2	Select account type	Savings/Checking/Deposit	Prompt for amount	-
R.1.3	Enter amount	Amount (₹100–₹10,000)	Dispensed cash + printed receipt / error	Validate balance and debit if enough funds exist

Example: Search Book in Library

ID	Requirement	Input	Output	Processing
R.1	Search Book	Key words	Matching book details	Search by title/author
R.1.1	Select search option	Option click	Prompt for keywords	-
R.1.2	Search and display	Keywords	Book title, author, year, ISBN, availability, etc.	Keyword matching and display

Example: Renew Book

- Involves login, selecting a book, validating, and confirming.
- Has multiple sub-requirements (R.2.1 to R.2.3).

3.2.9 Traceability

✦ **What is Traceability?**

- Traceability ensures that every requirement:

Can be **linked to its design component**

Can be **linked to code**

Can be **linked to its test cases**

✦ **Importance:**

- Confirms that **all requirements** are implemented and tested.
- Helps assess the **impact of changes** or bugs.

✦ **How to Ensure Traceability:**

- **Number every requirement uniquely**.
  - E.g., R.1, R.1.1, R.1.2, R.2, etc.

3.2.10 Organization of the SRS Document (IEEE 830)

Key Sections in SRS:

Section	What It Covers
1. Introduction	Purpose, scope, system environment, product overview
2. Overall Description	Product perspective, features, user classes, hardware/software environment
3. Functional Requirements	Grouped by user class or operation mode; each function with ID and details
4. External Interface Requirements	UI, hardware, software, communication interfaces
5. Other Non-Functional Requirements	Performance, security, safety, design constraints
6. Goals of Implementation	Optional suggestions (e.g., future-proofing, reusability, extensibility)

IEEE 830-1998 Standard for SRS

- Title
- Table of Contents
- 1. Introduction
  - 1.1 Purpose
  - 1.2 Project Scope
  - 1.3 Environmental Characteristics

- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices
Index

-

•This document specifies the requirements for the Task Manager Application.

• To create, manage, and track tasks with reminders and priorities, syncing across multiple devices.

• Outline the environment(Hardware and Software) with which the software will interact

IEEE 830-1998 Standard – Section 2 of SRS

- Title
- Table of Contents
- 1. Introduction
- 2. Overall Description
  - 2.1 Product Perspective
  - 2.2 Product Features
  - 2.3 User Characteristics
  - 2.4 Operating Environments
  - 2.5 Design and Implementation Constraints
  - User documentation

•This section explains how the product fits into the larger system, its relationship to other systems, and its dependencies. Interfaces with other software/hardware, Dependencies on external systems, High-level context diagram (optional).

This section lists the main functions and capabilities of the product.
- Create, read, update, delete tasks
- Set reminders
- Categorize tasks
- Synchronize tasks across devices

•This section describes the different user types (classes), their skills, experience, and technical knowledge relevant to using the product. Types of users (beginner, advanced, admin, etc.), Education/technical skill levels, Frequency of use

• specifies the hardware, software, network, and system configurations. Operating systems (Windows, macOS, Linux), Hardware requirements (Processor, RAM, storage), Browser versions (Chrome 115+, Firefox 100+), Network type (LAN, Cloud, 5G)

• This describes limitations and restrictions that affect design or coding decision. Regulatory compliance, Programming languages to be used, Database requirements, Communication protocols (HTTPS), Performance limitations

•This section lists the manuals, guides, online help, and training materials that will be delivered with the product for end users. User manual (PDF or printed), Installation guide, FAQ/help section in the application, Video tutorials and quick-start guides

IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Descriptio
- 3. Functional Requirements

    1. User Class(Operation mode) 1

        (a) Functional requirement 1.1

        (b) Functional requirement 1.2

    2. User Class(Operation mode) 2…….

- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

•Classify the functionalities either based on the specific functionalities invoked by different users
•Or the functionalities that are available in different modes etc. depending on what may be appropriate

•User Interfaces
•Hardware Interfaces
•Software Interfaces
•Communication Interfaces

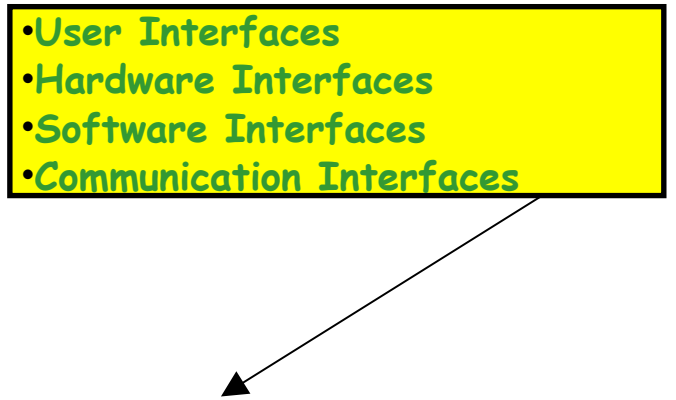
• Performance Requirements
• Safety Requirements
• Security Requirements

IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index



IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

Example: Personal Library Software – Functional & Non-Functional Requirements

Functional Requirement Categories:

1. **Manage Own Books**
   - Register book
   - Issue/return/query books
2. **Manage Friend Details**
   - Register/update/delete friends
3. **Manage Borrowed Books**
   - Record borrowed/deregister returned books
4. **Manage Statistics**
   - View count, investment, transaction summaries

Non-Functional Requirements:

- Free public DBMS (N.1)
- Runs on Windows & UNIX (N.2)
- Web-query support (N.3)

Summary Table: Requirement Types

Type	Description
Functional Requirement	Input → Output transformation, invoked directly by user
Non-Functional Requirement	Constraints (e.g., performance, platform, security, usability, etc.)
Goal of Implementation	Optional suggestions to developers (not validated during acceptance)
Traceability	Ability to trace requirement ↔ design ↔ code ↔ test

3.2.11 Techniques for Representing Complex Logic

When a **high-level functional requirement** involves **complex decision-making** or **multiple alternate flows**, describing it in plain text becomes difficult and error-prone.

- To handle such complexity, two techniques are commonly used:
  - **Decision Trees**
  - **Decision Tables**

These are especially useful in:

- Identifying **all possible conditions and outcomes**
- Designing **clear test cases**
- Making **logic easier to validate and implement**

When to Use:

Scenario	Use?
Few alternatives, simple logic	Plain text is sufficient
Many conditions, multiple outcomes	Use decision trees/tables
Need clear test case generation	Use decision tables

Decision Tree

✦ **Definition:**

- A **graphical representation** of decision logic.
- **Internal nodes**: conditions
- **Edges**: outcomes (yes/no or true/false)
- **Leaf nodes**: actions to be taken

🧪 **Example: Library Membership System (LMS)**

- **Supported Options:**
  - ➢ New Member
  - ➢ Renew Membership
  - ➢ Cancel Membership

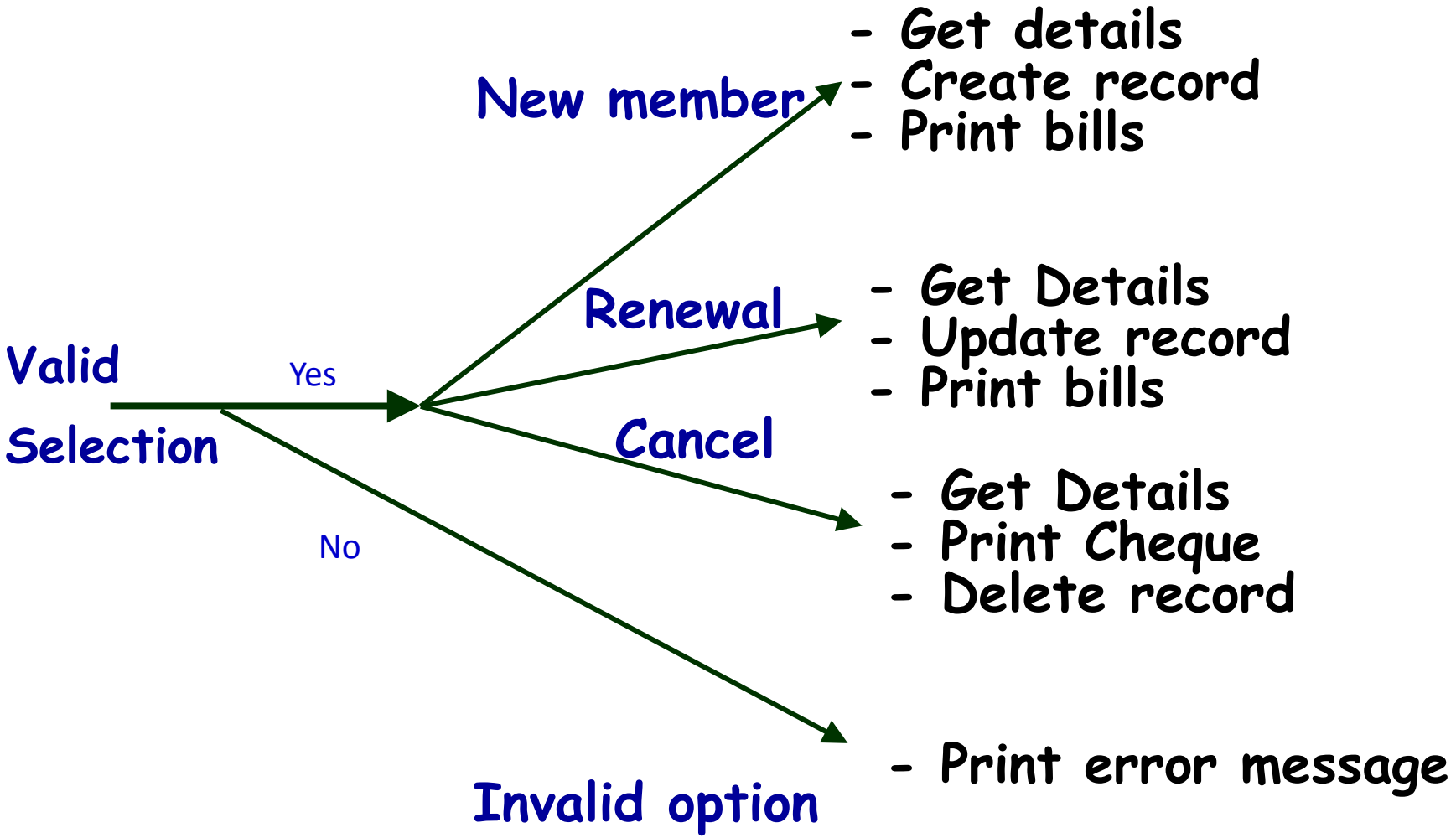
📌 **Logic (simplified):**

- If **invalid option**, → Show error message
- If **New Member**, → Ask details → Create record → Generate bill
- If **Renew**, → Ask member info → Validate → Update date → Generate bill / Show error
- If **Cancel**, → Ask name → Validate → Cancel → Print cheque → Delete record

- ➡️ This logic is shown using a **decision tree in Figure 4.3**.

- ✅ **Best For:**
  - Clear **hierarchical** and **multi-level decisions**
  - Easy visualization (when conditions are few)

Decision Tree Example



Decision Table

✦ **Definition:**

- A **matrix-style tabular representation** where:
- **Rows**: conditions (top) and actions (bottom)
- **Columns (Rules)**: unique combinations of condition outcomes

Decision Tree vs Decision Table

Feature	Decision Tree	Decision Table
Readability	Easier for small number of conditions	Better for large number of combinations
Order of Decision Making	Explicitly shown	Abstracted (not shown)
Multi-level Decision Representation	Supports it easily	Not intuitive for multi-level decisions
Compactness (for many conditions)	Becomes cluttered	More compact and structured
Test Case Generation	Manual	Easier and more direct

- 🧪 **Example: Decision Table for LMS**

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Valid selection	NO	YES	YES	YES
New member selected	-	YES	NO	NO
Renewal selected	-	NO	YES	NO
Cancel selected	-	NO	NO	YES
Actions				
Show error	✅			
Ask member info		✅		✅
Create record		✅		
Generate bill		✅	✅	
Ask for mem. details			✅	
Update expiry			✅	
Print cheque				✅
Delete record				✅

How to Read This Table:

- Each **column (Rule)** represents one scenario or decision path.
- **'×'** indicates the **action to be performed** under that rule.
- **'-'** indicates that the condition is **not relevant** or not evaluated for that rule.

3.3 Formal System Specification

Why Formal Methods?

Formal methods are mathematical approaches used in software engineering to **precisely describe, model, and verify systems**. They help ensure a system is **correctly implemented**—meaning it behaves exactly as its specification demands.

4.3.1 What is a Formal Technique?

A **formal technique**:

- Is a **mathematical method** to:
  - Specify systems (hardware/software)
  - Check whether a specification is possible (realisable)
  - Prove an implementation matches its specification
  - Verify system properties **without execution**

A formal specification language has:

- **Syntactic domain (syn):** Symbols + rules to form valid statements.
- **Semantic domain (sem):** Meaning/interpretation of those statements.
- **Satisfaction relation (sat):** Defines when the system model satisfies the specification.

📌 **System Development Hierarchy:**

Every stage (requirements → design → coding) serves as:

- **Implementation** of the previous stage
- **Specification** for the next stage
- Formal techniques can verify that **each stage aligns with the previous**.

Syntactic and Semantic Domains

➢ **Syntactic domain:**

Includes an alphabet and rules to form **well-formed formulas**.

➢ **Semantic domain:**

Varies by type:

- ➢ **Abstract Data Types:** Algebras, theories, etc.
- ➢ **Programs:** Input-output functions
- ➢ **Concurrent Systems:** States, events, transitions, etc.

Satisfaction Relation

- Used to check if a **system model** matches its specification.
- A **semantic abstraction function** maps system behavior to simplified representations.
- Two types:
  - Behavior-preserving
  - Structure-preserving

Model vs Property-Oriented Methods

- **Model-Oriented:**
  - Describes **how** the system behaves by **building a model**.
  - Uses **mathematical structures** like sets, sequences, tuples.
  - Example methods: **Z, CSP, CCS**

- **Property-Oriented:**
  - Describes **what** the system should do by listing **axioms or properties**.
  - Allows flexibility in implementation.
  - Example methods: **Axiomatic, Algebraic Specification**

📌 Example (Producer/Consumer):

- Property-oriented: State rules like "Consumer can only consume after production."
- Model-oriented: Define actions like produce(p) and consume(c).

📌 Usage:

- **Property-oriented** → Better for **requirements specification**
- **Model-oriented** → Better for **system design**

4.3.2 Operational Semantics

Operational semantics define **how system behavior is represented** during execution.

Common types:

1.Linear Semantics:

- Describes system behavior as **sequences of events/states**
- Concurrent actions shown by **interleaving** (e.g., a||b = a;b or b;a)

2.Branching Semantics:

- Uses a **graph** to show multiple paths from a state
- Better shows decision points but still uses interleaving for concurrency

3.Maximally Parallel Semantics:

- All concurrent actions run **together**
- Assumes full resources are always available (unrealistic)

4.Partial Order Semantics:

- Describes events with a **partial order**
- Some events must occur after others; **true concurrency is preserved**
- More **realistic** model for concurrent systems

Merits of Formal Methods

- ✅ **Precise specifications**
  ✅ **Early detection of design flaws**
  ✅ **Mathematical correctness (provable)**
  ✅ **No ambiguity in meaning**
  ✅ **Support for automated verification (e.g., theorem provers)**
  ✅ **Executable specifications = early feedback/prototyping**
- Example: In real-time systems, poor requirement specs cause **80% of cost overruns**, so formal methods can save time and money by catching problems early.

Limitations of Formal Methods

- ❌ **Hard to learn and use**
  ❌ **Incompleteness of logic = full correctness not always provable**
  ❌ **Complex systems = complex specifications (hard to manage)**
  ❌ **Large mathematical formulas = hard to understand**

Balanced Use: Formal + Informal

- Formal specs **do not replace** informal ones—they **complement** them.
- Use **formal methods** to define precise logic and verification steps.
- Use **informal reasoning** for better readability and documentation.
- Combine both for better understanding and correctness.

🔍 Example:

Jones (1980) suggests using formal methods to define verification rules, but informal language for argumentation and explanation.

Aspect	Property-Oriented	Model-Oriented
Focus	Desired properties (axioms)	Direct system behavior (model)
Suitable for	Requirements specification	System design specification
Flexibility	High (many possible implementations)	Low (tight model)
Tools/Examples	Axiomatic, Algebraic	Z, CSP, CCS

Formal methods enhance software quality through **precision**, **verification**, and **early error detection**, though they require **expertise and effort**.