Unit - II Software Engineering Requirement

1. Requirement Engineering

Definition: Requirement Engineering is the process of **gathering, analyzing, documenting, and validating** the needs and constraints of the stakeholders for a software system.

Its goal is to ensure that **the software meets user expectations and business needs**.

From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

🔄 2. Requirement Engineering Tasks

These tasks form the **backbone of the software development process**. There are typically 5 main tasks:

🔹 Inception

- Identify project stakeholders.
- Most projects begin when a business need is identified or a potential new market or service is discovered.
- The aim is
  - To have the basic understanding of problem
  - To know the people who will use the software
  - To know the exact nature of the problem.

🔹 Elicitation

Use interviews, questionnaires, and discussions to gather basic needs.

Elicitation means to define what is required.To know the objectives of the system to be developed is a critical job.

Requirement elicitation is difficult because numbers of problems are encountered:

Problems of scope: The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
Problems of understanding: The customers/users are not completely sure of what is needed. They don't have a full understanding of the problem domain. They omit information that is believed to be "obvious,". Specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous (unclear) or untestable.
Problems of volatility: The requirements change over time.

Elaboration (Analysis and Negotiation)

- Refine and expand requirements.
- Resolve conflicts, prioritize needs, and ensure feasibility.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.
- Each user scenario is parsed to extract analysis classes.
- The attributes of each analysis class are defined, and the services that are required by each class are identified.

Negotiation

It means discussion on financial and other commercial issues. In this step customer, user and stakeholder discuss to decode:

- To rank the requirements
- To decide priorities
- To decide risks
- To finalize the project cost
- Impact of above on cost and delivery

Specification

- Document all requirements in a structured form.
- Can be done using **Software Requirement Specification (SRS)** documents.
- A written document, a set of graphical models, a collection of scenarios, a prototype, Mathematical model.

- It serves as the foundation for subsequent software engineering activities. It describes the function, performance of a computer-based-system, constraints that will govern its development.

🔹 Validation

- Ensure that requirements are **complete, correct, and consistent**.
- Reviews, walkthroughs, and prototyping are used.
- Products are assessed for quality during the validation period.
- Provide clarification related to conflicting requirements, unrealistic expectations, etc.

🔹 Requirements Management

- Maintain and track changes to requirements throughout the project.
- Helps avoid scope creep and ensures traceability.

📑 3. Types of Requirements

Requirements are generally classified into three main types:

🔹 A. Functional Requirements

- Define **what the system should do**.
- Describe system behavior under specific conditions.
- Functional requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces and the functions that should be included in the software.
- These requirements should be complete and consistent.
- Completeness implies that all the user requirements are defined.
- Consistency implies that all requirements are specified clearly without any contradictory definition.
- 📌 *Example:* "The system shall allow users to log in using email and password."

🔹 B. Non-Functional Requirements

- Define **how the system performs** tasks (quality attributes).
- Includes performance, usability, reliability, and security.
- These requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not related directly to any particular function provided by the system.

Different type of Non Functional Requirement

Product Requirements:

- **Reliability Requirements** describe the acceptable failure rate of the software.
- **Usability Requirements** describe the ease with which users are able to operate.
- **Efficiency Requirements** describe the extent to which the software makes resources, the speed with which the system executes and the memory it consumes for its operation.
- **Portability Requirements** describe the ease with which the software can be transferred from one platform to another

Organizational Requirements:

(a) Implementation Requirements describe requirements such as programming language and design method.

(b) Standards Requirements describe the process standards to be used during software development. For example, ISO and IEEE standards.

(c) Delivery Requirements specify when the software and its documentation are to be delivered to the user.

External Requirements:

(a) **Interoperability Requirements** defines the way in which different computer-based systems interact with each other in one or more organizations.
(b) **Legislative Requirements** ensures that the software operates within the legal jurisdiction. For example, pirated software should not be sold.

(c) Ethical Requirements specifies the rules and regulations of the software so that they are acceptable to users.

- 📌 *Example:* "The system must handle 1000 transactions per second.

🔹 **C. Domain Requirements**

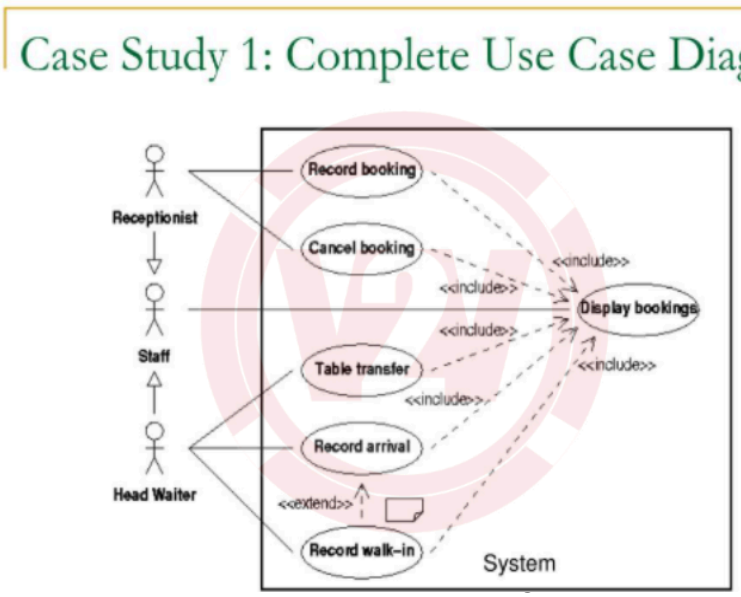
- Specific to the **domain or industry** of the software.
- Often driven by regulations, standards, or specific rules.
- 📌 *Example:* "A banking system must comply with RBI data security policies."

🎭 4. Developing Use-Cases

📘 **Definition:** A **use-case** is a written description of **how a user interacts** with the system to achieve a specific goal.
It helps in understanding **user behavior and system responses**.

🧩 **Components of a Use-Case:**

- **Use-Case Name**
- **Actor(s)** – Who interacts with the system
- **Preconditions** – What must be true before the use-case starts
- **Basic Flow** – Step-by-step normal interaction
- **Alternate Flow** – Variations in the steps
- **Exceptions** – What happens in case of errors
- **Postconditions** – Final state after use-case completes



📘 **SRS – Software Requirements Specification**

✅ 1. What is SRS? (Definition)

SRS is a **formal document** that describes the **functional and non-functional requirements** of the software system to be developed.
It acts as an **agreement** between the customer and the development team.

SRS should include both the user requirements for a system and a detailed specification of the system requirements.

Software requirement specification is a document that completely describes what the proposed software should do without describing how software will do it.

✅ 2. Need for SRS (Why SRS is Important)

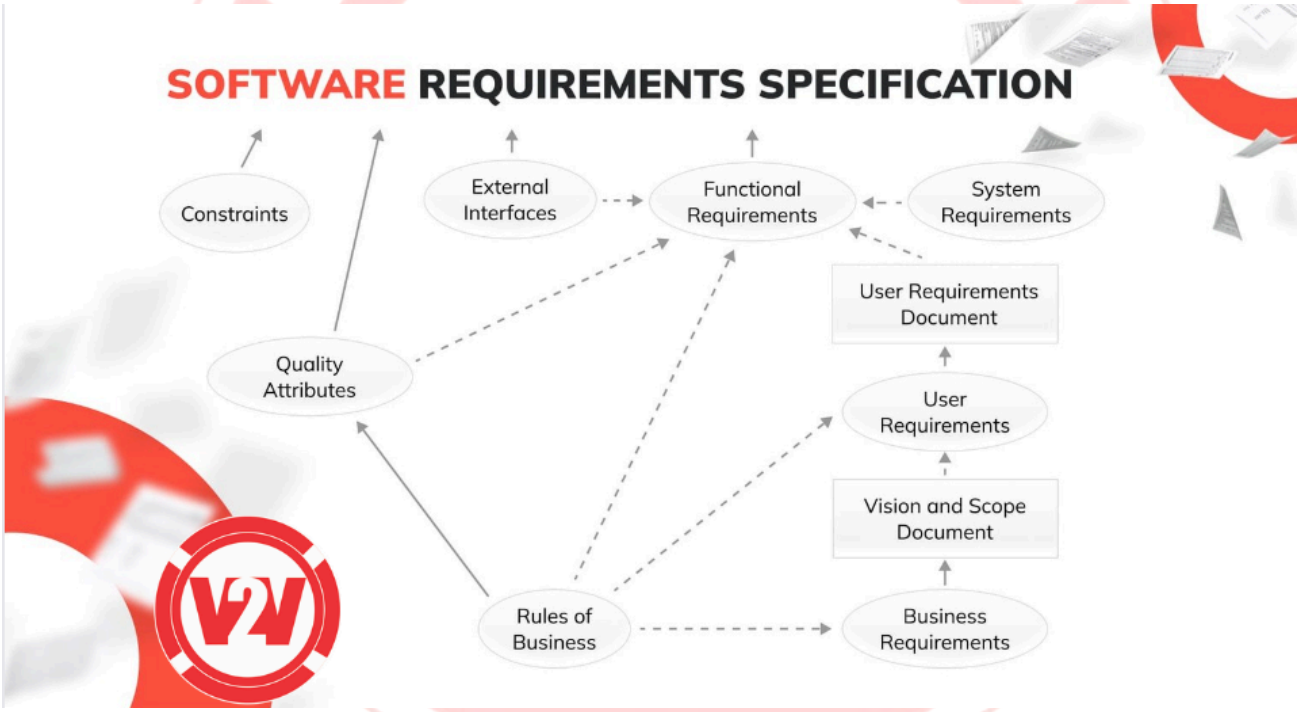
Reason	Explanation
✅ **Clear Communication**	Acts as a contract between stakeholders and developers
✅ **Requirement Clarity**	Helps avoid misunderstandings and assumptions
✅ **Project Planning**	Provides the foundation for cost, effort, and time estimation
✅ **Design & Development Base**	Used by designers and developers to build the system
✅ **Testing & Validation**	Testers use it to create test cases and validate functionality
✅ **Maintenance Reference**	Helps in future enhancements and debugging

1. An SRS is helping the clients understand their own needs or requirements.

2. An SRS is important because it establishes the basis for agreement between the client and the supplier on what the software product will do.

3. A high-quality SRS is a prerequisite to high-quality software.

4. SRS enables developers to consider user requirements before the designing of the system. This reduces rework and inconsistencies, which reduce the development efforts.
7. SRS needed to provide feedback, which ensures to the user that the organization understands the issues or problems to be solved.



Format of SRS Document

A standard SRS document follows the IEEE 830 standard. Here's a typical structure:

- Section 1: Product Overview and Summary
- Section 2: Development, Operating, and Maintenance Environments
- Section 3: External Interfaces and Data Flow
- Section 4: Functional Requirements
- Section 5: Performance Requirements
- Section 6: Exception Handling
- Section 7: Early Subsets and Implementation Priorities
- Section 8: Foreseeable Modifications and Enhancement
- Section 9: Acceptance Criteria
- Section 10: Design Hints and Guidelines
- Section 11: Cross-reference Index
- Section 12: Glossary of Terms

Characteristics of a Good SRS

Characteristic	Explanation
Correctness	SRS should describe the actual needs of the stakeholders
Completeness	It should include all significant requirements
Unambiguity	Each requirement should have only one interpretation
Consistency	No conflicting requirements or definitions
Verifiability	Each requirement must be testable (i.e., it can be verified)

Characteristic	Explanation
Modifiability	The document should be easy to update and modify
Traceability	Each requirement should be traceable to its source (client need, business rule)
Rank for Importance	All requirements are not equally important so we need to decide priority

🧩 Translating Requirement Model into Design Model

Translating the requirement model into the design model means converting the **"what" (requirements)** into the **"how" (design)**.
It bridges the gap between **user needs** (captured in the SRS or use-case model) and the **technical solution** (architecture and components of the system).

✅ 2. Purpose of Translation

- To **structure the software solution** based on user requirements
- To ensure the system design meets **all functional and non-functional requirements**
- To prepare the system for **implementation (coding)**

✅ 3. Steps in Translating Requirement Model into Design Model

---

🔹 **Step 1: Analyze the Requirements**

- Review the **SRS document**, **use-cases**, and **user stories**
- Identify key **system functions**, **interactions**, and **constraints**

🔹 **Step 2: Create the Data Design**

- Define **data structures** and **data storage needs**
- Use tools like **ER Diagrams**, **Class Diagrams**, or **Data Dictionaries**

🔹 **Step 3: Develop the Architectural Design**

- Decide **how the system will be structured**
- Identify **main components**, their responsibilities, and communication
- Output: **Architecture Diagram** or **Component Diagram**

🔹 **Step 4: Design Interfaces**

- Design **user interfaces (UI)** and **system interfaces**
- Consider **input forms**, **navigation**, and **user interaction**

🔹 **Step 5: Create Procedural/Component Design**

- Define **modules**, **functions**, and **logic flow**
- Describe control flow using **Activity Diagrams**, **Sequence Diagrams**, or **Flowcharts**

🔹 **Step 6: Verify and Validate the Design**

- Ensure the design **satisfies all the requirements**
- Conduct **reviews**, **walkthroughs**, or **prototyping**

🧠 **Core Software Design Concepts**

🔹 1. Abstraction

Definition: Abstraction is the process of **hiding unnecessary details** and showing only the essential features of an object or system.

Purpose:

- Helps manage **complexity**
- Makes software easier to understand and maintain

Types of Abstraction:

- **Functional Abstraction:** What a module does
- **Data Abstraction:** What data is stored and how it is used
- **Control Abstraction:** Describes control flow without implementation details

📌 *Example:* A function like print() hides the complexity of I/O operations.

🔹 2. Information Hiding

Definition: A design principle where **implementation details are hidden**, and only necessary interfaces are exposed to the user.

Purpose:

- Enhances **encapsulation and security**
- Reduces impact of changes in one part of the system

📌 *Example:* Hiding internal data structures in a class using private access modifiers.

🔹 3. Patterns

Definition: Design patterns are **proven solutions to common design problems** in software development.

Purpose:

- Promotes **reusability and best practices**
- Makes code easier to maintain and scale

Types of Patterns:

- **Creational** (e.g., Singleton, Factory)
- **Structural** (e.g., Adapter, Composite)
- **Behavioral** (e.g., Observer, Strategy)

📌 *Example:* Singleton pattern ensures only one instance of a class exists.

🔹 4. Modularity

Definition: Modularity is the concept of **dividing software into separate, independent modules** that perform specific tasks.

Benefits:

- Easier to develop, test, and maintain
- Promotes **code reuse and scalability**

📌 *Example:* In a banking app, separate modules for login, balance inquiry, and fund transfer.

🔹 5. Concurrency

Definition: Concurrency allows **multiple processes or threads to execute simultaneously** to improve performance.

Importance:

- Utilizes multi-core systems efficiently
- Improves **throughput and responsiveness**

📌 *Example:* A web server handling multiple client requests at the same time.

🔹 6. Verification

Definition: Verification is the process of ensuring that the software **correctly implements the specified design and requirements**.

Purpose:

- Detect and fix design flaws early
- Ensure **quality and correctness** before deployment

📌 *Example:* Code reviews, design walkthroughs, static analysis.

🔹 7. Aesthetics

Definition: Aesthetics in software design refers to the **look and feel**, **organization**, and **readability** of the software's structure and interface.

Why it's important:

- Enhances **user experience and satisfaction**
- Makes design intuitive and elegant
- Encourages **developer readability and collaboration**

📌 *Example:* A well-organized UI layout with consistent fonts, colors, and spacing.

🎯 Design Notations in Software Engineering

Design notations are **visual tools** used to represent the **structure and flow** of a system during the **software design phase**. They help communicate how the system works before coding begins.

🔷 1. Data Flow Diagram (DFD)

📘 **Definition:** A **Data Flow Diagram** is a graphical representation of the **flow of data** through a system, showing **processes**, **data stores**, **external entities**, and **data flows**.

A DFD represents system data in a hierarchical manner and with required levels of detail.

A data-flow diagram also known as bubble chart or work flow diagram.

A DFD maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, to show data inputs, outputs, storage points and the routes between each destination.

🧱 Basic Components of DFD:

Symbol	Element	Purpose
🟦 Rectangle	**External Entity**	Source or destination of data (e.g., user)
🟢 Circle/Oval	**Process**	A function that transforms data
🗃️ Open rectangle	**Data Store**	Storage of data (e.g., database, file)
➡️ Arrow	**Data Flow**	Direction of data movement

📊 Levels of DFD:

Level	Description
Level 0	**Context Diagram** – Shows the system as a single process and all external entities
Level 1	Shows main sub-processes and data flow between them
Level 2	Further breaks down Level 1 processes into detailed sub-processes
Level 3	This elaborates level 2 DFD and displays the process(s) in a detailed form.

TYPES OF DFDs :-

· There are two types of DFDs,

1. **Logical DFDs** are implementation-independent and describe the system, rather than how activities are accomplished. The logical DFD specifies the various logical processes performed on data i.e., type of operations performed.

2. **Physical DFDs** show how the system will be implemented. A physical DFD specifies who does the operations whether it is done manually or with a computer and also where it is done.

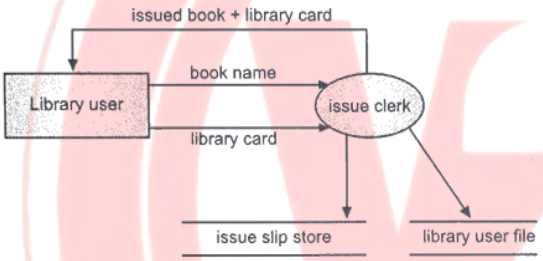


Fig. 2.13

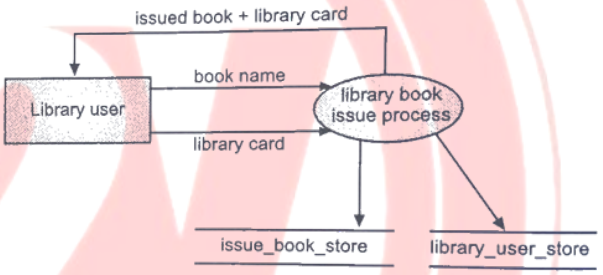


Fig. 2.14

✅ Advantages of DFD:

- Easy to understand and use
- Focuses on **data movement**, not control flow
- Helps identify system boundaries and interactions

· Let us consider an example of a Banking System. In a banking system, the customer of the bank (account holder) deposits money and gets the payment receipt. The context Diagram is shown in Fig. 2.16.

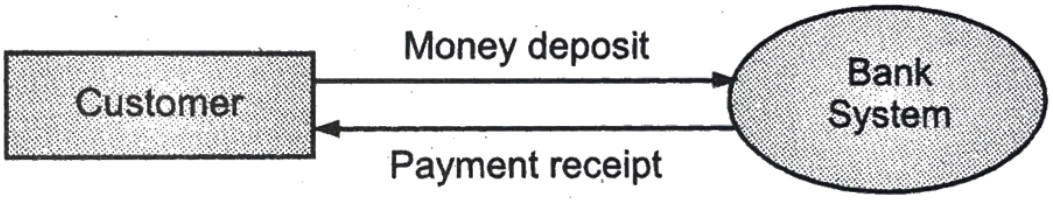


Fig. 2.16: Context Diagram of Bank System

- Level-1 DFD of the above context diagram, (Fig. 2.16) that depicts the process in some greater detail is shown in Fig. 2.17

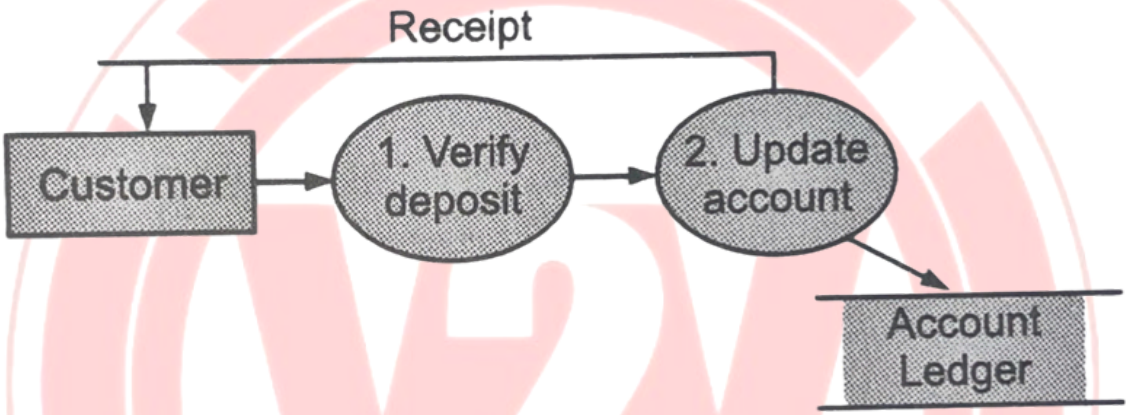
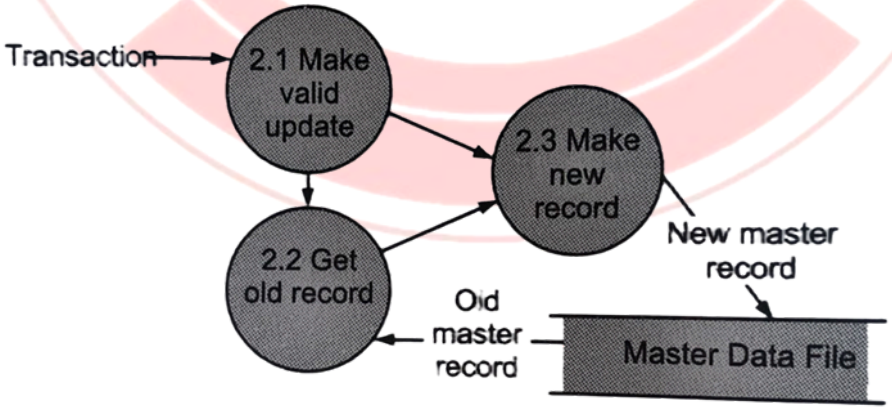


Fig. 2.17: Level-1 DFD of Bank System

The level-2 DFD of process-2 (update account) is shown in Fig. 2.18.



pdate account) of Bank System

Example 1: Draw DFD for College Management System.

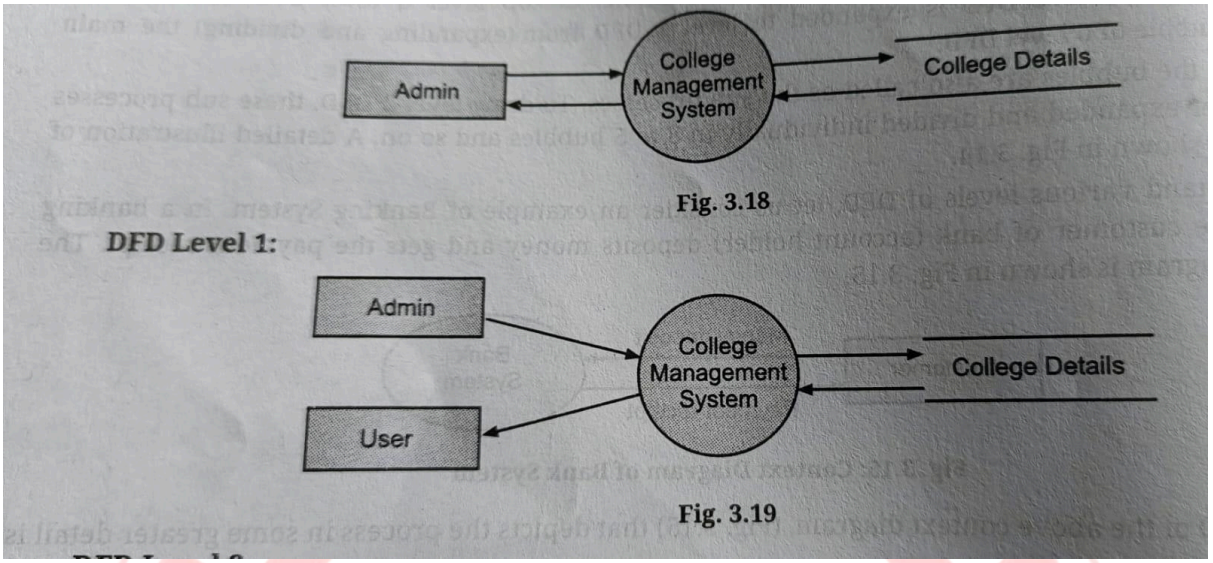


Fig. 3.18

DFD Level 1:

Fig. 3.19

DFD for Hospital Management System

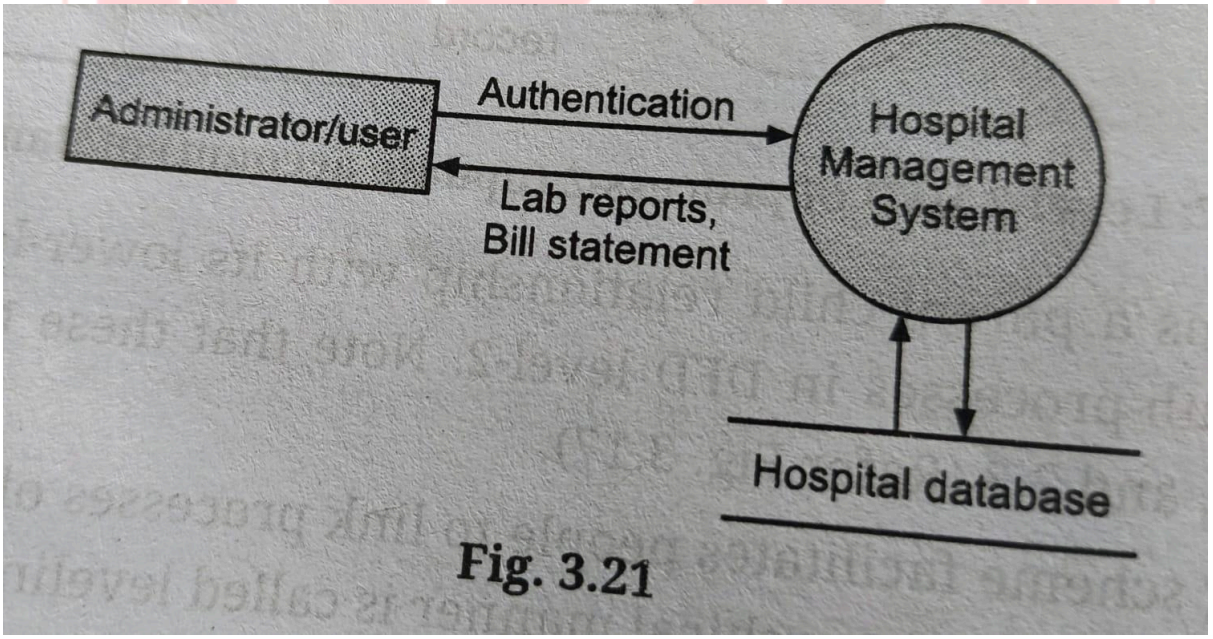


Fig. 3.21

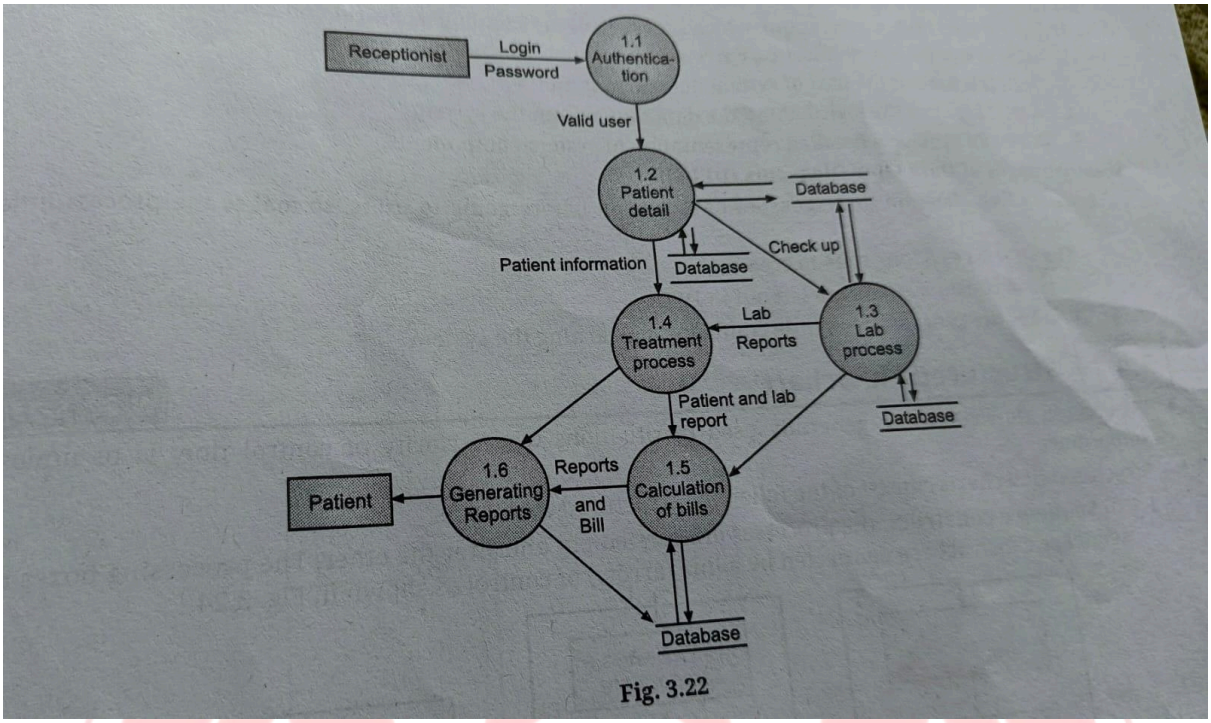


Fig. 3.22

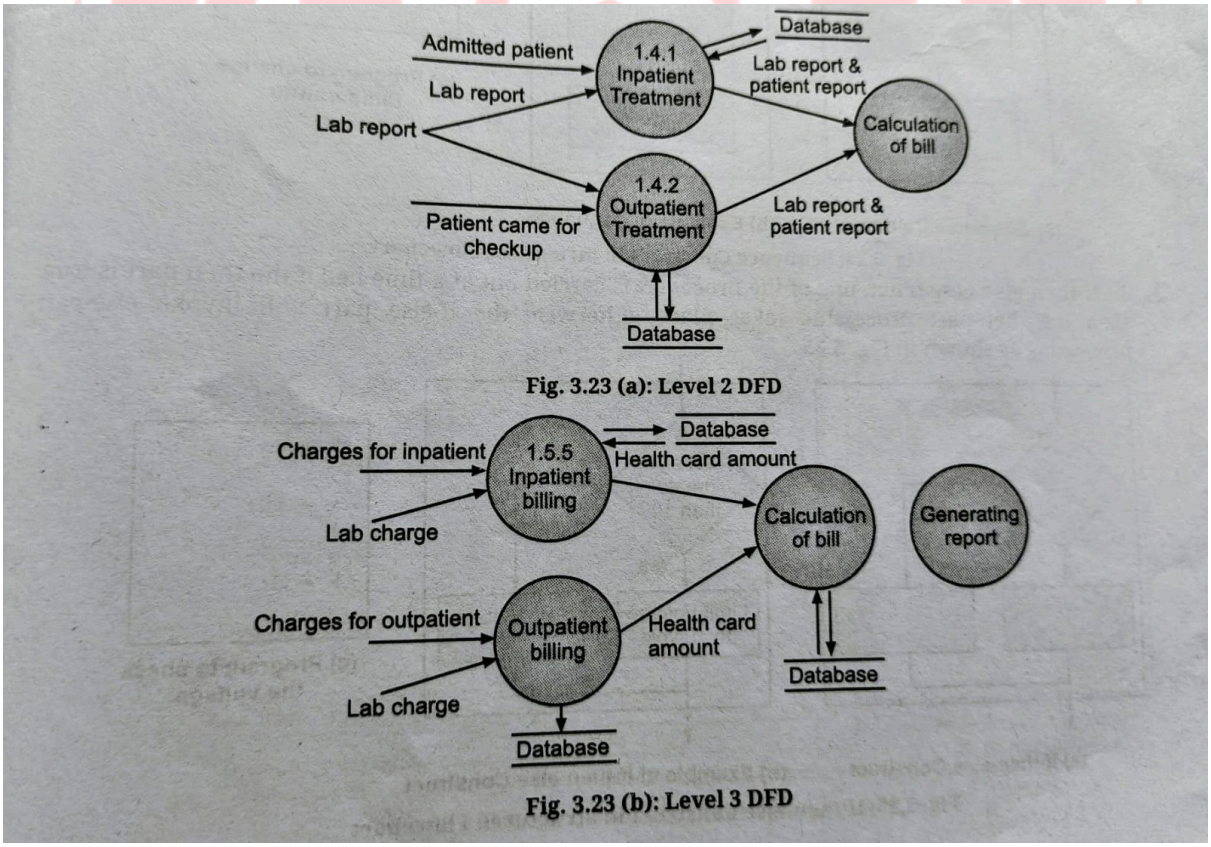


Fig. 3.23 (a): Level 2 DFD

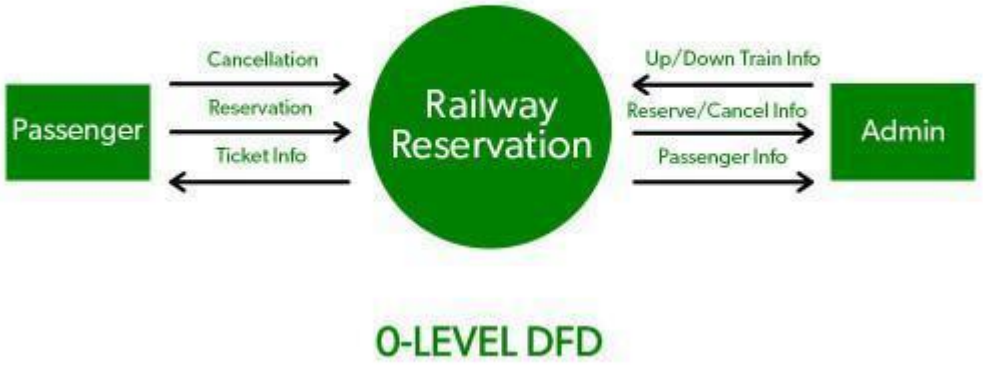
Fig. 3.23 (b): Level 3 DFD

📌 Example: ATM System DFD (Level 0)

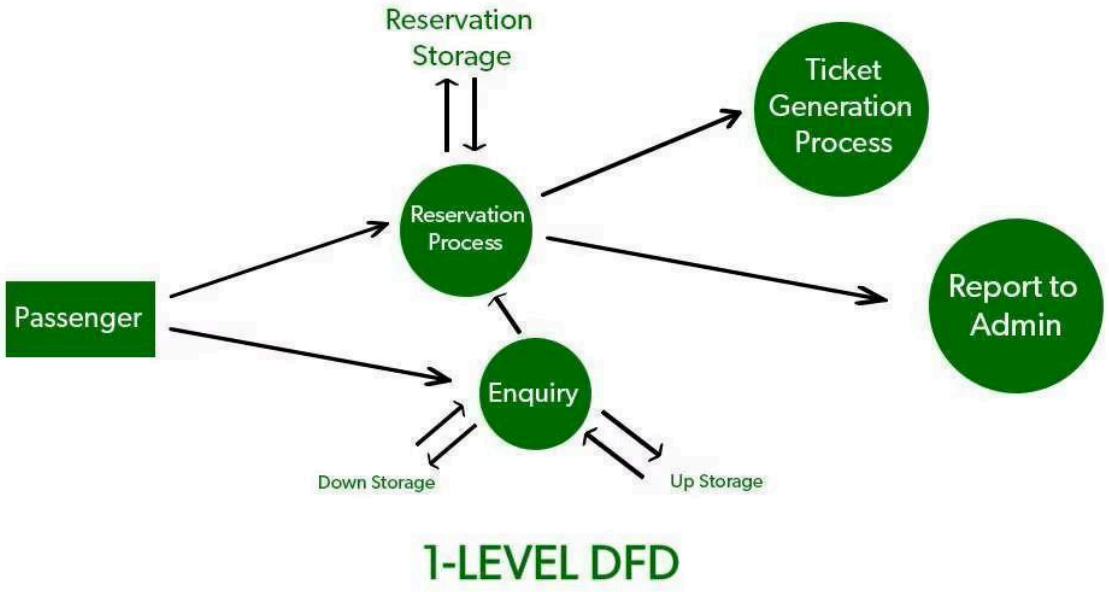
- External Entities: User, Bank
- Processes: ATM Machine
- Data Stores: Account Info
- Data Flows: PIN, transaction data, account info, cash

Railway Management System represented using **DFD Level 0, Level 1, and Level 2** diagram

🔵 Level 0 DFD



🔵 Level 1 DFD



🔵 Level 2 DFD

🔷 2. Structured Flowchart

📘 **Definition:** A **structured flowchart** is a diagram that shows the **step-by-step logic or flow of a program or process**, using standard symbols.

🧱 Basic Symbols Used:

Symbol	Meaning	Purpose
🔷 Oval	**Terminator**	Start/End of the process
⬛ Rectangle	**Process / Task**	A specific operation or instruction
🔺 Diamond	**Decision / Condition**	Yes/No or true/false decisions

Symbol	Meaning	Purpose
➡️ Arrow	**Flowline**	Shows the direction of control or process flow
· Parallelogram	**Input/Output**	Data input or output operation

---

✅ Advantages of Flowcharts:

- Clearly shows the logic of a process
- Helps in **debugging and testing**
- Improves communication between developers and clients

📌 Example: Login System Flowchart

1. Start
2. Input username and password
3. Check credentials
4. If valid → Go to dashboard
5. If invalid → Show error message
6. End