Unit - III Software Modelling & Design

🧩 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**.

🧱 Basic Components of DFD:

Symbol	Element	Purpose
🟦 Rectangle	**External Entity**	Source or destination of data (e.g., user)

Symbol	Element	Purpose
🟢 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

---

✅ Advantages of DFD:

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

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.

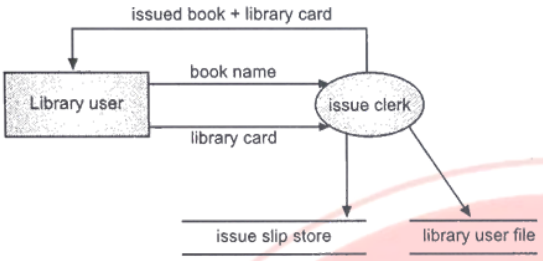


Fig. 2.13

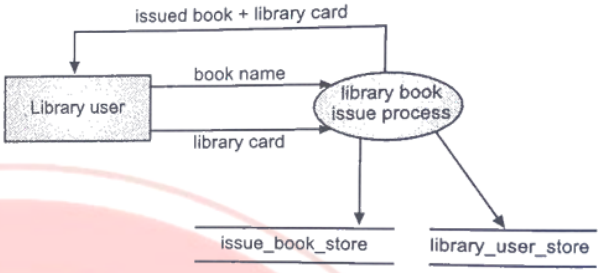


Fig. 2.14

✅ 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.

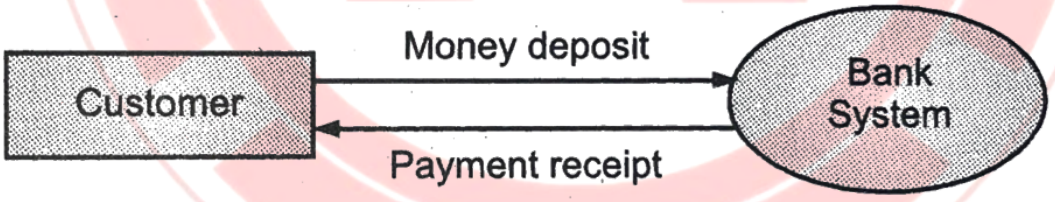


Fig. 2.16: Context Diagram of Bank System

· 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

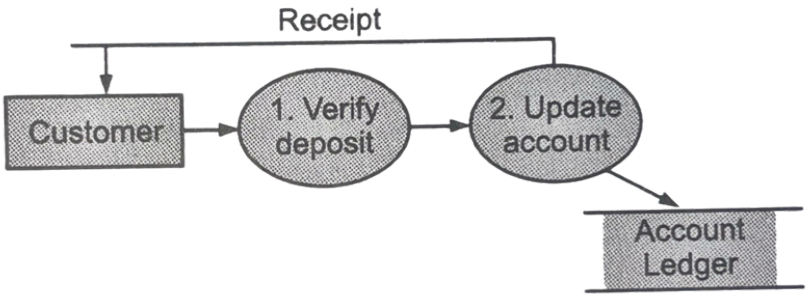
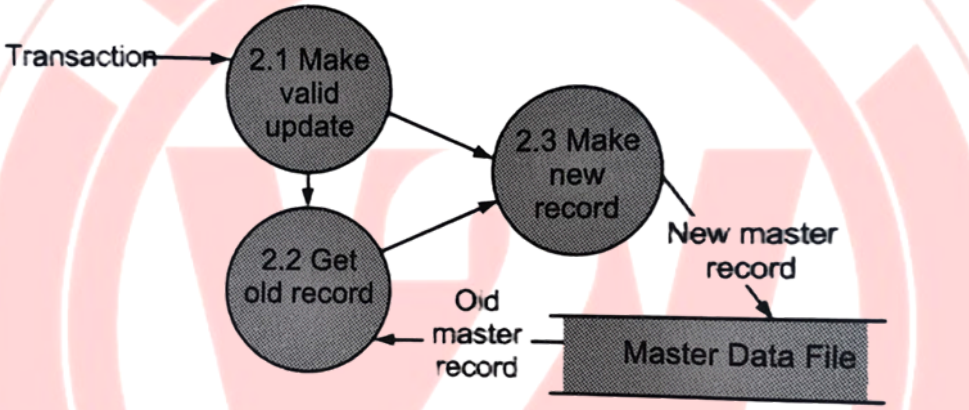


Fig. 2.17: Level 1 DFD of Bank System

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



pdate account) of Bank System

Draw DFD for College Management System.

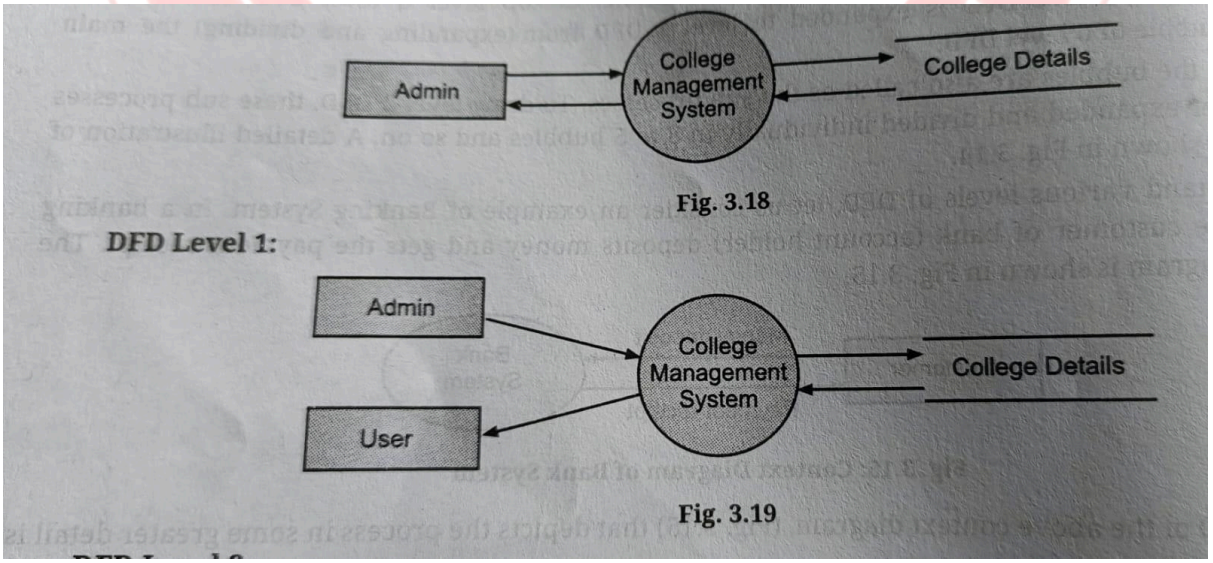


Fig. 3.18

DFD Level 1:

Fig. 3.19

DFD for Hospital Management System

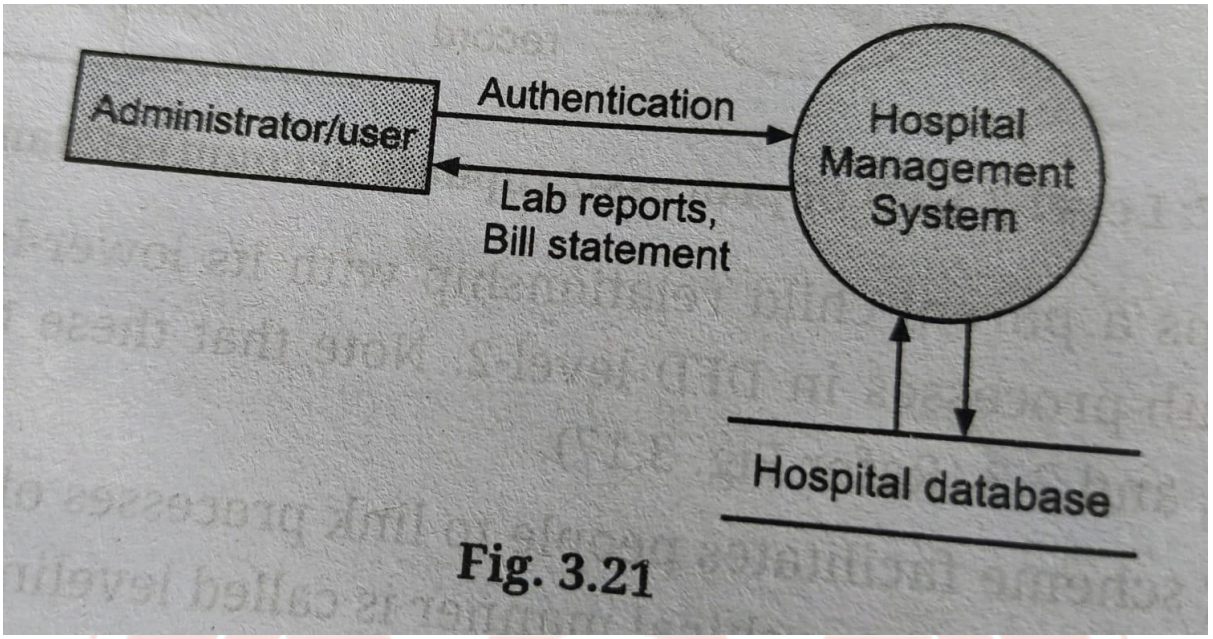


Fig. 3.21

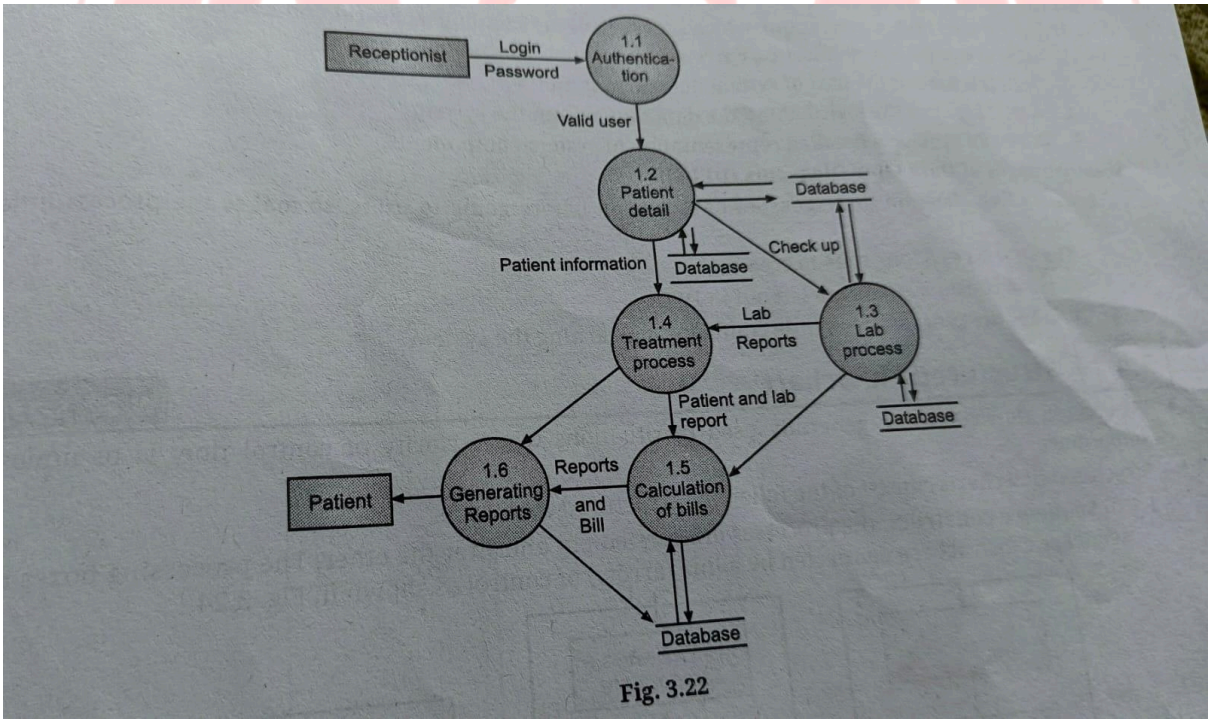


Fig. 3.22

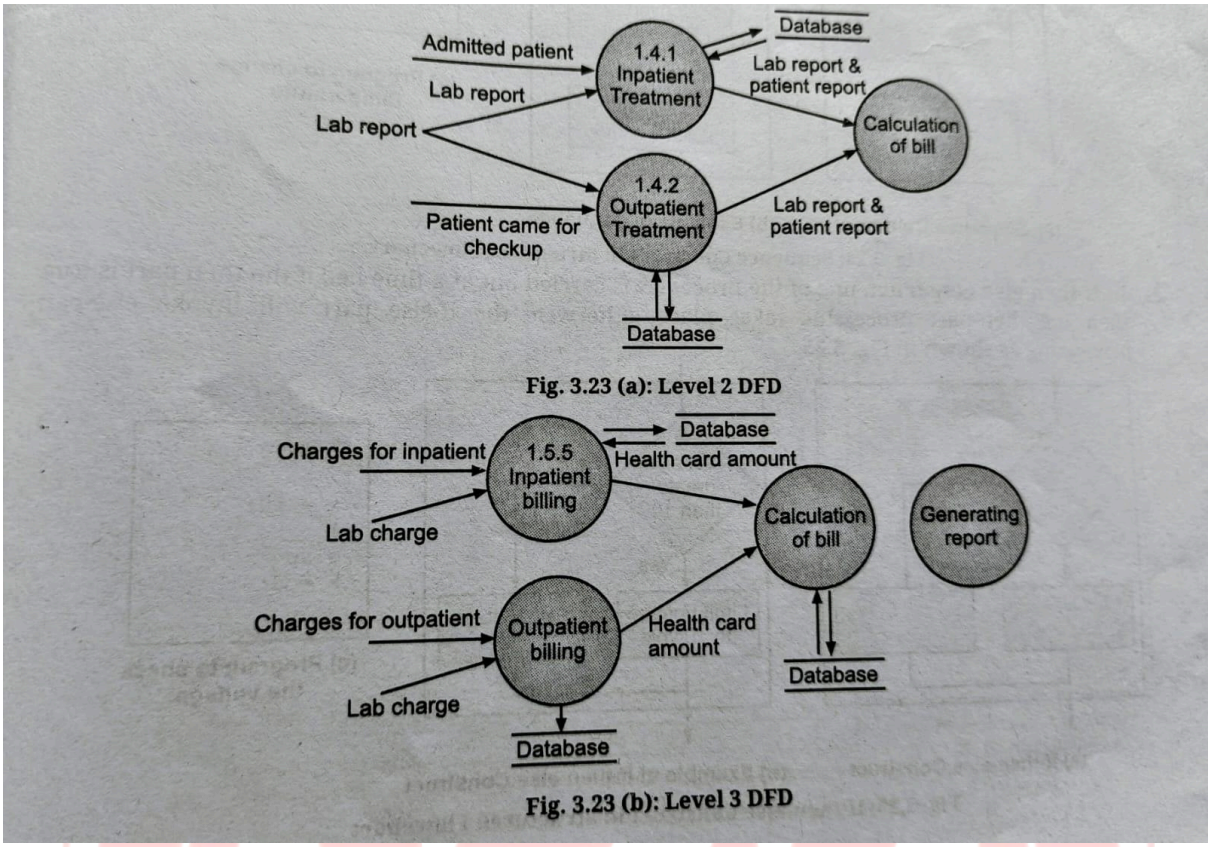


Fig. 3.23 (a): Level 2 DFD

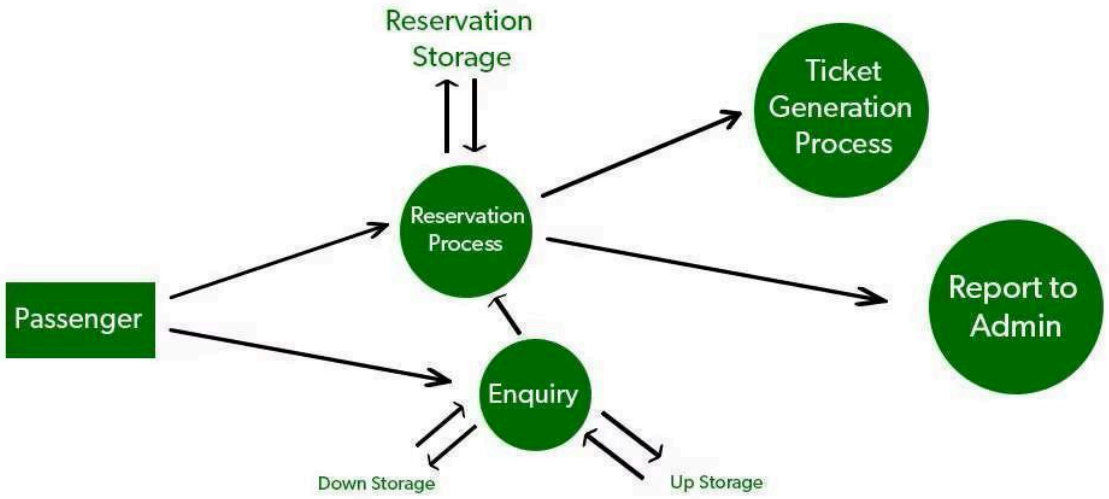
Fig. 3.23 (b): Level 3 DFD

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

🔵 **Level 0 DFD**

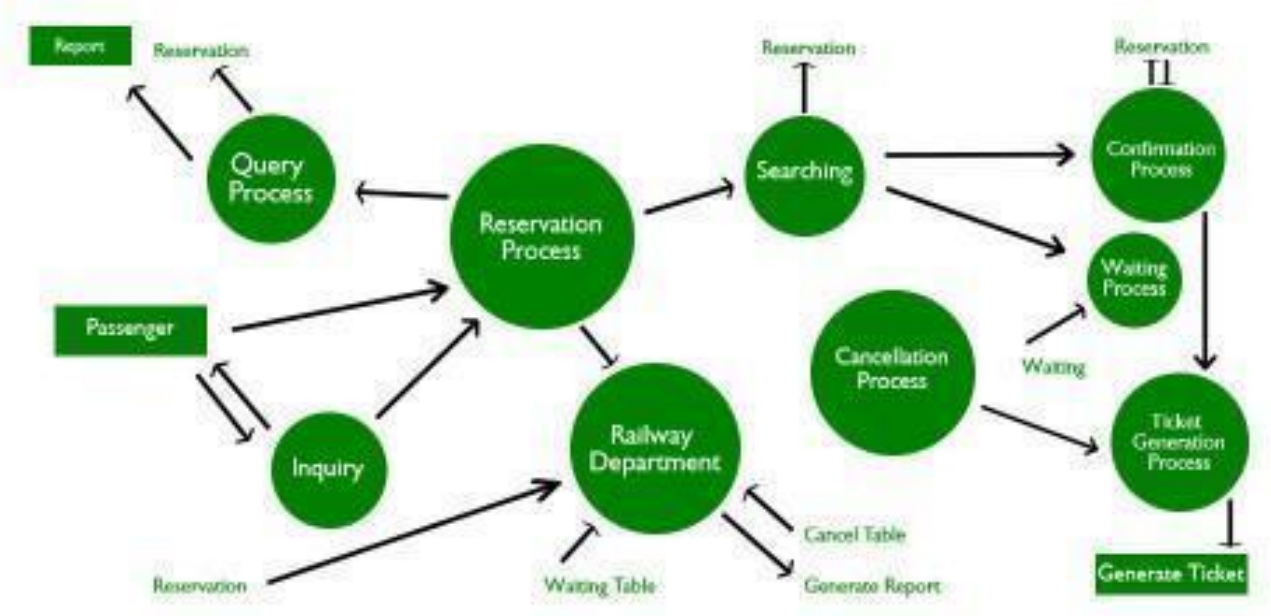
0-LEVEL DFD

🔵 Level 1 DFD



1-LEVEL DFD

🔵 Level 2 DFD



2-LEVEL 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
➡️ Arrow	**Flowline**	Shows the direction of control or process flow
· Parallelogram	**Input/Output**	Data input or output operation

---

If-then-else in flowchart

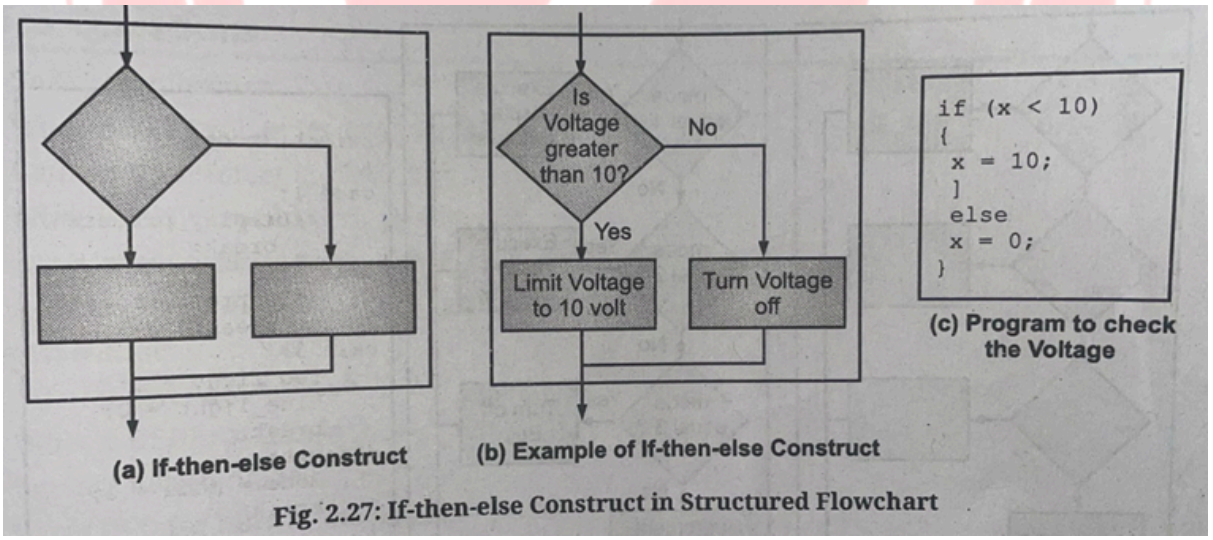


Fig. 2.27: If-then-else Construct in Structured Flowchart

While in Flowchart

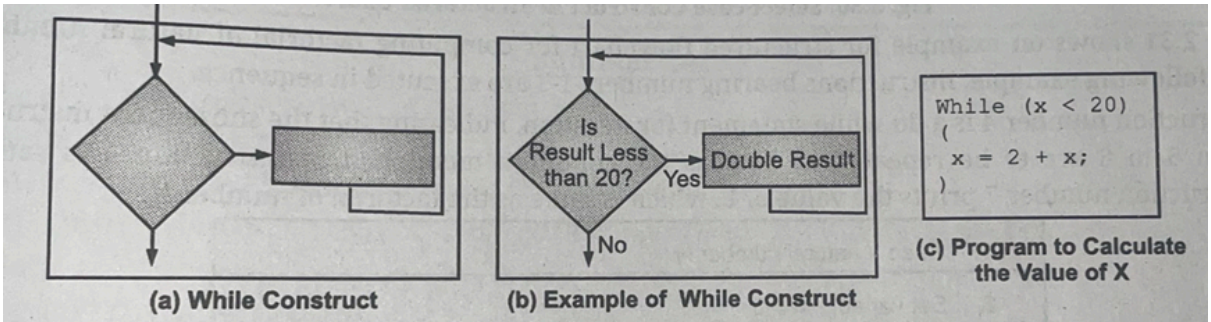


Fig. 2.28: While Construct in Structured Flowchart

Do While in Flowchart

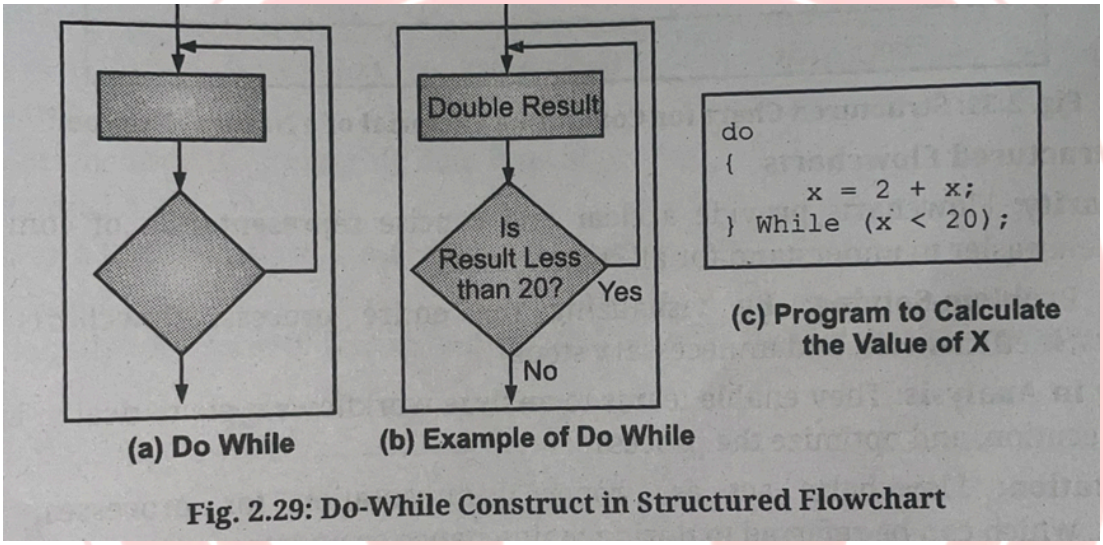


Fig. 2.29: Do-While Construct in Structured Flowchart

Switch in Flowchart

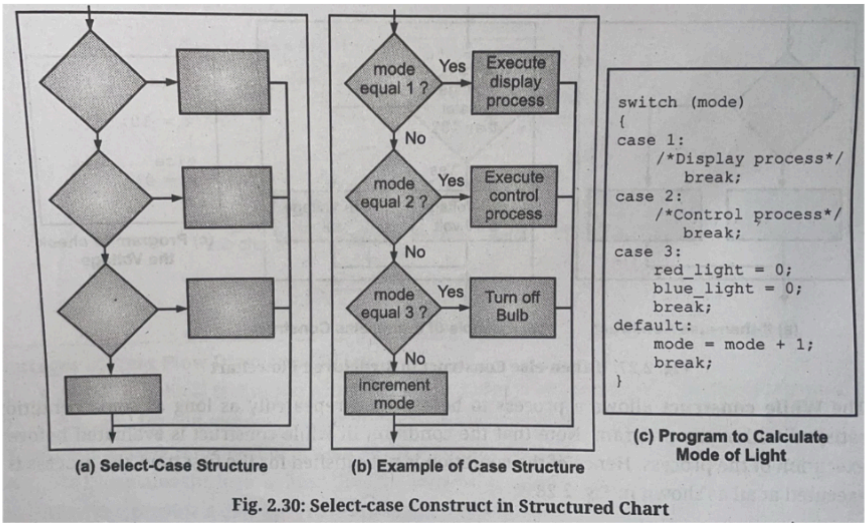


Fig. 2.30: Select-case Construct in Structured Chart

✅ Advantages of Flowcharts:

- Clearly shows the logic of a process
- Helps in **debugging and testing**
- Improves communication between developers and clients
- Visual Clarity
- Effective Problem-Solving: By visualizing the entire process, flowcharts help identify bottlenecks, inefficiencies, and unnecessary steps
- Efficiency in Analysis: They enable teams to analyze workflows systematically, identify flaws logic or execution, and optimize the process

---

📌 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