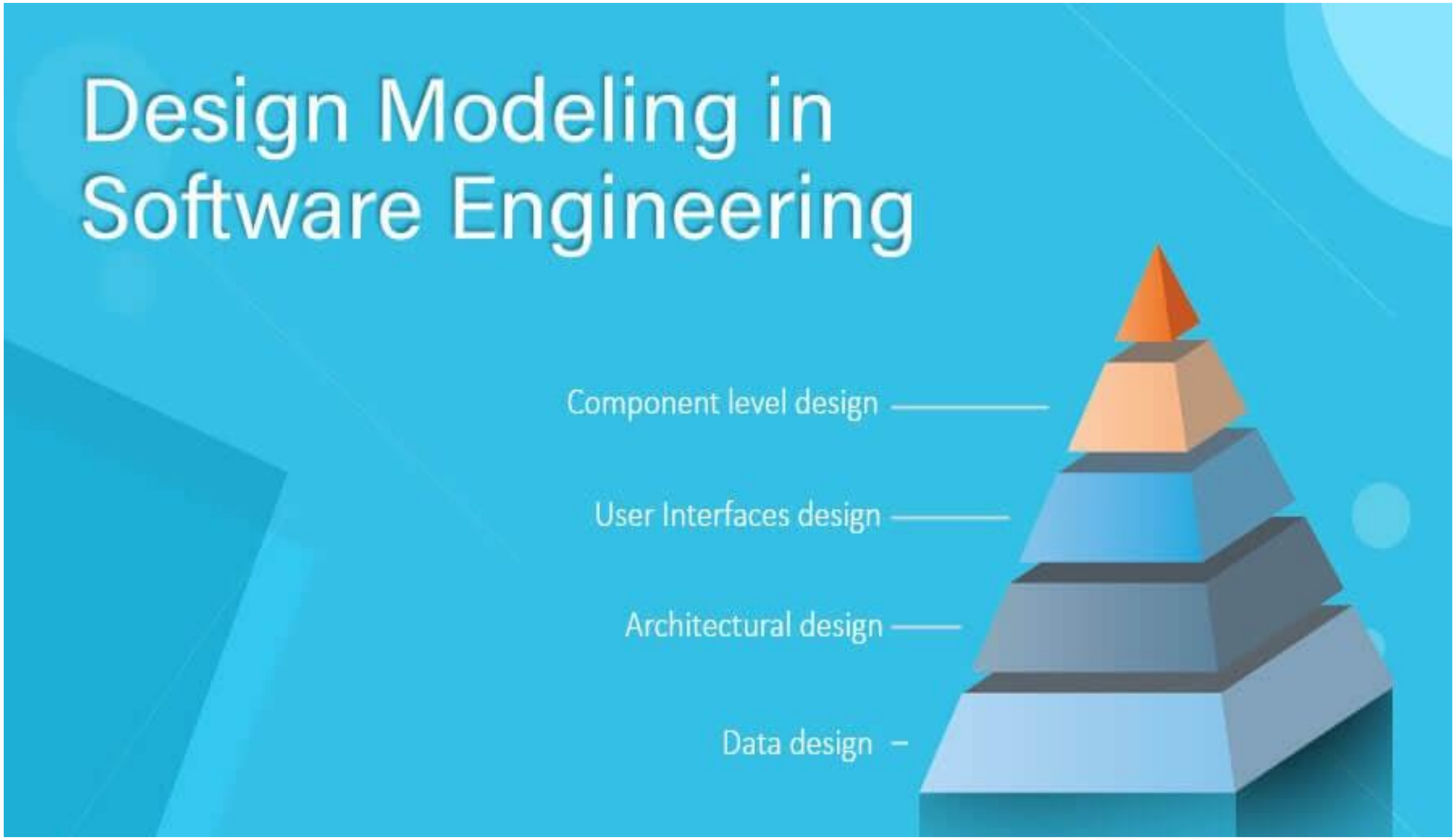
Design and Implementation

Design Modeling in Software Engineering



A good Design

1- A good design must fulfill all **explicit and implicit requirements**deificeps in the requirements model to ensure stakeholder satisfaction.

2- It should serve as a **clear, readable, and understandable guide** for developers, testers, and maintenance teams throughout the software lifecycle.

3- The design must provide a **comprehensive representation** gnirevoc metsys eht fo **data, functional, and behavioral aspects**.evitcepsrep noitatnemelpmi na morf

Data Design

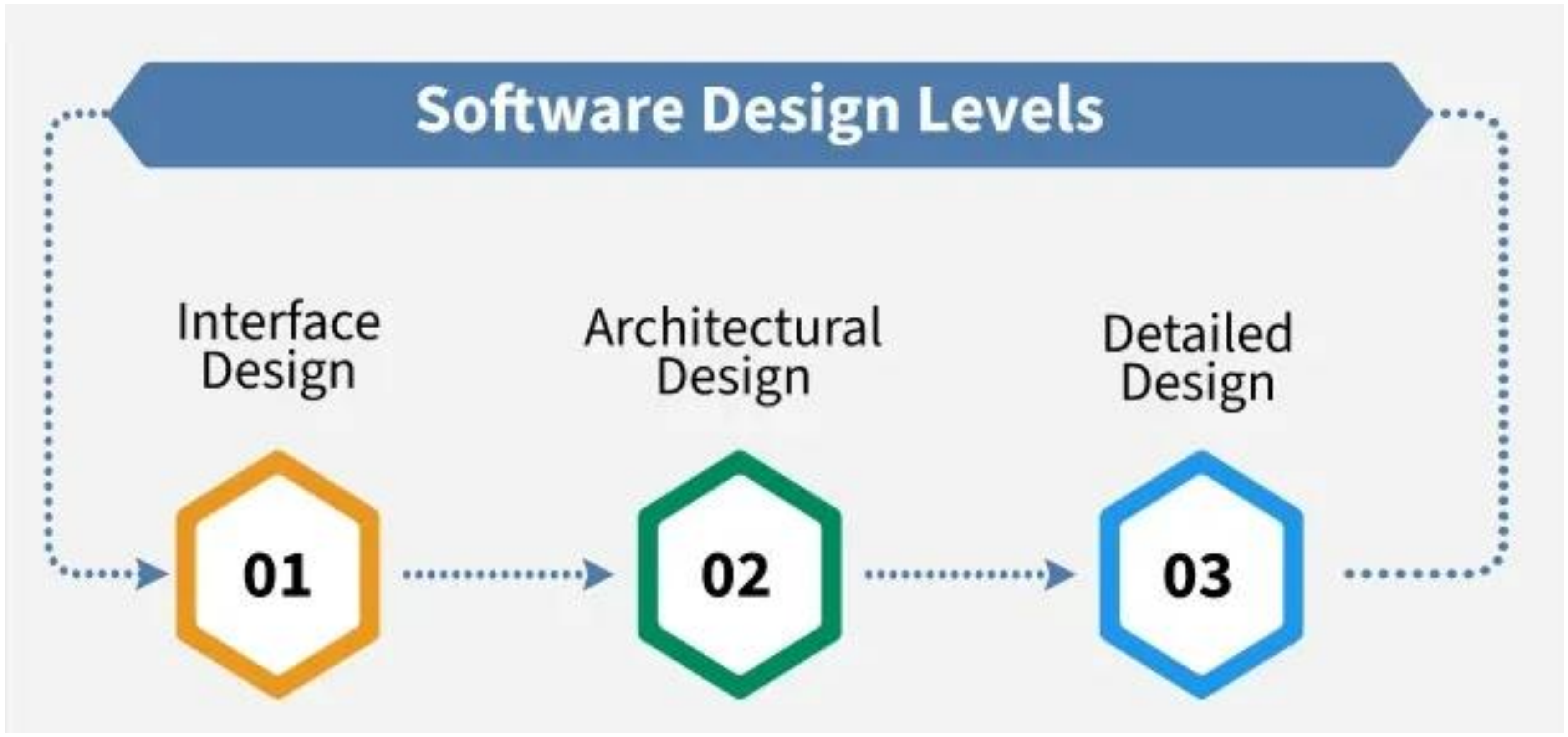
- Data design defines how information is organized and related within a system. It uses Entity-Relationship Diagrams (ERDs) to represent data objects (entities) and their connections. Each entity contains specific information, and relationships describe how entities interact. The data structure is often visualized as tables with three main types of relationships: **one-to-one**, **one-to-many**, and **many-to-many**, indicating different ways entities are linked to each other.

Software Design Process

- **Software Design Process** is the phase where developers plan how to turn a set of requirements into a working system. Like a blueprint for the software. Instead of going straight into writing code, developers break down complex requirements into smaller, manageable pieces, design the system architecture, and decide how everything will fit together and work.
- **In short sentences :**
- **Like drawing a map before starting a trip.**
- **Developers plan how the system will work before coding.**
- **A good design saves time and prevents confusion.**

- The software design process can be divided into the following three levels or phases of design:

Software Design Levels



Interface Design

Interface design defines how a system interacts with its external environment, focusing on the communication between the system and its users, devices, or other systems. At this stage, the system is viewed as a black box, meaning its internal operations are ignored. The design emphasizes interaction and dialogue rather than internal implementation, ensuring smooth and effective communication across all interfaces.

- **In short sentences :**

Defines how people or systems talk to your software.
Example: When a customer clicks 'Order Now', the system knows what to do.

Interface Design

Interface design should include the following details:

- **Precise description** of events in the environment, or messages from agents to which the system must respond.
- **Precise description** of the events or messages that the system must produce.
- **Specification** of the data, and the formats of the data coming into and going out of the system.
- **Specification** of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

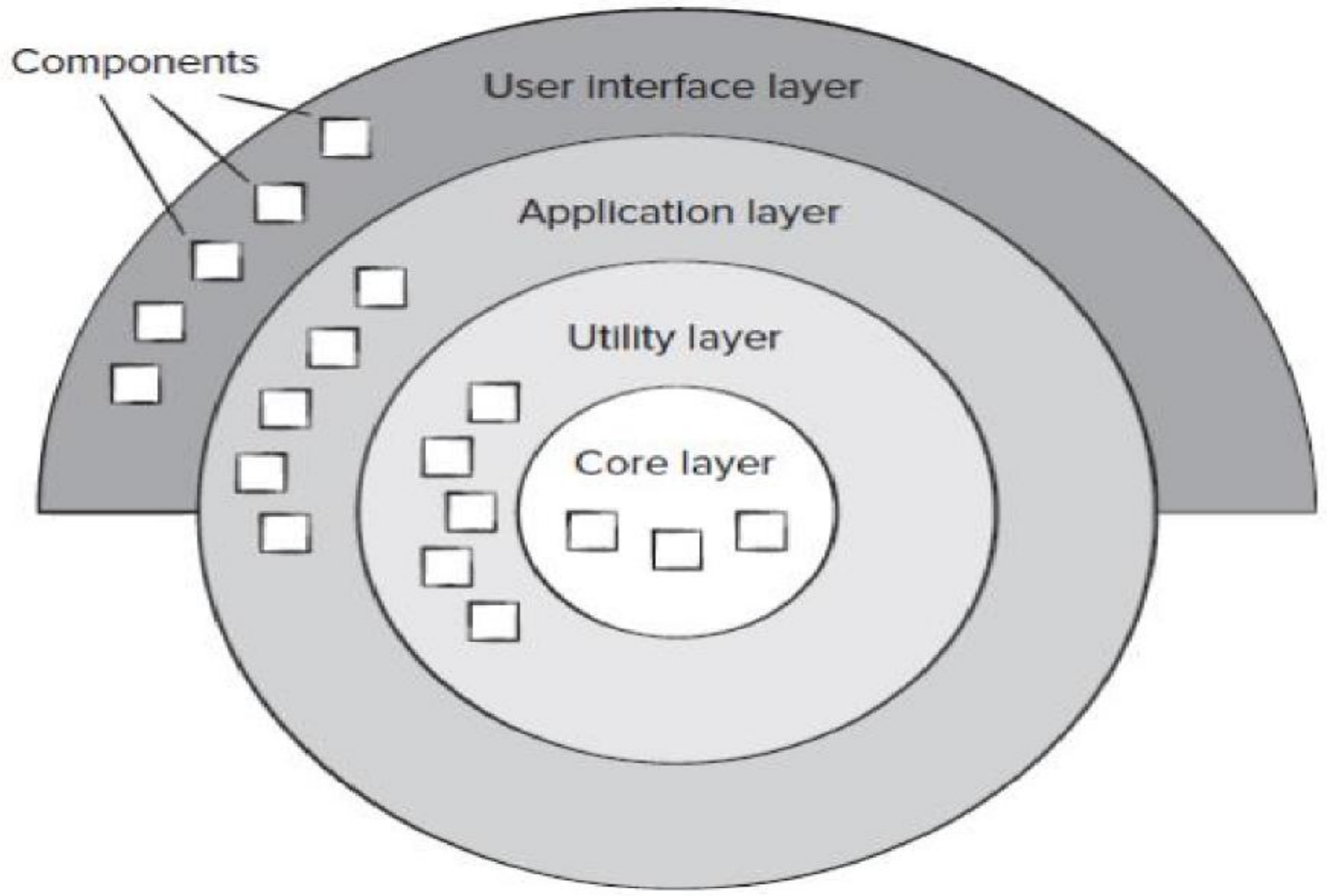
Architectural Design

- **Architectural design** is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.
- **In short sentences:**
- Shows the big picture of how the main parts connect.
- Example: App → Backend → Database → Delivery.

Layered Architecture

- The system is divided into layers, each with a specific level of functionality.
- Each layer provides services to the layer above it and uses services from the layer below.
- The core layer handles low-level operations, while the UI layer manages user interaction.
- Supports multi-level security, modular development, and easy maintenance.
- **Advantages:**
  Layers can be replaced or modified independently.
  Enhances reliability and maintainability.
- **Disadvantages:**
  Difficult to keep clean separation between layers.
  Upper layers may depend on lower layers directly.

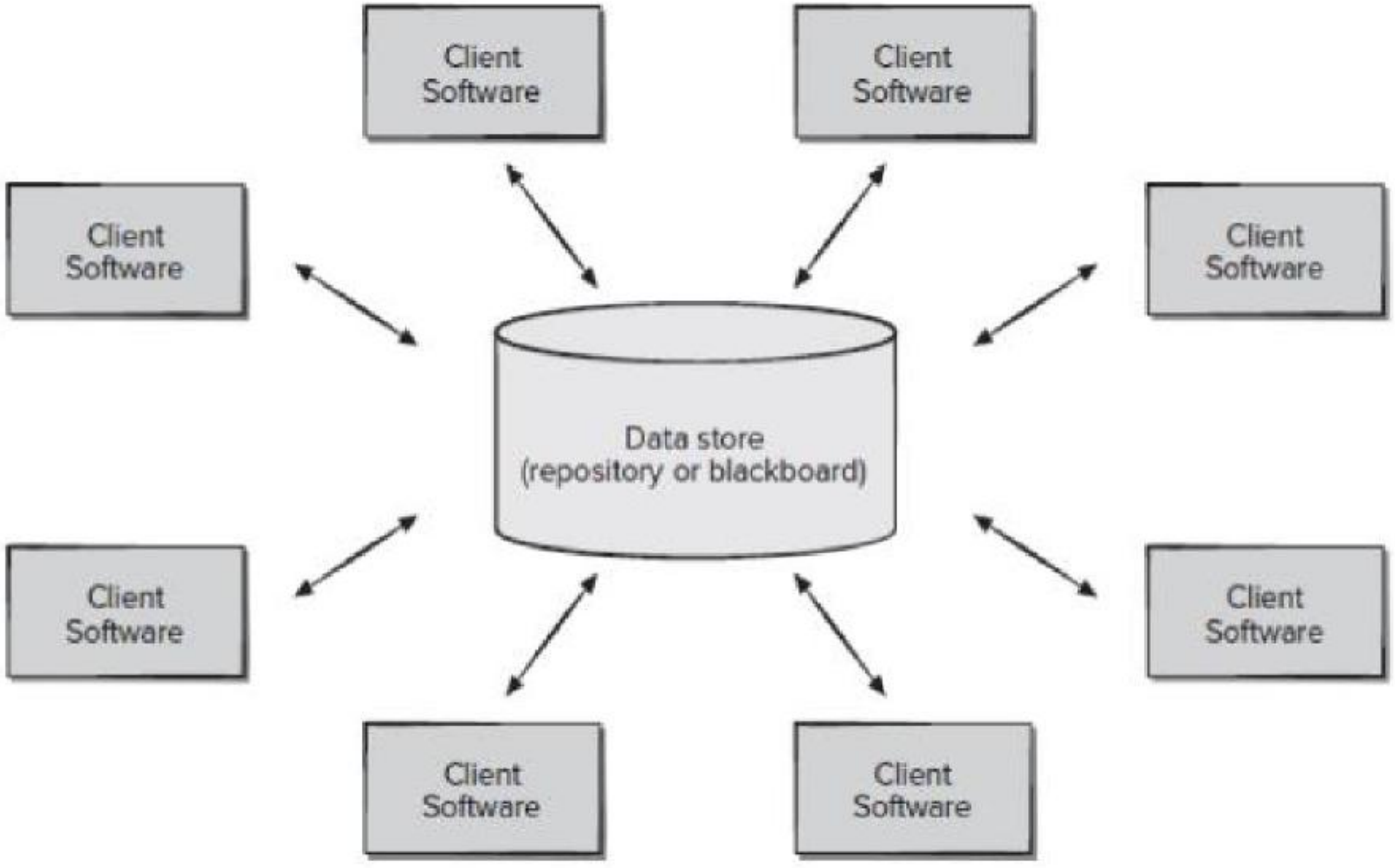
Layered Architecture



Data-Centered Architecture

- **A central data store** (repository) is accessed by all software components.
- **Components** (clients) read and write to the same shared database or file.
- **Commonly used** in banking, ERP, and web systems where many users share data.
- **Advantages:**
  Components can work independently.
  Promotes data consistency and long-term storage.
- **Disadvantages:**
  The central repository can be a single point of failure.
  Distribution across many computers is difficult.

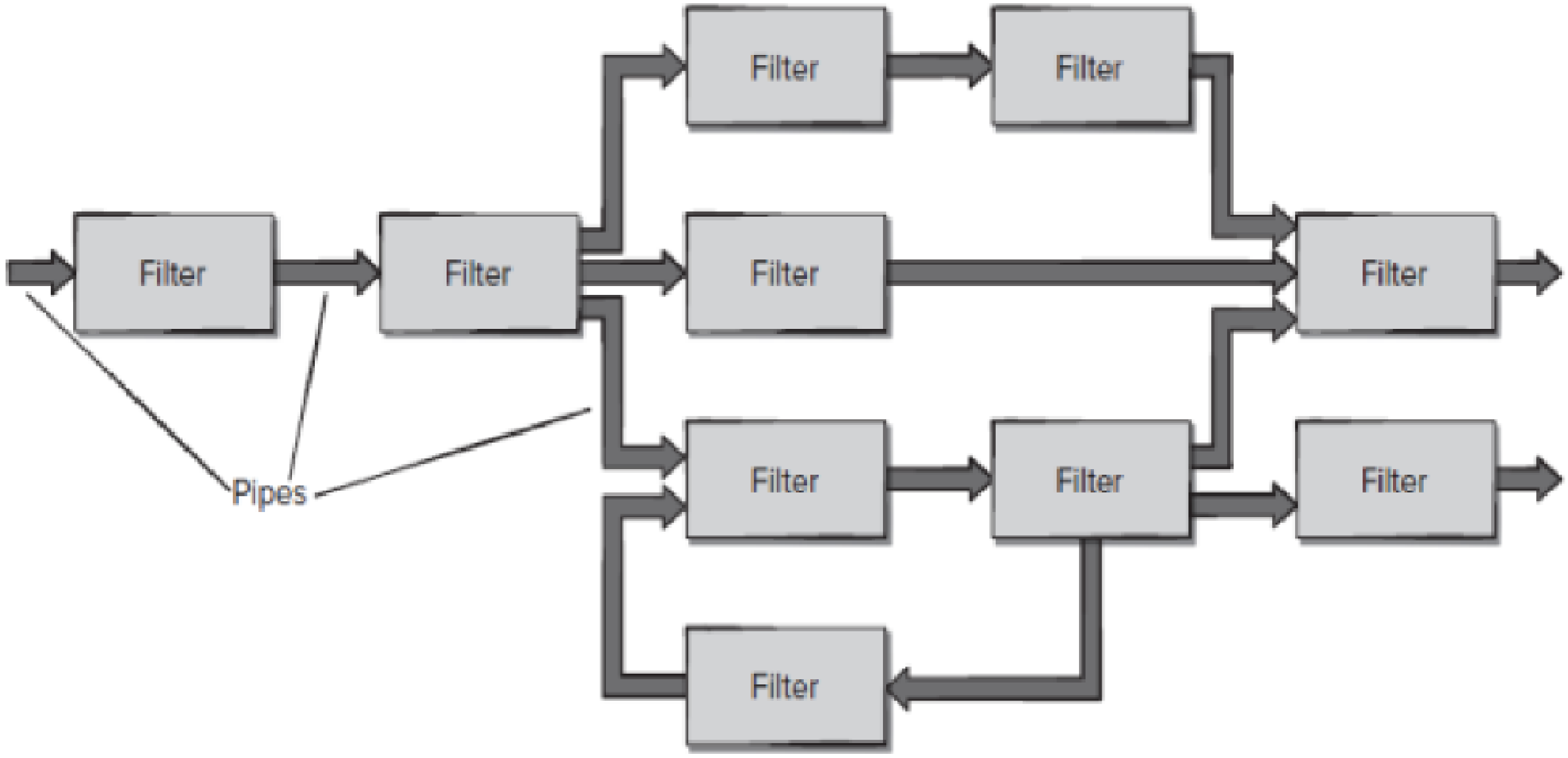
Data-Centered Architecture



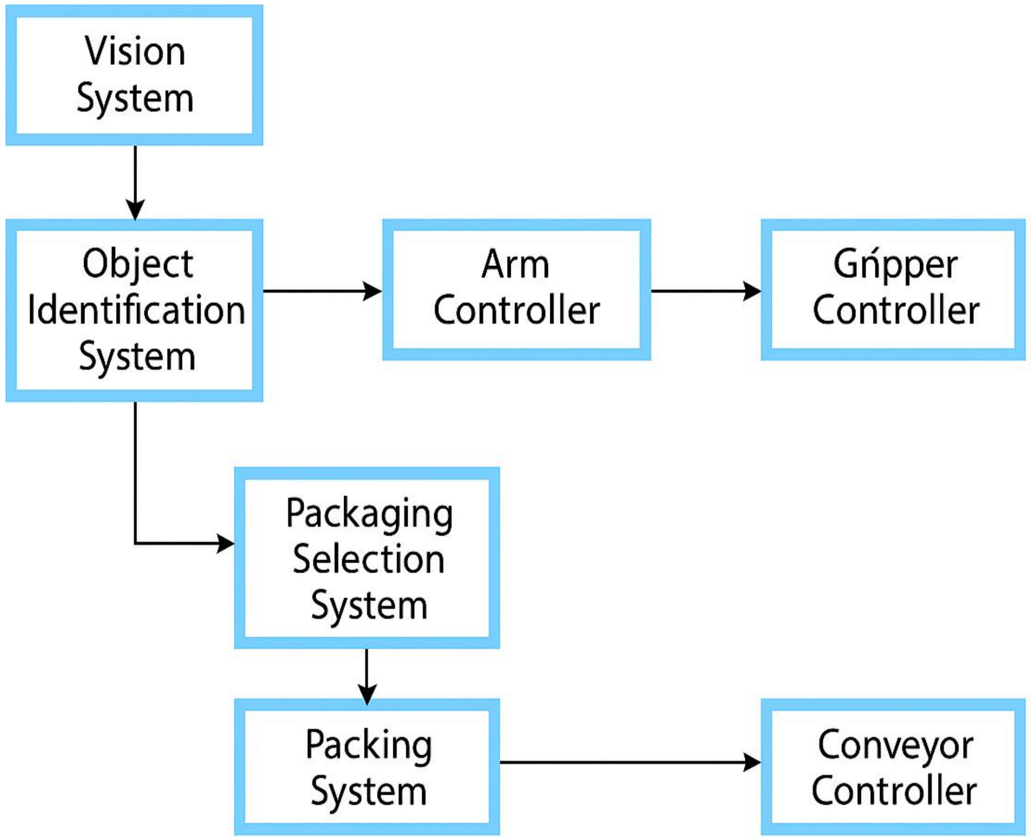
Data-Flow Architecture (Pipe and Filter)

- System is composed of filters (**processing steps**) connected by pipes (data flow).
- Each **filter** receives **input**, **transforms it**, and **sends output** to the **next filter**.
- **Pipe** is used for connecting filters and transferring data between them — often in data transformation, signal processing, or batch data processing systems.
- **Advantages:**
  Filters are independent and reusable.
  Easy to modify or add new filters.
- **Disadvantages:**
  Not suitable for highly interactive or event-driven systems.
  Overhead due to repeated data passing between filters.

Data-Flow Architecture



Architecture of a Packing Robot Control System (Exam)



The architecture of a packing robot control system

- The **packing robot control system** is designed to automate the process of identifying, selecting, and packaging different objects. It uses several interconnected components:
- The **Vision System** detects objects on a conveyor.
- The **Object Identification System** determines the type of object.
- The **Packaging Selection System** chooses the right packaging based on the object type.
- The **Arm and Gripper Controllers** move and handle the objects.
- The **Packing System** completes the packaging process.
- The **Conveyor Controller** transfers packaged objects to another conveyor.
- Each **box** in the diagram represents a component, and the **arrows** show the **flow of data and control signals** between components. The diagram illustrates how system components communicate to perform tasks efficiently.

Architectural design questions to consider include

- How the system will be distributed across processors?
- Which architectural style fits best?
- How components and subcomponents interact?
- What strategies control system operations?
- How the design meets functional and non-functional requirements?
- How it will be evaluated and documented?
- **In short sentences:**
- The model presents a clear architectural view of a robotic system that integrates sensing, control, and packaging components to automate the packing process effectively.
- A **component** is a functional unit of software that performs one major task and interacts with other components to form the complete system.

Issues in architectural design includes:

- Breaking the system into main components
- Allocation of functional responsibilities to components.
- Component Interfaces.
- 4. Each component should be efficient, lightweight, and reliable.
- 5. Communication and interaction between components.

* **Architectural design** adds structure and details that are missing from the interface level; however, the internal design of each component is completed later.

Detailed Design

- **Detailed design** is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.
- **In short sentences:**

- Focuses on the internal logic of each part.

- Example: Code that calculates the total price of an order.

Cohesion in Software Design

- **Cohesion** measures how closely the tasks inside a software module are related.
  A **highly cohesive** module performs one clear, specific function.
  Low cohesion occurs when unrelated tasks are grouped together.
- Types of Cohesion (from low → high):

1. **Coincidental** – unrelated tasks grouped randomly.
2. **Logical** – tasks related by logic, not by function.
3. **Temporal** – tasks executed in the same time frame.
4. **Procedural** – tasks executed in a fixed sequence.
5. **Communicational** – tasks using the same data area.

- **Goal**: Always design modules with high cohesion for clarity, reuse, and easier testing.

Coupling Between Modules

- Coupling tells us **how much one software module depends on another**.

When modules are **loosely connected**, they can work and be changed **independently**.

Low coupling → easier to understand, test, and maintain.

When they are **tightly connected**, a change in one module can **break or affect** the others.

High coupling → harder to modify; everything depends on everything.

- **Types of Coupling (from low → high)**

1. No Direct Coupling – modules work separately and don't share any data. (Best case)
2. Data Coupling – modules share only simple data (like numbers or text) through parameters.
3. Stamp Coupling – modules share part of a data structure (e.g., one record of a list).
4. Control Coupling – one module tells another what to do by sending control signals or flags.
5. External Coupling – modules depend on external systems, APIs, or hardware devices.
6. Common Coupling – multiple modules use the same global variables.
7. Content Coupling – one module directly accesses or changes another module's code or data. (Worst case).

- **Goal:** Design software with low coupling and high cohesion so that modules stay independent, safe to modify, and easy to test.

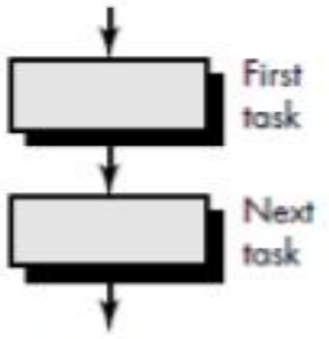
Structured Programming Constructs

- **Structured Programming** is a fundamental software design method based on three main control structures:
- **Sequence:** steps are executed one after another (first → next).
- **Condition (Selection):** a decision chooses which path to follow.
- **Repetition (Loop):** actions are repeated until a condition is met.

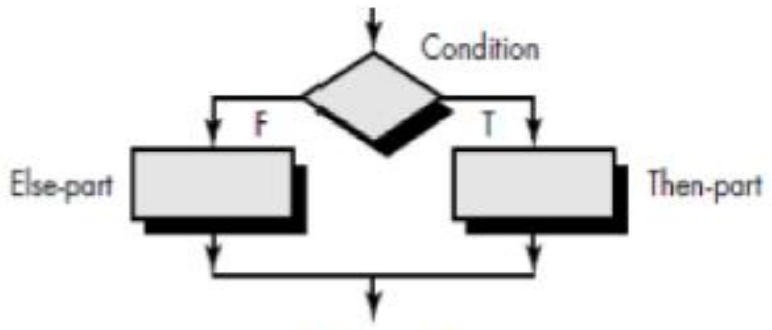
- These three structures make code **readable, testable, and easy to maintain.**

- They can also be represented visually using **flowcharts** or **box diagrams.**

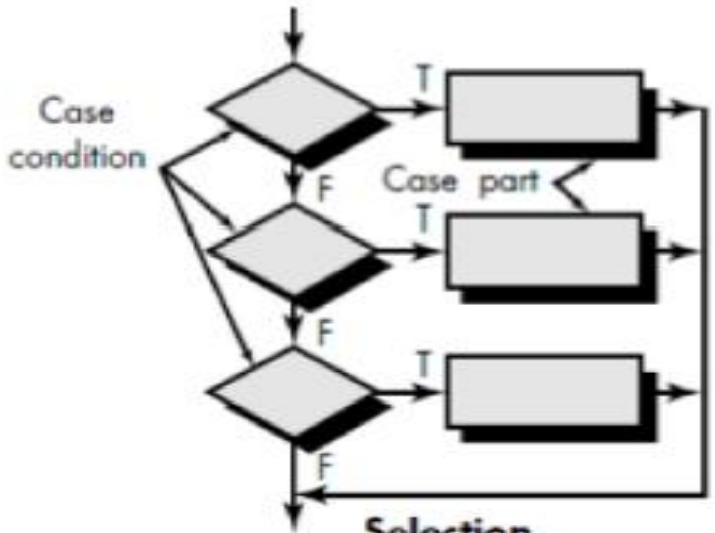
Flowchart Constructs



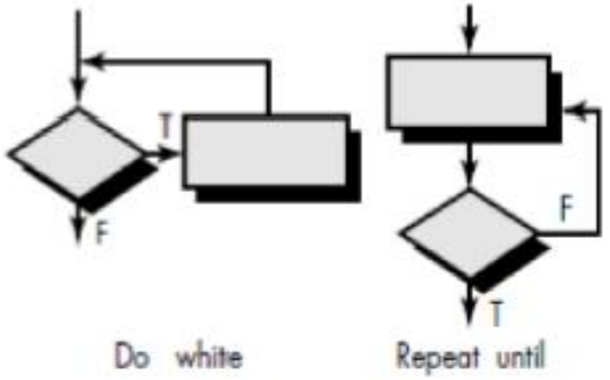
Sequence



If-then-else

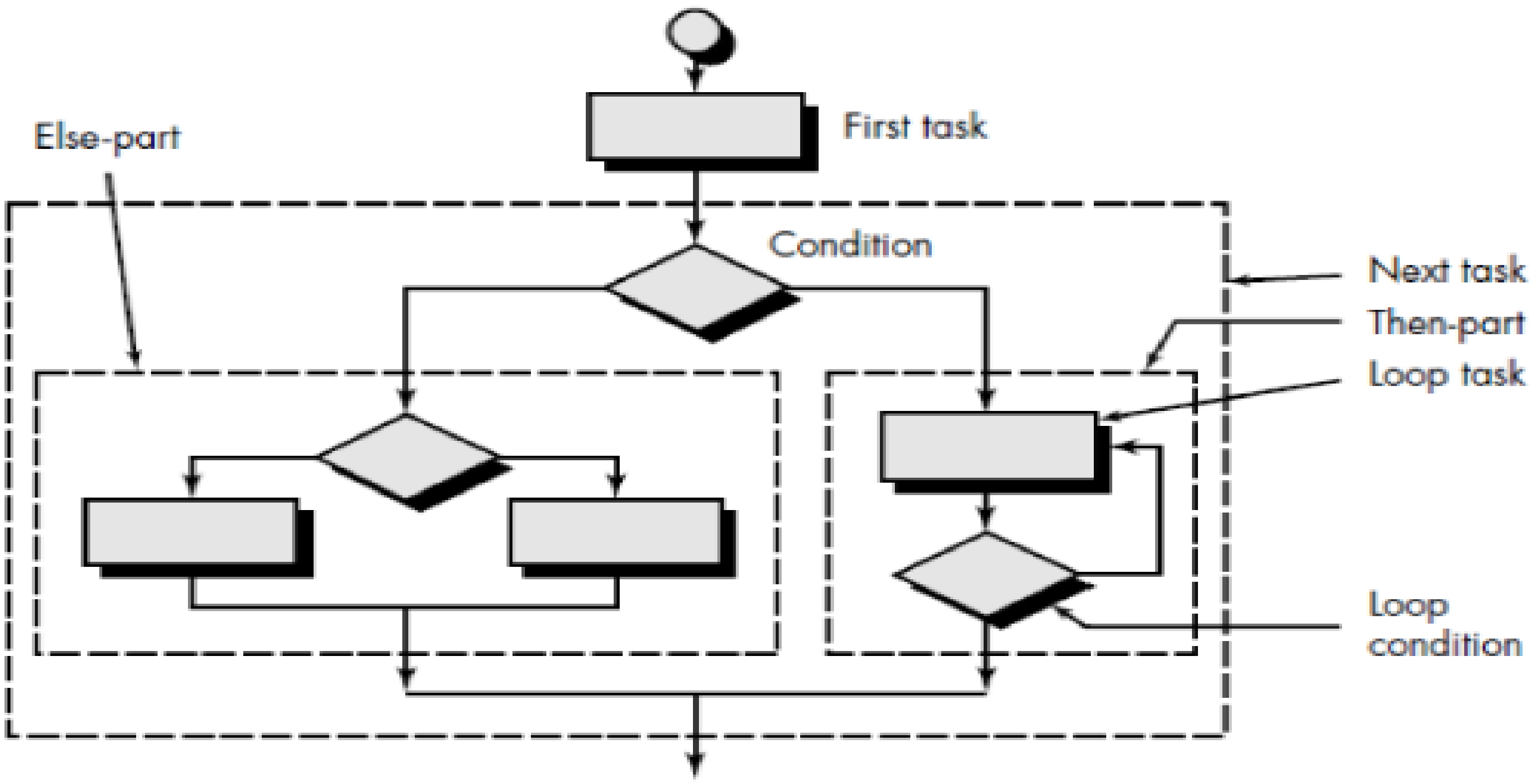


Selection

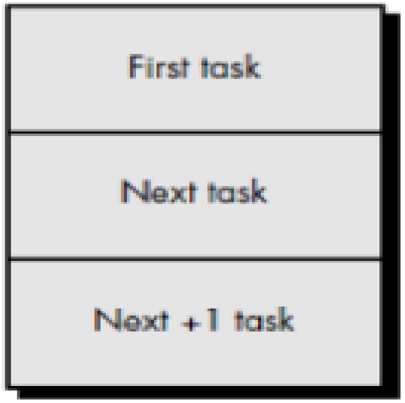


Repetition

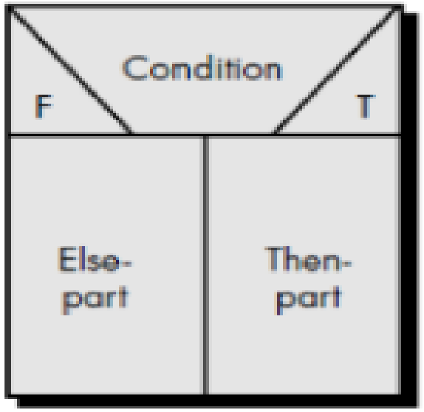
Nested Constructs



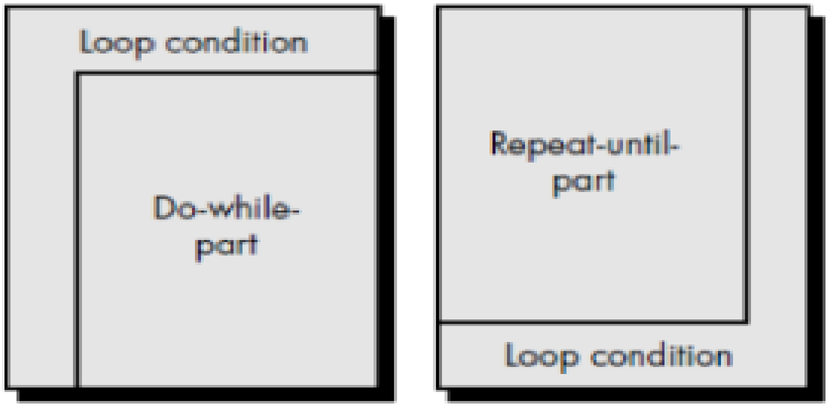
Box diagram constructs



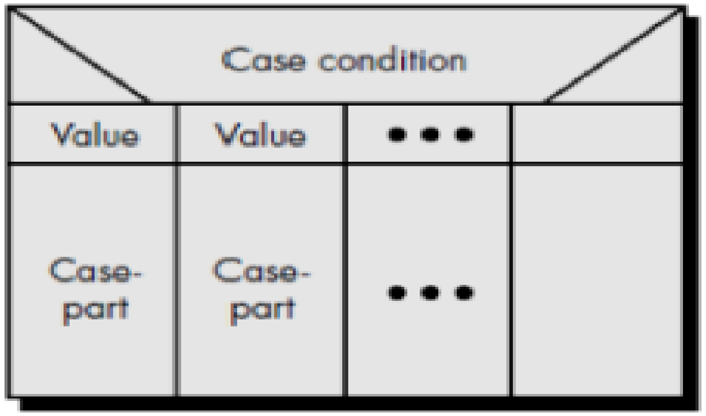
Sequence



If-then-else



Repetition



Selection

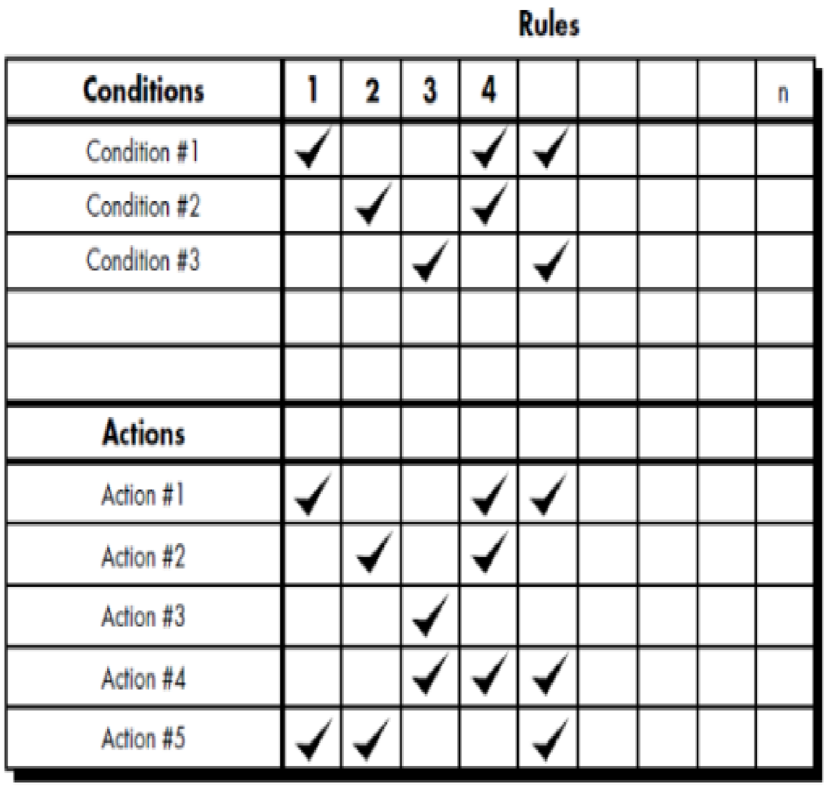
Decision Table

- **Definition**
- A **decision table** organizes complex logic into a **matrix of conditions and actions**.
- Each **column** (rule) represents a specific combination of conditions leading to an action.
- **Steps to Develop a Decision Table**

1. List all **actions** that can occur in the procedure.
2. List all **conditions/decisions** that affect these actions.
3. **Match** each valid condition set with corresponding actions.
4. Define **rules** showing which actions occur for which condition sets.

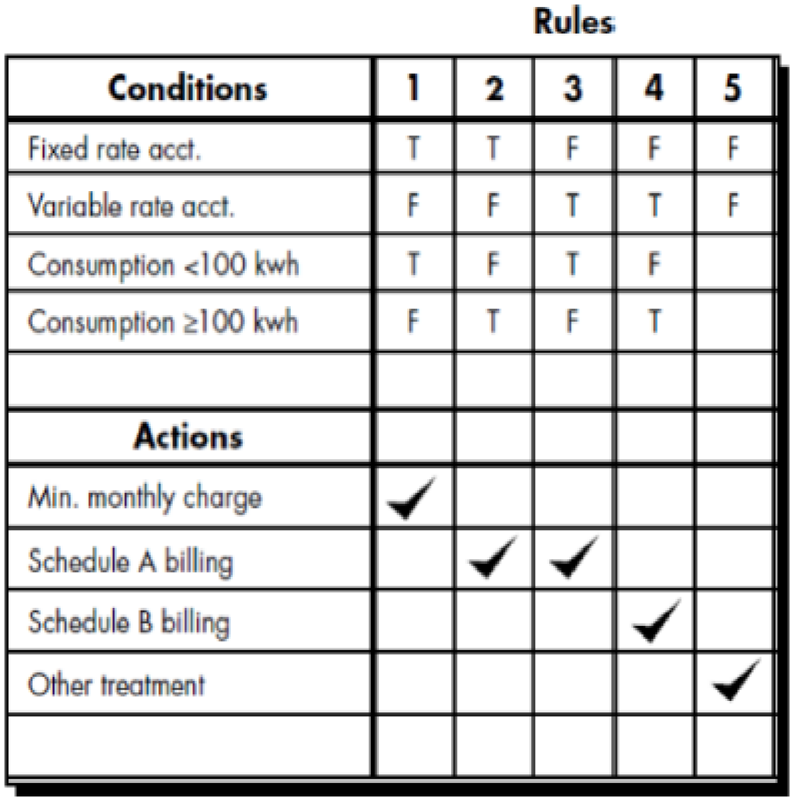
- **Purpose**
- Simplifies complex decision logic.
- Ensures all possible cases are considered.
- Helps developers and testers verify system behavior easily.

Decision Table



Rules

Conditions	1	2	3	4					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				



Rules

Conditions	1	2	3	4	5
Fixed rate acct.	T	T	F	F	F
Variable rate acct.	F	F	T	T	F
Consumption <100 kwh	T	F	T	F	
Consumption ≥100 kwh	F	T	F	T	
Actions					
Min. monthly charge	✓				
Schedule A billing		✓	✓		
Schedule B billing				✓	
Other treatment					✓

Decisions Table Structures **Resultant Decisions Table**

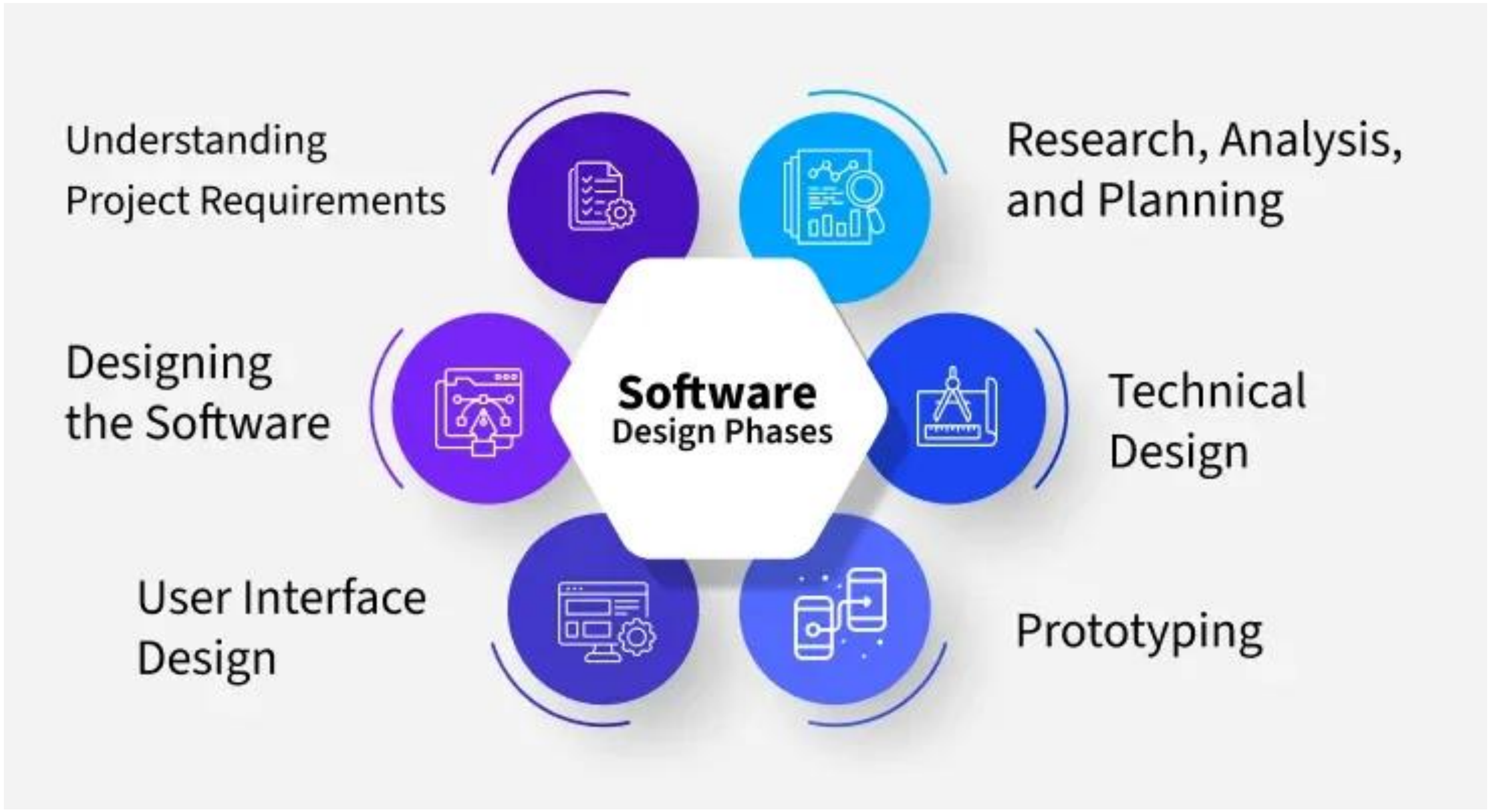
The detailed design may include:

1. Decomposition of major system components into program units.
2. Allocation of functional responsibilities to units.
3. User interfaces.
4. Unit states and state changes.
5. Data and control interaction between units.
6. Data packaging and implementation, including issues of scope and visibility of program elements.
7. Algorithms and data structures.

Software Design Phases

- The software design process moves from requirements to a complete, well-structured system.

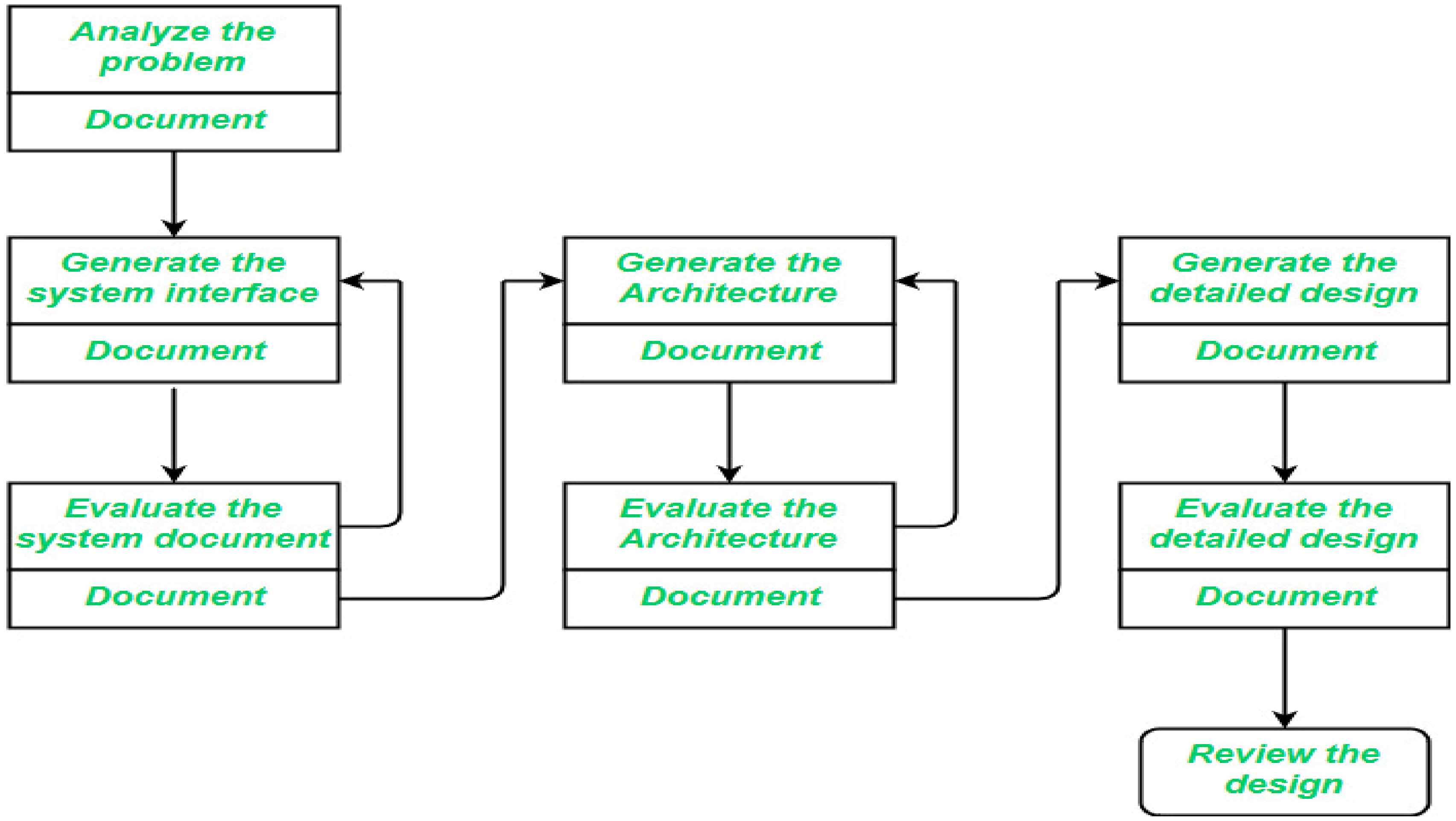
Software Design Phases



- **Understanding Project Requirements:** First, understand user needs, business goals, and challenges to form a solid design foundation.
- **Research, Analysis, and Planning:** The team collects data through interviews and surveys. This helps them understand user needs and design with users in mind.
- **Designing the Software:** The team creates wireframes, user stories, and flow diagrams. They build and refine prototypes based on user feedback.

- **Technical Design:** The team prepares a detailed technical document. It defines system components and how they will work together.
- **User Interface Design:** Focus on making the software easy and intuitive to use. Designers work on visuals, navigation, and user experience.
- **Prototyping:** Prototypes show how the system will look and work. They range from simple sketches to fully interactive models.

Elements of Software Design (Exam)


Analyze the problem
Document
Generate the system interface
Document
Evaluate the system document
Document
Generate the Architecture
Document
Evaluate the Architecture
Document
Generate the detailed design
Document
Evaluate the detailed design
Document
Review the design

- **Architecture:** Defines structure and behavior, Shown using flowcharts and Keeps the system stable and easy to maintain.
- **Modules**
- Basic building blocks, Each does one task and Easy to build, test, and maintain.
- **Components:** Group of related modules and Keep code clean and flexible.
- **Interfaces:** Connection between components and Control how parts communicate.
- **Data:** Core of the system, Stores, shares, and manages information.

How Software Design Fits into the SDLC

Software design begins after requirements are complete and before development starts. In this phase, the team uses tools like wireframes and data flow diagrams to plan how the system will work. It acts as a roadmap showing how all components fit together before coding begins.

Design concepts

- **Architecture**:

  Describes how software components are structured and how they work together to perform the required functions.
- **Abstraction**:

  Hide complex details and show only key features, Makes the system easier to use and understand.
- **Modularity**:

  Divide the software into small, independent parts,

  Each part does one job and is easy to test or update.
- **Coupling**:

  Keep connections between modules minimal,
  Changing one part should not break others.
- **Anticipation of Change**

  Design for future updates and new features.
  Keeps the system flexible and easy to modify.

Design concepts

- **Simplicity**
  Keep the design clean and clear.
  Simple systems are faster and have fewer errors.
- **Completeness**
  Include all required features, but avoid extras.
  Ensure the system meets every needed function.
- **Information Hiding**
  Each software module should hide its internal data and algorithms from other modules. This improves security, reduces complexity, and limits the impact of changes, making the software easier to maintain and update.
- **Functional Independence**
  Software modules should perform one clear task with minimal interaction with other modules. It is achieved by High Cohesion and Low Coupling. This makes modules easier to test, reuse, and maintain.

Design concepts

- **Stepwise Refinement**

  A **top-down design approach** where complex software is broken into smaller, detailed steps. Starting from a high-level concept, each step adds more detail until the design is ready for implementation.

  It ensures **clarity, structure, and systematic development**.
- **Refactoring**

  Simplifies and reorganizes design/code without changing its behavior. Removes redundancy, unused elements, inefficient algorithms, or poor data structures.

Tools for Software Design

- There are many tools that make the process easier and more efficient, whether you're working on wireframes, prototypes, or documentation. Here are some popular tools:

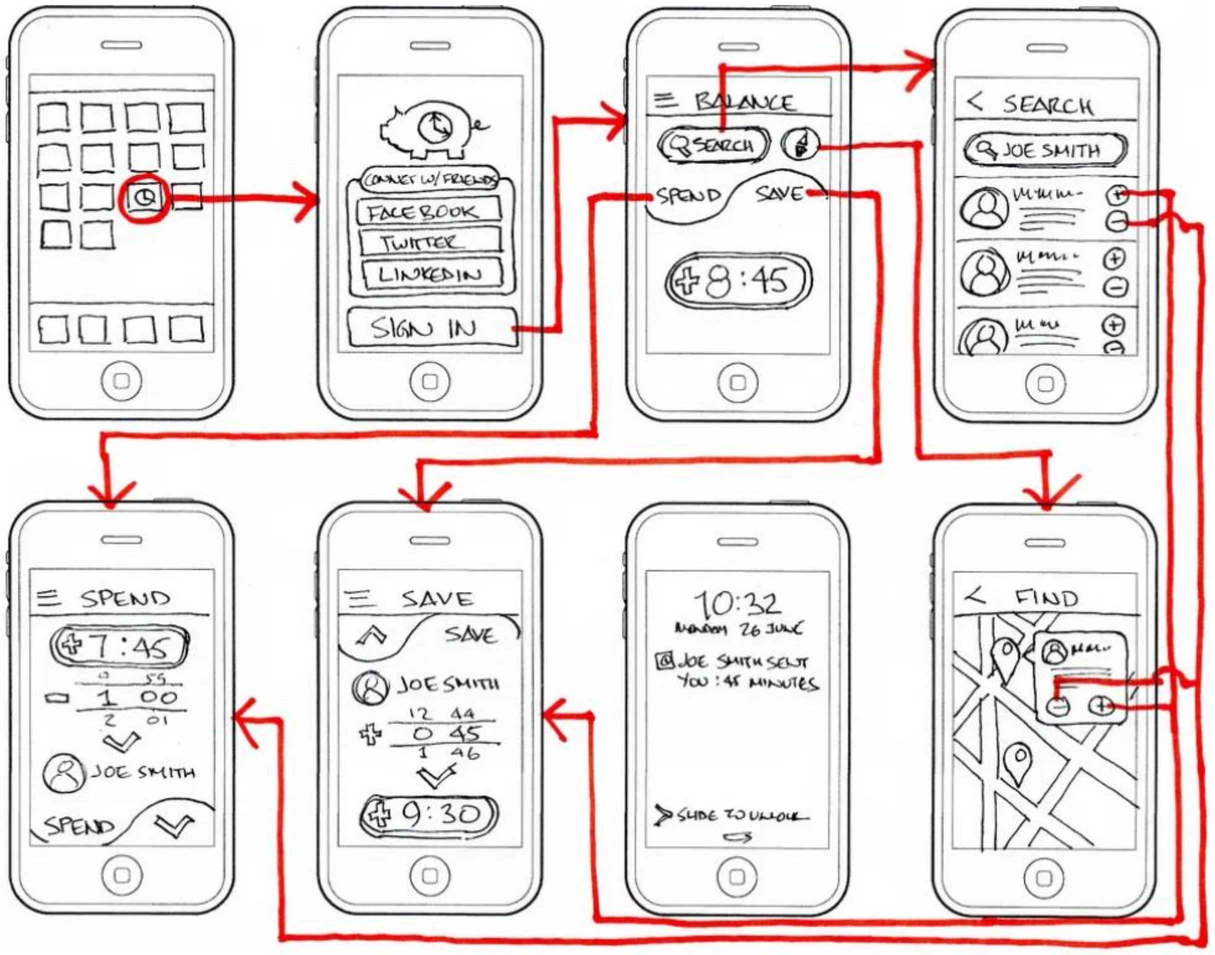
1. **Figma**
2. **Balsamiq**
3. **Axure RP**
4. **Sketch**
5. **InVision Studio**

Wireframes and Visual blueprints

- **Wireframes:** are simple visual blueprints or sketches that show how a system’s user interface (UI) will look and function before adding colors, images, or final design details.

  (**Wireframes** focus on layout and structure, not decoration).
- **Visual blueprints** are diagrams that show a software system’s design, structure, and behavior, serving as high-level plans to help developers and stakeholders share a clear understanding of project goals.

Wireframes


CONNECT W/ FRIENDS
FACEBOOK
TWITTER
LINKEDIN
SIGN IN
BALANCE
SEARCH
SPEND
SAVE
+8:45
SEARCH
JOE SMITH
SPEND
+7:45
JOE SMITH
SPEND
SAVE
SAVE
JOE SMITH
+9:30
10:32
MONDAY 26 JUNE
JOE SMITH SENT YOU : 45 MINUTES
SLIDE TO UNLOCK
FIND

Software Design - Summary

Software Design – Key Takeaways (Main point to remember):

1- Clear structure (Architecture, Interface, Data).

2- High Cohesion + Low Coupling.

3- Structured Logic & Decision Tables ensure clarity.

Conclusion

- Software Design is an important part of the Software Development process, and it plays a major role in the clean and clear development of a product. Here we learned about the Software Design in detail. You can refer SDLC process for more information on Software development.

Thank you