S.E. UNIT 4 : DESIGN Engineering

Design Concepts: Design within the Context of Software Engineering, The Design Process, Software Quality Guidelines and Attributes, Design Concepts - Abstraction, Architecture, design Patterns, Separation of Concerns, Modularity, Information Hiding, Functional Independence, Refinement, Aspects, Refactoring, Object-Oriented Design Concept, Design Classes, The Design Model , Data Design Elements, Architectural Design Elements, Interface Design Elements, Component-Level Design Elements, Component Level Design for Web Apps, Content Design at the Component Level, Functional Design at the Component Level, Deployment-Level Design Elements. Architectural Design: Software Architecture, What is Architecture, Why is Architecture Important, Architectural Styles, A brief Taxonomy of Architectural Styles.

🔹 What is Software Design?

Software design is the process of planning and structuring how a software system will work **before it's actually built** (coded). Think of it as creating a **blueprint** for a software application, just like an architect creates one for a building.

🔹 Key Elements Explained:

1. **"Set of principles, concepts, and practices"**
   These refer to the **rules** and **best practices** followed to ensure the design is clean, efficient, and maintainable.
   - Example: *Modular design, separation of concerns, DRY (Don't Repeat Yourself).*

2. **"Leads to a high-quality system or product"**
   Good design helps in building software that:
   - Works as intended
   - Is easy to understand and maintain
   - Performs well and is reliable

3. **"Stakeholder requirements, business needs, and technical considerations come together"**
   Design acts as a **meeting point** where:
   - **Stakeholders** (clients, users) express what they want
   - **Business needs** guide what's practical or profitable
   - **Developers and engineers** apply technical knowledge to make it happen

4. **"Creates a representation or model of the software"**
   This model is like a **visual or written plan** that describes how the system will be built.

    - Can be in the form of diagrams, flowcharts, or pseudo-code

5. **"Design model provides detail about..."**
   The model dives deep into things like:
    - **Software architecture** – the big-picture structure of the system
    - **Data structures** – how information will be stored and accessed
    - **Interfaces** – how different parts of the system or users will interact with it
    - **Components** – the individual building blocks of the software

---

🧱 Design Within the Context of Software Engineering

🔹 **"The importance of software design can be stated with a single word—*quality.*"**

- This means that **good software design is the key to high-quality software.**
- Without proper design, software may become buggy, hard to maintain, and difficult to scale.

🔹 **"Design is the place where quality is fostered in software engineering."**

- The design phase is where **decisions are made** about how the software should be structured, organized, and how components will interact.
- If quality is built into the design, it will show in the final product.
- It's like setting a strong foundation when constructing a building — if the foundation is weak, the whole structure suffers.

🔹 **"Design provides you with representations of software that can be assessed for quality."**

- The design gives us **visuals or models** (like UML diagrams, architecture diagrams, etc.) that can be **reviewed, evaluated, and improved** before any code is written.
- These representations help catch potential problems early.

🔹 **"Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system."**

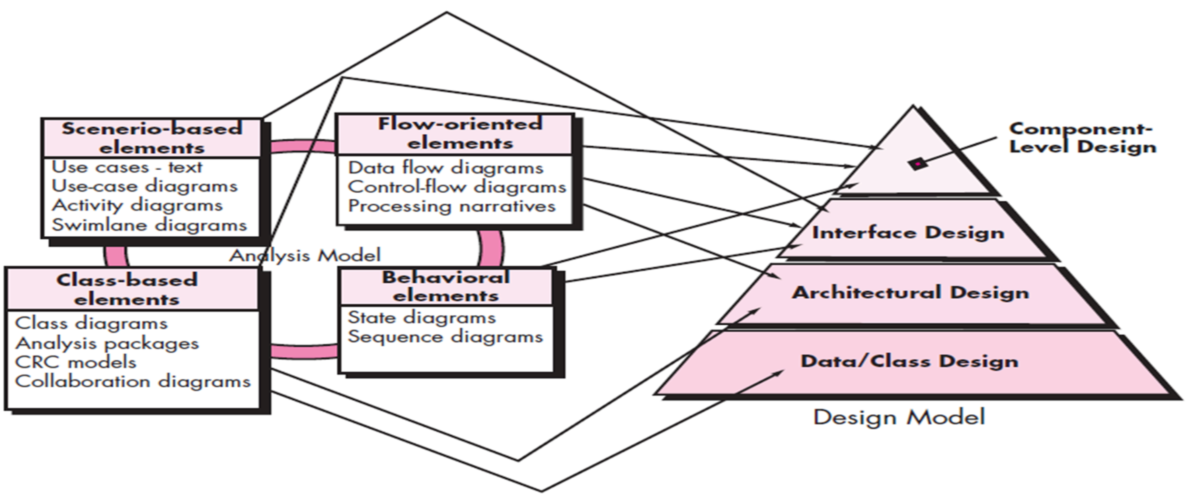
- Stakeholders (clients, users) tell us **what they need**.

- The **design translates those needs** into a plan that developers can follow to build the actual software.
- Without design, there's a risk of **misunderstanding the requirements** or implementing them poorly.

🔹 **"Software design serves as the foundation for all the software engineering and software support activities that follow."**

- Just like you can't build a house without a blueprint, you can't build reliable software without design.

- Every activity — **coding, testing, maintenance, updates** — depends on the design. A poor design leads to more bugs, slower development, and higher maintenance costs.



🔄 The Design Process in Software Engineering

🔹 **"Software design is an iterative process..."**

- **Iterative** means it's **done in cycles**, not just once. You design, review, improve, and repeat.
- Each cycle helps refine the design to better meet the requirements and fix any flaws.

> 🔁 Example: You might sketch out a basic layout for an app, realize it doesn't cover some user needs, then refine it again and again until it's right.

🔹 **"...through which requirements are translated into a 'blueprint' for constructing the software."**

- The **requirements** (what the software should do) are turned into a **detailed plan or design**—just like an architect makes a blueprint for a building before construction begins.

- This blueprint shows how the software will be **structured and behave**.

🔹 **"The design is represented at a high level of abstraction…"**

- This means the design starts from a **broad, overall view** (e.g., system architecture) instead of going straight into fine technical details.
- It focuses on **major components**, their **interactions**, and the **main goals** of the system.

🔹 **"…a level that can be directly traced to the specific system objective…"**

- Everything in the design should have a **clear connection to a system goal**.

- If the system's goal is, say, managing student attendance, the design must reflect components and functions related to that.
-

🔹 **"…and more detailed data, functional, and behavioral requirements."**

- As the design progresses, it becomes more detailed:
  - **Data design** → how data is stored, accessed, and managed.
  - **Functional design** → what each part of the software will do.
  - **Behavioral design** → how the system reacts to inputs or events (e.g., user clicks, errors).

🧩 Software Quality Guidelines and Attributes

✅ Three Characteristics of a Good Design

1. 📋 Fulfillment of Requirements

> *"The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders."*

- **Explicit requirements**: Clearly documented needs (features, functions, constraints).
- **Implicit requirements**: Unspoken expectations (usability, performance, scalability, etc.).

- A good design **must cover both** to ensure the final product is truly complete and satisfactory.

2. 📖 Readability and Understandability

> *"The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software."*

- The design should be:
  - **Clear**: Easy to follow and interpret.
  - **Well-documented**: With diagrams, comments, or specs that explain its parts.
  - **Useful**: For developers (to write code), testers (to create test cases), and support teams (for future maintenance).

3. 🧠 Comprehensive Coverage

> *"The design should provide a complete picture of the software, addressing the data, functional, and behavioural domains from an implementation perspective."*

- A robust design includes:
  - **Data domain**: What data the system will use and how it flows.
  - **Functional domain**: The system's actions or services.
  - **Behavioral domain**: How the system reacts to events, inputs, and interactions.
- This ensures that all major aspects of the system are well-planned and integrated.

🧩 Quality Guidelines for Software Design

These guidelines help ensure that your software design leads to a system that is **high-quality, maintainable, and efficient**.

1. 🏗️ A design should exhibit an architecture

- The design must define a **clear structure** showing how components interact.
- This helps in understanding the **overall layout** and **relationships** within the system.

2. 🧱 A design should be modular

- The system should be divided into **independent modules**, each responsible for a specific part of the functionality.
- Modularity improves **maintainability**, **reusability**, and **testability**.

3. 📊 A design should contain distinct representations of data

- Clearly define how **data is organized, stored, and accessed**.
- Helps in ensuring **data consistency** and **clarity** across the system.

4. 🧬 A design should lead to data structures that are appropriate for the classes

- Choose **data structures** that best support the responsibilities and operations of each **class or component**.
- This ensures **performance efficiency** and proper **data handling**.

5. ⚙️ A design should lead to components that exhibit independent functional characteristics

- Each component should be **focused** and do **one thing well**, with **minimal dependency** on others.
- This improves **reusability** and simplifies **modifications**.

6. 🔌 A design should lead to interfaces that reduce the complexity of connections

- Interfaces between components should be **simple, clear, and well-defined**.
- Reduces **coupling** and allows components to interact more **smoothly**.

7. 🔁 A design should be derived using a repeatable method

- The process of designing should follow a **standard approach or methodology**.
- This promotes **consistency**, **predictability**, and **quality assurance**.

8. 📝 A design should be represented using a notation that effectively communicates its meaning

- Use standard **notations** like **UML diagrams**, **flowcharts**, or **pseudo-code**.
- Helps **communicate** the design to developers, testers, and stakeholders clearly.

🎨 Design Concepts in Software Engineering

Definition:
Software design concepts refer to the **core ideas or principles** used to structure and organize the software system. These concepts are **applicable to both traditional and object-oriented development**.

🔑 Key Design Concepts

Concept	Meaning
Abstraction	Hides unnecessary details; shows only the **essential** features of a system.
Modularity	Breaks the system into **smaller, manageable parts** or modules.
Architecture	Defines the **overall structure** of the system, including components and their interactions.
Refinement	Step-by-step enhancement of a design by **adding more details gradually**.
Pattern	A **reusable solution** to a common design problem (e.g., design patterns like Singleton, MVC).
Information Hiding	Hides **internal details** of modules to reduce complexity and enhance security.

Refactoring	**Improves** internal structure of code/design without changing its behavior.

🧠 Why These Concepts Matter

- They provide a **foundation** for building scalable, maintainable, and efficient software.
- They **reduce complexity** and enhance **reusability, flexibility**, and **understandability** of the system.
- Applying these concepts leads to **cleaner**, more **robust designs**.

🔍 1. Abstraction in Software Design

Definition:
Abstraction is the process of **hiding unnecessary details** and showing **only the relevant information**. It allows you to focus on **what** an object does, instead of **how** it does it. It allows programmers to **work with a simplified view** while hiding low-level details.

📶 Levels of Abstraction

At different levels, the solution is described with varying levels of detail:

🧭 **High-level abstraction:**

- The problem is described in **simple, broad terms** (closer to human understanding).

⚙️ **Low-level abstraction:**

- The solution becomes more **technical and detailed**, closer to actual implementation.

🧩 Types of Abstraction

Type	Definition	Example
Procedural Abstraction	A **named sequence of instructions** that performs a specific task.	A function like printInvoice()—you know it prints the invoice, but not how it formats or fetches the data.
Data Abstraction	A **named data structure** that hides how the data is stored or managed.	A class Car with attributes like speed, fuelLevel—you interact with the object, not with raw memory or variables.

🏛️ 2. Software Architecture

📖 Definition:

> Software architecture is the **overall structure** of a software system and how its components **interact** with each other while maintaining **conceptual integrity**.

In simple terms:

- What parts make up the system
- How those parts work together
- How data flows between them

🧱 What It Includes:

1. **Program Components**
   (like modules, classes, services, functions)
2. **Interactions Between Components**
   (how data or control flows from one to another — e.g., via APIs, function calls, or messages)
3. **Data Structure and Flow**
   (how information is organized and passed)

🏗️ Real-Life Analogy: Building Architecture

Think of building a house:

- **Rooms, kitchen, bathrooms** = program components
- **Doors, hallways, plumbing, wiring** = interactions between components
- **Furniture placement, water pipes, electric circuits** = data structures and flow

Just as an architect plans how each part of the house fits and works together, a **software architect** plans how software components are organized and communicated to achieve system goals.

When designing software architecture, we consider not only **what the system is made of** but also **how it behaves**, **how well it performs**, and **how reusable it is**. These aspects are described using the following **three main properties**:

1️⃣ Structural Properties

- **What it means**:
  This refers to the **components** of a system (like modules, objects, or services) and **how they are connected or interact**.

- **Purpose**:
  It helps understand how the **system is organized** and how different parts **work together**.

- **Example**:
  In a food delivery app, structural components might include:
  - User Interface (UI)
  - Backend Services
  - Payment Module
  - Delivery Tracking Module
    The way these modules **talk to each other** (e.g., via API calls) defines the structural properties.

2️⃣ Extra-Functional Properties (also called Non-Functional Requirements)

- **What it means**:
  These describe how the system should behave in terms of **performance, security, reliability, scalability, and adaptability**.

- **Purpose**:
  Even if a system works correctly, it must also meet **quality expectations**.

- **Example**:
  For an online banking system:
  - It must respond in **under 2 seconds** (performance)

    - It must be **safe from cyberattacks** (security)
    - It must be available **24/7** (reliability)

3️⃣Families of Related Systems (Reusability)

- **What it means**:
  Architecture should support **reuse of design patterns** or **components** in **similar projects**.
- **Purpose**:
  To save time, reduce cost, and maintain consistency across different but related systems.
- **Example**:
  A company building multiple mobile apps (like a shopping app, food app, and pharmacy app) can **reuse the same login, user profile, or payment module**.

Architectural design can be **visualized or described** using different types of **models**, depending on what aspect of the system you're focusing on. Each model helps stakeholders understand the system from a specific perspective.

1️⃣Structural Models

- **What it does**: Shows the **components** of the system and **how they are organized**.
- **Focus**: Structure of the software.
- **Example**:
  A model showing modules like Login, Dashboard, Database, and how they are connected.

2️⃣Framework Models

- **What it does**: Identifies **standard templates** or **frameworks** that can be reused.
- **Focus**: Reusability and standardization.
- **Example**:
  Using the **Model-View-Controller (MVC)** framework in web development.

3️⃣Dynamic Models

- **What it does**: Describes the **runtime behavior** of the system.
- **Focus**: Interaction, events, and changes over time.
- **Example**:
  A sequence diagram showing how a user login request flows through the system.

4️⃣ Process Models

- **What it does**: Focuses on the **workflow** or **business logic** the system supports.
- **Focus**: Business or technical process.
- **Example**:
  A flowchart showing the steps involved in an online order from placing the order to delivery.

5️⃣ Functional Models

- **What it does**: Represents the **functions or operations** of a system in a **hierarchical way**.
- **Focus**: What the system does (functionality).
- **Example**:
  A function tree showing main functions like Manage Users, Process Payment

---

🎯 3. Patterns in Software Design

🔹 What is a Design Pattern?

> A **design pattern** is a **standard solution** to a **common problem** that occurs repeatedly in software design.

Brad Appleton defines it as:"A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context among competing concerns."

🔍 Why Use Patterns?

Design patterns help software designers by providing **tested, reusable solutions**. Think of them as **templates** that can be applied to common challenges during software development.

✅ Design Pattern Helps You Decide:

1. **Applicability**
   - 🔹 Can I use this pattern for the current problem?
2. **Reusability**
   - 🔹 Is this pattern reusable in other parts of the software or future projects?

3. **Adaptability**
   🔹 Can I use this pattern as a **guide** to create something similar but slightly different?

🧠 Real-Life Analogy:

Think of a design pattern like a **blueprint for building a house.**
You can reuse the same blueprint to build multiple houses, maybe with slight changes, like adding an extra room or changing the material—but the overall structure remains reliable and tested.

🧩 4. Separation of Concerns (SoC)

🔍 What is it?

Separation of Concerns is a software design principle that says:

> 💡 **"Break down a big, complex problem into smaller, manageable parts, where each part handles a specific concern (task or feature)."**

🔹 What is a "Concern"?

A **concern** is a **specific functionality, feature, or behavior** in your system.
Examples of concerns include:

- User interface
- Business logic
- Data storage
- Security
- Logging

🛠️ Why Separate Concerns?

By **separating** each concern:

- The software becomes **easier to understand**
- You can **change one part** without affecting others
- The code is **cleaner, reusable, and easier to maintain**

🧠 Real-Life Example:

Think of a **restaurant kitchen**:

- The **chef** cooks food (business logic)
- The **waiter** takes orders and serves (user interface)
- The **cashier** handles billing (data handling)

Each person focuses on their **own concern**. If you replace the waiter, the chef doesn't need to change anything!

🧩 5. Modularity

🔍 What is it?

Modularity is the design principle where a software system is **divided into separate, self-contained units** called **modules**.

> 💡 Each **module** is a building block that performs a specific function and can be developed, tested, and maintained **independently**.

🔄 How it's related to Separation of Concerns:

Modularity is the **practical application** of the *Separation of Concerns* principle.

- While *Separation of Concerns* is a **concept**,
- *Modularity* is how we **implement** that concept—by splitting code into modules.

🧱 What is a Module?

A **module** is a **self-contained component** of a software system that:

- Has a **specific task**
- Has a **name and interface**
- Can be reused and updated independently

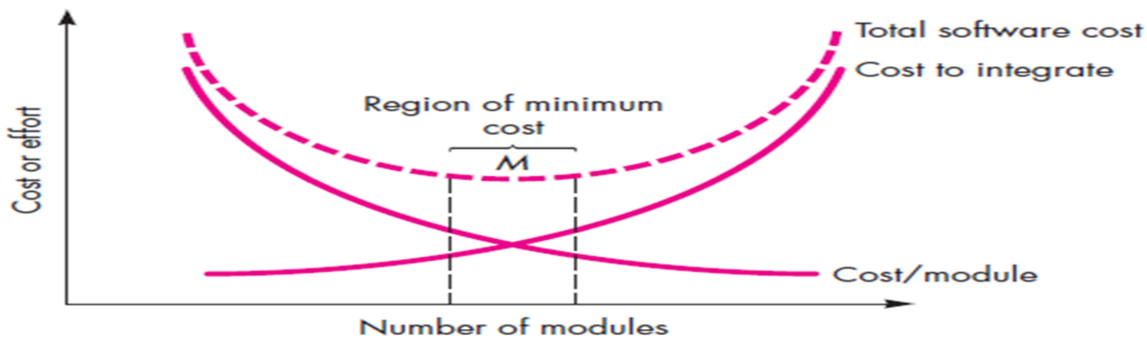
Examples:

- A login module
- A payment processing module
- A data validation module

🧠 Real-Life Analogy:

Think of a **car**:

- The **engine**, **wheels**, **brakes**, and **radio** are all **modules**.
- Each one can be developed or repaired **separately**, but together they form a **complete system**.



🔐 6. Information Hiding

💡 Definition:

Information Hiding is a design principle that ensures **internal details** of a module (like data structures and algorithms) are **hidden** from other modules.

> "Each module should hide *how* it works and only expose *what* it does."

Other modules should **only interact** with a module through its **public interface**—they shouldn't know or depend on its internal workings.

💻 Example:

Imagine a class in Java:

```
public class BankAccount {
    private double balance; // hidden data
    public void deposit(double amount) {
        balance += amount;
    }
```

```
    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        }
    }
}
```

- The balance is **hidden** (private).
- Other parts of the program can't directly modify it—they **must** use deposit() or withdraw().

🧠 Real-Life Analogy:

Think of a **vending machine**:

- You don't know how it processes your input internally.
- You only know the **interface**: insert coin, press button, get snack.
- The internal mechanics are **hidden** from you.

✅ Benefits of Information Hiding:

- 🔄 **Encapsulation**: keeps modules independent
- 🧪 **Easier to test and debug**: because each module is self-contained
- 🔁 **Improves reusability**: internals can change without affecting others
- 🔧 **Simplifies maintenance**: changes are localized

⚙️ 7. Functional Independence

💡 Definition:

Functional Independence means that each software module performs **one specific task** and does it **without relying heavily on other modules**.

📌 Key Idea:

A functionally independent module:

- Solves a **specific part** of the overall problem.
- Has a **simple, limited interface** to interact with other modules.
- Minimizes dependencies.

🔄 Relation to Other Concepts:

It is built upon:

- **Separation of Concerns** (each part solves one concern),
- **Modularity** (broken into components),
- **Abstraction** (focus on "what", not "how"),
- **Information Hiding** (internal details are hidden)

💻 Example:

In a **Library Management System**, you might have modules like:

- BookInventory: Manages books in the library.
- UserAccount: Handles member details.
- IssueReturn: Deals with issuing/returning books.

Each module:

- Performs one job.
- Can be tested and changed without breaking the others.

🧠 Real-Life Analogy:

Think of a **car**:

- The engine, brakes, and air conditioner are **independent components**.
- Each one does its job and has a simple interface (e.g., press the brake pedal).
- You can fix or upgrade the AC without touching the engine.

✅ Advantages of Functional Independence:

- 🧪 Easier to **test** (each part can be verified independently)
- 🛠️ Easier to **maintain** (fix one part without affecting others)
- 🔁 Easier to **reuse** (independent modules can be used elsewhere)
- ⚙️ Easier to **understand and develop** (divide and conquer)

🔧 8. Refinement

💡 Definition:

Refinement is the process of gradually adding more detail to a software design. It starts with a high-level overview and breaks it down step by step into smaller, more precise parts.

🔄 Stepwise Refinement:

- It follows a **top-down design** approach.
- You start with a general function or requirement.
- Then, **break it down** into smaller, more detailed parts.
- Repeat this process until each part is detailed enough to be implemented in code.

🧱 Hierarchy of Detail:

Example:

1. **High-level goal:** Process online order

2. Refine into sub-tasks:
    - Validate payment
    - Update inventory
    - Send confirmation email

3. Refine "Validate payment" into:
    - Check card details
    - Contact payment gateway
    - Confirm transaction

You continue refining until you reach the actual lines of code.

🧠 Real-Life Analogy:

Think of building a **house**:

- Start with a sketch.
- Then refine it into floor plans.
- Break that down into electrical, plumbing, and structural layouts.
- Finally, individual tasks like "install kitchen tap" or "wire bedroom light".

✅ Benefits of Refinement:

- Makes design **easier to manage**
- Helps in **clear understanding** of each component

- Encourages **organized and logical development**
- Bridges the gap between **requirements** and **code**

🧩 9. Aspects

💡 Definition:

An **Aspect** represents a **crosscutting concern** in a software system — a concern that **affects multiple parts** of the application but doesn't belong to just one module.

📌 Key Concepts:

- A **concern** is any requirement or feature in the software (e.g., logging, security, error handling).
- A **crosscutting concern** is a concern that **spans multiple modules**, making it hard to isolate using traditional modularization.
- An **aspect** allows you to **isolate and manage** these concerns **separately**, improving maintainability and design clarity.

🎯 Why It Matters:

- Helps avoid **code duplication** across modules.
- Makes the design **cleaner** and easier to maintain.
- Encourages **separation of concerns** even for cross-functional features.

🔄 How It Works:

In **Aspect-Oriented Programming (AOP)**, aspects are defined separately and then "woven" into the codebase wherever they are needed.

⚙️ Real-Life Example:

- Imagine a **security check** that must be done **before accessing any function** in a banking app.
- Instead of adding security logic in every function (which is repetitive), you define it as a **separate aspect**.
- This aspect is then applied to all relevant modules.

> "An aspect is implemented as a **separate module** that can be integrated across the codebase where needed."

✅ Benefits:

- Enhances **modularity**
- Reduces **redundant code**
- Makes **crosscutting concerns easier to manage**
- Improves **clarity and maintainability**

🔁 10. Refactoring

💡 Definition:

Refactoring is the process of **restructuring existing code or design without changing its external behavior**, in order to **improve internal structure** and **code quality**.

📌 Key Ideas:

- It's about **cleaning up** the design.
- Focus is on **making the code simpler, clearer, and more efficient**.
- Refactoring does **not change what the software does**, only **how** it does it internally.

🎯 Goals of Refactoring:

- Eliminate **redundant code**
- Remove **unused or outdated design elements**
- Improve **algorithm efficiency**
- Replace **poorly constructed data structures**
- Simplify **complex logic**

- Increase **readability** and **maintainability**

🔄 When to Refactor:

- Code becomes **hard to understand**
- You find **duplication**
- You're adding new features and want to **simplify the base first**
- Before or after **debugging**

⚙️ Real-Life Example:

Original code	Refactored version:

`if (user.isLoggedIn()) {`
`  if(user.getRole().equals("admin")){`
`    dashboard.showAdminPanel();`
`  }`
`}`
	`if (user.isAdmin()) {`
	`  dashboard.showAdminPanel();`
	`}`

✅ Same behavior, but cleaner and more readable.

✅ Benefits of Refactoring:

- Easier to maintain and extend
- Reduces chances of bugs
- Improves **developer productivity**
- Encourages **clean code practices**

🧩 11. Object-Oriented Design Concepts

💡 Definition:

Object-Oriented Design (OOD) is a method of design that organizes software around **objects**—real-world entities that have **state** (attributes) and **behavior** (methods).

📌 Key Concepts:

1. 🧱 Classes and Objects
   - A **class** is a blueprint for creating **objects**.
   - An **object** is an instance of a class with actual data.
   - Example:
     Car is a class, myCar is an object of class Car.

2. 🧬 Inheritance
   - Allows one class (child) to **inherit properties and methods** from another (parent).
   - Promotes **code reuse**.
   - Example:
     Class ElectricCar inherits from class Car.

3. 📩 Messages
   - Objects **communicate** by sending **messages** (method calls) to one another.
   - Example:
     myCar.startEngine() → a message to the object myCar.

4. 🔀 Polymorphism
   - One method name can perform **different tasks** depending on the object.
   - Example:
     shape.draw() could draw a circle, square, or triangle depending on the object.

🔄 Other OO Concepts:

- **Encapsulation**: Hiding internal details of objects and exposing only what is necessary.
- **Abstraction**: Simplifying complex reality by modeling classes appropriate to the problem.
- **Association, Aggregation, Composition**: Define relationships between objects.

✅ Benefits of OOD:

- Modular, reusable code
- Easier to maintain and extend
- Mirrors real-world systems
- Encourages better software architecture

🧩 12. Design Classes

💡 Definition:

Design classes refine **analysis classes** by adding the necessary **technical details** required for actual **implementation** of the system. They form the bridge between **analysis** and **coding**.These classes support the entire software infrastructure and align closely with the **software architecture layers**.

🧱 Types of Design Classes:

There are **five major types**, each serving a different purpose within the design architecture:

1. 🖥️ User Interface Classes

- Handle all interactions between the **user** and the **system**.
- Define UI elements like forms, buttons, menus, etc.
- **Example:**
  LoginScreen, DashboardUI, FormHandler

2. 🏢 Business Domain Classes

- Represent **core functionality** and **business logic**.
- Contain attributes and methods that model real-world entities.
- **Example:**
  Customer, Account, Invoice, Order

3. 🔄 Process Classes

- Handle **workflow** and **coordination** between domain objects.
- Implement business **processes**, often connecting UI with business domain classes.
- **Example:**
  OrderProcessor, PaymentService, ReportGenerator

4. 💾 Persistent Classes

- Manage data that must be **stored** and **retrieved** (e.g., from a **database**).
- Represent tables or collections in data storage.
- **Example:**
  UserDataStore, ProductRepository, DBManager

5. ⚙️ System Classes

- Enable system-level operations such as **communication**, **configuration**, **logging**, etc.
- Allow the software to function in its computing environment.
- **Example:**
  Logger, ConfigManager, SystemClock, EmailService

✅ Purpose of Design Classes:

- Guide implementation with **technical specificity**
- Support separation of concerns
- Ensure that each class has a **clear role** in the architecture
- Enhance **modularity**, **scalability**, and **maintainability**

🏗️ The Design Model

The design model serves as a bridge between the **analysis model** and the final software implementation. It can be viewed through two distinct dimensions:

1. 📈 Process Dimension

- **Definition**: Represents how the design model **evolves** over time as design tasks are executed during the software development process.
- **Key Points**:
  - Involves a series of steps or phases that take the design from initial concepts to a detailed implementation plan.
  - Illustrates the progression and refinement of the design as feedback and insights are gathered.
  - Each design task (like creating classes, interfaces, and relationships) contributes to the overall evolution of the model.

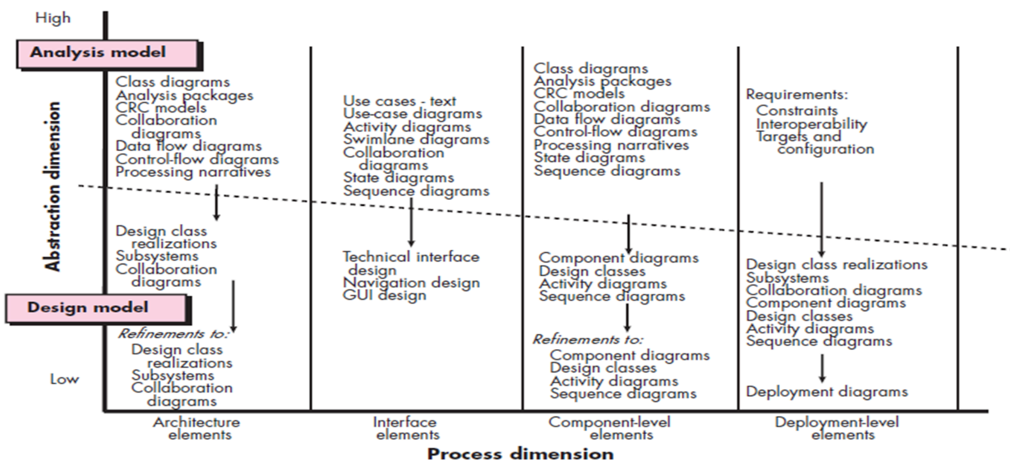
2. 🔍 Abstraction Dimension

- **Definition**: Indicates the **level of detail** in the design model as elements from the analysis model are transformed into design equivalents.

- **Key Points**:
    - Starts with high-level abstractions (e.g., major components and their interactions).
    - Progresses to more detailed representations, including specific algorithms, data structures, and interfaces.
    - Shows how each element is **refined iteratively** until it is ready for implementation in code.

📏 Boundary Between Analysis and Design Models

- The **dashed line** in the design model indicates the **boundary** between the analysis model and the design model.
- This boundary signifies a transition:
    - From understanding the **requirements** and **context** (analysis) to defining **how** those requirements will be fulfilled (design).
- It emphasizes that design is informed by analysis but focuses on creating a practical plan for implementation.



🗃️ 1. Data Design Elements

💡 Definition:

Data design involves creating a structured model of data and information that starts at a **high level of abstraction**. This model is progressively refined to more detailed representations suitable for implementation in a computer-based system.

📊 Process of Data Design:

1. **High-Level Data Model:**
   - Begins with an **abstract representation** of the data, focusing on entities and relationships without getting into implementation details.
   - **Example:** In a library system, high-level entities might include Books, Members, and Loans.

2. **Refinement to Implementation-Specific Models:**
   - As the design progresses, the data model becomes more specific, detailing how data will be stored, accessed, and manipulated.
   - **Example:** Transitioning from the concept of a Book entity to defining attributes like title, author, ISBN, and how they will be stored in a database.

3. **Implementation Representations:**
   - The final data design will be implemented using specific data structures, databases, or file formats, depending on the application requirements.
   - **Example:** Creating tables in a relational database to represent Books, Members, and Loans.

🏗️ Influence on Software Architecture:

- The **architecture of the data** has a significant impact on the overall architecture of the software that will process it.
- How data is structured and organized can determine:
  - The efficiency of data retrieval
  - The complexity of data manipulation
  - How easily the software can adapt to changes in requirements

🔑 Importance of Data Structure in Design:

- **Data Structure as a Foundation**: The way data is structured is crucial for the effectiveness and performance of the software.

- **Supports Business Logic**: Proper data design enables seamless integration with the business logic, ensuring that software functionalities can operate efficiently.

🏛️ 2. Architectural Design Elements

💡 Definition:

Architectural design in software is akin to the **floor plan of a house**, providing a structured overview of how the software system will be organized and how its components will interact.

📌 Key Points:

- **Overall View**: Architectural design elements offer a comprehensive view of the software, similar to how a floor plan outlines the layout and relationships between rooms and areas in a house.
- **Blueprint for Implementation**: Just as a house plan guides construction, the architectural design guides the development of the software, ensuring that all parts work together effectively.

📊 Sources of the Architectural Model:

The architectural model is derived from three primary sources:

1. **Application Domain Information**:
    - This includes knowledge about the specific **context** and **requirements** of the software being developed.
    - Understanding the domain helps define the key components and their interactions.
    - **Example**: In a healthcare application, the domain information would include data related to patients, doctors, appointments, and treatments.

2. **Specific Requirements Model Elements**:
    - This consists of detailed elements such as **data flow diagrams**, **analysis classes**, and their **relationships** and **collaborations**.
    - These elements provide insights into how data moves through the system and how different parts of the application will interact.
    - **Example**: A data flow diagram that shows how patient information is processed from registration to appointment scheduling.

3. **Architectural Styles and Patterns**:
    - Knowledge of existing **architectural styles** (e.g., layered architecture, microservices, client-server) and **design patterns** (e.g., MVC, event-driven) helps in structuring the software effectively.
    - Choosing the right style or pattern can greatly influence the software's maintainability, scalability, and performance.
    - **Example**: Using a microservices architecture to create a scalable application where each service can be developed and deployed independently.

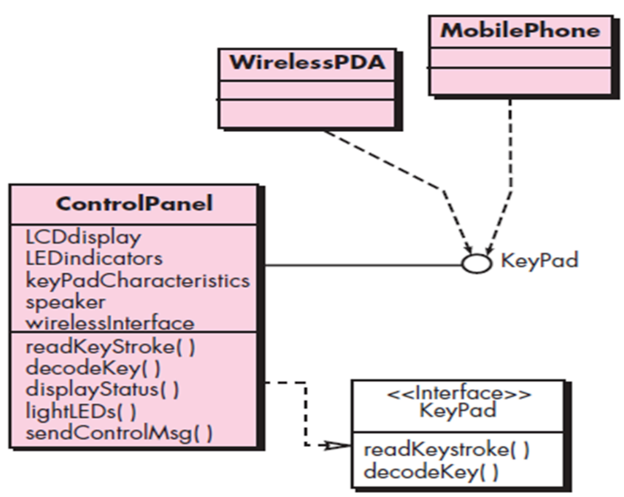
✅ Benefits of Architectural Design:

- **Clear Structure**: Provides clarity on the organization of the software and how components interact.
- **Guides Development**: Acts as a reference for developers to ensure that the implementation aligns with the intended design.
- **Facilitates Communication**: Helps stakeholders understand the software structure, making it easier to discuss changes and enhancements.

🖥️ 3. Interface Design Elements

💡 Definition:

Interface design elements in software illustrate the **information flows** into and out of the system and how this information is communicated among the various components defined in the architecture.



📌 Key Points:

- **Communication Framework**: Interface design establishes how different parts of the software system communicate, enabling both **external interactions** and **internal collaborations** among components.
- **Crucial for Usability**: A well-designed interface ensures that users can effectively interact with the software and that different system components can work seamlessly together.

🔑 Three Important Elements of Interface Design:

1. **User Interface (UI):**
    - This element defines how **users interact** with the software, including visual elements like buttons, menus, forms, and layouts.
    - The UI must be intuitive, user-friendly, and aligned with user requirements to enhance user experience.
    - **Example**: A web application might have a dashboard with interactive charts, buttons for navigation, and forms for user input.

2. **External Interfaces**:
    - These are interfaces that connect the software to **other systems**, **devices**, **networks**, or external information sources and consumers.
    - They define how the system communicates with outside entities, ensuring proper data exchange and interoperability.
    - **Example**: An e-commerce application may use APIs to connect to payment gateways, inventory management systems, and shipping services.

3. **Internal Interfaces**:
    - These interfaces define the communication between **various design components** within the software architecture.
    - They facilitate collaboration and data flow among internal modules, classes, or services.
    - **Example**: An internal interface might enable a UserService component to interact with a DatabaseService for retrieving and updating user information.

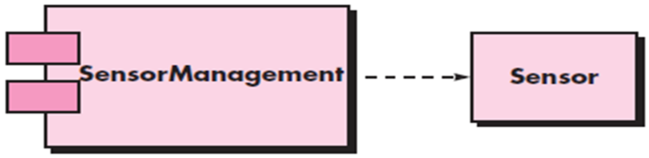
✅ Benefits of Interface Design Elements:

- **Enhanced Communication**: Ensures that all components, both internal and external, can effectively communicate with each other.
- **Improved Usability**: A well-designed UI makes the software accessible and easy to use for end-users.
- **Modularity**: Clear internal interfaces promote modular design, making it easier to update or replace components without affecting the entire system.

🛠️ 4. Component-Level Design Elements

💡 Definition:

Component-level design provides a detailed specification of each software component within the system. This design focuses on the internal workings of the components, including data structures, algorithms, and interfaces.



📌 Key Points:

- **Internal Detail**: This design phase delves into the specifics of how each component operates, ensuring that every aspect is well-defined for effective implementation.
- **Critical for Implementation**: A thorough component-level design is essential for developers to understand how to build and integrate components effectively.

🔑 Main Elements of Component-Level Design:

1. **Data Structures**:
    - Defines how **local data objects** are organized within the component.
    - Specifies the types of data used, including their relationships and how they are accessed or modified.
    - **Example**: A component might use a HashMap to store user sessions, where the key is the session ID and the value is the user data.

2. **Algorithmic Detail**:
    - Outlines the specific **algorithms** and **processing logic** that the component will use to perform its functions.
    - Includes steps for data manipulation, decision-making processes, and any calculations necessary for the component's operation.
    - **Example**: A sorting algorithm (like QuickSort or MergeSort) might be used within a component to organize data efficiently.

3. **Component Interface**:
    - Defines the **interface** through which other components or systems can interact with this component.
    - Specifies the available operations (methods) and the parameters required for each operation.
    - **Example**: A component might expose methods like getUserData(userId), updateUserData(userId, data), and deleteUser(userId).

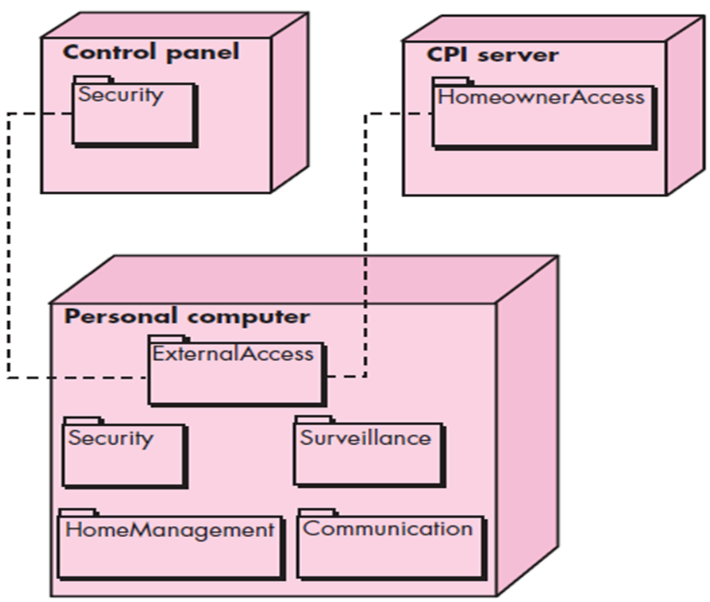
✅ Benefits of Component-Level Design:

- **Clarity and Detail**: Provides clear specifications for developers, reducing ambiguity and misunderstandings.
- **Modularity**: Encourages modular design by clearly defining how each component interacts with others, making it easier to replace or update components.
- **Efficiency**: Helps in optimizing performance by specifying appropriate data structures and algorithms tailored to the component's tasks.

🌐 5. Deployment-Level Design Elements

💡 Definition:

Deployment-level design elements detail how software functionality and subsystems will be allocated within the **physical computing environment** that will support the software. This includes the hardware, software, and network configurations required for deployment.



📌 Key Points:

- **Physical Allocation**: This design focuses on how components of the software system will be distributed across various physical resources, including servers, devices, and network connections.

- **Deployment is Crucial**: Proper deployment design ensures that the software operates efficiently in the intended environment, meeting performance and availability requirements.

📊 Stages of Deployment Design:

1. **Descriptor Form**:
    - The initial stage describes the **deployment environment** in general terms, outlining the types of servers, networks, and systems involved without getting into specific configurations.
    - **Example**: "The application will run on a cloud-based server environment with a load balancer and a database server."

2. **Instance Form**:
    - This stage uses a more detailed approach, explicitly describing the elements of the configuration.
    - Includes specifics about:
        - The **hardware specifications** (CPU, RAM, storage) for each server.
        - The **software stack** (operating systems, application servers, databases).
        - Network configurations (firewalls, routing, protocols).

    - **Example**:
        - "The application will be deployed on three virtual machines, each with 4 CPUs and 16 GB of RAM, running Ubuntu 20.04 with Apache Tomcat for the application server and MySQL for the database."

✅ Benefits of Deployment-Level Design:

- **Clarity on Infrastructure Needs**: Clearly defines the necessary hardware and software resources, which helps in resource allocation and budgeting.
- **Facilitates Planning**: Assists in identifying potential bottlenecks and ensuring scalability in the deployment environment.
- **Improves Reliability**: By explicitly detailing the deployment setup, it becomes easier to ensure redundancy, load balancing, and failover mechanisms are in place.