Design Engineering

Q. What is design. What are the qualities of good software design

Software design is the process of **defining the architecture, components, interfaces, and data** of a software system to satisfy specified requirements. It is a **blueprint** for software development that transforms **requirements (SRS)** into a **structured solution**.

Why is software design important?

- Helps in **understanding system structure** before coding.
- Reduces **errors, rework, and maintenance costs**.
- Ensures **scalability, efficiency, and reusability** of code.

Qualities of a Good Software Design

A **good software design** follows **design principles** that improve **maintainability, performance, and usability**. Below are the key **qualities of a good software design**:

1. Correctness

- The design must **meet the software requirements** specified in the **SRS document**.
- Example: If the requirement says "Login must allow only authorized users," the design must **include authentication and authorization**.

2. Simplicity

- A simple design is **easy to understand, modify, and debug**.
- **Avoids unnecessary complexity** and keeps code **clean**.
- Example: Using **modular functions** instead of a single long function improves readability.

3. Maintainability

- The design should allow **easy modifications and bug fixes**.
- Example: If a new **payment method** (e.g., UPI) needs to be added to an e-commerce site, it should not require **rewriting the entire payment module**.

4. Reusability

- Common functionalities should be **designed as reusable components**.
- Example: A **user authentication module** can be reused in multiple applications (web, mobile, API).

5. Scalability

- The software design must **support future growth** without major changes.
- Example: A social media app should allow **millions of users** without affecting performance.

6. Modularity

- The software should be **divided into independent modules** that can be developed and tested separately.
- Example: **MVC (Model-View-Controller) design** keeps data, UI, and logic separate.

7. Efficiency & Performance

- The design must ensure **optimized memory and CPU usage**.
- Example: Using **indexing in databases** reduces search time.

8. Security

- The design must **protect data and user privacy**.
- Example: Encrypting **user passwords** before storing them in a database.

9. Portability

- Software should **work across different platforms** with minimal changes.
- Example: A mobile app should run **on Android and iOS** without major code changes.

10. Testability

- The design should allow **easy testing** of each module.
- Example: Writing **unit tests** for individual functions before integrating them.

Q. Explain Design Principles

Design principles provide **guidelines** to create **efficient, maintainable, and scalable** software. Below are **10 key design principles** that ensure a **well-structured software system**.

---

1. The Design Process Should Not Suffer from 'Tunnel Vision'

- The design should **consider multiple solutions** before selecting the best one.
- Avoid **fixating on one approach** without evaluating alternatives.

Example:
A developer designs a banking system with **only web support** but later realizes mobile users also need access. **A broader vision** would have included mobile-friendly design from the start.

---

2. The Design Should Be Traceable to the Analysis Model

- Every part of the design should be **linked to a requirement** from the analysis phase.
- If a design **feature doesn’t relate** to any requirement, it's unnecessary.

Example:
In an **e-commerce app**, if the analysis model defines "Order Tracking," the design should **clearly implement order-tracking features**.

---

3. The Design Should Not Reinvent the Wheel

- **Reuse existing solutions** instead of creating new ones from scratch.
- Utilize **libraries, frameworks, and design patterns** to save time.

Example:
Instead of **writing a new login system**, use **OAuth or Firebase Authentication**.

---

4. The Design Should Minimize the Intellectual Distance

- The software should **closely represent the real-world problem** it is solving.
- The system should be **intuitive and natural** to users.

Example:
A **hospital management system** should use familiar terms like **Doctor, Patient, Appointment**, instead of complex technical names.

---

5. The Design Should Exhibit Uniformity and Integration

- The system should have a **consistent structure** and **integrated components**.
- The **user interface, database structure, and backend** should follow **the same design principles**.

Example:
A **web app** where **buttons, menus, and workflows** follow the same **color scheme and layout** across all pages.

---

6. The Design Should Be Structured to Accommodate Change

- The design should be **flexible** to allow future modifications **without major rewrites**.
- Use **modular design** and **design patterns** to support changes.

Example:
An **online shopping website** that initially supports **credit cards** should be **designed to easily add new payment methods** (like UPI, PayPal) later.

---

7. The Design Should Degrade Gracefully

- The software should **handle errors smoothly** without crashing.
- It should provide **helpful messages** when unexpected conditions occur.

Example:
A **banking system** should **show an error message** ("Insufficient funds") instead of crashing when a user tries to withdraw more than their balance.

---

8. Design Is Not Coding, and Coding Is Not Design

- **Design comes before coding** and focuses on **structure, architecture, and planning**.
- **Coding is the implementation** of the design.

Example:
Creating a **UML diagram** for an **E-commerce website** before writing a single line of code.

---

9. The Design Should Be Assessed for Quality as It Is Being Created

- **Check for errors** and **improvements** during the design phase, not after development.
- Use **design reviews and testing tools**.

Example:
Using **code reviews** and **static analysis tools** (like SonarQube) to assess the design quality **before development begins**.

---

10. The Design Should Be Reviewed to Minimize Conceptual Errors

- **Conceptual (semantic) errors** occur when the design does not match **real-world requirements**.
- Regular **reviews and validations** help detect these errors early.

Example:
A **hotel booking system** where **rooms can be booked even if they are occupied**—this is a conceptual error and should be **caught in the design review**.

Q. Explain design issues

Design issues in software engineering refer to **challenges, constraints, and considerations** that arise during the **software design process**. These issues impact the **efficiency, maintainability, usability, and performance** of the software system.

Key Design Issues in Software Engineering

1. Abstraction

✓ **Definition:** Hiding complex implementation details and exposing only essential features.
✓ **Challenge:** Finding the right level of abstraction to balance simplicity and functionality.
✓ **Example: A car dashboard** shows speed and fuel level but hides complex engine mechanisms.

---

2. Refinement

✓ **Definition:** Breaking down a complex system into smaller, manageable parts through stepwise refinement.
✓ **Challenge:** Maintaining clarity while refining components.
✓ **Example:** A **banking system** is divided into modules like **account management, transactions, and security**.

---

3. Modularity

✓ **Definition:** Dividing software into **independent modules** to enhance reusability and maintainability.
✓ **Challenge:** Ensuring **proper communication** between modules without tight coupling.
✓ **Example:** In a **website**, separate modules handle **user authentication, payments, and notifications**.

---

4. Architecture Selection

✓ **Definition:** Choosing a suitable **software architecture** based on system requirements.
✓ **Challenge:** Selecting between architectures like **Layered, Microservices, or Client-Server**.
✓ **Example: Netflix** uses a **Microservices Architecture** to handle different services like **streaming, recommendations, and user profiles** separately.

---

5. Control Hierarchy (Structural Design)

✓ **Definition:** Organizing the flow of control among system components.
✓ **Challenge:** Avoiding too many **levels of control**, which can increase complexity.

✓ **Example:** In an **ATM system**, the control hierarchy includes **User → ATM Interface → Bank Server**.

---

6. Structural Partitioning

✓ **Definition:** Arranging system components in a way that improves **performance and maintainability**.
✓ **Challenge:** Avoiding **overlapping responsibilities** and ensuring **clear separation** of concerns.
✓ **Example:** A **hospital management system** partitions functions into **Patient Management, Doctor Scheduling, and Billing**.

---

7. Data Structure Selection

✓ **Definition:** Choosing appropriate **data structures** for efficient data storage and retrieval.
✓ **Challenge:** Selecting between **arrays, linked lists, trees, or databases** based on system needs.
✓ **Example: Search engines like Google** use **trie data structures** for fast word lookups.

---

8. Interface Design

✓ **Definition:** Ensuring a smooth interaction between **users and the system** (User Interface) and **modules within the system** (System Interface).
✓ **Challenge:** Designing a **user-friendly, consistent, and responsive** interface.
✓ **Example: Mobile apps** use **intuitive UI elements** like buttons, icons, and gestures for navigation.

---

9. Error Handling & Exception Handling

✓ **Definition:** Designing mechanisms to **detect, report, and recover** from errors.
✓ **Challenge:** Ensuring errors are **handled gracefully** without crashing the system.
✓ **Example: Online banking** prompts a message like **"Invalid password. Please try again."** instead of displaying technical errors.

---

10. Concurrency

✓ **Definition:** Handling **multiple operations simultaneously** in a system.
✓ **Challenge:** Preventing **race conditions, deadlocks, and synchronization issues**.
✓ **Example: E-commerce websites** manage **thousands of users placing orders at the same time**.

---

11. Security

✓ **Definition:** Protecting the system from **unauthorized access, data breaches, and cyber threats**.
✓ **Challenge:** Implementing strong **authentication, encryption, and secure coding practices**.
✓ **Example: Two-Factor Authentication (2FA)** enhances security in banking apps.

Q. Explain Design Concept

Software **design concepts** are fundamental **principles** that guide developers in creating **efficient, maintainable, and scalable** software. These concepts help **translate requirements** into a **well-structured design**.

---

1. Abstraction

- Abstraction means **hiding details** and showing only the **essential features** of a system.
- It helps developers **focus on the bigger picture** instead of unnecessary details.

Example:

- A **car's steering wheel** hides the complex mechanism of the wheels turning.
- In programming, a **class** represents an abstraction:

```
class Car {
   void drive() { System.out.println("Car is moving"); }
}
```

The user doesn't need to know **how** the car moves—just that it does.

---

2. Refinement

- Refinement means breaking a **complex problem into smaller, simpler problems**.
- It helps in **step-by-step development**, making debugging and modifications easier.

Example:

- Instead of designing a **whole banking system at once**, break it into **Login System, Account Management, and Transaction System**.

---

3. Modularity

- A system is divided into **independent modules** that can be **developed and tested separately**.
- Each module performs a **specific function** and interacts with others through **defined interfaces**.

Example:

- A **library management system** can have modules like:
  - BookModule (Handles book records)
  - UserModule (Handles user details)
  - IssueModule (Manages book issuance)

Each module **works independently**, improving **reusability** and **maintainability**.

---

4. Software Architecture

- Architecture defines the **overall structure** of the software system.
- It includes **components, their interactions, and data flow**.

Example:

- **Client-Server Architecture:**
  - The **client (browser)** sends a request.
  - The **server (backend)** processes it and returns data.

---

5. Control Hierarchy (Structural Organization)

- It represents the **flow of control** between different modules.
- It helps in organizing **who controls whom** in a system.

Example:

- A **food delivery app** has:
  - User Module → Controls Order Module
  - Order Module → Controls Delivery Module

This ensures **proper structure** and avoids confusion.

---

6. Structural Partitioning

- Divides a system into **smaller, manageable parts** to enhance **scalability and efficiency**.
- Can be **horizontal (layered design)** or **vertical (feature-based design)**.

Example:

- **Horizontal Partitioning (Layered Architecture):**
  - UI Layer → Business Logic Layer → Database Layer
- **Vertical Partitioning (Feature-Based):**
  - Admin Features, User Features, Payment Features

---

7. Data Abstraction and Data Hiding

- **Data Abstraction:** Hides **complexity** and shows only **relevant information**.
- **Data Hiding:** Prevents **direct access** to data for security and integrity.

Example:

```
class BankAccount {
   private double balance;  // Data Hiding
   public double getBalance() { return balance; }  // Abstraction
}
```

Here, balance is **hidden** (Data Hiding), but the user can still **view** it through getBalance() (Abstraction).

---

8. Concurrency

- Concurrency allows multiple tasks to **run simultaneously** to improve performance.

Example:

- **Multithreading in Java:**
  - A game can **play background music** while **accepting user inputs**.
  - An **online store** can handle **multiple customer orders at the same time**.

---

9. Verification and Validation

- **Verification:** Ensures **the software is built correctly** (follows specifications).
- **Validation:** Ensures **the correct software is built** (meets user needs).

Example:

- **Verification:** Checking if a login feature follows password rules.
- **Validation:** Checking if the login feature actually works for users.

---

10. Simplicity

- The design should be **simple and easy to understand** to avoid unnecessary complexity.

Example:

- Instead of writing **2000 lines of code in one file**, break it into **separate modules**.

Q. Explain different architectural styles

Software architecture defines the **structure and organization** of a software system. An **architectural style** is a **design pattern** that provides a predefined structure for organizing the components of a system. These styles help improve **scalability, maintainability, security, and performance**.

The five major architectural styles are:

1. **Data Flow Architecture**
2. **Object-Oriented Architecture**
3. **Layered Architecture**
4. **Data-Centered Architecture**
5. **Call and Return Architecture**

---

1. Data Flow Architecture

Definition:

- In **Data Flow Architecture**, data moves **sequentially** through **processing components**.
- Each component **modifies the data** before passing it to the next stage.

Types of Data Flow Architecture:

1. **Batch Sequential Processing:**
    - Data is processed in **batches**, and each process runs **independently**.
    - Example: **Payroll Processing System** (Employee salary calculations).
2. **Pipelines and Filters:**

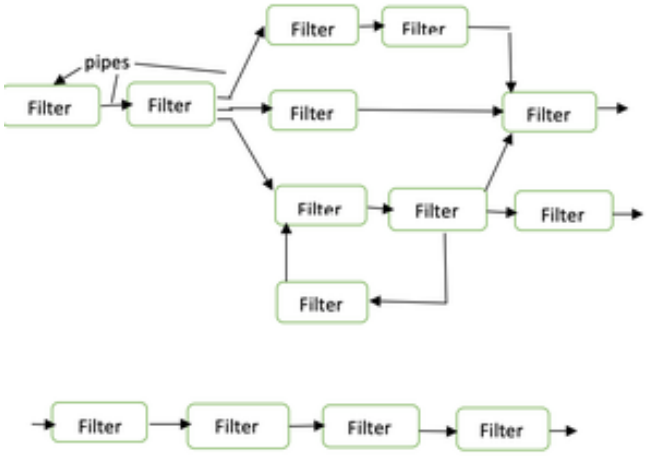
- The system consists of **filters (processing units)** connected by **pipelines**.
- Each filter processes the input and passes the output to the next filter.
- Example: **Compiler System** (Lexical Analysis → Syntax Analysis → Code Generation).

Advantages:

✓ **Modular design** (Each filter can be reused).
✓ **Easy to add or modify processing steps**.

Disadvantages:

✓ **Slow processing** (data must pass through multiple stages).
✓ **Not suitable for real-time systems**.



---

2. Object-Oriented Architecture

Definition:

- This architecture is based on the principles of **Object-Oriented Programming (OOP)**.
- The system is designed using **objects** that interact through **messages**.

Example:

- **Java-based Applications** (Spring, Hibernate).
- **Game Development** (Unity, Unreal Engine).
- **GUI-Based Applications** (Windows, Android Apps).

Advantages:

✓ **Encapsulation** improves **security and data hiding**.
✓ **Reusability** reduces **development time**.
✓ **Scalability** (Easy to add new features).

Disadvantages:

✓ **Higher memory consumption** due to objects.
✓ **Complex interactions** between objects.

---

3. Layered Architecture

Definition:

- The system is **divided into layers**, where **each layer performs a specific function**.
- Data flows **from one layer to another in a controlled manner**.

Example:

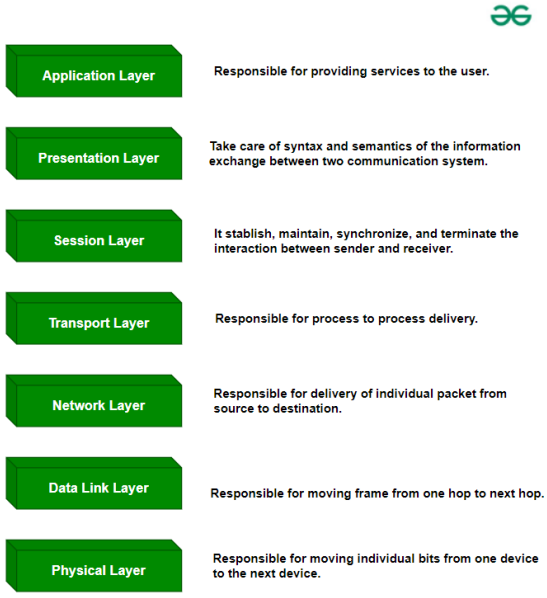
- **Web Applications** (Amazon, Gmail, Facebook).
- **Enterprise Software** (ERP, CRM).

Advantages:

✓ **High security** (Each layer has restricted access).
✓ **Easy to modify and maintain** (Changes in one layer don't affect others).

Disadvantages:

✓ **Slow performance** (Data must pass through multiple layers).
✓ **Difficult to modify lower layers** without affecting others.



---

4. Data-Centered Architecture

Definition:

- A **central database (repository)** is the **core** of the system.
- Different components interact with the database to fetch and store data.

Example:

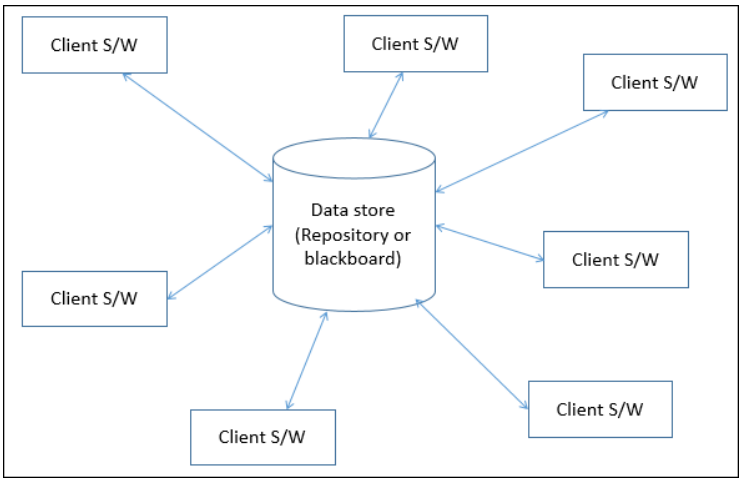
- **Database Management Systems (DBMS)** (MySQL, PostgreSQL).
- **Cloud Storage Systems** (Google Drive, Dropbox).
- **Library Management Systems** (Centralized database for books and users).

Advantages:

✓ **Centralized data management** (Ensures consistency and avoids redundancy).
✓ **Easy backup and recovery** (Data is stored in one place).

Disadvantages:

✓ **Single point of failure** (If the central database crashes, the system stops working).
✓ **Performance bottleneck** (Too many requests can slow down the database).



---

5. Call and Return Architecture

Definition:

- In **Call and Return Architecture**, the system consists of a **main program** that calls **subprograms** (functions or procedures).
- After execution, **control returns to the calling function**.

Types of Call and Return Architecture:

1. **Main-Subroutine Model:**
    - A **main program** calls multiple **subprograms**.
    - Example: **C and C++ programs** with multiple functions.
2. **Remote Procedure Call (RPC):**

- A function executes on a **remote server** instead of locally.
- Example: **Web Services** (SOAP, REST APIs).

Advantages:

✓ **Simplifies program flow** by dividing it into smaller functions.
✓ **Code reuse** (Functions can be used multiple times).

Disadvantages:

✓ **Difficult to modify** if function calls are deeply nested.
✓ **Network overhead** in remote calls (RPC).

Q. Explain Component Level Design

Component-Level Design (CLD) is the **detailed design** phase where **each software component** is designed based on **functional and non-functional requirements**. It transforms **high-level architecture** into **low-level, detailed structures** that developers can implement using **code**.

A **component** is a **modular, replaceable, and reusable** part of a system that performs a **specific function**.

Objectives of Component-Level Design

1. **Ensure modularity** – Each component is **independent** and reusable.
2. **Improve maintainability** – Changes in one component **do not affect** others.
3. **Enhance performance** – Well-designed components improve **efficiency**.
4. **Ensure scalability** – The system can **easily expand** by adding new components.
5. **Improve security** – Each component has **controlled access** to data and resources.

Principles of Component-Level Design

1. Single Responsibility Principle (SRP)

Each component should have **only one reason to change**.
✓ **Example:** A **User Authentication component** should only handle login/logout, not profile updates.

2. Open-Closed Principle (OCP)

A component should be **open for extension but closed for modification**.
✓ **Example:** Instead of modifying an existing **Payment component**, create a new **extension** for "UPI Payments".

3. Liskov Substitution Principle (LSP)

A subclass/component should be **replaceable** without breaking the system.
✓ **Example:** A **BankAccount class** should work correctly even if we substitute it with **SavingsAccount or CurrentAccount**.

4. Interface Segregation Principle (ISP)

A component should not have **unused functionalities**.
✓ **Example:** Instead of a **single Payment interface** with many methods, use **separate interfaces** for "CreditCardPayment", "UPIPayment", etc.

5. Dependency Inversion Principle (DIP)

High-level components should **not depend** on low-level implementations.
✓ **Example:** A **Notification System** should use an **interface** so it can switch between "SMS", "Email", or "Push notifications".

Elements of Component-Level Design

1. Data Structures

- Defines how **data is stored and accessed** in the component.
- Examples: **Arrays, Linked Lists, Hash Tables**.

2. Algorithms

- Defines **how operations** (sorting, searching) are performed.
- Example: **Quick Sort for fast searching in a database component**.

3. Interfaces

- Defines **communication** between components.
- Example: **REST API for a Weather Data Component**.

4. Error Handling

- Defines **how errors** are managed.
- Example: A **Payment component** should return **"Transaction Failed"** instead of crashing.

5. Security Mechanisms

- Ensures **data protection** and **restricted access**.
- Example: A **User Authentication Component** should use **password hashing**.

Steps in Component-Level Design

1. Identify Components

- Break the system into **logical modules**.
- Example: In an **E-commerce website**, we can have:
    - **User Authentication Component**
    - **Product Catalog Component**
    - **Payment Processing Component**

2. Define Component Interfaces

- Specify **how components interact**.
- Example: A **User Profile Component** exposes an API:
- GET /user/{id} → Fetch user data
- POST /user/update → Update user info

3. Design Component Internals

- Decide **data structures, algorithms, and logic** inside each component.
- Example: A **Recommendation System Component** might use a **Machine Learning model** for personalized suggestions.

4. Implement and Test Components

- Write **code** and test each component **independently**.
- Example: Use **JUnit** to test a **Login Component** in Java.

5. Integrate and Evaluate

- Connect components and **evaluate performance**.
- Example: An **Order Management System** should interact properly with **Inventory and Payment Systems**.

Types of Software Components

Type	Description	Example
User Interface (UI) Components	Handles user interaction	Login Page, Search Bar

Type	Description	Example
Business Logic Components	Implements core logic	Order Processing System
Data Access Components	Manages database interactions	MySQL Connector
Middleware Components	Facilitates communication between components	API Gateway
Security Components	Handles authentication & authorization	JWT Token System

---

Example: Component-Level Design of a Library Management System

A **Library Management System** has the following components:

1. User Management Component

- Manages user registration, login, and authentication.
- API Example: POST /register, POST /login.

2. Book Management Component

- Handles adding, deleting, and updating books.
- API Example: GET /books, POST /books/add.

3. Loan Management Component

- Handles book issuance and returns.
- API Example: POST /borrow/{book_id}, POST /return/{book_id}.

4. Notification Component

- Sends reminders for due dates.
- API Example: POST /send_reminder/{user_id}.

Advantages of Component-Level Design

✓ **Improves Maintainability** – Changes in one component do not affect others.
✓ **Enhances Reusability** – Components can be reused across projects.
✓ **Facilitates Scalability** – New features can be added easily.
✓ **Increases Development Speed** – Parallel development of components is possible.

Q. Explain System Level Design

System-Level Design (SLD) is the **high-level design phase** where the **entire system's architecture** is defined before moving to **detailed component-level design**. It focuses on:

- **Defining the system's structure and behavior**
- **Specifying how different components interact**
- **Ensuring scalability, reliability, and security**

Objectives of System-Level Design

✓ **Defines system architecture** – Organizes components and their relationships.
✓ **Ensures system integration** – How different modules work together.
✓ **Improves maintainability** – Designing systems for easy updates and modifications.
✓ **Enhances performance** – Optimizing resources for better efficiency.
✓ **Manages risks and failures** – Ensuring system stability under failures.

Key Elements of System-Level Design

1. System Architecture

- Defines **major components** and their **connections**.
- Example: In an **E-commerce system**, the main components are:
    - **User Interface (UI) Layer** – Web & mobile applications
    - **Business Logic Layer** – Order processing, payments
    - **Database Layer** – Stores user and product data

2. Data Flow and Control Flow

- Specifies **how data moves** within the system.
- Example: In a **Banking System**, a **fund transfer request** moves from **User → Bank API → Transaction Processing → Database Update**.

3. System Interfaces & APIs

- Defines **communication methods** between subsystems.
- Example: A **Weather API** provides external services with data via:
- GET /weather?city=Mumbai
- Response: { "temperature": 32°C, "humidity": 70% }

4. Security and Performance Considerations

- Ensures **authentication, authorization, and encryption**.
- Optimizes **response times and resource usage**.

- Example: A **Social Media System** encrypts passwords using **bcrypt hashing**.

Phases of System-Level Design

1. Requirements Analysis

- Understand **functional** and **non-functional** requirements.
- Example: A **Library Management System** must support **book lending, return, and user authentication**.

2. High-Level Architecture Design

- Select **architecture style** (Layered, Microservices, Client-Server, etc.).
- Example: A **cloud-based application** might use **Microservices Architecture**.

3. Defining Subsystems & Components

- Identify **key modules** and their interactions.
- Example: A **Hospital Management System** consists of:
    - **Patient Management**
    - **Appointment Scheduling**
    - **Billing & Payments**

4. Data Design

- Define **databases, data models, and storage mechanisms**.
- Example: An **E-commerce System** database contains:
- Table: Users → (UserID, Name, Email, Address)
- Table: Products → (ProductID, Name, Price, Stock)

5. Security & Reliability Planning

- Implement **encryption, authentication mechanisms, and failover strategies**.
- Example: A **Financial System** ensures security using **Two-Factor Authentication (2FA)**.

6. System Integration & Testing

- Combine **all components** and test for **compatibility and performance**.
- Example: A **Messaging App** is tested to handle **1 million users** without crashes.

Example: System-Level Design of an Online Banking System

Components:

1. **User Interface** – Mobile app & web portal
2. **Authentication Module** – Handles login, OTP verification
3. **Transaction Processing System** – Handles deposits, withdrawals
4. **Database** – Stores user and transaction data
5. **Security Module** – Ensures encryption and fraud detection

Data Flow:

User logs in → Authentication verified → User requests a fund transfer → Transaction processed & updated in the database → User gets confirmation

---

Comparison: System-Level Design vs. Component-Level Design

Feature	System-Level Design	Component-Level Design
Focus	Entire system architecture	Individual software components
Scope	High-level structure	Low-level implementation
Key Elements	Modules, subsystems, data flow	Data structures, algorithms, functions
Examples	Banking System, ERP, E-commerce	Payment Gateway, Notification Service

---

Advantages of System-Level Design

✓ **Improves Scalability** – Easy to expand the system.
✓ **Enhances Maintainability** – Components can be updated independently.
✓ **Ensures Reliability** – Reduces failure risks.
✓ **Optimizes Performance** – Efficient resource allocation.

Q. Explain User Interface Design Process

User Interface (UI) design is the process of creating interfaces that allow users to interact with a software system. The goal is to make the interface **intuitive, efficient, and user-friendly** while ensuring it meets **functional and aesthetic requirements**.

Objectives of UI Design

✓ **Enhance User Experience (UX)** – Ensure ease of use and accessibility.
✓ **Improve Efficiency** – Reduce the number of steps to complete a task.
✓ **Ensure Consistency** – Maintain uniform design across all screens.
✓ **Reduce Errors** – Guide users to prevent mistakes.
✓ **Increase Accessibility** – Make the UI usable for all, including differently-abled users.

User Interface Design Process

The UI design process consists of **five major steps**:

1. Understand the User & Requirements

- Conduct **user research** to understand their expectations.
- Identify **functional** and **non-functional** UI requirements.
- Example: A **banking app UI** should be **secure, easy to navigate, and mobile-friendly**.

2. Design the Information Architecture

- Organize information **logically** for better usability.
- Define the **navigation structure** (menus, categories, buttons).
- Example: A **news website** categorizes articles into **Politics, Sports, Entertainment, etc.**.

3. Create Wireframes & Prototypes

- **Wireframes** – Simple black-and-white sketches of UI layout.
- **Prototypes** – Interactive models to test user interactions.
- Example: A **wireframe of a login page** includes a **username field, password field, and login button**.

4. Develop the UI

- Choose a **color scheme, typography, and UI components**.
- Implement using technologies like **HTML, CSS, JavaScript** for web apps or **Android/iOS frameworks** for mobile apps.
- Example: A **shopping app** uses **red buttons for "Buy Now" and green buttons for "Add to Cart"**.

5. Testing & Evaluation

- Conduct **usability testing** to get user feedback.
- Perform **A/B testing** to compare different UI designs.

- Example: A **food delivery app** tests if users prefer **"Order Now" on the homepage or inside the menu**.

Golden Rules of UI Design

1. Place the user in control – Allow undo/redo, provide clear navigation.
2. Reduce user effort – Minimize clicks, automate repetitive tasks.
3. Ensure consistency – Use the same buttons, colors, and font styles throughout.
4. Provide feedback – Show loading indicators, success messages, and error alerts.
5. Simplify design – Avoid clutter and unnecessary elements.

Example: UI Design for an E-Commerce Website

Step 1: Understand Users – Customers expect **easy navigation, product filters, and a secure checkout process**.
Step 2: Information Architecture – Products are categorized into **Electronics, Clothing, Home Decor, etc.**
Step 3: Wireframe & Prototype – Sketch a homepage with **a search bar, product grid, and cart button**.
Step 4: Develop UI – Choose **blue and white colors**, use **icons for navigation**, and add **"Buy Now" buttons**.
Step 5: Testing – Check if **users find the checkout process smooth and intuitive**.

Q. What are the Golden Rules of user interface design

The **Golden Rules of UI Design** ensure that user interfaces are **consistent, user-friendly, and efficient**. These rules improve **usability, accessibility, and user satisfaction**.

---

1. Strive for Consistency

✓ Maintain a **uniform design** in fonts, colors, buttons, and layouts.
✓ Use **consistent commands, icons, and actions** across all screens.
✓ Example: **Microsoft Word, Excel, and PowerPoint** have similar toolbars for ease of use.

---

2. Visibility of System Status (Offer Informative Feedback)

✓ The system should **inform users about ongoing processes**.
✓ Provide **loading indicators, progress bars, or success/error messages**.
✓ Example: **WhatsApp** shows **"typing…"** when someone is responding.

---

3. Match Between System and the Real World (Design Dialog to Yield Closure)

✓ Use language, symbols, and workflows that users **understand from the real world**.
✓ Ensure the system provides **clear beginning, middle, and end interactions**.
✓ Example: **Trash Bin in Windows** mimics a real-world trash can.

---

4. User Control and Freedom (Permit Easy Reversal of Actions)

✓ Allow users to **undo, redo, and navigate freely** without restrictions.
✓ Example: **Undo/Redo options** in text editors like MS Word.

---

5. Error Prevention and Simple Error Handling

✓ Design interfaces to **prevent mistakes** before they happen.
✓ Use **confirmation dialogs** before critical actions (e.g., deleting files).
✓ Example: **Gmail warns before sending an email without an attachment** if the text mentions "attached".

---

6. Reduce Short-Term Memory Load (Recognition Rather Than Recall)

✓ Display **menus, hints, and tooltips** to help users recognize information instead of memorizing it.
✓ Example: **Google search suggests recent searches instead of requiring users to type again**.

---

7. Enable Frequent Users to Use Shortcuts

✓ Provide **keyboard shortcuts, gestures, or command-line options** for expert users.
✓ Example: **Ctrl + C (Copy) and Ctrl + V (Paste) save time in editing tasks**.

---

8. Aesthetic and Minimalist Design

✓ Avoid unnecessary elements that distract users.
✓ Keep the design **clean, simple, and visually appealing**.
✓ Example: **Google's homepage is minimalistic, with only a search bar and logo**.

---

9. Help Users Recognize, Diagnose, and Recover from Errors

✓ Provide **clear and human-readable error messages**.
✓ Offer solutions to **fix errors instead of just displaying codes**.

✓ Example: Instead of "Error 404," a website should display **"Page Not Found – Try Searching or Check the URL."**

---

10. Help and Documentation

✓ Provide **accessible help sections, FAQs, and tutorials**.
✓ Allow users to find guidance **without relying on external support**.
✓ Example: **Microsoft Office includes a "Help" section with step-by-step guides**.

Q. Explain Analysis Model and Design Model

Both the **Analysis Model** and **Design Model** are essential in software development. They help transform user requirements into a structured system.

---

1. Analysis Model

The **Analysis Model** is a representation of user requirements in a structured way. It focuses on **"what"** the system should do, rather than **"how"** it will do it.

Purpose:

✓ Understand user needs.
✓ Identify system functionalities.
✓ Provide a foundation for system design.

Components of Analysis Model:

1. **Use Case Diagram** → Shows how users interact with the system.
2. **Data Flow Diagram (DFD)** → Represents how data moves in the system.
3. **Entity-Relationship Diagram (ERD)** → Defines data relationships.
4. **Class Diagram** → Identifies objects and their relationships.

Example:

For an **online shopping system**, the analysis model will describe:
✓ Users can **browse products, add to cart, and make payments**.
✓ The system **stores customer details and order history**.

---

2. Design Model

The **Design Model** translates the **Analysis Model** into a **detailed system architecture**. It focuses on **"how"** the system will be built.

Purpose:

✓ Convert functional requirements into **technical specifications**.
✓ Define **data structures, algorithms, and interfaces**.
✓ Optimize **performance, scalability, and security**.

Components of Design Model:

1. **Architectural Design** → Defines software structure (Layered, Microservices, etc.).
2. **Component-Level Design** → Breaks the system into modules/components.
3. **Data Design** → Specifies database structure.
4. **Interface Design** → Defines user and system interfaces.

Example:

For an **online shopping system**, the design model will describe:
✓ **Database design** (tables for users, orders, products).
✓ **Modules** for **user authentication, payment gateway, and inventory management**.
✓ **User Interface (UI) layout** for the website and mobile app.

Key Differences Between Analysis Model & Design Model

Feature	Analysis Model	Design Model
Focus	What the system should do	How the system will work
Main Goal	Understand and document requirements	Develop system structure and architecture
Components	Use Cases, DFDs, ERDs, Class Diagrams	Architecture, Component Design, Data Design
Example	User can place an order	Uses SQL database to store orders
Output	Software Requirement Specification (SRS)	Technical design documents

Q. Explain how an analysis model is translated to Design models

The **Analysis Model** focuses on **what** the system should do, while the **Design Model** defines **how** it will be implemented. The transition from analysis to design ensures that the software is structured, scalable, and efficient.

---

1. Steps to Translate Analysis Model into Design Model

Step 1: Identify Architectural Structure

What happens?
✓ Decide the **overall architecture** of the system (e.g., Layered, Microservices, Client-Server).
✓ Define **modules, components, and their interactions**.

Example:
For an **E-commerce system**, we decide:
✓ **Frontend** → Web UI using HTML/CSS.
✓ **Backend** → Node.js for handling user requests.
✓ **Database** → MySQL for storing product and order data.

Step 2: Transform Functional Requirements into Components

What happens?
✓ Convert **Use Cases and Data Flow Diagrams (DFD)** into software **modules and functions**.
✓ Define **classes, methods, and interactions**.

Example:
For **"User Registration"**, we convert the analysis model's **User entity** into a **User Class** with:
✓ Attributes: Name, Email, Password, etc.
✓ Methods: registerUser(), loginUser(), etc.

Step 3: Define Database Schema

What happens?
✓ Translate **Entity-Relationship Diagram (ERD)** into **relational tables**.
✓ Define **primary keys, foreign keys, and relationships**.

Example:
From **ERD**, we define a database table:

User_ID	Name	Email	Password
101	John	john@gmail.com	********

Step 4: Develop Component-Level Design

What happens?
✓ Convert **Data Flow Diagrams (DFD)** into **class diagrams, sequence diagrams, and**

flowcharts.
✓ Define **functions, attributes, and interactions** between components.

Example:
✓ "Order Processing" module interacts with **User, Payment, and Inventory** components.
✓ The **Payment component** calls processPayment() method to handle transactions.

Step 5: Design User Interface (UI/UX)

What happens?
✓ Convert UI **wireframes into interactive screens**.
✓ Define **user navigation flow and interface elements**.

Example:
✓ From wireframes, design a **Login Page** with:

- **Text Fields** for email & password.
- **Buttons** for Login & Signup.
- **Error messages** for invalid inputs.

Step 6: Define Software & Hardware Requirements

What happens?
✓ Specify **technology stack** (e.g., programming languages, frameworks, servers).
✓ Define **hardware resources** (e.g., cloud storage, processing power).

Example:
✓ Use **Java Spring Boot** for backend, **ReactJS** for frontend, and **AWS** for cloud hosting.

Q. Explain Coupling and Cohesion

When designing software, two critical concepts play a key role in ensuring **modularity, maintainability, and efficiency**:
✓ **Cohesion** (How well a module's elements work together)
✓ **Coupling** (How dependent one module is on another)

Cohesion (High is Good)

Cohesion refers to **how strongly related and focused the responsibilities of a single module are**. A module with **high cohesion** does one specific task and does it well.

Types of Cohesion (From Worst to Best)

1. **Coincidental Cohesion (Worst)**

- Unrelated functions are placed in the same module.
- Example: A function handling **file operation, database access, and UI updates** together.

2. **Logical Cohesion**
    - Functions performing similar activities are grouped but require a **control flag** to decide which one to execute.
    - Example: A single function that **handles all types of user inputs** (keyboard, mouse, voice).
3. **Temporal Cohesion**
    - Functions that execute at the **same time** are grouped together.
    - Example: **Initialization functions** for logging, security setup, and database connections in one module.
4. **Procedural Cohesion**
    - Functions that execute in a **sequence** are grouped together.
    - Example: A module that first **validates input**, then **saves data**, then **logs the transaction**.
5. **Communicational Cohesion**
    - Functions that operate on the **same data** are grouped together.
    - Example: A module that **fetches and processes customer orders** using the same database.
6. **Functional Cohesion (Best)**
    - Each module performs **one task and only one task**.
    - Example: A **"Send Email" module** that handles everything related to sending emails only.

High cohesion is preferred as it leads to better reusability, maintainability, and readability.

Coupling (Low is Good)

Coupling refers to **the degree of dependency between two modules**. A system with **low coupling** means **modules can function independently**, making it easier to modify and maintain.

Types of Coupling (From Worst to Best)

1. **Content Coupling (Worst)**
    - One module directly modifies the internal **data or code of another module**.

- Example: Function A **modifies private variables** inside Function B.

2. **Common Coupling**
   - Multiple modules share the **same global data**.

     Example: Different modules modifying the same **global variable**.
3. **Control Coupling**
   - One module **controls** another by **passing control variables**.
   - Example: Function A **passes a flag** to Function B, which then behaves differently based on that flag.
4. **Stamp Coupling (Data Structure Coupling)**
   - Modules share **data structures**, leading to unnecessary dependencies.
   - Example: A function passing an entire **user profile object** when only the **user ID** is needed.
5. **Data Coupling (Best)**
   - Modules share **only necessary data** via parameters.
   - Example: Function A passes **only the user ID** to Function B instead of the entire user profile.

Low coupling is preferred because it improves reusability, scalability, and flexibility.

Difference Between Coupling and Cohesion

Feature	**Cohesion (High is Good)**	**Coupling (Low is Good)**
Definition	How well elements within a module work together	How dependent one module is on another
Goal	Make modules **self-contained**	Make modules **independent**
Effect of Change	Changes in one function **do not** impact others	Changes in one module **may affect** others
Example (Good)	A "Login" module that only handles login tasks	A "Login" module **only interacting via function calls**
Example (Bad)	A module handling **both login and report generation**	A "Login" module **modifying another module's database directly**

Q. What are the benefits of high cohesion and low coupling

Achieving **high cohesion** and **low coupling** is a **best practice** in software design. It leads to **better maintainability, scalability, and reliability** of a system.

Benefits of High Cohesion (Good)

High cohesion means a module **focuses on a single well-defined task** and has **strong internal unity**.

1. Improves Maintainability

- Since a **module is focused on a single task**, it's **easier to understand and modify**.
- Developers can **quickly identify** what a module does without getting confused.

2. Enhances Reusability

- A highly cohesive module **performs a single function**, making it **easier to reuse** in other parts of the project or different projects.

3. Increases Readability & Understandability

- Code is **organized and structured logically**, making it easier to **read and comprehend**.

4. Facilitates Debugging & Testing

- When an issue occurs, it's **easier to pinpoint the problem** because the module has **one clear responsibility**.
- Testing is simplified because there are **fewer interdependencies**.

5. Reduces Complexity

- A **clear separation of concerns** makes the system **modular** and reduces confusion.

Benefits of Low Coupling (Good)

Low coupling means **modules are independent**, minimizing the effect of changes in one module on another.

1. Makes Code More Flexible & Extensible

- Since modules **don't heavily depend on each other**, new features can be added **without breaking existing functionality**.

2. Improves Scalability

- A loosely coupled system allows **easy expansion**, making it suitable for large-scale projects.

3. Reduces Risk of Side Effects

- A change in one module **does not heavily impact** other modules, reducing **unexpected bugs**.

4. Enhances Testability

- Since modules are **independent**, they can be **tested individually** without worrying about dependencies.

5. Enables Parallel Development

- Different teams can work **on separate modules simultaneously** without interfering with each other.

Summary Table: High Cohesion vs. Low Coupling Benefits

Feature	High Cohesion	Low Coupling
Maintainability	Easy to update or modify	Less impact on other modules
Reusability	Modules can be reused in different projects	Independent modules make reusability easier
Readability	Code is clear and well-organized	Modules interact in a clean, predictable way
Testing	Easier to test since modules do one task	Each module can be tested separately
Flexibility	A module can be modified **without breaking** internal logic	Modules can be changed without affecting others
Debugging	Easy to find and fix errors	A change in one module does not cause widespread issues
Scalability	More modular structure allows for growth	New features can be added easily

Q.