**Lecture 1**

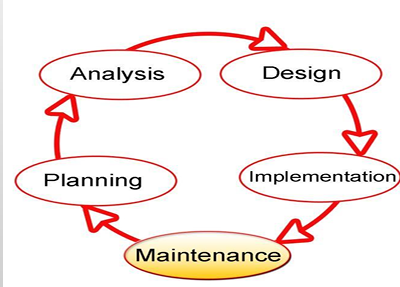
**Stages in Software Process**

A software process is a step-by-step set of activities needed to build a software system.

Even though there are many different ways (models) to develop software, they all usually include these main stages:

1. **Specification –** Decide and clearly define what the system should do (requirements of the software).
2. **Design and Implementation –** Plan the structure/organization of the system and then build (code) it.
3. **Validation –** Test the system to make sure it does what the customer actually wants.
4. **Evolution –** Update or change the system later when the customer’s needs change.

A software process model is simply a general representation (or diagram) of these steps.  
It shows the process from a certain point of view to help us understand it better.



**Requirement Analysis**

Requirement Analysis is the process of finding out what the customer or user needs and writing it down clearly.  
It means deciding what services the system should provide and what limitations (rules or restrictions) it must follow.

The steps in the Requirements Engineering Process are:

1. **Feasibility Study –** Check if building the system is possible with the available technology and if it is affordable (within budget).
2. **Requirements Elicitation and Analysis –** Talk to stakeholders (people who will use or be affected by the system) to understand what they need or expect.
3. **Requirements Specification –** Write down all the requirements in clear detail.
4. **Requirements Validation –** Make sure the written requirements are correct, complete, and really match what the customer wants

**Requirement Analysis – Key Challenges**

While doing requirement analysis, some common problems (challenges) are:

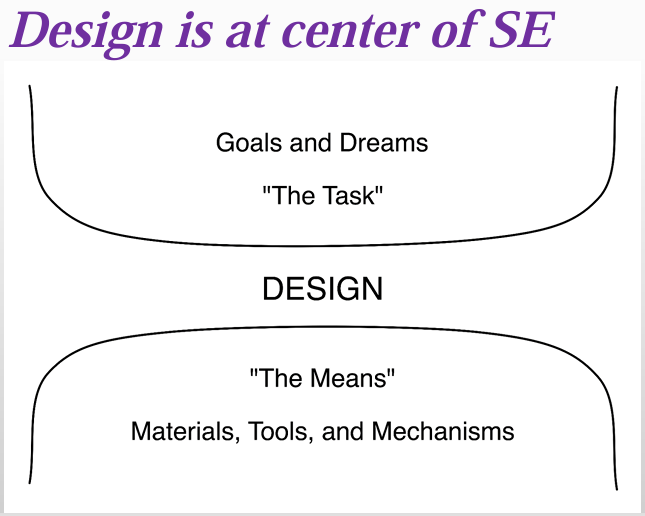
1. **Stakeholder Issues** – Stakeholders (users, customers, managers) may have **different opinions** or may not explain their needs clearly.
2. **Engineer/Developer Issues** – Developers may **misunderstand the requirements** or may not have enough technical knowledge about the domain.
3. **Management Issues** – Managers may set **unrealistic deadlines, goals, or priorities** that make requirement gathering harder.
4. **Expert Issues** – Sometimes experts are **not available** or may give advice that is **too complex** for others to understand.
5. **Budget Issues** – The **cost limit** may restrict what features can be included.
6. **Planning Issues** – Poor or unclear **planning** can lead to missing, incomplete, or confusing requirements.
7. **Novelty Issue** – If the project is **new or unique**, there may be no previous examples, making it difficult to define requirements.

**Design**

* Design means planning how the application (software) will be built.
* It explains the different parts of the system and how these parts will work together.
* The design is usually written down in the form of documents, diagrams, and text to give a clear picture before actual coding starts.

**Some key points**

* Design helps us connect tools and processes with our goals.
* Without design, tools and technologies are just random things with no direction.



**Design Continue…**

* No matter how much software engineering improves, design will always be the **central part.**
* Design **produces artifacts** (documents, diagrams, models) that help in reaching the goals.
* It **turns new ideas** into a practical and **workable form.**
* It creates a **balance between system complexity and project goals** (by dividing system into modules or layers), so the system is both useful and manageable

**Implementation**

* **Implementation** means writing the actual **code** for the system.
* The code should always be written **according to the design** (never randomly).
* Before compiling (running) the program, a developer should **carefully check the code** by reading it line by line.
* Here, **“correct”** means the code really does what the requirements and design expect.

**What is *Author-Inspection*?**

* **Author-Inspection** is when the **developer who wrote the code** checks it themselves.
* The author (programmer) carefully **reviews their own code** to make sure it is correct, clear, and matches the requirements, **before giving it to others or running tests**.

**Testing**

* Testing means giving input to the application and checking if the output matches the requirements written in the Software Requirements Specification (SRS).
* It helps to find defects (mistakes or errors) in the system.
* **Important:** Testing can prove defects exist, but it can never prove that the software has no defects at all.

**Types of Testing**

1. **Black-Box Testing –** Testing the system from the outside (focus only on input and output, without looking at the code).
2. **White-Box Testing –** Testing the system from the inside (looking at the actual code, logic, and internal paths).

**Verification vs Validation (V & V)**

**Verification = Checking the process**

* Ask: *Did we build the product correctly according to the design/specifications?*
* Focus: Following the rules and steps properly.
* Example: Suppose you are baking a cake. Verification means checking the recipe step by step (Did I add sugar? Did I preheat the oven?).

**Validation = Checking the final result**

* Ask: *Did we build the right product that the customer actually wants?*
* Focus: Meeting the user’s needs.
* Example: After baking, validation means tasting the cake to see if it’s actually delicious and what the customer wanted.

**Testing – Tips for Good Testing**

1. **Test early and often** – Start testing as soon as possible and keep testing regularly.
2. **Test with extreme values** – Try very **small** and very **large** inputs (e.g., 0, max limit) to see how the system reacts.
3. **Test borderline values** – Check values at the **edges** (just before or after the valid range).
   * Example: If age input is allowed from 18–60, test with 17, 18, 60, 61.
4. **Test with “illegal” values** – Give **invalid inputs** (like letters where numbers are expected) to see if the system handles errors properly.
5. **Vary test cases** – Use **different kinds of data** to cover more situations.
6. **Don’t repeat the same test data** – Unless required, avoid running the **same test again and again**, use new inputs to find hidden issues.

**Maintenance**

* **Maintenance** means the work done on the software **after it has been delivered to the customer**.
* Its purpose is to keep the software **useful, correct, and up to date**.

**Types of Maintenance**

1. **Corrective Maintenance (Defect Removal)**

* Fixing errors or bugs found after the software is delivered.
* Example: Repairing a broken login function.

1. **Perfective Maintenance (Enhancement)**

* Adding new features or improving existing ones.
* Example: Adding a “dark mode” option to an app.

1. **Adaptive Maintenance**

* Modifying the software so it works in a **new environment** (new hardware, OS, or database).
* Due to need
* Example: Updating an app to run on the latest Android/iOS version.

1. **Preventive Maintenance**

* Making changes to **avoid future problems or failures**. No defect in the system but you think if some event triggered then the system can be crash or fail.
* Example: Cleaning up code or updating libraries to reduce security risks.

**What is design?**

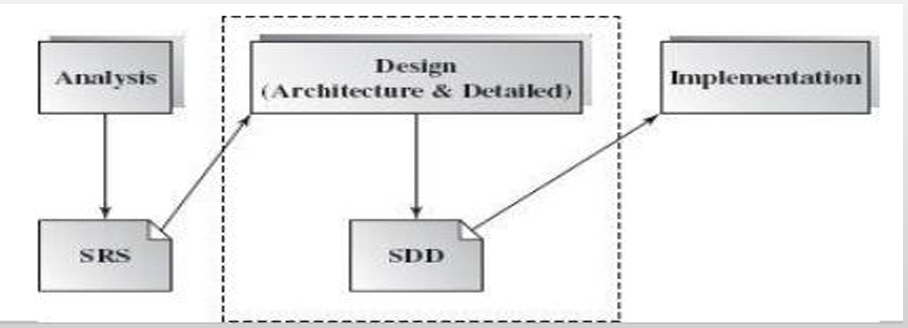
* Design means creating a plan for building an object or system.
* It works like a roadmap or a strategy to achieve a specific goal.
* It is about how the system functions.
* In software design, the goal is to create a model that meets all the customer’s requirements and makes implementation (coding) successful.
* Design also defines:
  + **Specifications** (what to build)
  + **Plans and processes** (how to build)
  + **Costs and activities** (resources needed)
  + **Rules and constraints** like legal, social, environmental, safety, and economic limits.

**In short:** Design is a detailed plan that helps turn ideas into a working system while respecting all requirements and restrictions.

**Design and Requirements**

* Requirements specification describes WHAT the system should do.
* Design describes HOW the system will actually work.
* It breaks the system into smaller parts (decomposition) so it is easier to understand and build.
* This division also helps to split the work among different developers in a team.
* Design also ensures non-functional requirements like good performance, easy maintenance, and reusability.
* It also considers the target technology (like databases, programming language, or platforms).

**In short:** Design is the step that converts “what to build” into “how to build it.”

****

**Design as Decision Making**

In requirement engineering and design, a lot of decisions have to be made:

1. **Decisions of how to accomplish something** – Example: Which algorithm or method to use.
2. **Decisions of how to represent something –** Example: Should data be stored in a database table, a file, or a data structure.

**Relation between Design and Decision Making**

* **Design =** A collection of decisions about how the system will be built.
* Each design choice affects how well the system meets requirements.

**Types of Decisions in Design**

* **Capturing decisions –** Writing down what decisions were made and why.
* **Analyzing and comparing alternatives –** Looking at different options before choosing the best one.
* **Managing alternatives –** Keeping track of possible solutions.
* **Reflecting on past decisions –** Learning from previous choices to make better designs in the future.
* **Creating reusable knowledge –** Turning design experience into guidelines or patterns that can be reused in future projects.

**In short:** Design is really about making smart decisions, recording them, and reusing that knowledge to build better systems in the future.

**Design – Software Architecture**

* **Software Architecture** is about the **big design decisions** in a project.
* These decisions are very important—if made wrongly, the **whole project can fail**.

**What Software Architecture Defines**

1. **Components of the system** – The main parts or modules of the software.
2. **Interaction between components** – How these parts share functionality and data with each other.
3. **Control management** – How control (flow of execution) is passed from one component to another.

**In short:** Software architecture is the “high-level design” that shows the main building blocks of a system and how they work together.

A diagram of software development

AI-generated content may be incorrect.

**Desing Approaches**

**1. Top-Down Design**

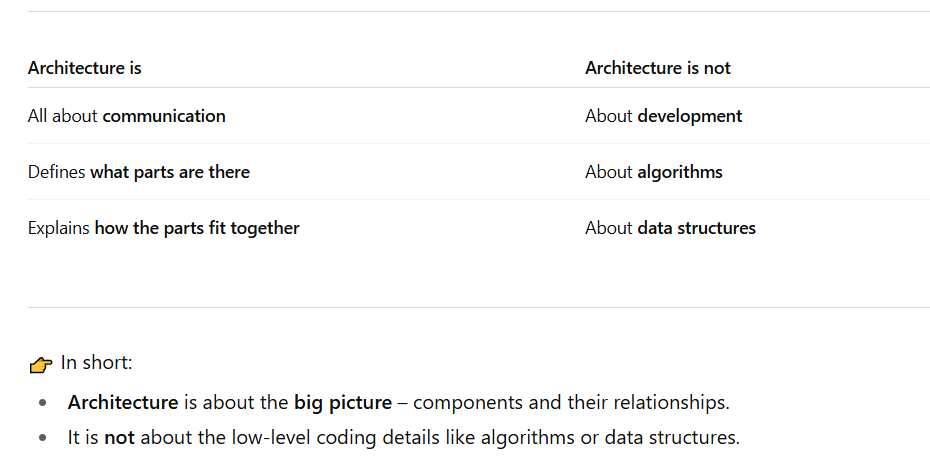
* Start with the **big problem (whole system)**.
* Break it down step by step into **smaller sub-problems**.
* Keep dividing until each part is small enough to solve.
* **Limitation**: Pure top-down is not practical for very large systems because it can become too complex.
* **Example:** Designing a car by first planning the **whole car**, then dividing into **engine, body, wheels**, and then into smaller components.

**2. Bottom-Up Design**

* Start with **existing small solutions (components/modules)**.
* Combine and adapt them to **fit the larger problem**.
* Builds the big system by assembling smaller, already available pieces.
* **Example:** Designing a car by first gathering **ready-made parts** (engine, tires, seats), then assembling them into the full car.

**What is architecture?**

Software architecture is the “high-level design” that shows the main building blocks of a system and how they work together.

****

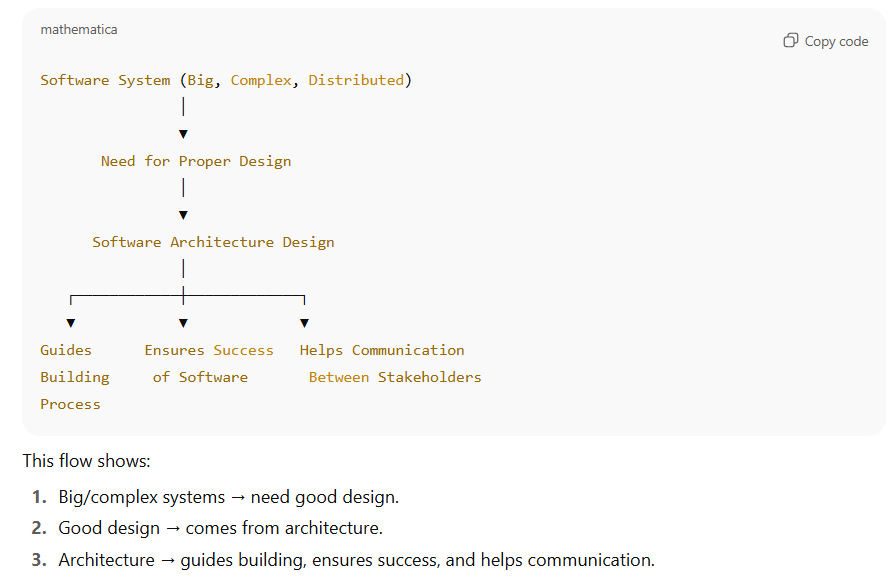
**Importance of Design**

As software systems keep getting bigger, more complex, and spread out, it becomes very important to design them properly.

* Every software, no matter what it is made for, should have a clear architecture design that guides how it is built and developed.
* The success of a software product mostly depends on how good its architecture design is.
* Architecture is like a small and clear model that shows how the system is organized and how different parts work together.
* These architecture designs also help in communication between all the people (stakeholders) who are involved in making the system.

**Q: Where most of the work is done in software?**

**Ans:** Most of the work is done in the maintenance phase. But for the maintenance system understanding is most important so in order to understand the system design plays an important role.

****

**Importance of Design**

* If the design is poor, the software may not meet the required needs.
* It may not adjust easily when new requirements or changes are needed in the future.
* It may not be reusable, meaning the code can’t be used again in other projects.
* It may show unexpected errors or slow performance.
* Without good planning in the architecture stage, making the software can take too much time and cost more money.

**Q: How to Ensure Reusability?**

* Modular Design
* Follow Coding Standards

**Level of Design**

**Architectural Design (High-Level Design)**

**Architecture** shows the **overall structure** of the system – the main modules and how they connect to each other.

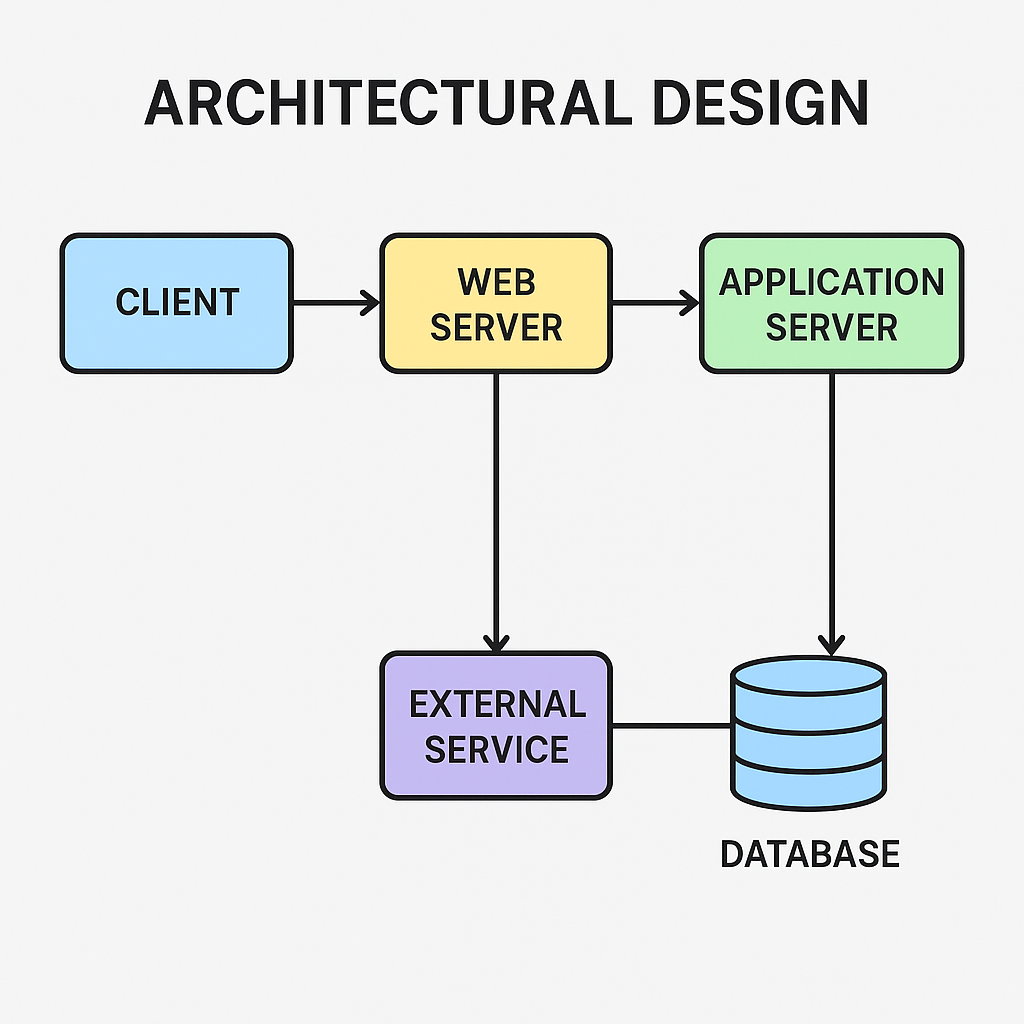
It is a design that focuses on the **main use-cases** (important features the system must handle).

It also takes care of **non-functional requirements** like performance, security, and reliability.

Architectural design is usually **hard to change later (like building)**, so it must be planned carefully.

A **high-level design document** normally has an **architecture diagram** that shows:

* The main components,
* How they interact (interfaces),
* The networks or connections needed.

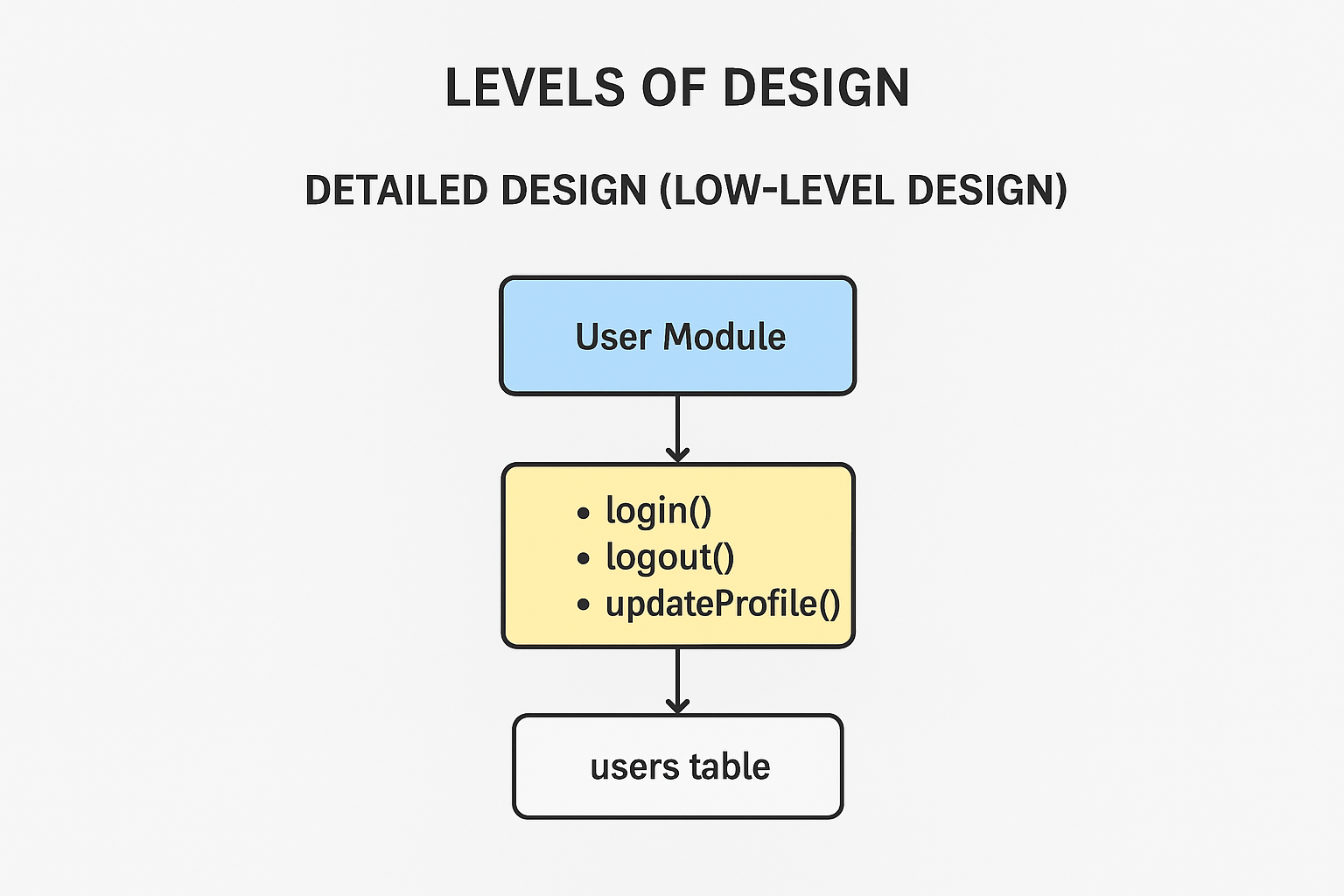


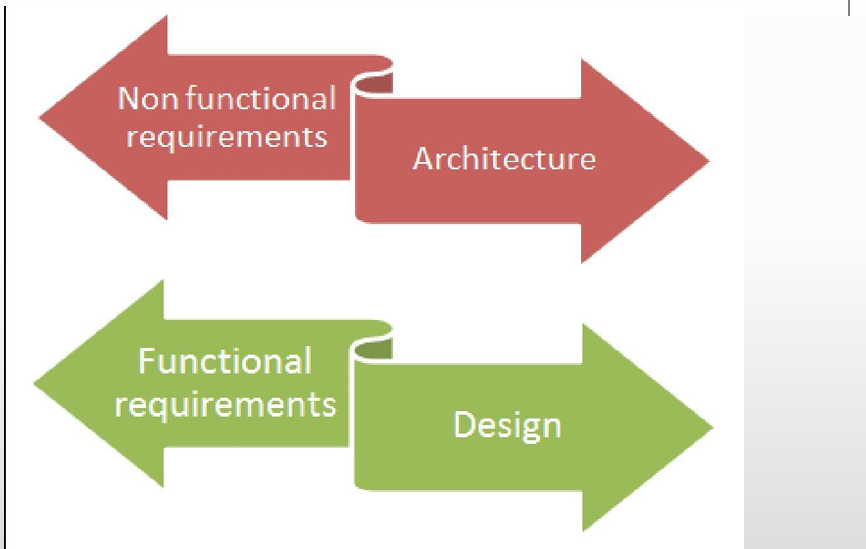
**Detailed Design (Low-Level Design)**

* Shows the **inside structure** of each main module.
* Considers the **programming language** to be used (Java, Python, etc.).
* Provides **enough detail** so developers can directly write the code.

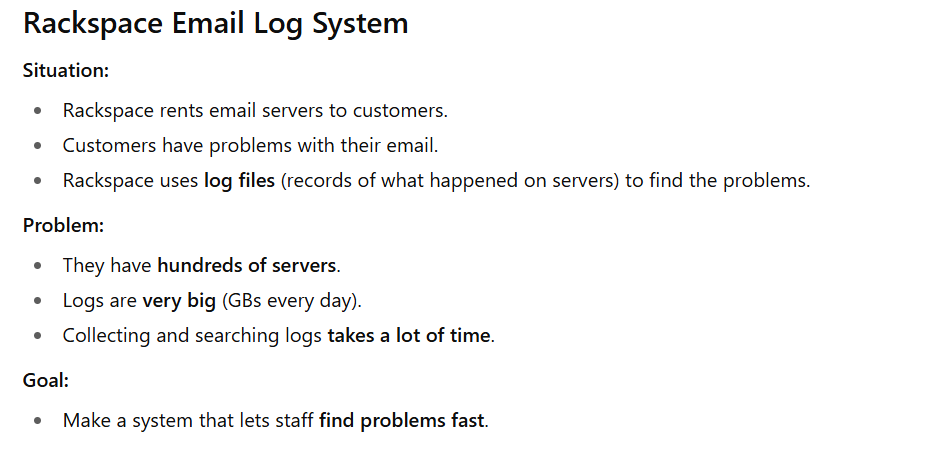
**Example**

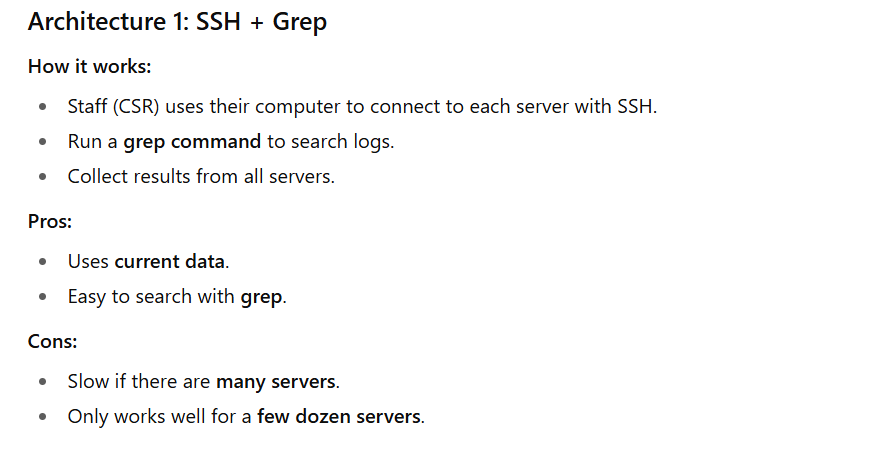
* High-Level Design: "User Module"
* Low-Level Design: "User class with login (), logout (), update-Profile () methods, and a user’s database table."





**Scenario**

****

****

**A diagram of a computer network

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A diagram of a computer network

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A diagram of a computer network

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Lecture 2**

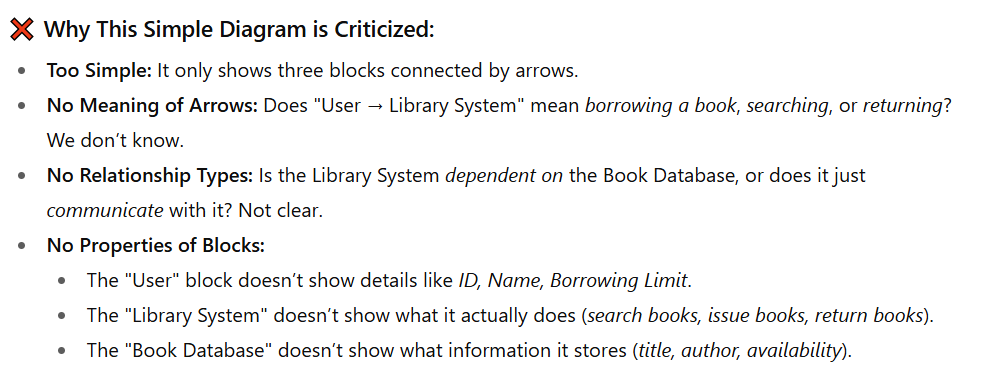
**Architecture Styles**

**Basic Architectural Representations**

The most common way to explain software architecture is by using **simple and informal block diagrams**. These diagrams show the main parts (entities) and how they are connected (relationships).

However, **people often criticize** these diagrams because they are too simple. People say these diagrams are not good enough because they only show boxes and arrows without much explanation.

* **They don’t explain the exact meaning of the connections.**  
  The arrows or lines just show that things are connected, but they don’t tell us *how* they are connected or *what kind of connection* it is (for example: “sending data,” “controlling,” “storing,” etc.).
* **They don’t explain the types of relationships.**  
  We can’t tell whether the parts are “dependent,” “communicating,” “sharing data,” or something else.
* **They don’t explain the clear details (properties) of each part of the architecture.**  
  The diagram doesn’t show the important features of each block, like what it does, what inputs it takes, or what outputs it gives.



**Better Representation**

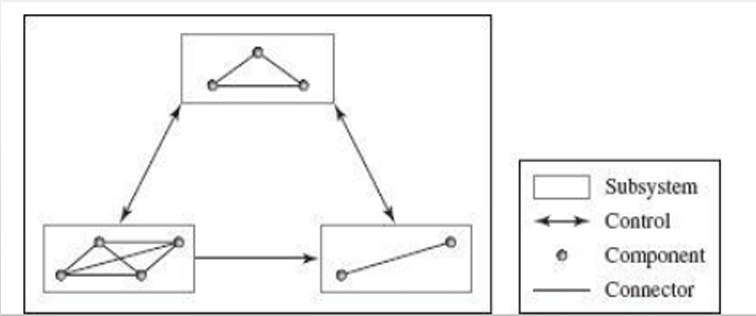
A screenshot of a computer code

AI-generated content may be incorrect.

**Box and Line Diagrams**  
These diagrams show the general shape or outline of an architecture design.

They are very **high-level and abstract** – they don’t explain what kind of relationships exist between the parts, or the clear details (visible properties) of the sub-systems.

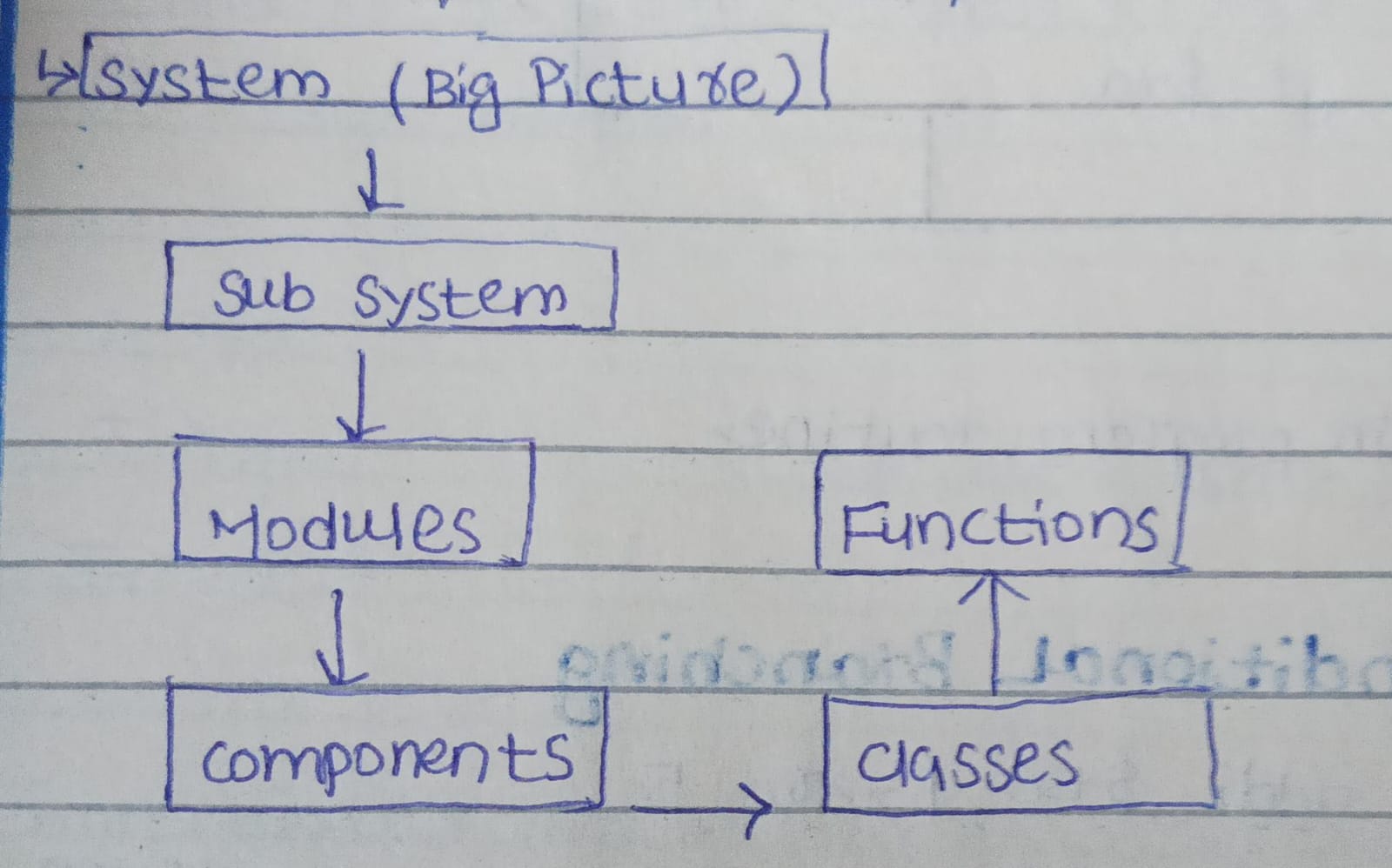
Still, they are **useful for talking with stakeholders** (people interested in the project) and for planning the project.



**Continue…**

* The **parts of a system** can be divided based on what they do (functionality), where they are placed (location), and how they look or work when the program is running (runtime images).
* The **structure of the system** (topology) shows how these parts are arranged. This can be in different forms like layered, flat, star-shaped, centralized, or spread out (distributed).

**How System made up off?**



**Use of Architectural Models**

A full software architecture description should not only explain the parts (elements) and how they connect, but also the rules (constraints) and how the system will behave when running. This helps developers clearly understand *what* to build and *how* to build it.

**The main tasks of a software architect are:**

* Divide the system into smaller parts (subsystems) and explain how these parts communicate with each other.  
  **Example:** In an online shopping app, divide into *User System*, *Order System*, and *Payment System*.
* Make sure each software part can be set up (configured), built (developed), delivered, installed (deployed), and even replaced later if needed.  
  **Example:** The Payment System can be updated or replaced with PayPal without changing the whole app.
* Define each part’s interface clearly so it can connect and work properly with other parts or subsystems.  
  **Example:** The Payment System provides a clear interface (like a menu) with an option called **makePayment()**. When the Order System needs to take money, it just calls **makePayment()** from the Payment System’s interface.
* Set up how different subsystems control and interact with each other, for example through data flow, control flow, or message passing.  
  **Example:** When a user places an order, data flows from *User System → Order System → Payment System*.
* Think about and compare different architecture styles, then choose the one that best fits the specific problem.  
  **Example:** Choose a layered architecture for an online shopping app (UI layer, business layer, database layer).

**Architectural Design Decisions**

Architectural design is a creative process, so the way it is done can change depending on what kind of system is being built.

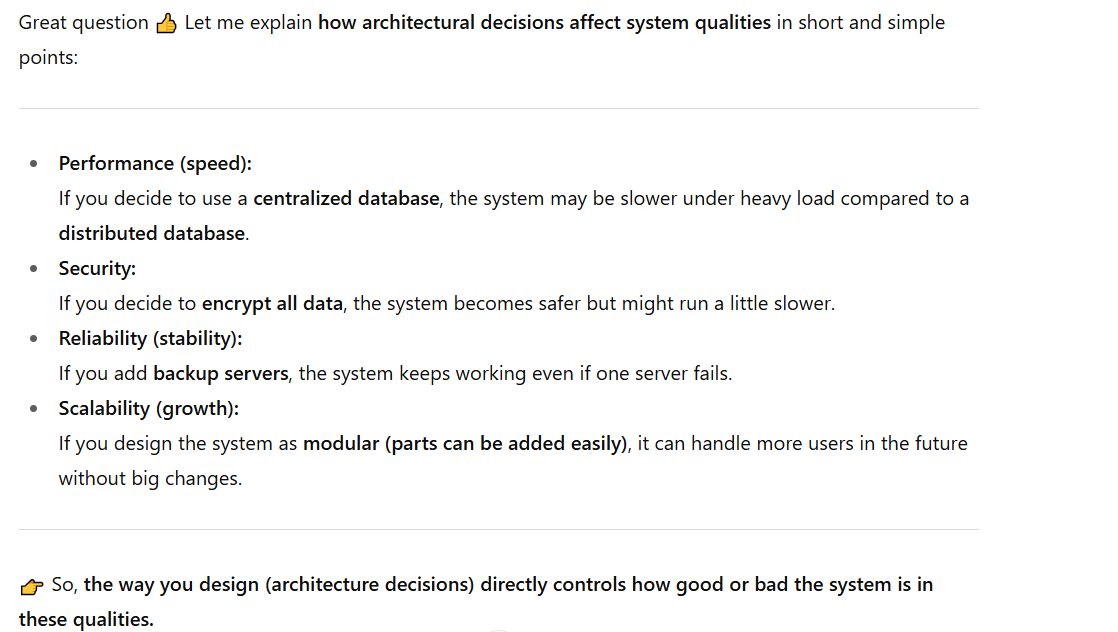
Still, there are some common decisions that are part of almost every design process, and these decisions usually affect the system’s non-functional features (like performance, security, reliability, or scalability).

**Some common decisions?**

**A close-up of a text

AI-generated content may be incorrect.**

**How does architecture affect the nonfunctional requirements?**



**Architectural Design Decisions** **Continue…**

Should we use a ready-made (generic) application architecture that already exists?  
**Example:** Use a standard e-commerce architecture instead of designing from scratch.

How should the system be spread out (distributed)?  
**Example:** Run the database on one server and the application on another for better performance.

Which architectural style or model is the best fit?  
**Example:** Choose a client-server model for a banking system.

What method will we use to organize the system’s structure?  
**Example:** Use a layered structure (UI layer, business layer, database layer) for an online shopping app.

How will we break the system into smaller modules or parts?  
**Example:** Split into User Module, Order Module, and Payment Module.

What type of control strategy will manage how parts interact?  
**Example:** Use message passing so services communicate through messages instead of direct calls.

How will we check (evaluate) if the chosen design is good enough?  
**Example:** Test if the system can handle 10,000 users at once without crashing.

What is the best way to write down and document the architecture?  
**Example:** Use UML diagrams and written notes so developers can easily understand the design.

**Architectural Design Decisions Continue...**

**Architecture Reuse**  
Systems that belong to the same domain (area) often have similar architectures because they use the same main ideas.

A group of similar software products is made using the same basic design, and then small changes are added to match what different customers want.

**Architectural Views**  
Each architecture model shows only one side (view) of the system.

For example, it may show how the system is divided into modules, how processes work together when running, or how parts of the system are spread across a network.

To properly design and document the system, you usually need to show **multiple views** of the software architecture.