Software Design and Architecture

[Software Architecture] – Week 03, L01

Lecture Outlines

- **Fundamentals of software architecture**
  - ✓ What is software architecture?
  - ✓ Why is it important?

- **Key tasks in software architecture**
  - ✓ Stakeholders concerns
  - ✓ Identify architectural views, styles, and patterns
  - ✓ Identify major component's and interfaces
  - ✓ Evaluating and validating the Architecture
  - ✓ Introducing policies for design synchronicity

- **Problem-solving in architecture**

- **What's next…**

Fundamentals of Software Architecture

- Architecture **is**:
  - ✓All about communication.
  - ✓What ‘parts’ are there?
  - ✓How do the ‘parts’ fit together?
- Architecture **is not**:
  - ✓About development.
  - ✓About algorithms.
  - ✓About data structures.

Fundamentals of Software Architecture

What is Architecture ?

- Refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures.

- Architecture captures three primary dimensions:
  - ✓ Structure
  - ✓ Communication (behavior)
  - ✓ Nonfunctional requirements

- Focuses on those aspects of a system that would be difficult to change once the system is built.

Fundamentals of Software Architecture

- Let's get straight to the point, formally, we define software architecture as
- (1)The **foundational software design activity** that evaluates and translates **software requirements** (both functional and non-functional) into a **collection of design elements** that specify structural and behavioral aspects of the **major components of the system**, together with their provided quality and interrelationships required to **support the detailed design** and **construction** of software systems.

- (2) and the **product** resulting from such activity.

Extra: Software architecture **(big-picture)**

1-Process is deciding how the **major components** of the system (frontend, backend, database, etc.) will be structured and interact.

2-**Product** is the high-level diagrams, component models.

- **Example**
- Imagine you are building an **e-commerce website** (like Amazon).

- **Architecture activity (process):**
  You decide the system needs:
  - A **frontend** (user interface) where customers browse and order.
  - A **backend** (server logic) to process orders.
  - A **database** to store products and users.
  - A **payment service** to handle credit cards.

- **Architecture product (blueprint):**
  A diagram showing these components (Frontend → Backend → Database → Payment Service) and how they talk to each other.

Extra: Comparison between **Software Architecture** and **Detailed Design**

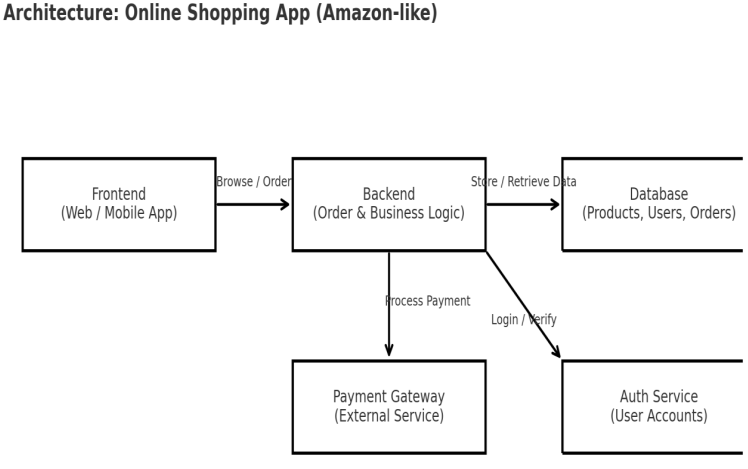
- **Software Architecture (big picture)**
  - **Process:** Deciding how the major components of the system (frontend, backend, database, etc.) will be structured and interact.
  - **Product:** The blueprint (high-level diagrams, component models).
  - **Example:** For an online shopping app → deciding that you need a frontend (website/app), backend (server logic), payment service, and database.

- **Software Design (detailed picture)**
  - **Process:** Working out the details inside those components — the classes, functions, data structures, algorithms, and how they solve the requirements.
  - **Product:** The design diagrams/documents (class diagrams, sequence diagrams, flowcharts, etc.).
  - **Example:** For the same online shopping app → designing how the “ShoppingCart” class works, what methods it has (addItem(), removeItem(), checkout()), and how it communicates with the Payment class.

Extra: Comparison between Software Architecture and Detailed Design

- 1. **Architecture = High-level Interaction (Big Picture)**
- **Focus:** Which components exist and how they connect.
- **Shows:** Subsystems or major modules, not individual classes.
- **Interaction** = component-to-component.
- **Example (Online Shopping App, Architecture):**

  Frontend ↔ Backend ↔ Database ↔ Payment Gateway. The diagram would show arrows like: "Frontend sends requests to Backend," "Backend stores data in Database," "Backend calls Payment Gateway."
- **So here, we don't care how each part works internally — only which components exist and how they talk.**



Extra: Continue

2. Design = Low-level Interaction (Detailed Picture)

Focus: How components are implemented internally.

Shows: Classes, methods, data structures, algorithms.

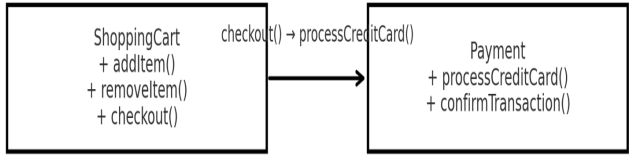
Interaction = object-to-object / class-to-class.

Example (Online Shopping App, Design):

In the **Backend**, you zoom in and design:

- **ShoppingCart class** (methods: addItem(), removeItem(), checkout())
- **Payment class** (methods: processCreditCard(), confirmTransaction())
- So, the **ShoppingCart** calls **Payment.processCreditCard()** when user **checkout**.
- **So here, we *do* care about method names, data flow, and logic.**

Detailed Design (Backend Example): Classes & Methods



Extra: you can use the **same diagram types across both phases** In **Architecture and Detailed design but:**

- In **Architecture**, they are **high-level** (just showing subsystems, actors, or major flows).
- In **Design**, they are **detailed** (down to classes, methods, attributes, exact interactions).

Fundamentals of Software Architecture

- From this definition, a few things are of interest and need further explanation:
  - ✓ Foundational design
  - ✓ Transforming requirements (from textual base into a graphical base)
  - ✓ Collection of design elements (diagrams)
  - ✓ Major system components together with their provided quality
  - ✓ Support (an input to) detailed design and construction
- Let's take a more detailed look at these…

Fundamentals of Software Architecture

- On " ... **foundational software design…"**
  - ✓ Software architecture provides the groundwork "basic" essential for meeting requirements (functional + non-functional)
  - ✓ This suggest that architecture is not an optional activity, it is a major activity.
  - New development efforts should approach software architecture as a forward engineering activity that leads to the implementation of systems and **Extra :not as a reverse engineering *analyzing the system after the system has been built and implemented***.
  - ✓ As foundational design, it is where designing for quality begins
  - ✓ Not considering quality attributes of the system during the software architecture activity can be a grave mistake!

Fundamentals of Software Architecture

- On "...**translates software requirements**..."
  - ✓ Software architecture provides abstractions for software requirements in the design domain
  - ✓ Design elements (diagrams) need to be created so that there is a mapping between requirements and design element.
    - When we do this, we transform requirements from textual form into a graphical form
    - This allows us to provide easy means for tracing requirements through the development life-cycle.
  - ✓ To create design elements from requirements, it is assumed that requirements are understood, however, this is not always the case
    - This suggest that architects must be proficient in activities related to requirements engineering.
    - **Extra**: This statement means that before architects can create design elements based on requirements, they need to fully understand those requirements. However, understanding the requirements isn't always straightforward. Therefore, architects need to be skilled in requirements engineering, which involves gathering, analyzing, and clarifying what the system needs to do. This ensures that the design aligns with what is actually needed.

Fundamentals of Software Architecture

- On "…**collection of design elements**…"
  - ✓ No one structure or diagram can fully describe the software architecture.
    - Think about this: can you evaluate a system's usability and performance with one diagram?
  - ✓ This suggest that architectures are composed of multiple structures (or diagrams).
  - ✓ This is also because architecture design aims for communicating the system structure with different stakeholders. Therefore, different diagrams are required for different stakeholders.

Extra:

Why one diagram is not enough in Software architecture?

- A single diagram cannot show everything (usability, performance, security, data flow, deployment).
- Different stakeholders (developers, testers, managers, customers) care about different views of the system.
- That's why multiple diagrams (structures) are used in software architecture.
- Example: Online Shopping App
  - Component Diagram – shows major parts (frontend, backend, database, payment) and their interactions (for architects).
  - Class/Sequence Diagrams – show detailed logic and method calls (for developers).

- In short: Software architecture = collection of diagrams. Each one highlights a different aspect of the system so the right people can understand what they need.

Fundamentals of Software Architecture

- On "…**major components of the system, together with their provided quality**…"
  - ✓ This means that architectural work focuses on the **major components**, the **quality properties**, and **services** (functionality) that these components exhibit and provide to other components.

Extra about the Figure:

software architecture, it's not enough to say "this component provides a service" — you also need to describe the quality of that service.

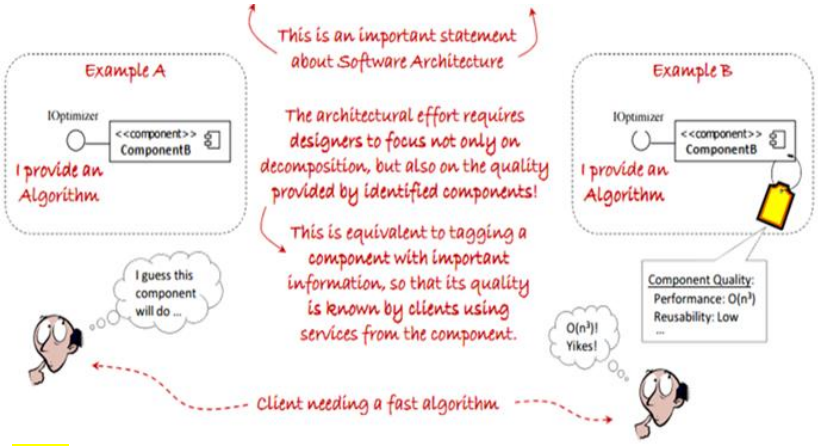
◆ Example A (Left Side)

- **Component B provides an algorithm.**
- **But no quality info is given.**
- **The client just assumes it's "good enough."**
- **Risk: The component might be too slow or not reusable.**

◆ Example B (Right Side)

- **Same component B, but now tagged with quality properties.**
- **Example:**
  - **Performance: O(n²) (slow for large inputs).**
  - **Reusability: Low.**
- **Now the client knows the limitations and may decide not to use it.**

◆ Why this matters Architecture is not just about structure, but also about qualities (performance, scalability, security, reliability). Adding quality tags makes components transparent → clients can make enlightened choices.



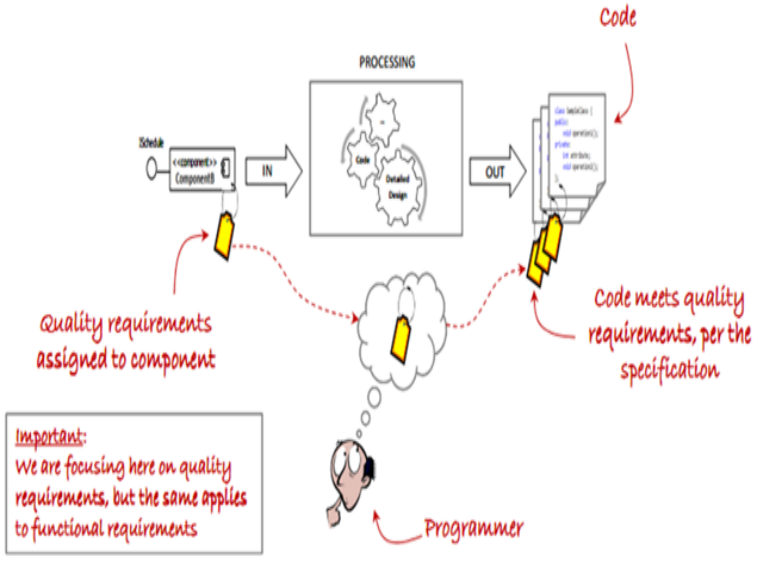
Extra: When we design software **architecture**, we don't just list the big parts (components).
We also describe:
Major components – the building blocks (e.g., Frontend, Backend, Database).
Quality properties – how they behave in terms of performance, security, scalability, etc.
Services (functionality) – what they actually provide to other components.

Fundamentals of Software Architecture

- More on "…**major components of the system, together with their provided quality**…"
  - ✓ Expected quality requirements identified during architecture trickles down all the way to the implementation of components.

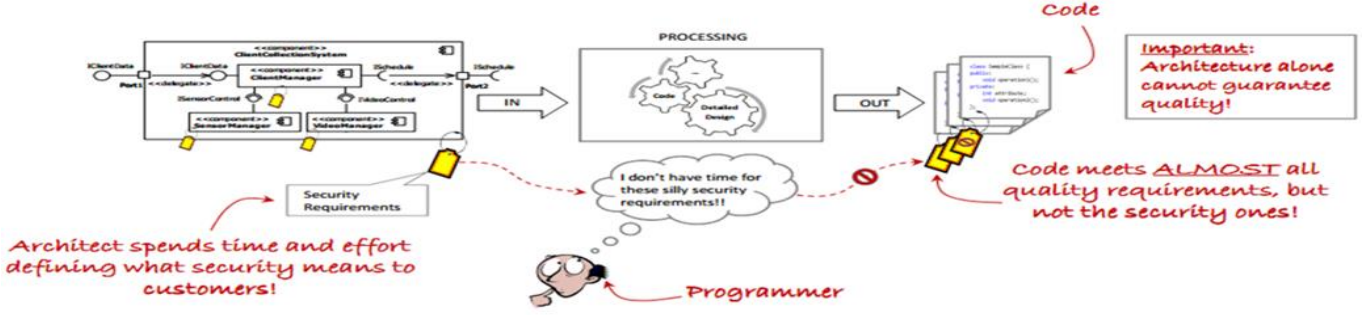
Extra:

1. **Assign quality to components**
   - In the architecture, each component is tagged with expected qualities.
   - **Example**: Component B must be fast (performance), reliable, and secure.
2. **Design & Coding phase**
   - Programmers take these components into detailed design and coding.
   - The quality tags act like requirements guiding how the code should be written.
3. **Output (code)**
   - The produced code should meet the quality requirements (as specified in the architecture).
   - **Example**: If performance was required, the code should use efficient algorithms.
4. **Important note**
   - The same approach applies not only to quality requirements (e.g., performance, security) but also to functional requirements (what the system should do).



Fundamentals of Software Architecture

- On "…**support detailed design and construction**…"
  - ✓ Although **architecture** focuses on the quality properties of the system, it must also result in a design that supports efficient **detailed** design and **construction** of the system.
    - Architects benefit greatly from having proficiency in general programming design concepts.
  - ✓ This suggests that architecture alone *cannot guarantee the quality of the system*!



•Extra:
•Architect defines **quality requirements** (e.g., security).
•Programmer ignores some (e.g., "no time for security").
•Final code meets **most qualities** but **misses security**.
•**Result:** Architecture alone ≠ guarantee → qualities must be **implemented in code**.

Fundamentals of Software Architecture

- To wrap up …
  - ✓Architectural design should be an early stage of the system design process
  - ✓Represents the link between specification and design processes
  - ✓Often carried out in parallel with some specification activities
  - ✓It involves identifying **major system components** and their **communications** with required **quality attributes**

Key Tasks in Architectural Design

- Formally, key tasks that need to be performed during the software architecture design effort include:
  - Identifying **stakeholders** concern (requirements & quality attributes)
  - Identifying appropriate architectural **views**
  - Identifying architectural **styles** and **patterns**
  - Identifying **influences** of architectural decisions in organizations
  - Identifying the system's major **components** and interfaces
  - **Evaluating** and validating the architecture
  - Establishing policies for ensuring architectural design **synchronicity.**
  - **Extra: synchronicity means** It means making sure **all parts of the system follow the same design rules** so everything fits together**.**
  - **Example:** If the rule is *"use one database system (MySQL)"*, then all teams must use **MySQL**, not mix MySQL in one module and MongoDB in another.

Extra:

1. Architectural Views

•**Definition:** Different perspectives to describe the system for different stakeholders.

•**Purpose:** Communicate structure and behavior clearly to users, developers, testers, managers, etc.

Example:

Physical (deployment) view: servers, nodes, network.

2. Architectural Styles

Definition: A **general template/approach** for organizing system structure.

Purpose: Provides a **high-level solution shape**.

Example: **Distributed System.**

3. Architectural Patterns

Definition: Reusable solutions to recurring and repeated architectural problems (more specific than styles).

Purpose: Guide detailed design decisions inside the style.

Examples:

Client –Server, Filters and pipes

Key Tasks in Architectural Design

- **Identifying stakeholders concerns**
  - ✓ Stakeholders are persons, groups, or organizations that have a **direct** or **indirect** stake or interest in the system.
    - They include systems engineers, software engineers, hardware engineers, project management, customers, testing teams, quality assurance teams, members of the configuration management team, etc.
  - ✓ A stakeholder's **concern** provides high-level information about desired **characteristics** of the software system.
    - The software architect must ensure that the software to be developed addresses the concerns of all stakeholders." it considers the needs and interests of everyone involved."
    - Stakeholders' concerns serve as driving force behind architectural decisions (important)

Extra:

Direct Stakeholders	Indirect Stakeholders
Directly use or operate the system	Do not use it directly, but are affected by it
Drive **functional requirements**	Drive **non-functional / external requirements**
Examples: End Users, **Admins**, Developers, Testers	**Examples:** Regulators, Investors, Marketing Team, **Managers**

Key Tasks in Architectural Design

- **Identifying stakeholders concerns (cont.)**
  - ✓ The software architect must identify and understand the different ways stakeholders influence the system.
    - These need to be elicited before any design effort can begin.

- **Extra**: Imagine you're designing a new mobile app for a university. The stakeholders include students, professors, and administrators.
  - **Students** want an easy way to view their schedules and grades.
  - **Professors** need a tool to upload grades and communicate with students.
  - **Administrators** require features to manage courses and student records.
- Before you start designing the app, you need to talk to each group to understand their specific needs and concerns.
- **For example: Students** might want a user-friendly interface, while **Administrators** are more concerned with security and data management.

Key Tasks in Architectural Design

➢ **Some important quality attributes of software systems that might concern the stakeholders .**

Quality	Description
Usability	The degree of complexity involved when learning or using the system.
Modifiability	The degree of complexity involved when changing the system to fit current or future needs.
Security	The system's ability to protect and defend its information or information system.
Performance	The system's capacity to accomplish useful work under time and resource constraints.
Reliability	The system's failure rate.
Portability	The degree of complexity involved when adapting the system to other software or hardware environments.
Testability	The degree of complexity involved when verifying and validating the system's required functions.
Availability	The system's uptime.
Interoperability	The system's ability to collaborate with other software or hardware systems. Extra: interoperability is when different computer programs or devices can easily work together. For example, Microsoft Office and Google Workspace both can open, edit, and save the same types of files like .docx for documents. This means you can make a file in Microsoft Word and then open and edit it in Google Docs without any problems. This is really helpful for people who use different programs but need to work on the same files.

Extra:

None functional Requirements (NFR) Versus Quality Attributes

We can say they are closely related but not exactly the same.

1. NFR (**system-level**):
   Describes *how well* the system should perform , so it is a global quality goal (e.g., performance, security).
2. Quality Tag (**component-level**):
   Assigns that goal to a specific component so it's clear what part of the system must meet it.

- **Example** (Online Shopping App)System
- **NFR (Performance):**
  - ✓ "System must process 1000 orders per second."
- **Quality Tags:**
  - ✓ **Backend Order Service Component** → Performance ≤ 100 ms per order.
  - ✓ **Database Component** → Performance ≥ 5000 queries/sec.

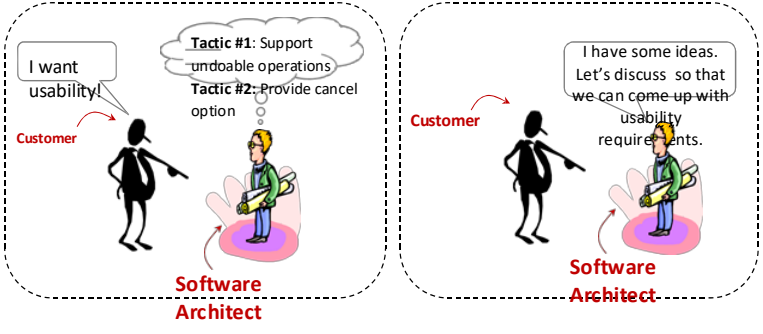
Key Tasks in Architectural Design

- Notice that these quality attributes also describe **high-level information** about desired characteristics of the software system.
  - ✓ In their current form, they are **insufficient to develop** the system.
  - ✓ For a system to exhibit any of these qualities, **design decisions (tactics)** must be made to **support** the achievement of these qualities.

Key Tasks in Architectural Design

- The term **tactic** is a design decision that affects the control of a quality attribute response.

➢In many cases, these quality attributes are identified through the requirement phase, leaving the architect responsible for specifying how to meet these quality attributes. During this process, **tactics** are identified for each desired quality attribute. (important)



Extra: Tactic: A **design decision** the architect makes to achieve that quality attribute.

In short:

Quality attribute = goal (what we want).

Tactic = design decision (how we achieve it).

Examples

1. **Quality Attribute:** Performance
   Tactic: Add caching to reduce response time.
2. **Quality Attribute:** Security
   Tactic: Encrypt all sensitive data.

So you can say:

A tactic **supports** a quality attribute.

A tactic **controls** the response of a quality attribute.

A tactic is the way we **achieve** a quality attribute.

Key Tasks in Architectural Design

Examples on some tactics that relate to the following qualities:

- Tactics for ***Security***
  - ✓ Resisting Attacks: authenticating users, Limit exposure, Limit access, only on need-to-know basis, etc.
  - ✓ Detecting Attacks: intrusion detection
- Tactics for ***Testability***
  - ✓ Event logging : log data and operations throughout the system. Allow testers to enable/disable this feature so that when enabled, events are displayed in the console to give insight into the system's operations and data.
- Tactics for ***Modifiability***
  - ✓ Localize changes: modularization, abstraction, encapsulation
  - ✓ Prevention of ripple effects: encapsulation, reduce coupling
- Tactics for ***Availability***
  - ✓ Redundancy, Task monitor, Watchdog timer , etc.
- Tactics for ***Performance***
  - ✓ Increase computation efficiency, reduce computational overhead, introduce concurrency, etc

Extra: Tactics are specific design decisions that help ensure the software meets certain quality attributes.

•**Security**: To protect the system, we might require user authentication (like passwords) and limit access to sensitive data. **For example**, only admins can access student records.

•**Testability**: By logging events, testers can see what the system is doing. **For example**, if a bug occurs, the logs can help trace what went wrong.

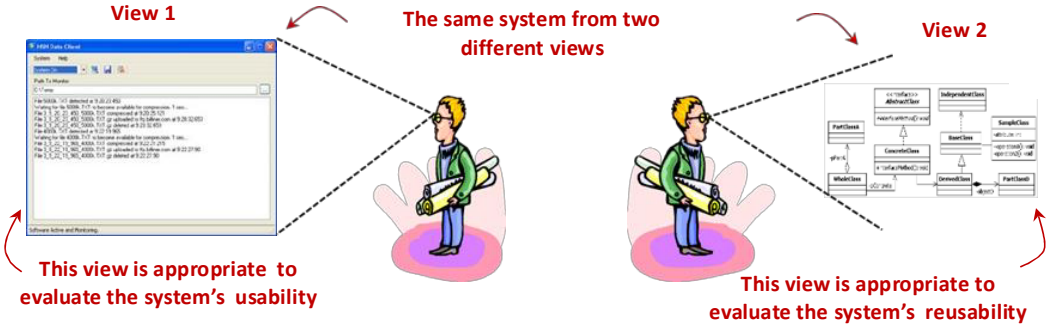
•**Modifiability**: We organize code into modules so that changes in one part don't break others. **For example**, if we update the login feature, it won't affect the grade submission module.

•**Availability**: We use backups and monitoring to ensure the system is always up. **For example**, if one server fails, another can take over without downtime.

•**Performance**: We optimize code to run faster, like reducing unnecessary calculations. **For example**, processing grades quickly even when many students are online.

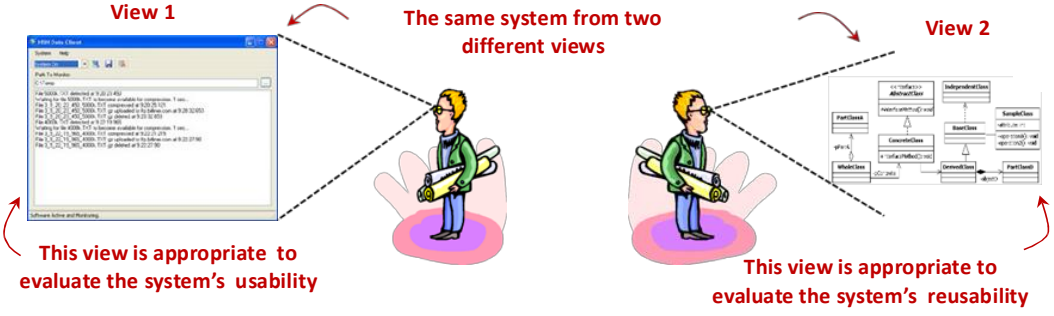
Key Tasks in Architectural Design

- **Identifying appropriate *architectural views***
  - ✓ In complex software systems, there can be numerous stakeholders with different backgrounds.
    - These stakeholders have different perception about the system, which influence the way they evaluate the system's design



Key Tasks in Architectural Design

- **Identifying appropriate *architectural views***
  - ✓ For this reason, architectural designs must support **different architectural views** used to evaluate the design from a particular **stakeholder's perspective.**
  - ✓ An architectural ***view*** is a representation of the system.
    - Different representations are required to evaluate certain properties of the system.



Key Tasks in Architectural Design

➢ **Identifying architectural styles and patterns**

- ✓ They provide an overall strategy for designing a family of software systems. (e.g. distributed systems family)
- ✓ They provide reusable architectural solutions, documented in a way that is easily understood and applied.
- ✓ Decisions based on styles and patterns benefit from years of documented experience.
- ✓ The concept of architectural styles and patterns are fundamental to the efficient creation of software architectures.
- ✓ Today, numerous styles and patterns exist so architects must be aware of these so that they can identify and determine the appropriateness of a particular style or pattern for their system's design
- ✓ We will discuss major design patters in more details later in the course
- ✓ Architects must understand the effect of architectural decisions on customers, budget, schedule, and resource availability

Extra:

- ✓ Architectural styles and patterns are reusable design strategies that guide architects in building efficient systems while balancing customer needs, cost, and schedule.

Key Tasks in Architectural Design

- **Evaluating and Validating the Architecture**
  - ✓ Long and iterative process
  - ✓ Failure to do so can have significant impact in effort and cost incurred to develop the system.
    - It is well known that defects found earlier on in the development process take much less effort to correct than if found at later stages.
  - ✓ The result should determine if the software architecture is *sufficiently* complete to support the development of the system.
  - ✓ We'll have more to say about this later on in the course…

Extra: Evaluating and Validating the Architecture is crucial because it ensures the design is solid before moving forward.

Key Point: Fixing problems early in the process is cheaper and easier than addressing them later.

Outcome: The evaluation confirms whether the architecture is ready to support the system's development.

This process, though lengthy, helps avoid costly mistakes down the line:

Extra Example: Consider a large-scale e-commerce platform like Amazon.

- **Early Evaluation**: During the design phase, architects evaluate the system's ability to handle millions of users simultaneously, ensuring the architecture supports scalability and performance.
- **Result**: If they identify that the database design won't handle peak traffic, they can rework the design early on.
- **Impact**: Catching this issue early avoids major performance problems during high-traffic events like Black Friday. Fixing it later, after the system is live, would be far more costly and disruptive.

Key Tasks in Architectural Design

- **Introduce policies for design synchronicity**
  - ✓ Design synchronicity is a measurement of the degree of how well the software implementation reflects its design, both in architectural form and detailed design form.
  - ✓ Obviously, we want high synchronicity, but this is not always the case.
    - It is very easy to deviate from design, especially on multi-year efforts.
  - ✓ For any software architecture effort to result in successful implementation, all subsequent phases and activities must be synchronized with the architecture.
    - To do this, a well-defined and understood process must be in place.
    - This includes the maintenance phase, which can go on for years after a software has been deployed!

Extra: Design synchronicity
Meaning: How closely the final software matches the original design.

Example: Initial plan: Banking app with secure login + fund transfer.

During development: Team adds extra features (themes, chat) and changes security without updating design docs.

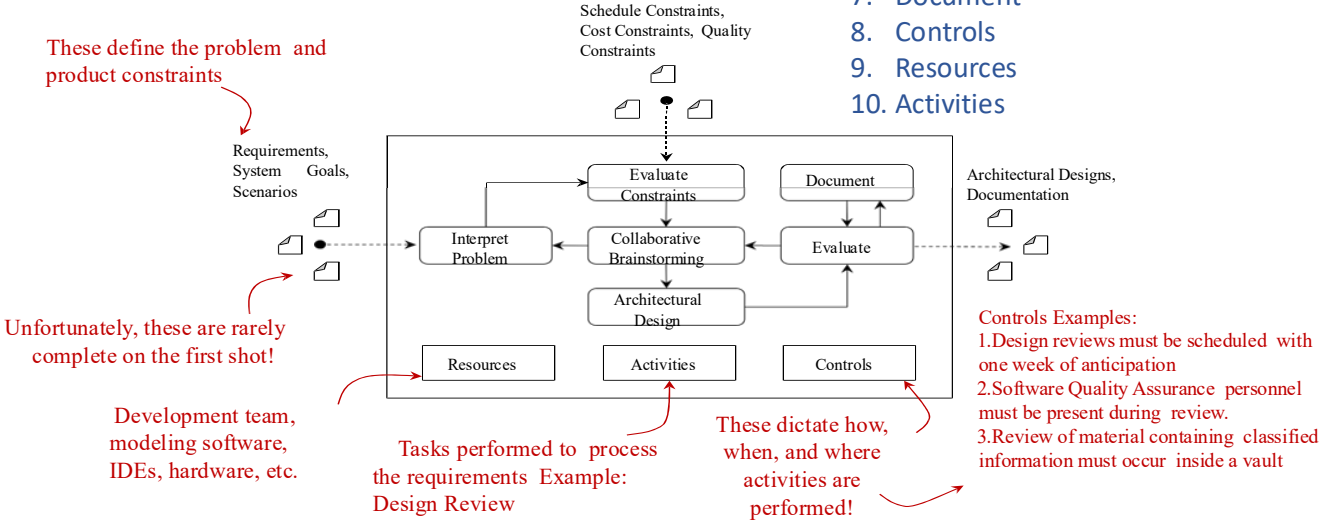
Result: Final product ≠ original design → low synchronicity.

Best Practice: Document, review, and approve all changes → ensures high synchronicity.

Problem Solving During Architecture

Extra:

1. Requirements, System Goals, Scenarios
2. Interpret Problem
3. Evaluate Constraints
4. Collaborative Brainstorming
5. Architectural Design
6. Evaluate
7. Document
8. Controls
9. Resources
10. Activities



Summary

- This session, we presented **fundamentals concepts** of software architecture and **key tasks** that need to be performed during software architecture, including:
  - ✓ Identifying stakeholders concerns (requirements)
  - ✓ Identify architectural views, styles, and patterns
  - ✓ Identify major component's and interfaces
  - ✓ Evaluating and validating the Architecture
  - ✓ Introducing policies for design synchronicity

- We've also mentioned issues with software quality and requirements (but not in details)

- Next session will focus on **requirements engineering**, providing enough information to successfully create good requirements.