
# Software Design and Architecture

## [Software Architecture] – Week 03, L02

# Lecture Outlines

- **Software Architecture Process**
  - ✓ Understand and Evaluate Requirements
  - ✓ Design the Architecture
  - ✓ Evaluate the Architecture
  - ✓ Document the Architecture
  - ✓ Monitor and control implementation
- **Requirements Engineering**
  - ✓ Elicitation
  - ✓ Analysis
  - ✓ Specification
  - ✓ Validation
- **What's next..**

# Software Architecture Process

- On a larger scale, the **process** for creating software architectures can be executed using the following tasks:
  - ✓ Understand and evaluate requirements
  - ✓ Design the architecture
  - ✓ Evaluate the architecture
  - ✓ Document the architecture
  - ✓ Monitor and control implementation

# Understand and Evaluate Requirements

- **Requirements engineering**
  - ✓ The discipline within software engineering that is concerned with the systematic approach to requirements specification, mainly through the following activities:
    - Elicitation
    - Analysis
    - Specification
    - Validation

- Software architects spend a great deal of time working with software requirements.
  - ✓ Even after requirements are specified, software architects find themselves going back and forth between requirements and design.
  - ✓ In some cases, architects are completely immersed(Deep and involved) in the requirements phase, playing a key role in the specification of requirements.

# Understand and Evaluate Requirements (Elicitation)

- **Elicitation**
  - ✓ Activity that deals with identifying stakeholders, uncovering what the customer needs and wants, and with determining the non-functional requirements.
  - ✓ Begins by identifying all *sources* of information that can be used to generate requirements.
  - ✓ Different sources can have similar but different visions for the system.
    - Common sources of requirements include:
      - Stakeholders
      - Goals
      - Domain knowledge
      - Operational and organizational environment

# Understand and Evaluate Requirements (Elicitation)

- **Common Elicitation Techniques**
  - ✓ Interviews
  - ✓ Facilitated meetings
  - ✓ Observation
  - ✓ Scenarios

# Understand and Evaluate Requirements (Analysis)

- **Analysis**
  - ✓ Requirements are analyzed in their raw form to address and solve issues such as requirements that are *contradicting*, *incomplete*, *vague*, or just *wrong*.
  - ✓ Extra: For example, if one requirement states the system must operate 24/7, but another says it needs daily maintenance downtime, these are contradictory and need to be resolved.
  - ✓ Allows architects to clear the air in regard to what needs to be done before devising more detailed designs.
  - ✓ The following tasks can be performed during analysis:
    - Requirement **classification**
    - Requirement **prioritization**
    - Requirement **negotiation**
    - Conceptual **modeling**

## Extra: Example: Real Software Example: Banking App Development

Imagine a team is developing a mobile banking app, and the initial requirements include:

| Requirement | Issue Found | Resolution |
|---|---|---|
| **R1: Secure login** | Vague – "secure" not defined (could mean password, 2FA, fingerprint, etc.). | Clarify the exact login method with stakeholders. |
| **R2: Fast & seamless user experience** | May **conflict** with R1 because strong security can slow down the app. | Optimize design to balance speed and security (e.g., efficient encryption, smar caching). |
| **R3: Multiple payment methods** | Incomplete – doesn't specify which methods (credit card, PayPal, crypto, etc.). | Define the required payment methods clearly. |
| **R4: 24/7 availability** | Contradicts the need for system updates/maintenance that may cause downtime. | Plan for rolling updates or redundant servers to keep service available. |

# Understand and Evaluate Requirements (Analysis)

## ➢ Requirement classification

- ✓ Performed for identifying the nature of each requirement
  - – Functional vs. non-functional
  - – Product vs. process
  - – Imposed vs. Derived

| Criteria | Description |
|---|---|
| Functional vs. Non-Functional | Classification that differentiates between requirements that specify the functional aspects of the system vs. the ones that place constraints on how the functional aspects are achieved. |
| Product vs. Process | Requirement placed on the system product vs. requirements placed on the process employed to build such product. |
| Imposed vs. Derived | Requirements imposed by stakeholders vs. requirements that are derived by the development team. |

Extra:

**1. Functional vs. Non-Functional**

- **Functional** → What the system does.

**Example**: "The system must allow users to log in".

- **Non-Functional** → How the system behaves.

**Example**: "The system must load within 2 seconds."

**2. Product vs. Process**

- **Product requirement** → what the delivered software must have or do.

**Example**: "The app must support online payments."

- **Process requirement** the **way of developing** that product (the steps, methods, or standards used).

**Example**: "The app must be developed using Agile methodology."

**3. Imposed vs. Derived**

- **Imposed** → Given directly by stakeholders.

**Example**: "The app must run on iOS and Android."

- **Derived** → Identified by the development team to meet higher goals.

**Example**: "To support both platforms, the app should use Flutter framework."

# Understand and Evaluate Requirements (Analysis)

## ➢ Requirement prioritization and negotiation

- ✓ Helps identify the most important functions of the software system.
- ✓ When done properly, it can help refine the projected schedule by determining which requirements need to be processed first.
- ✓ Can be used to determine different builds of the software
- ✓ Can help during negotiation when conflicts between requirements are identified.

Extra:

1. **Identify key functions** → e.g., login security is more important than adding a theme color.
2. **Refine schedule** → do critical features first (payment, security) → less urgent features later.
3. **Plan builds/versions** → Release 1: core features, Release 2: chatbot + extras.
4. **Support negotiation** → if conflict → focus on security now, move chatbot to future update.

# Understand and Evaluate Requirements (Analysis)

- **Conceptual modeling**
  - ✓ Conceptual models are created to further identify the requirements by understanding their context, discovering the bounds of the software system, and conceptualizing how the system interacts with its environment.
  - ✓ In many projects, this is where architectural design begins, since system decomposition is essential to developing effective conceptual models. (important)

- **Extra: Conceptual Modeling**
- **Purpose:** Understand requirements, system context, and boundaries.
- **Boundaries:** Decide what the system will include and exclude.
  - ✓ ***Inside****:* login, account balance, money transfer.
  - ✓ ***Outside:*** ATM hardware, cash withdrawal.
- **Shows:** How the system interacts with users and its environment.
- **Example: Online banking system** → users log in, check balance, transfer money (but ATMs are not part of this system).

# Understand and Evaluate Requirements (Specification & validation)

- **Specification and validation**
  - ✓ Activity where the results of elicitation an analysis are formally captured and documented in an appropriate format for the use and review of all stakeholders.

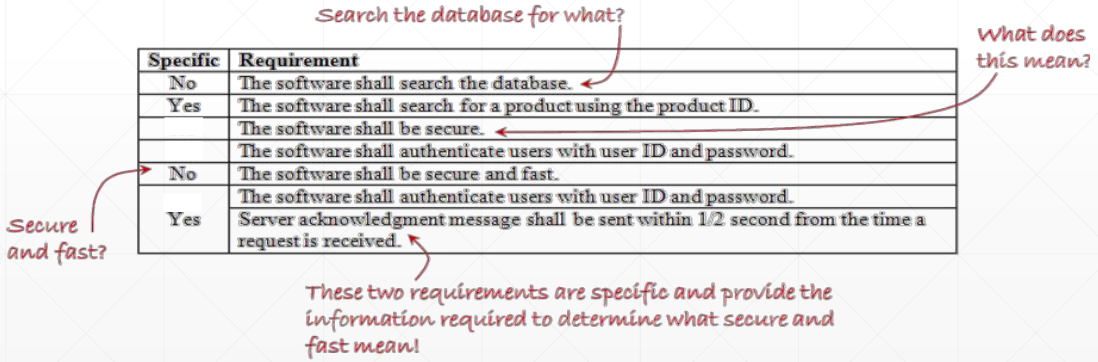
  Extra: Stakeholders review the classified requirements to ensure accuracy, completeness, and feasibility.

  - ✓ When specifying requirements, it is important that each requirement exhibit certain characteristics desired for designing successful systems. **Requirements must be**:

| | |
|---|---|
| **Specific** | **Consistent** |
| **Correct** | **Attainable= ”Achievable”, Possible”** |
| **Complete** | **Verifiable** |

# Understand and Evaluate Requirements

- On being **specific**,
  - ✓ Requirements need to be specified in a clear, concise" short and brief," and exclusive manner.
  - ✓ Clear requirements are not open to interpretation; unclear or ambiguous requirements lead to incorrect designs, incorrect implementations, and deceptive validation during test.
  - ✓ Concise requirements are brief and to the point and are therefore easier to understand.
  - ✓ Exclusive requirements specify one, and only one thing, making them easier to verify.




| Specific | Requirement |
|---|---|
| No | The software shall search the database. |
| Yes | The software shall search for a product using the product ID. |
| | The software shall be secure. |
| | The software shall authenticate users with user ID and password. |
| No | The software shall be secure and fast. |
| Yes | The software shall authenticate users with user ID and password. |
| | Server acknowledgment message shall be sent within 1/2 second from the time a request is received. |

Example of specific and nonspecific requirements:

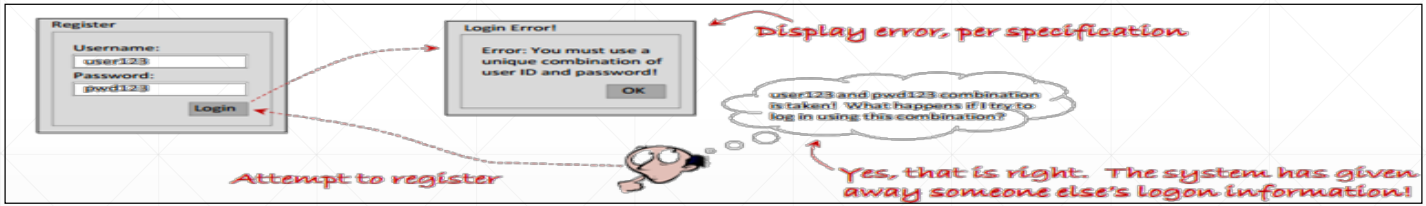
➢ **Extra: Specific Requirements Non-Specific Requirements:**

1. **"The software shall search for a product using the product ID."**
   - ✓ **Specific**: Yes, because it clearly states what needs to be searched (product) and how (using product ID).
2. **"The software shall authenticate users with user ID and password."**
   - ✓ **Specific**: Yes, as it specifies the authentication mechanism (user ID and password).
3. **"Server acknowledgment message shall be sent within 1/2 second from the time a request is received."**
   - ✓ **Specific**: Yes, it provides a measurable performance requirement (response time within 1/2 second).
4. **"The software shall search the database."**
   - ✓ **specific**: No,This is vague because it doesn't specify what to search for in the database. It lacks context and detail about the search criteria or the expected outcome.
5. **"The software shall be secure."**
   - ✓ **specific**: No, Security is a broad concept. This requirement doesn't specify what aspects of security are important, such as whether it refers to data encryption, secure user authentication, or another security measure.
6. **"The software shall be secure and fast."**
   - ✓ **specific**: No, This requirement combines two qualitative attributes—security and speed—without detailing what "secure" or "fast" means in measurable terms.

# Understand and Evaluate Requirements

- On being **correct**,
  - ✓ Requirements need to be correct in the sense that they must accurately describe a desired system function.
  - ✓ Incorrect requirements can lead to incorrect or undesired behavior.

- The diagram you're describing seems to illustrate a sequence of interactions around user registration and login for a software system, highlighting an error handling scenario where a username and password combination is not unique. Here's a detailed breakdown based on the described elements:




**Extra: Scenario Overview:**

1. **Attempt to Register:**
   A user tries to register with a username and password via a registration form.
2. **Error Display:**
   The system checks if the username and password combination is unique.
   Since the combination is already taken, the system generates an error message stating, "Error! You must use a unique combination of user ID and password!"
3. **User's Reaction:**
   This prompts the user to either choose a different username or password to ensure uniqueness.

# Understand and Evaluate Requirements

- On being **complete**,
  - ✓ Requirements must be complete both individually and as collective set.
    - This means that each requirement should be specified thoroughly so that it absolutely describes the functions required to meet some need.
    - Collectively, requirements need to provide complete specification of the software's required functionality in the software requirements specification (SRS).
      - Extra:
        - Individually → each requirement should be fully detailed and clear.
        - Collectively → together, they should describe the entire functionality in the Software Requirements Specification (SRS).

        In short: Complete requirements = nothing missing, everything specified.
  - ✓ Incomplete requirements lead to incomplete designs, which in turn leads to incomplete construction of the software system.

Notice how the original requirement is broken into two complete requirements!

This is a good requirement in the sense that it is specific and correct.

| Complete | Requirement |
|---|---|
| No | The software shall generate product reports. |
| Yes | The software shall generate product reports consisting of product description, picture, and price. |
| | Product reports shall be in PDF format. |

However, if we know the requirements for product reports, then a complete version of this requirement specifies this information. This help disambiguate the notion of a product report so that it can be implemented easily in code.

# Understand and Evaluate Requirements

- On being **consistent**,
  - ✓ Requirements are consistent when they do not preclude "prevent" the design or construction of other requirements.
- On being **attainable**,
  - ✓ Requirements that are unattainable serve no purpose.
  - ✓ Attainability can be determined for both product and process.
  - ✓ Requirements are attainable if they can realistically be achieved within the project constraints, which include time, technology, budget, and resources.

**Extra:**

**Consistent**

- No conflicts between requirements.
- ✅ Example: "System must support online payments" & "System must support credit card payments" (work together).
- ❌ Conflict: "System must be offline only" & "System must support online payments."

**Attainable**

- Realistic within time, budget, technology, resources.
- ✅ Example: "App must handle 1,000 users at once."
- ❌ Not attainable: "App must handle 1 billion users on a $500 budget."

💡 **Tip:** Good requirements = **clear, non-conflicting, and realistic.**

| Attainable | Requirement |
|---|---|
| No | The software shall execute on all future operating systems. |
| Yes | The software shall execute on the Microsoft Windows 7 platform. |

# Understand and Evaluate Requirements

**Extra:** **A requirement is verifiable if there is a suitable way to measure or test it to confirm that the requirement has been met. This could be through tests, demonstrations**

- On being **verifiable**,
  - ✓ Perhaps the most obvious desirable characteristic of requirements.
  - ✓ Requirements that cannot be verified cannot be claimed as met.
  - ✓ Inability to verify requirements point to a serious flaw early on in the development process

| Verifiable | Requirement |
|---|---|
| No | The system shall maximize communication speed. |
| Yes | The system's data rate shall be no less than 1Mbps. |

•**Extra:**

- ✅ **Verifiable Requirements**
- **Good requirement** = measurable, testable, clear.
- **Bad requirement** = vague, not testable.
- **Examples:**
- ❌ *Not verifiable:* "The system shall maximize speed."
- ✅ *Verifiable:* "The system's data rate shall be at least **1 Mbps**."
- 💡 **Tip:** Always make requirements measurable → use **numbers, limits, or conditions**.

## Summary

- In this session, we presented fundamentals concepts of **requirements engineering**, and provided enough information to successfully create good requirements.

- In the next session, we'll discuss how to design the software architecture using the popular **4+1 View Model**, which includes:
  - ✓ Logical view
  - ✓ Development View
  - ✓ Process View
  - ✓ Physical View
  - ✓ User View