Software Architecture

Unit -3

Architecture Description, Documentation & Evaluation

Early Architecture Description Languages - Domain & Style specific ADLs, Extensible ADLs - Documenting Software Architectures - Architecture Evaluation - ATAM

* Architecture Description

Architecture Description

Elements or Specifications

(i) Components

(ii) Connectors

(iii) Interactions

Constraints

(i) Static vs. Dynamic

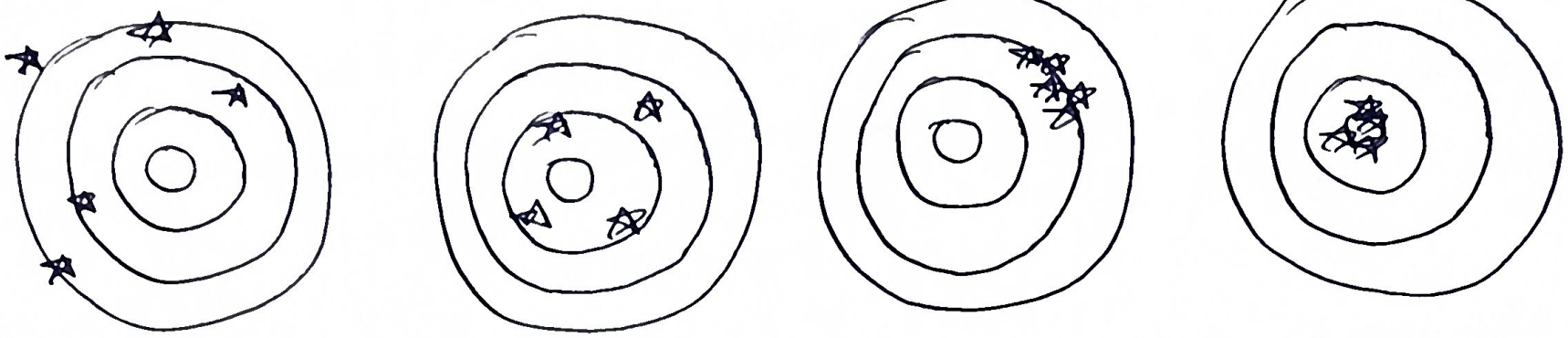
(ii) Structural vs Behavioral

(iii) Functional vs Non-Functional aspects

Example : Lifetime predictivity of an asset in the factory

* Architecture Evaluation Aspects

(i) Ambiguity
(ii) Accuracy
(iii) Precision



Accuracy-X
Precise - X

Accuracy - ✓
Precise - X

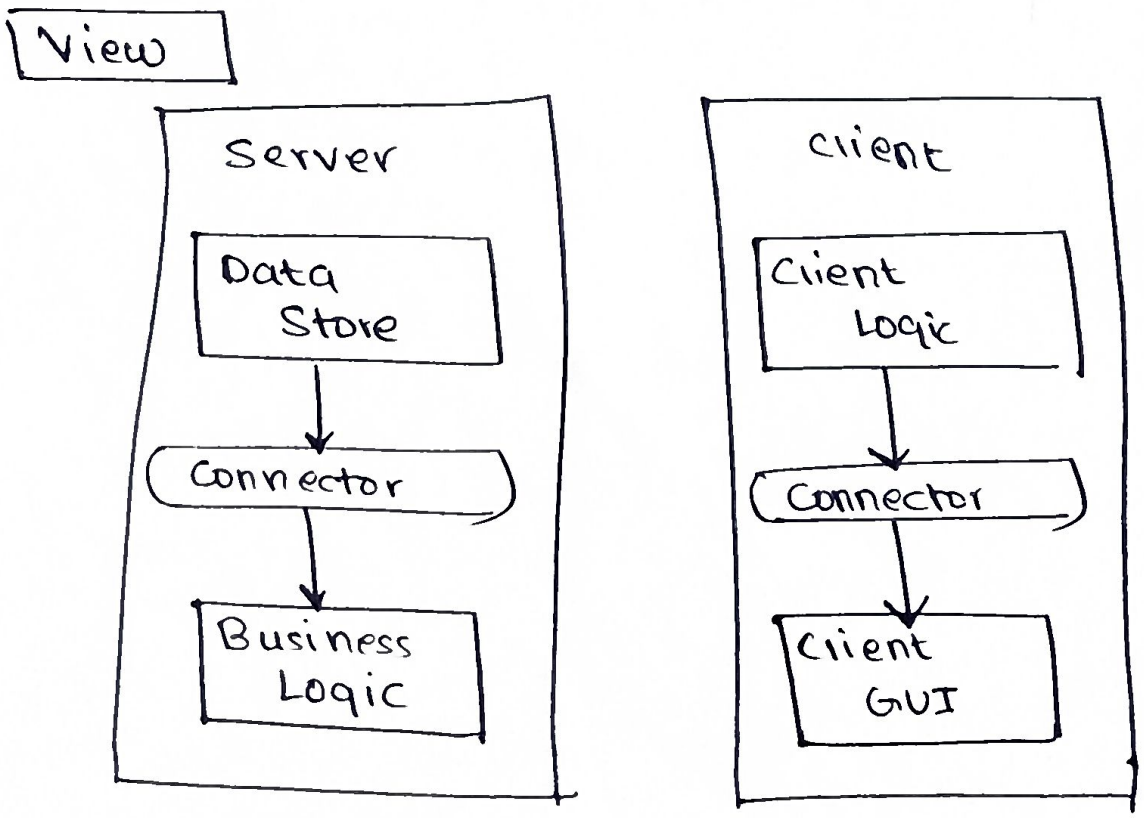
Accuracy - X
Precision - ✓

Accuracy - ✓
Precision ✓

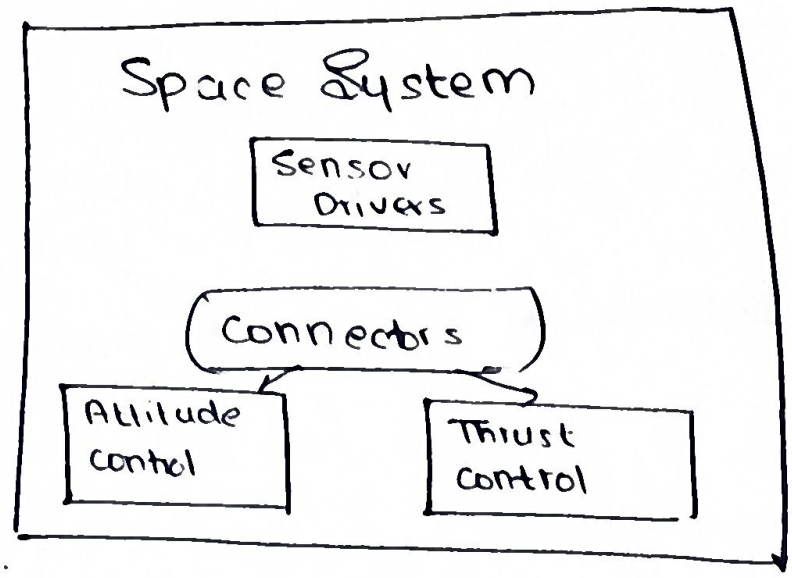
* ~~Architecture~~ Evaluation through Architecture Styles / Systems

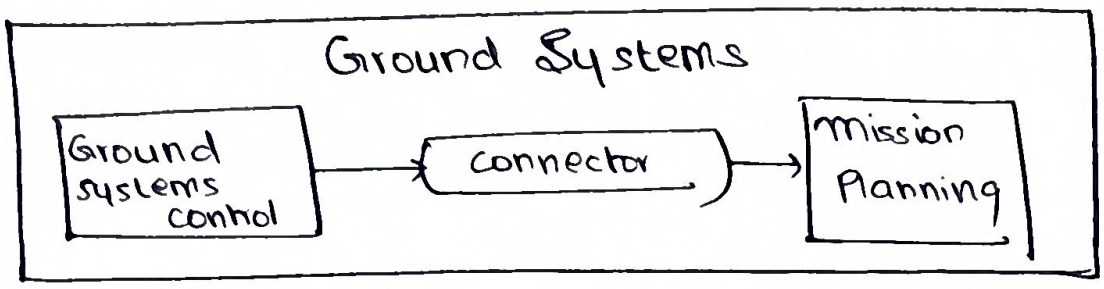
A. View - overall representation of the systems

B. Viewpoint - a filter applied on the view, providing context applicable to each system.

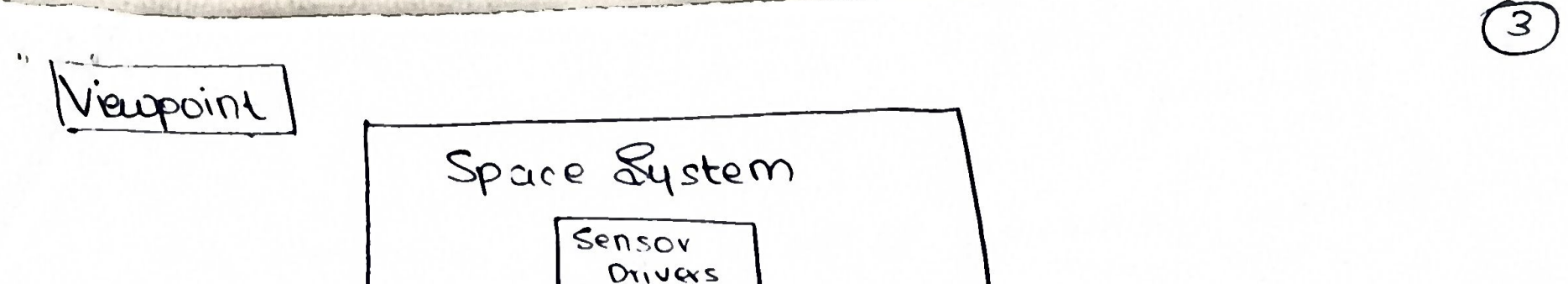


Viewpoint





* Architecture Evaluation through Visualization



FCC DVAE

A. Fidelity (FaithFulness to model - adherence)

→ Stick to the specifications

→ Reuse architecture wherever possible

→ Meaningful annotations and labelling

→ Documentation about the model

→ Innovates enhance models

B. Consistency

→ Maintain consistency of the model

→ Coordination expected throughout (ideation to deployment)

→ Peer to peer consistency of model

→ Master-slave consistency

→ Push & pull consistency

C. Comprehensibility

→ Clear and well-structured architecture diagrams

→ Doumentation that explains the relationship between components

→ Use of familiar architecture patterns to ensure understanding across different stakeholders

→ Minimal complexity to avoid confusion & ensure easy adoption by new developers

D. Dynamism

→ Architecture should support changes over time without significant rework

→ Flexibility to adapt to new requirements, technologies or scaling demands

→ Support for real-time or on-demand updates

→ Mechanisms to handle dynamic workloads or user demands

E) View Coordination

→ Proper coordination between different architectural views (eg. Logical, physical, process views)

→ Alignment between the high-level design and lower-level implementation details.

→ Ensure consistency between different perspectives - eg. developer, business & operations views

→ Cross-validation between views to avoid conflicts and ensure a coherent structure.

F. Aesthetics

→ Clean, organized & visually appealing architecture diagrams

→ Use of consistent styles and symbols across all representations

→ Simplification of architecture design to focus on the most important aspects.

→ Intuitive representation, making it easier for stakeholders to grasp key concepts at a glance.

G. Extensibility

→ Architecture should allow for adding new features or modules without major redesign

→ Component-based design to facilitate future extensions

→ Use of APIs / other integration points for external systems / features

* Architecture Analysis

Definition : Architectural analysis is the activity of discovering important system properties using the system's architectural models.

*** 8 Dimensions of Architecture Analysis

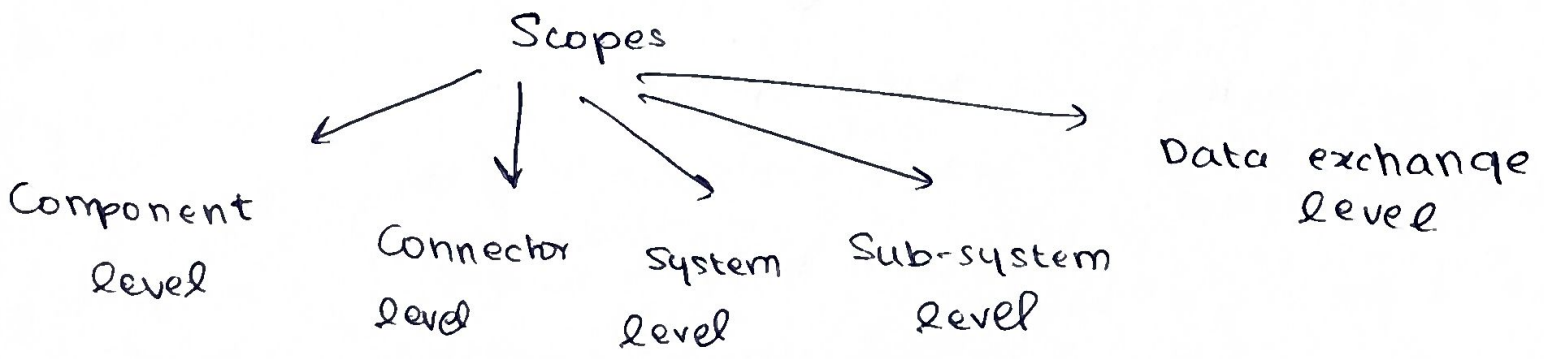
1. The goals of analysis
2. The scope of analysis
3. The primary architectural concern being analyzed.
4. The level of formality of the associated architectural models
5. The type of analysis
6. The level of automation
7. The system stakeholders to whom the results of analysis may be relevant.
8. The applicable analysis techniques.

* Architecture Analysis Challenges

1. Name Inconsistencies → eg. a function named getUserInfo() & another called fetchUserDetails() cause confusion in tracking down the usage
2. Interface Inconsistencies → a method needs 3 inputs, but in one component, the method is called w/ only 2 params ⇒ runtime errors
3. Behavioral Inconsistencies
   ↳ a checkout fn behaves differently when invoked in the mobile app vs. website (discounts not applied consistently)
4. Interaction Inconsistencies → one module sends data in XML format while another module expects it in JSON format — can break communication between components
5. Refinement Inconsistencies ↙

An initial high-level design specifies a security check before every data access, but the lower-level implementation forgets to include this check in some places - can lead to potential security vulnerabilities

* Scopes of Analysis of Architecture



A. Component-level Analysis - eg. evaluating the login module's security, performance & integration with the user DB.

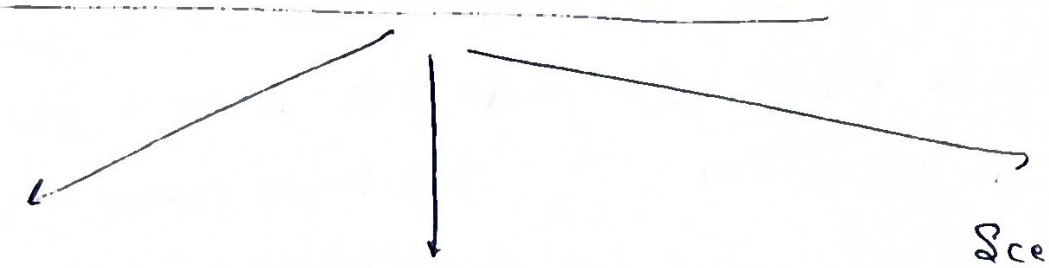
B. Connector level Analysis - eg. Analyzing API's gateway's performance

c. System-level Analysis - Assessing overall interaction between modules

D. Sub-System level Analysis - eg. examining the order management sub-system in an e-commerce platform, including order placement & shipment tracking

E. Data Exchange level - eg evaluate secure & reliable data exchange between banking apps, central banking services etc

★ Techniques for the Evaluation of Architecture



Static analysis techniques

(syntax-based)

↓

using a tool like SonarQube to scan the source code for errors without running the program

Dynamic analysis techniques

(implementation focused)

↓

Running a performance tests to measure how the system behaves under load during execution

Scenario-based analysis techniques

(impact focused)

↓

Imagine a real-world situation, like a system failure in a hospital, to see how it would impact patient care and hospital operations

↙

Soln: regular checks of assets

* Summary of Architecture Analysis

- Analysis
  - Goals
    - Completeness
    - Consistency
      - Name
      - Interface
      - Behaviour
      - Interaction
      - Refinement
    - Compatibility
    - Correctness
  - Scope
    - Component & Connector level
    - Subsystem & System level
    - Data Exchange
    - Different Abstraction levels
    - Architecture comparison
  - Concerns
    - Structural
    - Behavioral
    - Interaction
    - Non-Functional
  - Models
    - Informal
    - Semiformal
    - Formal
  - Type
    - Static
    - Dynamic
    - Scenario-based

* ATAM - Architecture Trade Off Analysis Method

→ Architecture evaluation is essential for ensuring that a software system meets both functional and non-functional requirements.

→ It helps balance trade-offs and manage risks related to the system's quality attributes like performance, scalability & security

The Architecture Trade-Off Analysis Method (ATAM)

- ATAM is a structured method used to evaluate software architecture by identifying and analyzing trade-offs between different quality attributes.

→ It is used during the design phase to ensure that the architecture aligns with the system's business goals and technical constraints.

→ Two key aspects considered in ATAM are:

Business Drivers in Software Architecture

Includes

(i) critical system functionality

(ii) constraints - technical, managerial, economic

(iii) business goals - broader objectives of the project

(iv) Stakeholders - customers, developers, managers

(v) Quality Attributes - non-functional goals - security, performance, scalability

Technical Constraints

(i) Hardware and operating systems requirements

(ii) Middleware & programming languages

(iii) System Interactions

Types of Scenarios in Architecture Evaluation

(i) Use-case scenarios - how stakeholders envision using the system in real-life

(ii) Growth scenarios - Focus on how the system will handle planned modifications or expansions

(iii) Exploratory scenarios - Test the system's adaptability by considering major potential changes.

Architectural Approaches to meet Quality Goals

→ must adopt approaches that meet Non-Functional Requirements

(i) Completeness - cover all necessary aspects

(ii) Consistency - make sure that there are no conflicting elements within the architecture

(iii) Compatibility - architecture should be able to work with other systems / components

* Implementation of Software Architecture

The phases involved are as follows:

1. Implementation phase
2. Mapping & Traceability phase - track how well architecture is implemented
3. Dynamic & Non-functional properties - guidelines for system's runtime behavior & quality
4. One-way & Two-way mapping - how changes in implementation are handled

1. Implementation Phase

- → designed architecture is changed into a working system

→ final product is judged based on how well the architecture holds up during implementation & how it meets the project requirements

Step 1 : Map Architecture to Implementation

→ Map architecture to actual code & systems

→ Define components, set up interactions, ensure all components work together.

Step 2 : Get everything in working mode

→ Ensure that the system components, such as connectors and interfaces are functioning properly

→ This involves selecting the right programming languages and, tools and libraries

Step 3 : Make sure to follow architecture framework

→ Architecture framework serves as the guide throughout the Implementation process.

→ Ensures that the system is being built according to the overall architectural principles & goals established earlier

2. Mapping and Traceability Phase

→ Traceability ensures that all requirements set out in the architecture design are reflected in the final implementation.

Step1: Create a Traceability Matrix

→ serves as a tool to track the progress of requirements throughout the implementation process

→ acts like a checklist

	Gate A	Gate B	Gate C	Gate D	Gate E	Gate F
Project Stages	Requirement Evolution	High-level Design	Detailed Design	Implementation	Testing	Deployment
Requirements Progression	Requirements Version 1		Req Version 3-5		Req Version 7	
Requirements Traceability		Design & Business Rule Document		Implementation and testing refer		confirm complete trace

Step 2 : Define Components & Connectors

→ defines how functionality is broken down and how components communicate (traceability matrix ensures those are correctly implemented)

Step3 Define interfaces

→ Define how communication happens in components through methods & API calls

Step4 : Design Configurations

→ Configurations show how components & connectors are organized

→ made as a linked graph

Step5 : Design Rationale

→ Explain the reasons behind architectural decisions

→ Not directly implemented in code, influences implementation by providing context for why certain connectors / components were chosen.

3. Dynamic and Non-Functional Properties

→ define how the system behaves under different conditions and well it meets certain quality attributes

A. Dynamic Properties

→ dictate how the system should behave when it runs

→ some properties can be translated into implementation skeletons or sometimes complete implementations.

B. Non-Functional Properties

→ security, scalability — very hard to implement

→ use techniques like testing, inspections & documentation

4. One-way & Two-way Mapping

→ Mapping helps ensure consistency between the architecture and its implementation.

One-way mapping - all changes originate from the architecture and are pushed into the implementation

→ Architecture drives development, changes are made only when the architecture evolves

Req → Test

Test → Req

Two-way mapping - changes can be made either at the architecture or implementation level, and updates are reflected in both directions.

* Six Thinking Hats

Process - planning for action

Facts - What we know, and what needs to be found out

Feelings - intuitions, gut instinct, no reasoning

Creativity - ideas, possibilities

Benefits - positive points

Cautions - weaknesses, risks, difficulties, dangers