Course Contents:

Unit 4. Software Architecture analysis and design: requirements for architecture and the life-cycle view of architecture design and analysis methods, architecture-based economic analysis: Cost Benefit Analysis Method (CBAM), Architecture Tradeoff Analysis Method (ATAM). Active Reviews for Intermediate Design (ARID), Attribute Driven Design method (ADD), architecture reuse, Domain –specific Software architecture.

Unit-4

Software Architecture analysis and design: The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.

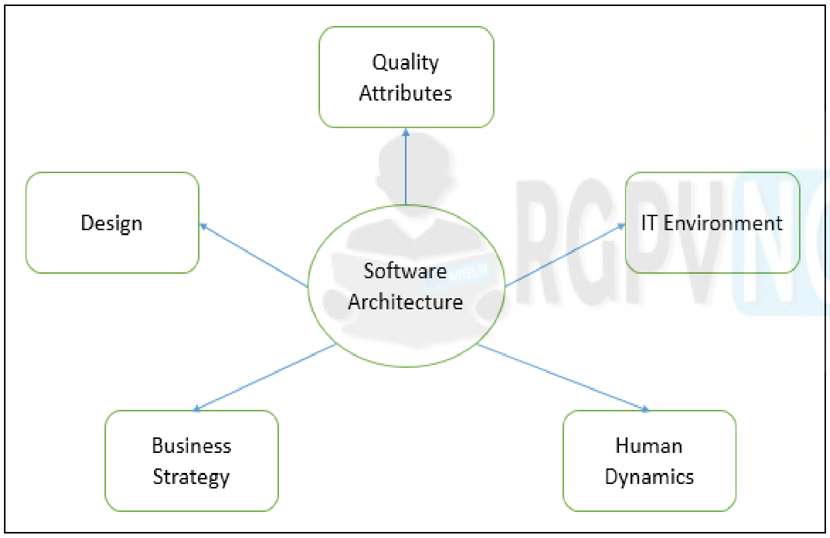

Figure 4.1: Software Architecture Design factors

Software Architecture and Design segregates into two distinct phases: Software Architecture and Software Design.

In **Architecture**, non-functional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Software Architecture

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

- It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of –
    - Selection of structural elements and their interfaces by which the system is composed.
    - Behavior as specified in collaborations among those elements.
    - Composition of these structural and behavioral elements into large subsystem.
    - Architectural decisions align with business objectives.
    - Architectural styles guide the organization.

Software Design

Software design provides a design plan that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows −

- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.

It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.

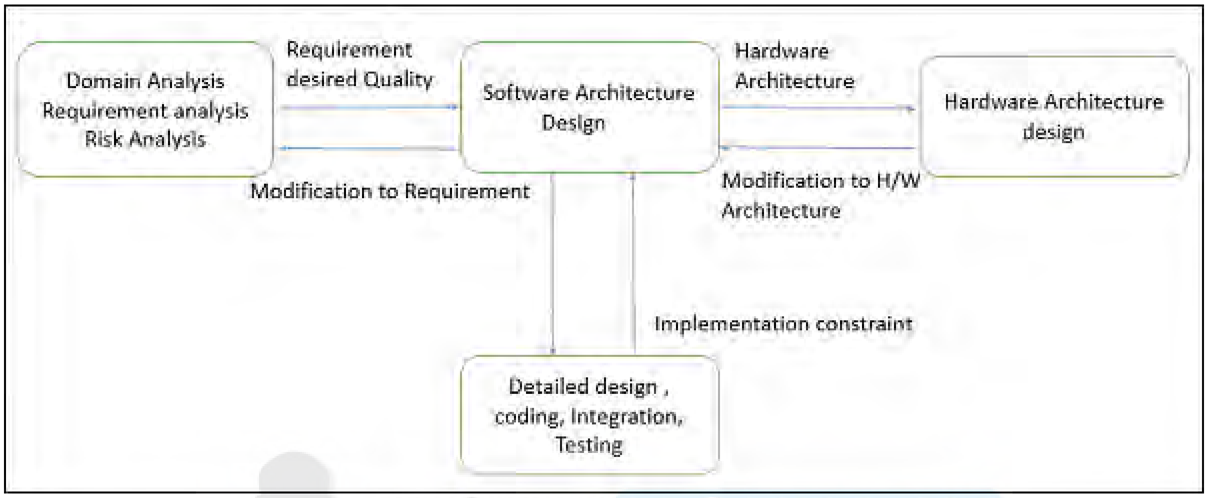
Domain Analysis
Requirement analysis
Risk Analysis
Requirement
desired Quality
Modification to Requirement
Software Architecture Design
Hardware Architecture
Modification to H/W Architecture
Hardware Architecture design
Implementation constraint
Detailed design , coding, Integration, Testing

Figure 4.2: Software Architecture Design Task

Requirements for architecture: The software needs the architectural design to represent the design of software. IEEE defines architectural design as “the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.” The software that is built for computer-based systems can exhibit one of these many architectural styles. Each style will describe a system category that consists of:

- A set of components (e.g., a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system. Some of the goals are as follows −

- Expose the structure of the system but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization’s market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Common Architectural Design: The following table lists architectural Design that can be organized by their key focus area –

Table 4.1: Common Software Architectural Design

Category	**Architectural Design**	**Description**
Communication	Message bus	Prescribes use of a software system that can receive and send messages using one or more communication channels.
	Service–Oriented Architecture (SOA)	Defines the applications that expose and consume functionality as a service using contracts and messages.
Deployment	Client/server	Separate the system into two applications, where the client makes requests to the server.
	3-tier or N-tier	Separates the functionality into separate segments with each segment being a tier located on a physically separate computer.
Domain	Domain Driven Design	Focused on modeling a business domain and defining business objects based on entities within the business domain.
Structure	Component Based	Breakdown the application design into reusable functional or logical components that expose well-defined communication interfaces.
	Layered	Divide the concerns of the application into stacked groups (layers).
	Object oriented	Based on the division of responsibilities of an application or system into objects, each containing the data and the behavior relevant to the object.

The life-cycle view of architecture design and analysis methods: The architecture design process focuses on the decomposition of a system into different components and their interactions to satisfy functional and nonfunctional requirements. The key inputs to software architecture design are –

- The requirements produced by the analysis tasks.
- The hardware architecture (the software architect in turn provides requirements to the system architect, who configures the hardware architecture).

The result or output of the architecture design process is an architectural description. The basic architecture design process is composed of the following steps –

Understand the Problem

- This is the most crucial step because it affects the quality of the design that follows.
- Without a clear understanding of the problem, it is not possible to create an effective solution.
- Many software projects and products are considered failures because they did not actually solve a valid business problem or have a recognizable return on investment (ROI).

Identify Design Elements and their Relationships

- In this phase, build a baseline for defining the boundaries and context of the system.
- Decomposition of the system into its main components based on functional requirements. The decomposition can be modeled using a design structure matrix (DSM), which shows the dependencies between design elements without specifying the granularity of the elements.
- In this step, the first validation of the architecture is done by describing a few system instances and this step is referred as functionality based architectural design.

Evaluate the Architecture Design

- Each quality attribute is given an estimate so in order to gather qualitative measures or quantitative data, the design is evaluated.
- It involves evaluating the architecture for conformance to architectural quality attributes requirements.
- If all estimated quality attributes are as per the required standard, the architectural design process is finished.
- If not, the third phase of software architecture design is entered: architecture transformation. If the observed quality attribute does not meet its requirements, then a new design must be created.

Transform the Architecture Design

- This step is performed after an evaluation of the architectural design. The architectural design must be changed until it completely satisfies the quality attribute requirements.
- It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.
- A design is transformed by applying design operators, styles, or patterns. For transformation, take the existing design and apply design operator such as decomposition, replication, compression, abstraction, and resource sharing.
- The design is again evaluated, and the same process is repeated multiple times if necessary and even performed recursively.
- The transformations (i.e., quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively.

Key Design Principles: Following are the design principles to be considered for minimizing cost, maintenance requirements, and maximizing extendibility, usability of architecture –

- Separation of Concerns.
- Single Responsibility Principle.
- Principle of Least Knowledge.
- Minimize Large Design Upfront.
- Do not Repeat the Functionality.
- Prefer Composition over Inheritance while Reusing the Functionality.
- Identify Components and Group them in Logical Layers.
- Define the Communication Protocol between Layers.
- Define Data Format for a Layer.
- System Service Components should be Abstract.
- Design Exceptions and Exception Handling Mechanism.
- Naming Conventions.

The methods and activities, and notes which artifacts are inputs to the method, outputs from the method, or both.

QAW: Quality Attribute Workshop, ADD: Attribute-Driven Design, ARID: Active Reviews for Intermediate Designs (ARID), CBAM: Cost-Benefit Analysis Method (CBAM), ATAM: Architecture Tradeoff Analysis Method (ATAM).

Table 4.2: Methods and Life-Cycle Stages

Life-Cycle Stage	QAW	ADD	ATAM	CBAM	ARID
Business needs and constraints	Input	Input	Input	Input	
Requirements	Input; output	Input	Input; output	Input; output	
Architecture design		Output	Input; output	Input; output	Input
Detailed design					Input; output
Implementation					
Testing					
Deployment					
Maintenance				Input; output	

Architecture-based economic analysis: It assists the objective assessment of the lifetime costs and benefits of evolving systems, and the identification of legacy situations, where architecture or a component is indispensable but can no longer be evolved to meet changing needs at economic cost. Need to understand the economic impact of architecture design decisions: the long term and strategic viability, cost-effectiveness, and sustainability of applications and systems. Economics-driven software development can increase quality, productivity, and profitability, but comprehensive knowledge is needed to understand the architectural challenges involved in dealing with the development of large, architecturally challenging systems in an economic way.

Cost Benefit Analysis Method (CBAM): The CBAM facilitates architecture-based economic analysis of software-intensive systems. This method helps the system's stakeholders to choose among architectural alternatives for enhancing the system in design or maintenance phases.

Inputs to the CBAM: The inputs include-

- the system's business/mission drivers.
- a list of scenarios.
- the existing architectural documentation.

Steps of the CBAM: This method includes the following steps:

1. Collate scenarios: Collate the scenarios elicited during the ATAM exercise and give the stakeholders the chance to contribute new ones. Prioritize these scenarios based on satisfying the business goals of the system and choose the top one-third for further study.

2. Refine scenarios: Refine the scenarios, focusing on their stimulus/response measures. Elicit the worst, current, desired, and best-case quality-attribute-response level for each scenario.

3. Prioritize scenarios: Allocate 100 votes to each stakeholder to be distributed among the scenarios, where the stakeholder's voting is based on considering the desired response value for each scenario. Total the votes and choose the top 50% of the scenarios for further analysis. Assign a weight of 1.0 to the highest rated scenario. Relative to that scenario, assign the other scenarios a weight that becomes the number used in calculating the architectural strategy's overall benefit. Make a list of the quality attributes that concern the stakeholders.

4. Assign intra-scenario utility: Determine the utility for each quality-attribute-response level (worst-case, current, desired, best-case) for the scenarios under study. The quality attributes of concern are the ones in the list generated during Step 3.
5. Develop architectural strategies for scenarios and determine their expected quality attribute- response levels: Develop (or capture already developed) architectural strategies that address the chosen scenarios and determine the expected quality-attribute-response levels that will result from implementing these architectural strategies. Given that an architectural strategy may affect multiple scenarios, this calculation must be performed for each affected scenario.
6. Determine the utility of the expected quality-attribute-response levels by interpolation: Using the elicited utility values (that form a utility curve), determine the utility of the expected quality-attribute-response level for the architectural strategy. Determine this utility for each relevant quality attribute enumerated in the previous step.
7. Calculate the total benefit obtained from an architectural strategy: Subtract the utility value of the current level from the expected level and normalize it using the votes elicited previously. Sum the benefit of a particular architectural strategy across all scenarios and relevant quality attributes.
8. Choose architectural strategies based on return on investment (ROI) subject to cost and schedule constraints: Determine the cost and schedule implications of each architectural strategy. Calculate the ROI value for each remaining strategy as a ratio of benefit to cost. Rank the architectural strategies according to the ROI value and choose the top ones until the budget or schedule is exhausted.
9. Confirm results with intuition: Of the chosen architectural strategies, consider whether they seem to align with the organization's business goals. If not, consider issues that may have been overlooked while doing this analysis. If significant issues exist, perform another iteration of these steps.

Outputs of the CBAM: Outputs include

- a set of architectural strategies, with associated costs, benefits, and schedule implications.
- prioritized architectural strategies, based on ROI.
- the risk of each architectural strategy, quantified as variability in cost, benefit, and ROI values.

Architecture Tradeoff Analysis Method (ATAM): The ATAM helps a system's stakeholder community understand the consequences of architectural decisions on the system's quality attribute requirements. These consequences are documented in a set of risks and tradeoffs that constitute the main output of the ATAM.

Inputs to the ATAM: Inputs include the-

- system's business/mission drivers.
- existing architectural documentation.

Steps of the ATAM: This method includes the following steps:
1. Present business drivers: A project spokesperson (ideally the project manager or system customer) describes which business goals are motivating the development effort and identifies the primary architectural drivers (e.g., high availability, time to market, or high security).
2. Present architecture: The architect describes the architecture, focusing on how it addresses the business drivers.
3. Identify architectural approaches: The architect identifies, but does not analyze, architectural approaches.
4. Generate quality attribute utility tree: The quality factors that make up system "utility" (performance, availability, security, modifiability, etc.) are specified down to the level of scenarios, annotated with stimuli and responses, and prioritized.

5. Analyze architectural approaches: Based on the high-priority factors identified in the utility tree, the architectural approaches that address those factors are elicited and analyzed (e.g., an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). Architectural risks, sensitivity points, and tradeoff points are identified.
6. Brainstorm and prioritize scenarios: A larger set of scenarios is elicited from stakeholders and prioritized through a voting process.
7. Analyze architectural approaches: The highest ranked scenarios are treated as test cases— they are mapped to the architectural approaches previously identified. Additional approaches, risks, sensitivity points, and tradeoff points may be identified.
Outputs of the ATAM: Outputs include a-

- list of architectural approaches.
- list of scenarios.
- set of attribute-specific questions.
- utility tree.
- list of risks and non-risks.
- list of risk themes.
- list of sensitivity points.
- list of tradeoffs.

Active Reviews for Intermediate Design (ARID): The ARID method blends Active Design Reviews with the ATAM, creating a technique for investigating designs that are partially complete. Like the ATAM, the ARID method engages the stakeholders to create a set of scenarios that are used to "test" the design for usability—that is, to determine whether the design can be used by the software engineers who must work with it. The ARID method helps to find issues and problems that hinder the successful use of the design as currently conceived.
Inputs to ARID: Inputs include-

- a list of seed scenarios.
- the existing architectural/design documentation.

Steps of ARID: This method includes the following steps:
1. Present the design: The lead designer presents an overview of the design and walks through the examples. During this time, participants follow the ground rule that no questions concerning implementation or rationale are allowed, nor are suggestions about alternate designs. The goal is to see if the design is "usable" to the developer, not to find out why things were done a certain way or to learn about the secrets behind implementing the interfaces. This step results in a summarized list of potential issues that the designer should address before the design can be considered complete and ready for production.
2. Brainstorm and prioritize scenarios: Participants suggest scenarios for using the design to solve problems they expect to face. After they gather a rich set of scenarios, they winnow them and then vote on individual scenarios. By their votes, the reviewers actually define a usable design—if the design performs well under the adopted scenarios, they must agree that it has passed the review.
3. Apply the scenarios: Beginning with the scenario that received the most votes, the facilitator asks the reviewers to craft code (or pseudo-code) jointly that uses the design services to solve the problem posed by the scenario. This step is repeated until all scenarios are covered or the time allotted for the review has ended.
Output of ARID: The output includes a list of "issues and problems" preventing successful use of the design.

Attribute Driven Design method (ADD): The ADD method defines a software architecture by basing the design process on the quality attributes the software must fulfill. ADD documents a software architecture in a number of views; most commonly, a module decomposition view, a concurrency view, and a deployment view. ADD depends on an understanding of the system's constraints and its functional and quality requirements, represented as six-part scenarios.

Inputs to ADD: Inputs include a-

- set of constraints.
- list of functional requirements.
- list of quality attribute requirements.

Steps of ADD: This method includes the following steps:

1. Choose the module to decompose: The module selected initially is usually the whole system. All required inputs for this module should be available (constraints and functional and quality requirements).

2. Refine the module according to the following steps:

a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.

b. Choose an architectural pattern that satisfies the architectural drivers. Create (or select) the architectural pattern based on the tactics that can be used to achieve the architectural drivers. Identify children modules required to implement the tactics.

c. Instantiate modules and allocate functionality from the use cases using multiple views.

d. Define interfaces of the child modules: The decomposition provides modules and constraints on the types of interactions among the modules. Document this information in the interface document for each module.

e. Verify and refine use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the children modules for further decomposition or implementation.

Output of ADD: The output includes a decomposition of the architecture, documented in at least three views: module decomposition, concurrency, and deployment.

Architecture Reuse: Maximizing reuse has always been an important goal of software development. It's better to re-use than to expend the cost of creating something new, testing it, and releasing it for the first time with the risk of hidden problems that all new software has. Languages, particularly object-oriented ones, have been developed to make reuse easier. But a language alone isn't enough to provide cost effective reuse. The bulk of reusable software comes from skilled developers and architects who are able to identify and leverage reuse opportunities.

Reusable Asset: The following are some examples of reusable software assets:

- Architectural frameworks
- Architectural mechanisms
- Architectural decisions
- Constraints
- Applications
- Components

There are three perspectives to look at when reusing software: code (implementation), design, and framework or architecture. Architects should look to reuse significant application frameworks such as layers that can be applied to many different types of applications. Developers should look to designs and Patterns that can be reused to produce desired behaviour or robust structures. They should also look at how to reduce the amount of code that needs to be written by leveraging stable components and code that has been proven in production environments.

To assess and select assets to reuse in project, there is a need to understand the requirements of the system's environment. Also need to understand the scope and general functionality of the system that the stakeholders require. There are several types of assets to consider, including (but not limited to): reference architectures; frameworks; patterns; analysis mechanisms; classes; and experience. We can search asset repositories (internal or external to your organization) and industry literature to identify assets or similar projects.

We need to assess whether available assets contribute to solving the key challenges of the current project and whether they are compatible with the project's architectural constraints. We also need to analyze the extent of the fit between assets and requirements, considering whether any of the requirements are negotiable (to enable use of the asset). Also, assess whether the asset could be modified or extended to satisfy requirements, as well as what the tradeoffs in adopting it are, in terms of cost, risk, and functionality.

Domain – specific Software architecture: A Domain-Specific Software Architecture (DSSA) is an assemblage of software components, specialized for a particular domain, generalized for effective use across that domain and composed in a standardized structure (topology) effective for building successful applications.

Domain specific software architecture comprises:

- A reference architecture, which describes a general computational framework for a significant domain of applications.
- A component library, which contains reusable chunks of domain expertise.
- An application configuration method for selecting and configuring components within the architectural to meet particular application requirements.

DSSAs involve several domain engineering activities. The regions of the problem space (domains) are mapped into domain-specific software architectures (DSSAs) which are specialized into application-specific architectures and then implemented. The three key factors of DSSA are:

1. Domain: Must have a domain to constrain the problem space and focus development
2. Technology: Must have a variety of technological solutions like tools, patterns, architectures & styles, legacy systems to bring to bear on a domain
3. Business: Business goals motivate the use of - Minimizing costs: reuse assets, when possible, and Maximize market: develop many related applications for different kinds of end users

These three factors together apply technology to domain-specific goals which is made firm by business knowledge.

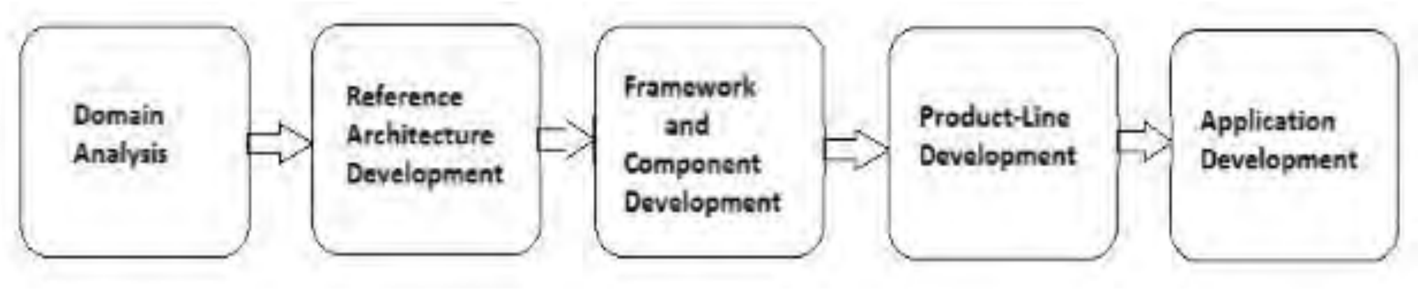
Domain Analysis
Reference Architecture Development
Framework and Component Development
Product-Line Development
Application Development

Figure 4.3: The overview of the DSSA process