Module 1: Introduction to Software Architecture

1. What is Software Architecture?

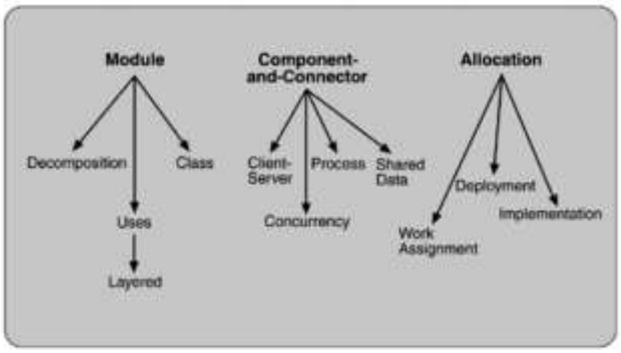
- Software architecture refers to the high-level design and structure of a software system.
- It defines the system's components, their relationships, and the principles governing their organization.
- A Software architecture is a set of Structures; A structure is a set of elements and relations between them.

2. Definitions of Software Architecture

- Various definitions exist, but common elements include system structure, components, and the way they interact.
- It emphasizes the system's design decisions, which impact its quality attributes.

3. Architecture Structure, Views, Style, and Patterns

- **Structure**: The arrangement of components and their relationships within a software system.
  - Structures primarily represent the underlying organization, components, and relationships within a system or entity.
  - They define the fundamental building blocks and how they interact.
  - Structures are concerned with the system's internal workings, its physical or logical components, and how they are connected.
  - Architects and designers use structures to establish the fundamental blueprint or foundation of a system or entity.
- **Type of structures**: Module, Component and Connector, and Allocation.



  - **Module Structure:**
    - **Description**: The module structure focuses on the decomposition of a system into smaller, manageable modules or components.
    - **Purpose**: It helps in organizing the software or system into coherent, functional units, making it easier to develop, test, and maintain.

    - **Examples**: In software architecture, this structure might represent classes, libraries, or packages. In a hardware architecture, it could represent integrated circuits or hardware modules.
    - **Types of Module Structure**:
        - **Decomposition:** This architectural style involves breaking down a system into smaller, manageable modules or components. Each module focuses on specific functionality, and the system's overall behaviour emerges from the interactions between these modules. It promotes modularity and encapsulation.
        - **Uses Structure:** In this style, software modules are organized around their usage or relationships. Modules are grouped based on how they are used within the system, fostering a clear understanding of dependencies and interactions.
        - **Layered:** Layered architecture organizes components or modules into horizontal layers, where each layer has a specific responsibility. Communication typically occurs only between adjacent layers, promoting separation of concerns and ease of maintenance.
        - **Class:** This style emphasizes the use of object-oriented principles like inheritance and polymorphism. Classes or objects represent different entities, and relationships are established through inheritance hierarchies, interfaces, or generalization relationships.
- **Component and Connector Structure:**
    - **Description**: The component and connector structure emphasizes the components of a system and the connections or interactions between them.
    - **Purpose**: It provides insight into the runtime behavior of the system and how different parts of the system collaborate.
    - **Examples**: In software architecture, components could be software modules, and connectors could be communication protocols or data flows. In hardware architecture, components might represent devices, and connectors could be buses or communication links.
    - **Types of C&C Structures**:
        - **Process**: This style focuses on decomposing a system into separate processes or components that communicate through well-defined interfaces. Each process may run independently and perform a specific function, and communication between them occurs via message passing or inter-process communication.
        - **Concurrency**: Concurrency architecture deals with systems that need to handle multiple tasks or operations simultaneously. It emphasizes the concurrent execution of components, often employing techniques like multithreading, parallel processing, or distributed computing.

        - **Shared Data**: In this style, components interact by sharing a common data repository. This repository serves as a centralized source of data that multiple components can access, modify, or query. It's commonly used in database systems.
        - **Client-Server**: The client-server architectural style separates a system into two main parts: clients, which request services or resources, and servers, which provide those services or resources. Clients communicate with servers over a network, often following a request-response model.
  - **Allocation Structure:**
    - **Description**: The allocation structure shows how software modules or components are mapped to hardware or physical resources.
    - **Purpose**: It helps in understanding the deployment of software on hardware and ensures that the system's performance and resource requirements are met.
    - **Examples**: This structure indicates which software components run on which servers, processors, or nodes in a networked system.
    - **Types of Allocation Structures:**
      - **Deployment**: Deployment architecture focuses on how software components are physically distributed across hardware or computing nodes. It deals with issues like load balancing, fault tolerance, and scalability.
      - **Implementation**: Implementation architecture outlines how system functionality is divided among different modules, components, or subsystems. It defines how various parts of the system are built, their interfaces, and their interactions.
      - **Work Assignment**: Work assignment architecture pertains to the allocation of tasks or responsibilities among individuals or teams in a software development project. It ensures that the right people are assigned to the right tasks, considering their skills and expertise.
- **View**: Views are abstractions or perspectives of the system's structure, tailored to specific stakeholders' needs and concerns.
  - They provide a way to present relevant information to different parties without overwhelming them with unnecessary details.
  - Views focus on a subset of the elements and relationships within the structure, emphasizing aspects or viewpoints.
  - Views can include diagrams, reports, or presentations that highlight specific elements and relationships to support decision-making or understanding.
- **Types of Views:** Module, Component and Connector, and Allocation.

- **Module View:**
    - **Description**: The module view provides a perspective of the system that focuses on the organization of software modules or components.
    - **Purpose**: It helps developers understand how the system's functionality is divided into discrete modules and how they interact.
    - **Examples**: Class diagrams in UML are a common example of a module view in software architecture.
- **Component and Connector View:**
    - **Description**: The component-and-connector view emphasizes the runtime behaviour and interactions between components in the system.
    - **Purpose**: It helps architects and developers analyse the system's dynamic behaviour, including communication patterns and data flows.
    - **Examples**: Sequence diagrams, deployment diagrams, or communication diagrams in UML can be used for component and connector views.
- **Allocation View:**
    - **Description**: The allocation view illustrates how software modules or components are allocated to specific hardware or physical resources.
    - **Purpose**: It assists in system deployment planning, resource allocation, and ensuring that the system meets performance and scalability requirements.
    - **Examples**: Deployment diagrams in UML are often used for allocation views in software architecture.

- **Note**: Both structures and views involve sets of elements and relationships between them, but they serve different purposes and focus on different aspects of an architecture or system. Structures are just set of elements and relations, however views are tailored to specific stakeholders' needs and concerns; along with the elements and relations.
- **Architectural Patterns(style)**: A set of architectural design principles and conventions. Reusable solutions to common and recurring architectural problems.
    - Layered Architecture
    - Client-Server Architecture
    - Microservices Architecture
    - Event-Driven Architecture
    - MVC (Model-View-Controller)
    - Component-Based Architecture
    - Peer-to-Peer Architecture

4. What is a Good Architecture?

- A good architecture meets functional and non-functional requirements efficiently.
- It is adaptable, maintainable, and scalable.
- Balances trade-offs between conflicting goals, such as performance vs. maintainability.

5. Importance of Software Architecture

- Guides the development process, ensuring a clear path.
- Affects system qualities like performance, reliability, and security.
- Facilitates communication among stakeholders.
- **Quality Assurance**: A well-designed architecture helps in identifying and addressing potential issues early in the development process, improving software quality.
- **Cost-Effective**: It can reduce development and maintenance costs by providing a structured approach to software development.
- **Scalability**: A good architecture allows for easy scalability as the system requirements evolve.
- **Risk Mitigation**: It helps in managing risks by providing a clear roadmap for development and making informed decisions.

6. Contexts of Software Architecture

- **Technical Context**: Concerned with the system's technical aspects and its interaction with hardware and software components.
- **Business Context**: Focuses on aligning the architecture with business goals, such as cost reduction or revenue generation.
- **Professional Context**: Involves ethical and legal considerations, adhering to industry standards, and best practices.
- **Project Life Cycle Context**: Addresses architectural decisions at various project stages, from inception to maintenance.

7. Architecture Competence

- Refers to the skills and knowledge required to design and evaluate software architectures.
- Involves understanding architectural styles, patterns, and trade-offs.
- Architects should possess both technical expertise and communication skills to bridge the gap between stakeholders.

Software architecture is the blueprint for designing and organizing software systems. It plays a crucial role in meeting requirements, ensuring system quality, and aligning with business objectives across various contexts. Successful architects possess the competence to make informed design decisions that balance competing factors.

Module 2: Software Quality Attributes

Understanding Quality Attributes: Quality attributes, also known as non-functional requirements or "ilities," define the desired characteristics and properties of a software system beyond its basic functionality. They encompass aspects like performance, reliability, security, and usability, which are essential for delivering a successful software product.

Types of Quality Attribute:

1. Interoperability: Interoperability refers to the ability of a software system or component to interact and function effectively with other systems or components, often using standard interfaces, protocols, or data formats. It ensures seamless communication and data exchange between heterogeneous systems.

- **Source of Stimulus:** External systems or components.
- **Stimulus:** Need for seamless interaction and data exchange.
- **Environment:** Heterogeneous system environments.
- **Artifact:** System interfaces and protocols.
- **Response:** Support for standardized protocols and data formats.
- **Response Measure:** Compliance with industry standards.
- **Design Tactics:**
    - Implement standard communication protocols (e.g., REST, SOAP).
    - Use data serialization formats like JSON or XML.
    - Employ API versioning strategies.
- **Design Checklist:**
    - Assess compatibility with external systems.
    - Ensure adherence to industry interoperability standards.
    - Test integration with third-party components.

2. Testability: Testability is the degree to which a software system or component can be easily tested to verify its correctness, identify defects, and ensure that it meets specified requirements. Testable systems facilitate efficient and thorough testing processes.

- **Source of Stimulus:** Development and QA teams.
- **Stimulus:** Need for efficient testing and debugging.
- **Environment:** Development and testing environments.
- **Artifact:** Codebase and testing infrastructure.
- **Response:** Facilitate test automation, debugging, and test coverage analysis.
- **Response Measure:** Code coverage metrics, reduced debugging time.

- **Design Tactics:**
    - Implement unit testing frameworks.
    - Support mocking and stubbing for dependencies.
    - Design for observability and logging.
- **Design Checklist:**
    - Ensure code modularity for isolated testing.
    - Integrate continuous integration and testing pipelines.
    - Enable tracing and monitoring for debugging.

3. Usability: Usability refers to the ease with which users can interact with and navigate through a software application or system. It encompasses factors like user interface design, intuitiveness, efficiency, and overall user satisfaction.

- **Source of Stimulus:** End-users.
- **Stimulus:** Need for a user-friendly interface and intuitive workflows.
- **Environment:** User interaction with the software.
- **Artifact:** User interface (UI) and user experience (UX) design.
- **Response:** Intuitive navigation, responsive design, and user satisfaction.
- **Response Measure:** Usability testing results, user feedback.
- **Design Tactics:**
    - Conduct user research and usability testing.
    - Implement responsive and accessible UI design.
    - Provide clear user guidance and error handling.
- **Design Checklist:**
    - Ensure a consistent and visually appealing UI.
    - Validate accessibility compliance.
    - Gather user feedback and iterate on design.

4. Performance: Performance measures how well a software system or component responds to various loads and executes tasks within acceptable timeframes. It includes aspects like response time, throughput, and resource utilization.

- **Source of Stimulus:** High user load or system resource constraints.
- **Stimulus:** Need for responsive and efficient system behaviour.
- **Environment:** Varies based on usage patterns.
- **Artifact:** System components affecting performance (e.g., databases, algorithms).

- **Response:** Efficient resource utilization, low response times, and scalability.
- **Response Measure:** Response time, throughput, resource utilization metrics.
- **Design Tactics:**
    - Optimize algorithms and data structures.
    - Implement caching mechanisms.
    - Use load balancing and horizontal scaling.
- **Design Checklist:**
    - Profile and analyse performance bottlenecks.
    - Conduct load testing and capacity planning.
    - Monitor and alert on performance metrics in production.

5. Scalability: Scalability is the ability of a software system to handle increasing workloads or user demands by efficiently adding resources or components. It ensures that the system can grow without significant degradation in performance.

- **Source of Stimulus:** Growing user base or increased workload.
- **Stimulus:** Need to handle increasing demand without system degradation.
- **Environment:** Evolving system usage patterns.
- **Artifact:** System architecture and deployment.
- **Response:** Ability to handle higher loads through horizontal or vertical scaling.
- **Response Measure:** Scalability metrics, such as requests per second.
- **Design Tactics:**
    - Use distributed architectures (e.g., microservices).
    - Employ cloud-based auto-scaling solutions.
    - Opt for stateless components where possible.
- **Design Checklist:**
    - Design for elasticity and load distribution.
    - Monitor system performance under varying loads.
    - Plan for data partitioning and distribution.

6. Modifiability: Modifiability, also known as maintainability, reflects how easily a software system can be modified or extended to accommodate changes in requirements, technologies, or business needs. Highly modifiable systems are flexible and cost-effective to maintain.

- **Source of Stimulus:** Evolving business requirements or changing technologies.
- **Stimulus:** Need for easy adaptation and maintenance.
- **Environment:** Ongoing development and change cycles.
- **Artifact:** Codebase, software architecture, and design.
- **Response:** Support for code maintainability, extensibility, and ease of change.
- **Response Measure:** Time and effort required for modifications.
- **Design Tactics:**
    - Apply modular and component-based architecture.
    - Implement design patterns (e.g., SOLID principles).
    - Use dependency injection for decoupling.
- **Design Checklist:**
    - Assess code readability and maintainability.
    - Conduct code reviews and refactoring as needed.
    - Ensure backward compatibility when making changes.

7. Security: Security encompasses measures taken to protect a software system from unauthorized access, data breaches, and malicious attacks. It involves implementing safeguards, access controls, encryption, and monitoring to ensure the confidentiality, integrity, and availability of data and resources.

- **Source of Stimulus:** Threats and vulnerabilities.
- **Stimulus:** Need for protection against unauthorized access, data breaches, and malicious attacks.
- **Environment:** Networked systems with potential security risks.
- **Artifact:** Security measures within the software.
- **Response:** Implement security mechanisms, access controls, and encryption to mitigate risks.
- **Response Measure:** Compliance with security standards, incident response time.
- **Design Tactics:**
    - Employ authentication and authorization mechanisms.
    - Apply encryption to sensitive data.
    - Conduct security audits and penetration testing.
- **Design Checklist:**

    - Identify and address common security vulnerabilities.
    - Stay updated on security best practices.
    - Have an incident response plan in place.

8. Availability: Availability measures the extent to which a software system or service is operational and accessible when needed by users. It involves redundancy, fault tolerance, and disaster recovery measures to minimize downtime and ensure uninterrupted service.

- **Source of Stimulus:** System failures or downtime.
- **Stimulus:** Need for uninterrupted system operation.
- **Environment:** Varies based on system criticality.
- **Artifact:** System components and infrastructure.
- **Response:** Implement redundancy, fault tolerance, and disaster recovery measures.
- **Response Measure:** Uptime percentage, mean time to recovery (MTTR).
- **Design Tactics:**
    - Use load balancing for high availability.
    - Employ failover mechanisms and redundant components.
    - Back up critical data and have a disaster recovery plan.
- **Design Checklist:**
    - Evaluate the impact of component failures.
    - Perform availability testing and simulate failure scenarios.
    - Establish clear recovery procedures.

9. Integration: Integration refers to the process of connecting and making different software systems, components, or services work together seamlessly. It involves defining communication protocols, data formats, and interactions to achieve interoperability and data exchange.

- **Source of Stimulus:** Multiple systems or components needing to work together.
- **Stimulus:** Need for seamless data and process integration.
- **Environment:** Distributed systems with heterogeneous components.
- **Artifact:** Integration points, APIs, and data formats.
- **Response:** Support for data transformation, communication protocols, and error handling.
- **Response Measure:** Successful integration of systems and minimal data loss.
- **Design Tactics:**
    - Use standardized data exchange formats (e.g., JSON, XML).
    - Implement message queues or publish-subscribe systems.

    - Provide clear API documentation and versioning.
- **Design Checklist:**
    - Validate data compatibility between integrated systems.
    - Conduct end-to-end integration testing.
    - Monitor integration points for errors and latency.

Other Quality Attributes:

- Various other quality attributes may be relevant to specific projects, such as compliance with industry regulations, environmental considerations, or cultural adaptability. Each of these attributes will have its unique stimulus, response, response measure, design tactics, and design checklist based on the specific context.

Design Trade-Offs:

- Design trade-offs involve balancing multiple quality attributes when making architectural decisions. It's often necessary to compromise on one attribute to optimize another. For example, improving performance may reduce modifiability, or enhancing security may increase complexity. Architects must carefully consider these trade-offs based on project priorities and constraints.

Module 3A: Capturing Architecturally Significant Requirements (ASR)

1. Gathering ASR:

- **Initial Steps:** Allocation of responsibilities, Coordination model, Data Model, Management of Resources, Mapping among Architectural Elements, Binding Time Decisions, Choice of technology**.**
- **Define Architecturally Significant Requirements (ASR):** ASRs are those requirements that have a significant impact on the system's architecture, influencing key design decisions. They often relate to quality attributes like performance, security, and scalability.
- **Gathering (ASR) from Requirement Document, Business Goals & Stakeholders:** ASRs can be extracted from various sources, including formal requirement documents, discussions with stakeholders, and alignment with the organization's business goals.
- **Capturing ASR in a Utility Tree for Further Refinement:** A Utility Tree is a visual representation that helps structure and refine ASRs. It breaks down high-level goals into subgoals and identifies the ASRs associated with each goal.
- **Architecture Design Strategy and Attribute-Driven Design Method:** Choosing the right design strategy and method is crucial. Attribute-Driven Design focuses on identifying and prioritizing ASRs to drive architectural decisions.
- **Challenges in Identifying ASRs:** Identifying ASRs can be challenging due to vague or conflicting requirements, evolving stakeholder needs, and the need to balance various quality attributes.

3. Quality Attribute Workshop:

- Quality attribute workshops are collaborative sessions involving stakeholders and architects to identify and prioritize ASRs.
- Steps include:
  - Presentation and Introduction
  - Business Presentation
  - Architectural Plan Presentation
  - Identification of actual drivers
  - Scenario Brainstorming
  - Scenario Consolidation
  - Scenario Prioritization
  - Scenario Refinement

4. Understanding Business Goals from Sponsors:

- **PALM: A Method for Eliciting Business Goals:** PALM (**Pedigreed**, **Attribute**, **eLicitation**, and **Method**) is a method for eliciting and understanding business goals from project sponsors. It helps align architectural decisions with the organization's objectives.
- **Identifying Architectural Drivers:** Architectural drivers are ASRs that significantly impact the system's architecture. Identifying them helps in focusing architectural efforts on the most critical aspects.
- **Understanding Scenarios for Each Architectural Driver via Brainstorming with Stakeholders:** Scenarios illustrate how ASRs manifest in real-world usage. Brainstorming with stakeholders helps create detailed scenarios that inform architectural decisions.

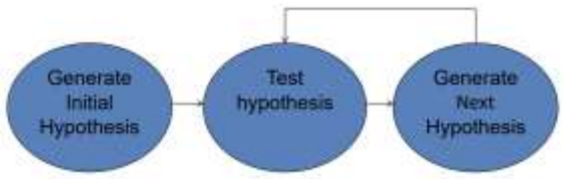
7. Prioritizing Scenarios:

- Prioritizing scenarios involves assessing their importance and impact on the architecture. This helps in allocating resources to address the most critical scenarios.
- **Building a Utility Tree:** A Utility Tree structures ASRs and their relationships, helping architects prioritize and refine ASRs. It provides a clear hierarchy of goals and subgoals.

Module 3B: Architecture Design

1. Design Strategy:

- The design strategy outlines the high-level approach for creating the system's architecture. It considers factors like architectural style, technology stack, and development process.
    - **Decomposition**: The whole system is decomposed into parts. Each part may inherit all of part of the quality attribute requirements from the whole
    - **Designing to Architecturally Significant Requirements**: Must satisfy ASR with design.
    - **Generate and Test**



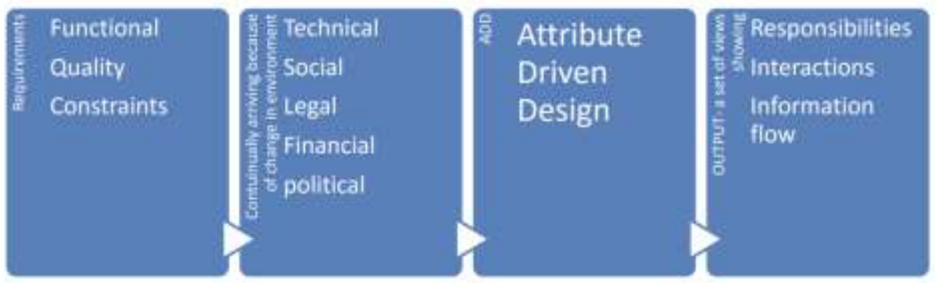
        - <u>Initial hypotheses in the context of design strategy often emerge from various sources:</u>
            - **Requirements:** Hypotheses can be based on the project's requirements, especially those related to specific design decisions or architectural features.
            - **Stakeholder Input:** Input from stakeholders, such as clients, users, or subject matter experts, can lead to hypotheses about user needs and preferences.

    - **Prior Experience:** Architects and designers may draw on their prior experience and knowledge of best practices to formulate hypotheses.
    - **Market Research:** Market research and competitive analysis can also inform hypotheses about what features or design elements are likely to be successful.
- <u>Testing a hypothesis involves a systematic process of validation or verification:</u>
    - **Define Success Criteria:** Clearly define what success looks like for the hypothesis. What specific outcome or result are you trying to achieve?
    - **Experiment or Implementation:** Implement the hypothesis or design element in a controlled manner. This may involve creating prototypes, conducting <u>A/B testing</u>, or making specific architectural decisions.
    - **Collect Data:** Gather data and metrics related to the hypothesis. This could include user feedback, performance measurements, or other relevant data points.
    - **Analyze Results:** Evaluate the data to determine whether the hypothesis was validated or refuted. Did the outcome align with the success criteria?
    - **Iterate:** If the hypothesis is not validated, consider iterating on the design or making adjustments based on the results. If it is validated, move forward with the validated design.
- <u>Knowing when you are done testing a hypothesis depends on the nature of the hypothesis and the project:</u>
    - If the hypothesis is related to a specific feature or design element, you are done when you have achieved the desired outcome or met the success criteria.
    - For broader architectural hypotheses, you may continue to refine and validate them throughout the project's lifecycle.
    - In iterative development processes like Agile, you are never truly "done" with hypothesis testing; it becomes an ongoing part of the development cycle.
- <u>Generating the next hypothesis is often a continuous and iterative process:</u>
    - **Learn from Results:** Analyze the results of the previous hypothesis testing. What did you learn from the data? Did it reveal new insights or user needs?
    - **Identify Gaps or Opportunities:** Identify gaps in the current design or areas where improvements can be made. Look for opportunities to enhance user experience or address performance issues.
    - **Prioritize:** Prioritize the most critical gaps or opportunities based on their potential impact on the project's goals and objectives.

- **Formulate Hypotheses:** Based on your analysis and priorities, formulate new hypotheses that address the identified gaps or opportunities. These hypotheses should be specific, measurable, and actionable.
- **Repeat the Testing Process:** Implement, test, and validate the new hypotheses following the same testing process outlined earlier.

2. Attribute-Driven Design:

- Attribute-Driven Design is an approach that begins with identifying and prioritizing ASRs and then making design decisions to satisfy these attributes. (Iterative Method)



- Steps involve:

1. Choose an element of the system to design.
2. Identify the ASRs for the chosen element.
3. Generate a design solution for the chosen element.
4. Inventory remaining requirements and select the input for the next iteration.
5. Repeat steps 1–4 until all the ASRs have been satisfied.

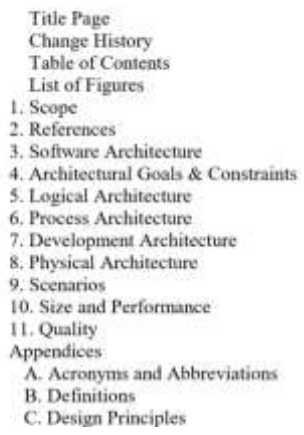
3. Architecting in Agile Projects:

- Agile projects emphasize iterative and collaborative development. Architects need to adapt their approach to align with agile principles.
- **How Much Architecture:** Agile projects require just enough architecture to meet current needs while allowing for flexibility as requirements evolve.
- **Agility and Architecture Methods:** Balancing agility with architectural concerns involves choosing lightweight methods and practices that suit the project's context.
- **Example of Agile Architecting:** Practical examples of how architecture evolves in an agile project context.
- **Guidelines for the Agile Architect:** Recommendations for architects working in agile projects, such as active collaboration, incremental design, and continuous evaluation of ASRs.

    1. Commitment and accountability of success-critical stakeholders
    2. Stakeholder “satisficing” (meeting an acceptability threshold) based on success-based negotiations and tradeoffs

3. Incremental and evolutionary growth of system definition and stakeholder commitment
4. Iterative system development and definition.
5. Interleaved system definition and development allowing early fielding of core capabilities, continual adaptation to change, and timely growth of complex systems without waiting for every requirement and subsystem to be defined.
6. Risk-driven anchor point milestones, which are key to synchronizing and stabilizing all of this concurrent activity.

Module 4: Documenting Software Architecture:



Title Page
Change History
Table of Contents
List of Figures
1. Scope
2. References
3. Software Architecture
4. Architectural Goals & Constraints
5. Logical Architecture
6. Process Architecture
7. Development Architecture
8. Physical Architecture
9. Scenarios
10. Size and Performance
11. Quality
Appendices
A. Acronyms and Abbreviations
B. Definitions
C. Design Principles

Outline of a Software Architecture Document

1. Importance of Architecture Documentation: Documentation of software architecture is crucial for several reasons:

- **Communication:** It serves as a means of effective communication among team members, stakeholders, and future maintainers. It provides a common understanding of the system's structure and design.
- **Maintainability:** Well-documented architecture aids in the long-term maintainability of the software. It helps developers understand how the system works, making it easier to make changes or troubleshoot issues.
- **Decision Support:** Architecture documentation captures design decisions, rationale, and trade-offs. It helps in revisiting and justifying these decisions and guides future architectural choices.
- **Onboarding:** New team members can quickly familiarize themselves with the software's structure and design principles through documentation.
- **Compliance:** In regulated industries, documentation is often a requirement for compliance with standards and regulations.

2. Architecture Views:

- Architecture documentation often employs multiple views or perspectives to describe different aspects of the system. Common views include:
    - **Module View:** Describes the structural organization of the software, such as components, modules, and their relationships.
    - **Component and Connector View:** Focuses on interactions, communication, and dependencies between components.
    - **Deployment View:** Illustrates how software components are physically distributed across hardware or environments.
    - **Process View:** Emphasizes the runtime behaviour of the system, including concurrency, processes, and threads.

- **Data View:** Describes data structures, storage, and data flow.
- **Scenarios View:** Captures how the system behaves in specific use cases or scenarios.

3. Quality Attribute Views:

- These views provide insights into how the architecture addresses specific quality attributes:
    - **Security View:** Describes security mechanisms, access controls, and threat models to protect the system.
    - **Communication View:** Focuses on communication patterns, protocols, and data exchange mechanisms.
    - **Reliability View:** Addresses fault tolerance, error handling, redundancy, and recovery mechanisms.

4. Combining Views:

- Architecture documentation often combines multiple views to provide a comprehensive understanding of the system. These views complement each other and offer a holistic perspective.

5. Philippe Kruchten's 4+1 View:

- Philippe Kruchten's "4+1 View Model" is a widely used approach to architecture documentation. It consists of four primary views:

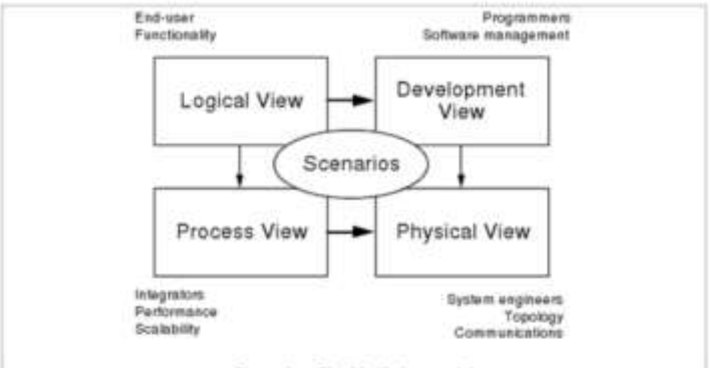
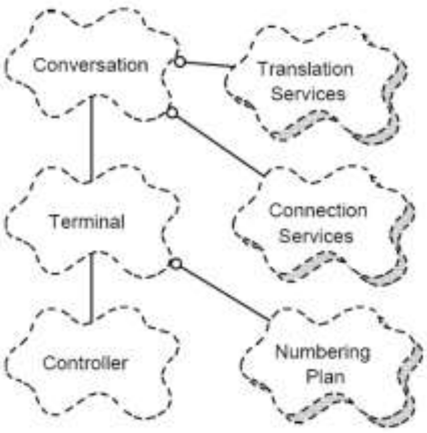


Figure 1 — The "4+1" view model

    - **Logical View:** Describes the system's high-level structure, such as modules, classes, and relationships.



- **Development View:** Focuses on the software's organization in terms of development, including source code structure, components, and libraries.

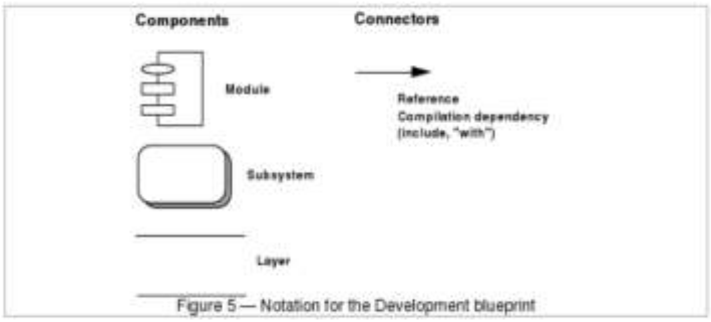
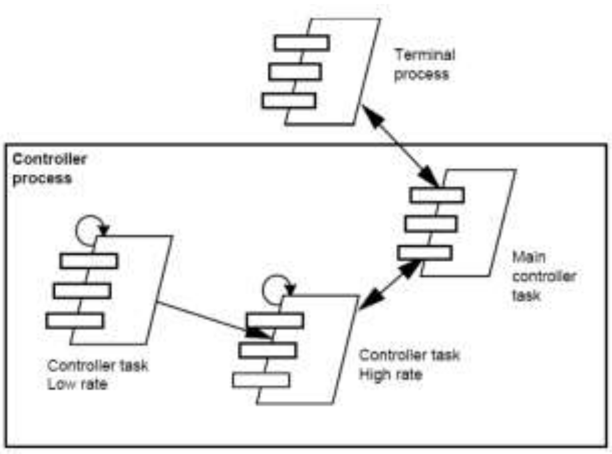
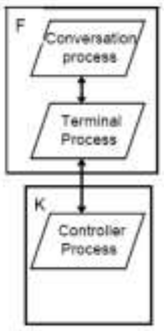


Figure 5 — Notation for the Development blueprint

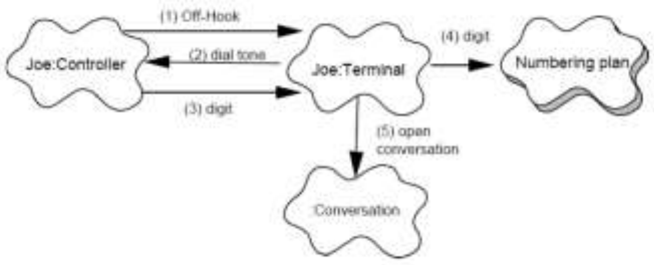
- **Process View:** Emphasizes the system's runtime behavior, including concurrency and processes.



- **Physical View:** Illustrates how the software is physically deployed across hardware.



- The "+1" view is the **Scenarios View,** which captures how the system behaves in specific scenarios or use cases. It ties the other views together.



6. Kruchen's four views

- **Logical**: The elements are "key abstractions," which are manifested in the object-oriented world as objects or object classes. This is a module view.
- **Process:** This view addresses concurrency and distribution of functionality. It is a component-and-connector view.
- **Development:** This view shows the organization of software modules, libraries, subsystems, and units of development. It is an allocation view, mapping software to the development environment.
- **Physical:** This view maps other elements onto processing and communication nodes and is also an allocation view (which others call the deployment view).

7. Documentation Package:

- A documentation package typically includes:
    - **Architectural Overview:** High-level description of the system's architecture, its components, and interactions.
    - **Architectural Views:** Diagrams and descriptions of different architecture views (e.g., module view, deployment view).
    - **Quality Attribute Documentation:** Details on how quality attributes like security, reliability, and performance are addressed.
    - **Rationale:** Explanation of design decisions, trade-offs, and justifications.
    - **Use Cases or Scenarios:** Descriptions of how the system behaves in specific situations.
    - **Change History:** Records changes made to the architecture over time.
- Documenting software architecture ensures that the knowledge about the system's design and behaviour is captured and accessible to all relevant stakeholders, contributing to successful development, maintenance, and evolution of software systems.

Module 5: Layered architecture, ATAM and Testing

Layered Architecture:

Layered architecture, also known as the n-tier architecture, is a software design pattern where a complex software system is organized into distinct layers, each responsible for specific functionalities. The layers are arranged hierarchically, with each layer providing services to the layer above it and consuming services from the layer below it. This separation of concerns enhances modularity, maintainability, and scalability in software systems.

Different Layers and Guidelines:

1. **Presentation Layer:**
    - **Responsibilities:** The presentation layer is responsible for handling user interaction, user interface rendering, and user experience.
    - **Guidelines:**
        - **Client-Side Caching vs. Server-Side Caching:** Choose between client-side and server-side caching based on factors like data size, network latency, and data volatility. Use client-side caching for frequently accessed data within the user's device. Employ server-side caching for shared or global data.
        - **Memcached:** Implement Memcached or similar caching mechanisms to store frequently used data in memory for rapid retrieval, improving application performance.
        - **Asynchronous Communication with Web Server:** Employ asynchronous techniques like AJAX to enhance user experience by allowing non-blocking data retrieval and real-time updates.
2. **Business Layer:**
    - **Responsibilities:** The business layer contains the core business logic, rules, and processes of the application.
    - **Guidelines:**
        - **Application Facade:** Use an application facade design pattern to provide a simplified and unified interface to the business layer. This hides the complexity of internal modules from external clients.
        - **Implement Session Management:** Implement session management to maintain user state and manage user-specific data.
        - **Use Workflow Engine:** A workflow engine can automate and streamline complex business processes, making the system more efficient.
        - **Aspect-Oriented Design:** Apply aspect-oriented design to modularize cross-cutting concerns like logging, security, and error handling. This keeps the core business logic clean and maintainable.

3. **Data Layer:**
    - **Responsibilities:** The data layer is responsible for data storage, retrieval, and management.
    - **Guidelines:**
        - **Object-Relational Mapping (ORM):** Utilize ORM frameworks such as Hibernate (Java) or Entity Framework (.NET) to abstract database interactions. ORM simplifies data access and enhances maintainability.
        - **SQL Injection Attacks:** Guard against SQL injection vulnerabilities by implementing input validation and using parameterized queries. Protect sensitive data and database integrity.
4. **Service Layer:**
    - **Responsibilities:** The service layer provides services and APIs to external systems, enabling integration.
    - **Guidelines:**
        - Design and implement services using RESTful APIs, SOAP, or other communication protocols to facilitate interoperability and integration with external systems.

Layered architecture promotes separation of concerns and modularization, making it easier to develop, maintain, and scale software systems. By adhering to the guidelines for each layer, developers can create well-structured and robust applications that meet user requirements and adhere to best practices in software design and development.

Architecture Evaluation (ATAM: Architecture Trade-off Analysis Method):

The Architecture Trade-off Analysis Method (ATAM) is a comprehensive approach to evaluating software architecture. It focuses on identifying and analyzing trade-offs between competing quality attributes and ensuring that architectural decisions align with project objectives. ATAM is used to assess architectural decisions and their impact on a system's qualities.

Factors for Evaluation:

1. **Evaluation by the Designer:** The initial evaluation is conducted by the software architect or design team responsible for creating the architecture. It involves assessing how well the architecture meets the intended objectives and quality attribute requirements.
2. **Peer Review:** In this phase, peer architects or experienced developers review the architecture. The aim is to gain additional insights, identify potential issues, and ensure that the architecture adheres to best practices.
3. **Peer Review Steps:** The peer review typically involves the following steps:
    - Identifying critical scenarios or use cases that exercise key architectural decisions.
    - Reviewing architectural decisions, including the rationale behind them.

    - Assessing the architecture's ability to meet quality attribute requirements.
    - Analysing the architecture for potential risks and trade-offs.

4. **Analysis by Outsiders:** In this step, independent external experts evaluate the architecture. They bring an unbiased perspective and provide insights that may not be apparent to internal team members. Their analysis can offer valuable validation and identification of risks.

Trade-off Analysis Method:

- ATAM employs a trade-off analysis method to evaluate architectural decisions and their impact on quality attributes. The process involves the following components:
    - **Explanation:** The architectural team presents the architecture to a panel of evaluators, explaining key decisions, quality attribute concerns, and critical scenarios.
    - **Scenarios:** Critical scenarios, which represent real-world system interactions and usage, are discussed and analyzed in-depth.
    - **Attribute Utility:** The team assesses how well the architecture satisfies quality attribute requirements, assigning utility values to each attribute.
    - **Trade-offs:** Evaluators identify trade-offs between different quality attributes and architectural decisions. Trade-offs occur when optimizing one attribute may negatively impact another.
    - **Risks:** Potential architectural risks and their consequences are identified and analyzed.

Participants:

- The ATAM typically involves the following participants:
    - Evaluation team (software architects and experts).
    - Stakeholders (representatives from the project and end-users).
    - External reviewers (independent experts).
    - Moderator (facilitates the evaluation process).

Roles:

- During an ATAM evaluation, several roles are assigned:
    - **Moderator:** Facilitates the evaluation sessions and ensures that the process adheres to the ATAM guidelines.
    - **Presenter:** Represents the architectural team and presents the architecture to the evaluation panel.
    - **Note-taker:** Records discussions, findings, and recommendations.

Outputs of ATAM:

- The ATAM evaluation process produces several valuable outputs:

    - Findings and observations about the architecture.
    - A list of architectural risks and their impact.
    - Recommendations for mitigating identified risks.
    - A refined architecture that addresses discovered issues.

Phases of the ATAM:

- The ATAM typically consists of the following phases:
    - **Phase 1: Preparation:** Define the evaluation objectives, select the evaluation team, identify critical scenarios, and prepare the presentation of the architecture.
    - **Phase 2: Evaluation:** Present the architecture to the evaluation panel, discuss critical scenarios, analyze quality attribute utility, and identify trade-offs and risks.
    - **Phase 3: Review:** Analyze trade-offs, assess risks, prioritize them, and validate that the architecture aligns with the project's goals and objectives.
    - **Phase 4: Documentation:** Document the findings, architectural risks, recommendations, and the refined architecture in an evaluation report.

Evaluation Method (Lightweight Architectural Evaluation):

Lightweight architectural evaluation methods are simplified approaches for assessing architecture, often used in agile and iterative development environments. These methods focus on specific architectural aspects or quality attributes and are less formal than ATAM.

Common lightweight evaluation methods include:

- **Architectural Design Review:** Informal discussions and reviews of architectural decisions, often conducted within the development team.
- **Checklists:** Using predefined checklists to verify that architectural best practices and quality attribute concerns are addressed.
- **Prototyping:** Building architectural prototypes to validate design decisions and gather feedback.
- **Architectural Analysis Tools:** Employing automated tools to analyze code or architectural diagrams for adherence to architectural guidelines.

Lightweight architectural evaluations are more agile and suited to fast-paced development environments where rapid feedback and adjustments are essential. However, they may not provide the depth of analysis offered by a comprehensive method like ATAM.

Architecture Conformance techniques during implementation

Architecture conformance techniques during implementation ensure that the implemented system adheres to the intended software architecture and design. These techniques help maintain architectural integrity and alignment with quality attribute requirements. Here are several key conformance techniques:

1. **Code Reviews:**
    - **Description:** Regular code reviews involve peers and senior developers assessing the codebase to ensure it aligns with the architectural design and follows coding standards.
    - **Benefits:** Code reviews help catch architectural violations early in the development process, fostering architectural conformance.
2. **Architectural Inspections:**
    - **Description:** Architectural inspections involve in-depth reviews focused on architecture-related aspects of the code, including modules, component interactions, and adherence to architectural decisions.
    - **Benefits:** Architectural inspections provide a more thorough examination of architecture-related concerns and can reveal subtle issues that may be missed in standard code reviews.
3. **Static Analysis Tools:**
    - **Description:** Static analysis tools automatically analyze the codebase for architectural violations, coding rule violations, and adherence to design guidelines.
    - **Benefits:** These tools offer a systematic and objective way to identify architectural non-conformities and coding issues. Examples include SonarQube and Checkmarx.
4. **Dynamic Analysis Tools:**
    - **Description:** Dynamic analysis tools monitor the system's runtime behavior to detect runtime violations of architectural constraints, performance bottlenecks, and memory leaks.
    - **Benefits:** These tools provide insights into how the system behaves in real-world scenarios, helping to uncover architectural issues that may only manifest at runtime. Examples include profiling tools and memory analysis tools.
5. **Automated Testing Suites:**
    - **Description:** Automated tests, including unit tests, integration tests, and system tests, can verify that the implemented code functions as expected within the architectural framework.
    - **Benefits:** Automated testing ensures that architectural components and interfaces remain intact during development and subsequent changes.
6. **Continuous Integration (CI) and Continuous Deployment (CD):**

- **Description:** CI/CD pipelines automatically build, test, and deploy code changes, allowing for immediate feedback on architectural conformance.
- **Benefits:** CI/CD helps identify and resolve architectural issues early in the development cycle, reducing the risk of architectural drift.

7. **Architectural Linters:**
    - **Description:** Architectural linters are specialized tools that analyze code for architectural violations. They focus on ensuring code adheres to predefined architectural rules and standards.
    - **Benefits:** Architectural linters provide specific checks for architectural conformance and can enforce architectural guidelines.
8. **Change Control Boards (CCB):**
    - **Description:** CCBs are responsible for reviewing and approving changes to the software's architecture. Any proposed architectural changes or deviations from the original design should be evaluated by the CCB.
    - **Benefits:** CCBs help maintain control over architectural decisions and ensure that changes align with the project's goals and quality attribute requirements.
9. **Documentation and Design Guidelines:**
    - **Description:** Maintain comprehensive architectural documentation and design guidelines that developers can reference during implementation to ensure alignment with the architectural design.
    - **Benefits:** Clear documentation and guidelines serve as a reference point for developers, reducing the likelihood of architectural drift.
10. **Regular Architecture Review Meetings:**
    - **Description:** Conduct periodic architecture review meetings where architects and developers discuss ongoing implementation efforts and assess architectural conformance.
    - **Benefits:** Regular meetings facilitate communication, identify potential issues early, and provide a forum for addressing architectural concerns.

In the context of software architecture and testing, several techniques and processes are used to ensure that the implemented system adheres to the intended architectural design and that any violations or deviations are identified and addressed. Here are some key activities related to architecture and testing:

Architecture Reconstruction:

Architecture reconstruction is the process of reverse-engineering the software architecture from existing code and documentation. It aims to create a clear and accurate representation of the system's architecture based on the implemented code.

Architecture reconstruction helps bridge the gap between the implemented system and the original design. It provides a visual or textual representation of the architecture, making it easier to assess conformance and identify deviations.

- **Raw View Extraction:**
  - Raw view extraction involves extracting architectural information directly from the code artifacts, such as source code files, configuration files, and database schemas. This information can include module dependencies, component interactions, and data flows.
  - Extracting raw views helps capture the current state of the system's architecture as reflected in the codebase. It provides a basis for analyzing the implemented architecture and comparing it to the intended design.
- **View Fusion:**
  - View fusion is the process of combining different architectural views into a cohesive and comprehensive whole. These views may include module views, component and connector views, deployment views, and others.
  - View fusion creates a holistic perspective of the system's architecture by integrating information from various sources and views. It helps architects and testers understand how different architectural elements relate to each other.
- **Finding Violations:**
  - Finding violations involves actively searching for discrepancies between the implemented system and the intended architecture. This includes identifying instances where architectural decisions or constraints have been violated.
  - Detecting violations early allows for timely corrective actions. Violations may include deviations from design patterns, incorrect component interactions, or the use of unauthorized technologies.