Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit I

Topics:

- The plot
- Introduction
- SA Importance

The Plot

- In the world of technology, starting from small children to young people and starting from young to old people everyone using their Smartphones, Laptops, Computers, PDAs etc to solve any simpler or complex task online by using some software programs, there everything looks very simple to user.
- Also that’s the purpose of a good software to provide good quality of services in a user-friendly environment.
- There the overall abstraction of any software product makes it looks like simple and very easier for user to use. But in back if we will see building a complex software application includes complex processes which comprises of a number of elements of which coding is a small part.

The Plot

- After gathering of business requirement by a business analyst then developer team starts working on the Software Requirement Specification (SRS), sequentially it undergoes various steps like testing, acceptance, deployment, maintenance etc. Every software development process is carried out by following some sequential steps which comes under this Software Development Life Cycle (SDLC).
- **In the design phase of Software Development Life Cycle the software architecture is defined and documented.**

Introduction

- ***Software Architecture defines fundamental organization of a system and more simply defines a structured solution. It defines how components of a software system are assembled, their relationship and communication between them.***
- It serves as a blueprint for software application and development basis for developer team.
- Software architecture defines a list of things which results in making many things easier in the software development process:

1. A software architecture defines structure of a system & behavior of a system.
2. A software architecture defines component relationship & defines communication structure.
3. A software architecture balances stakeholder's needs & influences team structure.
4. A software architecture focuses on significant elements & captures early design decisions.

SA Importance

- Software architecture comes under design phase of software development life cycle. It is one of initial step of whole software development process. Without software architecture proceeding to software development is like building a house without designing architecture of house.
- So software architecture is one of important part of software application development. In technical and developmental aspects point of view below are reasons software architecture are important.
- Selects quality attributes to be optimized for a system.
- Facilitates early prototyping.
- Allows to be built a system in component wise.
- Helps in managing the changes in System.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit I

Topics:

- Advantages & Disadvantages
- Architectural Characteristics

Advantages and Disadvantages

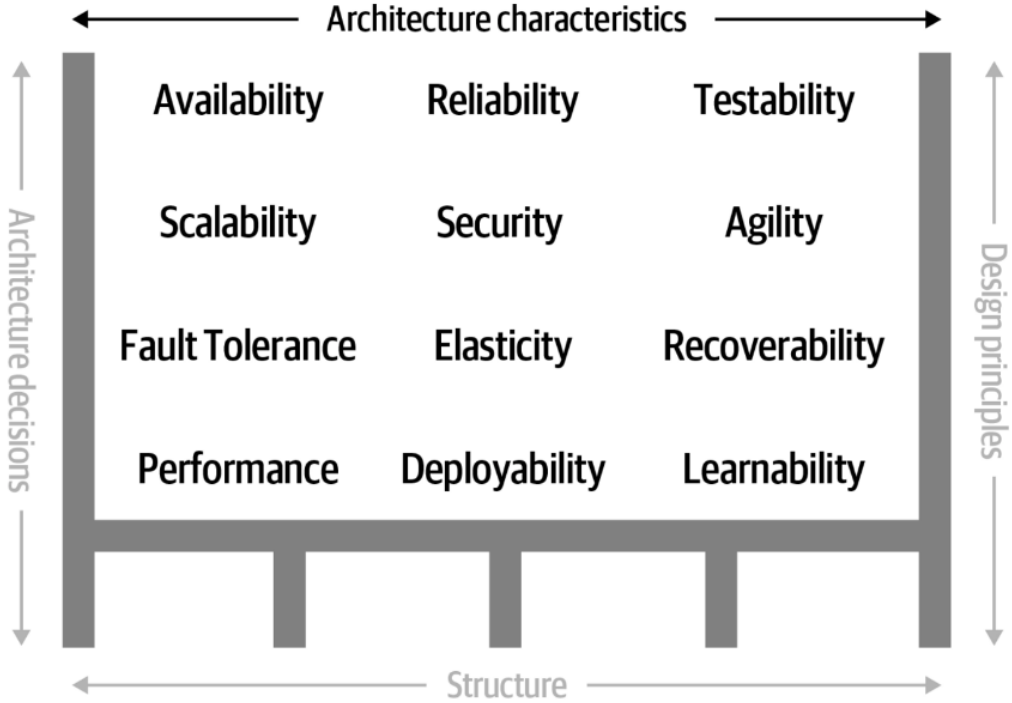
Advantages of Software Architecture :

- Provides a solid foundation for software project.
- Helps in providing increased performance.
- Reduces development cost.

Disadvantages of Software Architecture :

- Sometimes getting good tools and standardization becomes a problem for software architecture.
- Initial prediction of success of project based on architecture is not always possible.

Architecture Characteristics



1. Understandability:

- When defining the Architecture Structure our goal should not be just to make an effective software architecture structure. But It should able to communicate easily, quickly understood by development teams and stakeholders at the same time it should meet the business requirements. When a new developer joins the product team they should able to understand the software architecture with a short introduction.

2. Usability :

- Achieving the Usability of a software product depends on a number of factors like target users, UX experience, and ease of using Product features. We need to consider what exactly Users want and What we are providing to users.

3. Securability:

The Application exposed on the web always has a risk of cyber-threats, if the application accessed by unauthorized users. Application security is responsible to stop or reduce cyber-threats, accidental actions, data theft, or loss of information. There are numerous ways to secure the application like authentication, authorization, auditing, and data encryption.

4. Reliability & Availability:

- Reliability is an attribute of the system responsible for the ability to continue to operate under predefined conditions.
- Availability of the Application is calculated based on Total Operation Time divided by Total Time this is expressed in percentage like 99.9%, it is also expressed in the number of 9s.

5. Interoperability:

- Most of applications services are required to communicate with external systems to provide full-fudged services. A well-designed software architecture facilitates how well the application is interoperable to communicate and exchange the data with external systems or legacy systems.

6. Testablity:

- An industry estimates 30 to 40 percent of the cost is taken by Testing. The role of Software Architect to ensure they design every component can be testable. A Testable Architecture should clearly show all the interfaces, application boundaries, and integration between components.

7. Scalability:

- Over time business will grow and the number of users of the application will grow 1000’s to 100000’s. When the load gets increased the application should able to scale without impacting the performance. There are two types of scaling vertical scaling/scaling up and horizontal scaling or scaling out.

8.Performability:

Performance is the ability of the application to meet timing requirements such as speed & accuracy. The performance score is generally measured on throughput, latency, and capacity.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

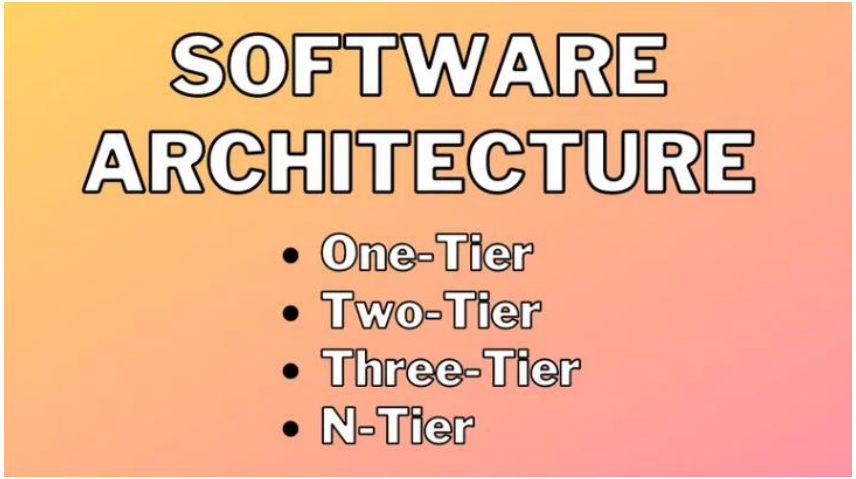
Unit I

Topics:

- Evolution of SA
- Fundamentals of SE: SDLC & Models

Evolution of SA

- Software Architecture consists of One Tier, Two Tier, Three Tier, and N-Tier architectures.
- A “tier” can also be referred to as a “layer”.
- Three layers are involved in the application namely Presentation Layer, Business Layer, and Data Layer.



Evolution of SA

1. Presentation Layer:

It is also known as the Client layer. The top most layer of an application. This is the layer we see when we use the software. By using this layer we can access the web pages. The main function of this layer is to communicate with the Application layer. This layer passes the information which is given by the user in terms of keyboard actions, mouse clicks to the Application Layer.

2. Business Layer:

It is also known as Business Logic Layer which is also known as the logical layer. As per the Gmail login page example, once the user clicks on the login button, the Application layer interacts with the Database layer and sends required information to the Presentation layer. It controls an application’s functionality by performing detailed processing. This layer acts as a mediator between the Presentation and the Database layer. Complete business logic will be written in this layer.

Evolution of SA

3. Data Layer:

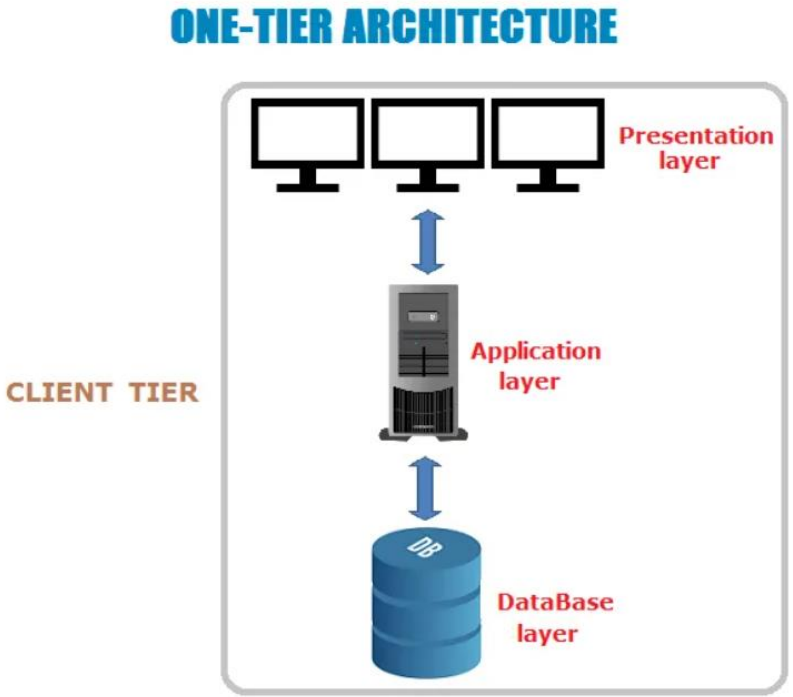
The data is stored in this layer. The application layer communicates with the Database layer to retrieve the data. It contains methods that connect the database and performs required action e.g.: insert, update, delete, etc.

Evolution of SA

1. One tier Architecture:

-One-tier architecture has all the layers such as Presentation, Business, Data Access layers in a single software package.

- Applications that handle all the three tiers such as MP3 player, MS Office come under the one-tier application.



Evolution of SA

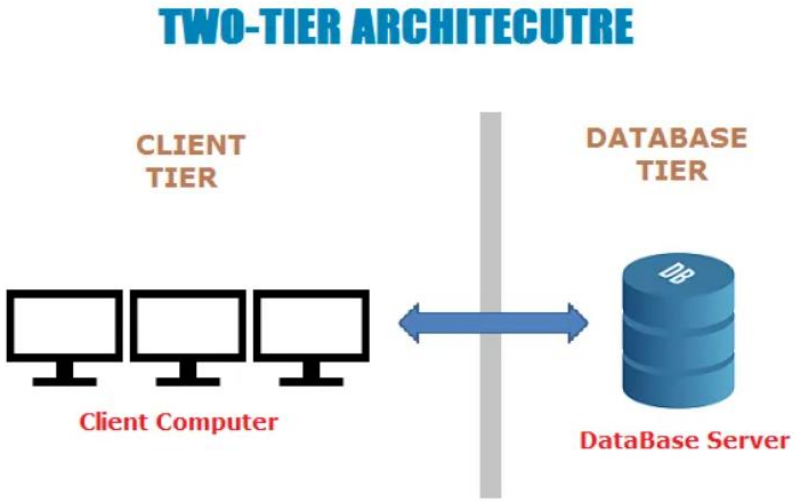
1. Two tier Architecture:

-The Two-tier architecture is divided into two parts:

1. Client Application (Client Tier)

2. Database (Data Tier).

- The communication takes place between the Client and the Server. The client system sends the request to the server system and the Server system processes the request and sends back the data to the Client System



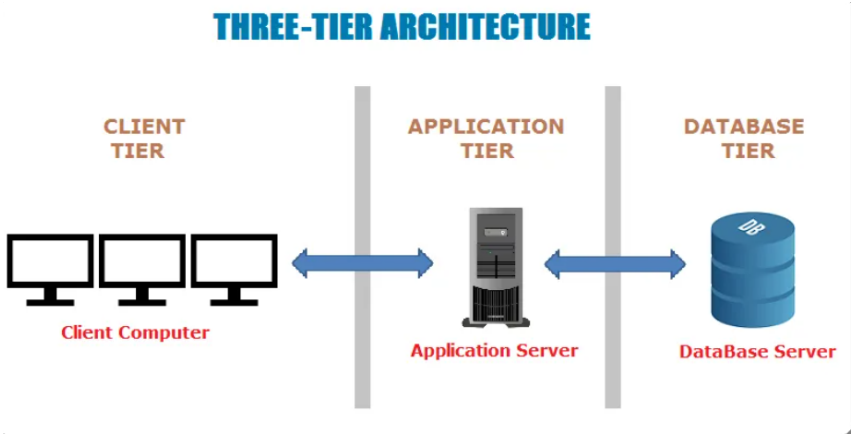
Evolution of SA

1. Three tier Architecture:

-The Three-tier architecture is divided into three parts:

1. Presentation layer (Client Tier)
2. Application layer (Business Tier)
2. Database layer (Data Tier)

The client system handles the Presentation layer, the Application server handles the Application layer, and the Server system handles the Database layer.



Evolution of SA

1. N tier Architecture:

- An N-tier architecture divides an application into logical layers and physical tiers.
- Layers are a way to separate responsibilities and manage dependencies. Each layer has a specific responsibility.
- A higher layer can use services in a lower layer, but not the other way around.

Fundamentals of SE

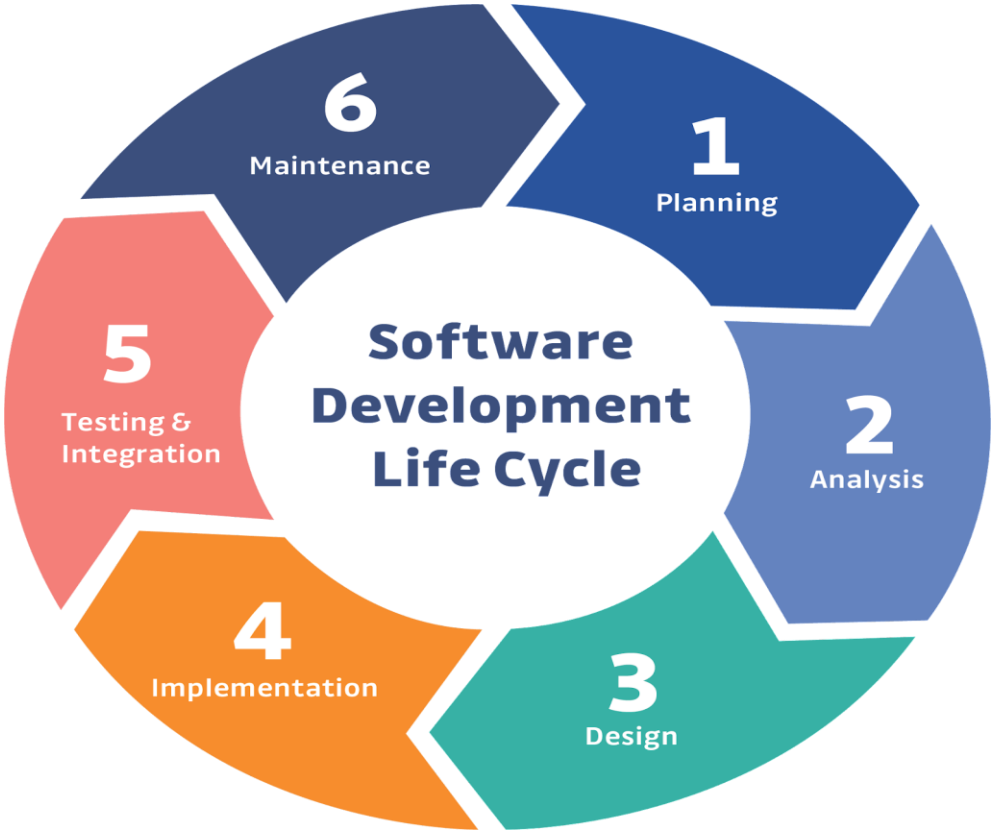
- **SDLC Model:**

A software life cycle model (also termed process model) is a pictorial and diagrammatic representation of the software life cycle.

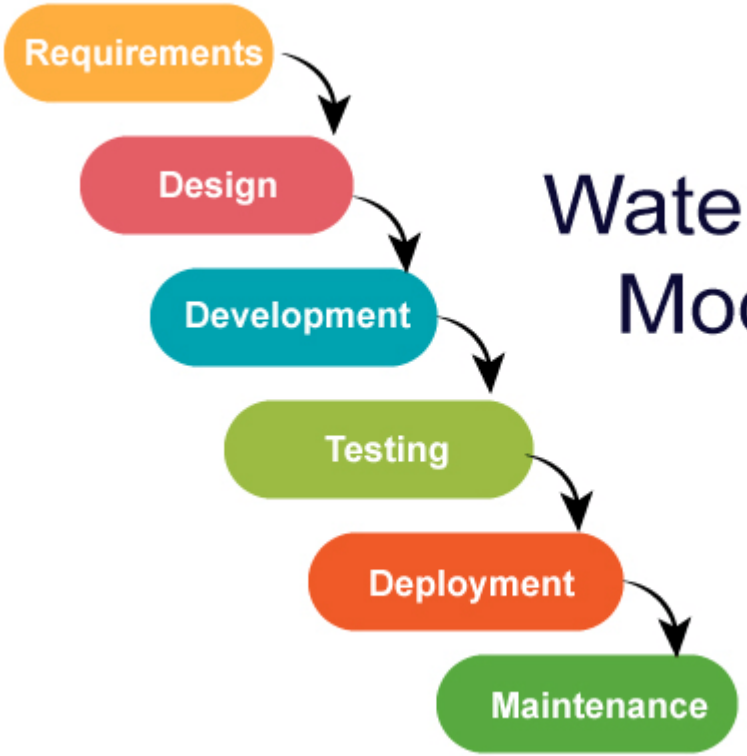
In other words, a life cycle model maps the various activities performed on a software product from its inception to retirement.

Different life cycle models may plan the necessary development activities to phases in different ways.

SDLC Phases


Software Development Life Cycle
1 Planning
2 Analysis
3 Design
4 Implementation
5 Testing & Integration
6 Maintenance

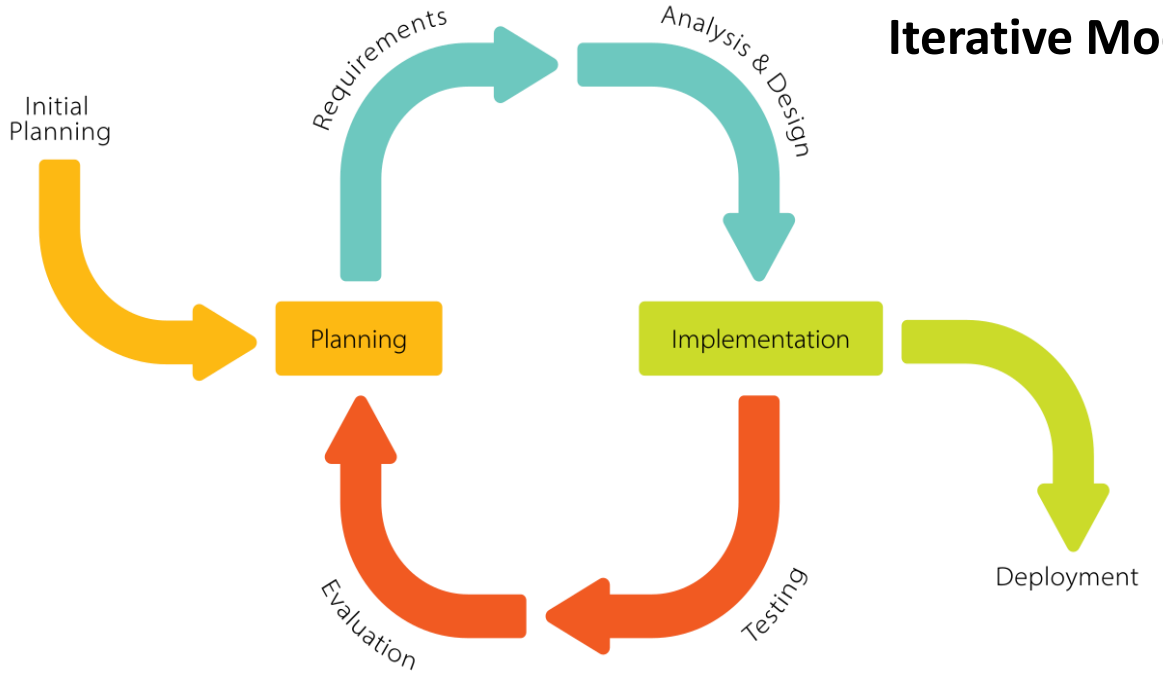
SDLC Models



Waterfall Model

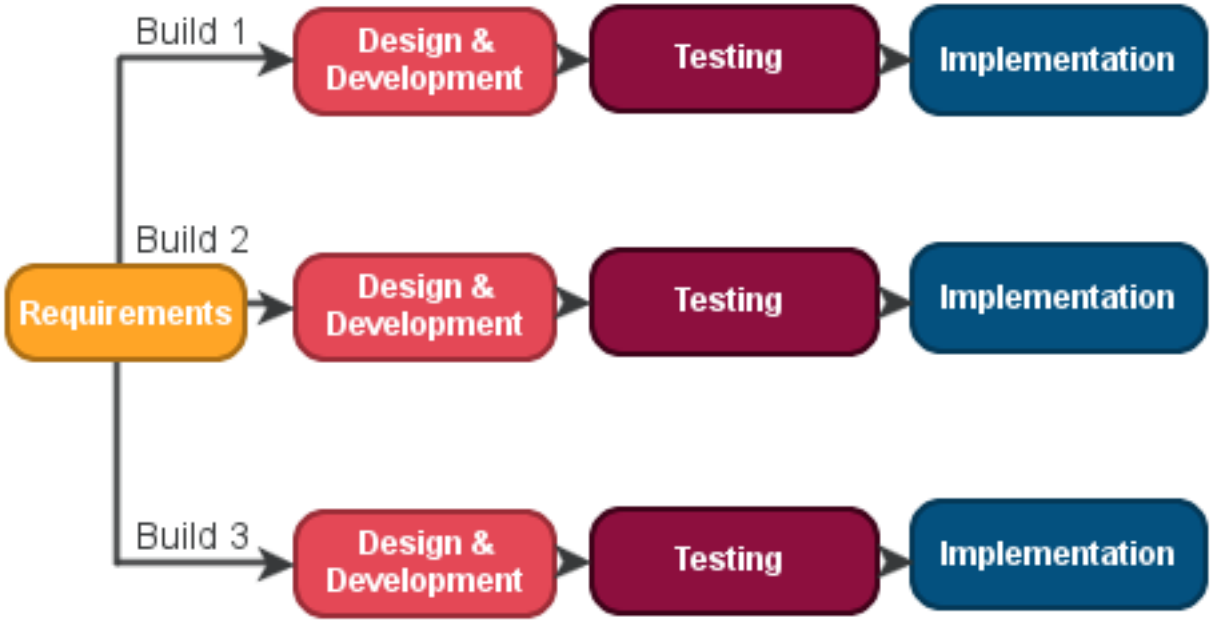
SDLC Models

Iterative Model

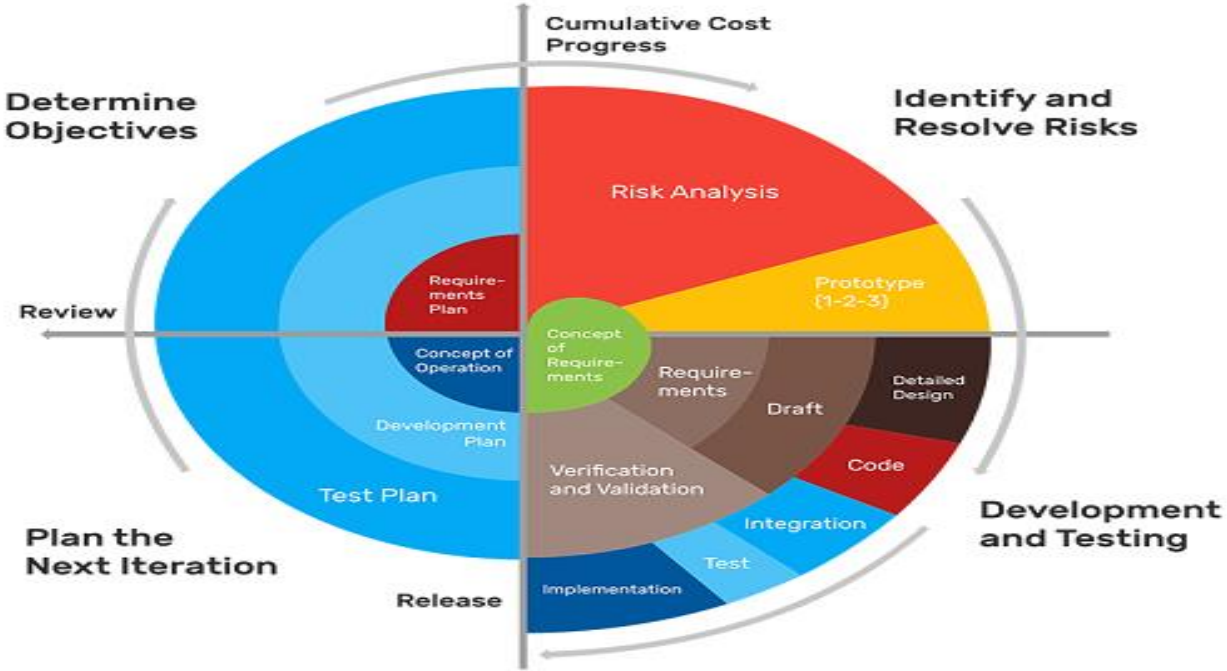


SDLC Models

Incremental Model

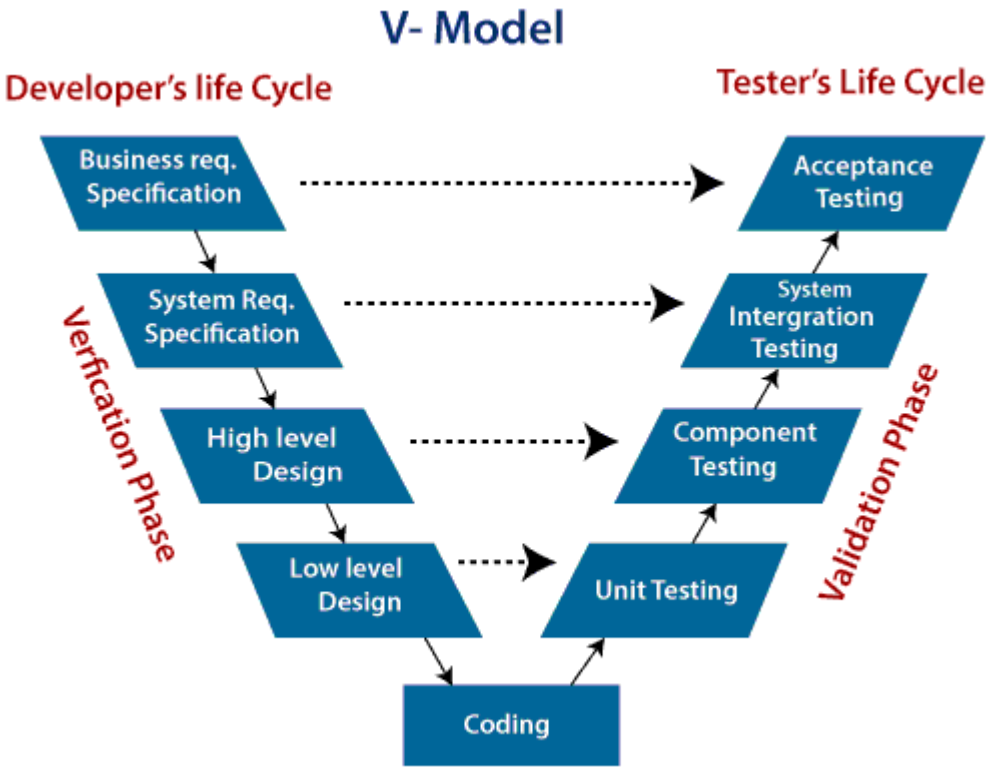


SDLC Models



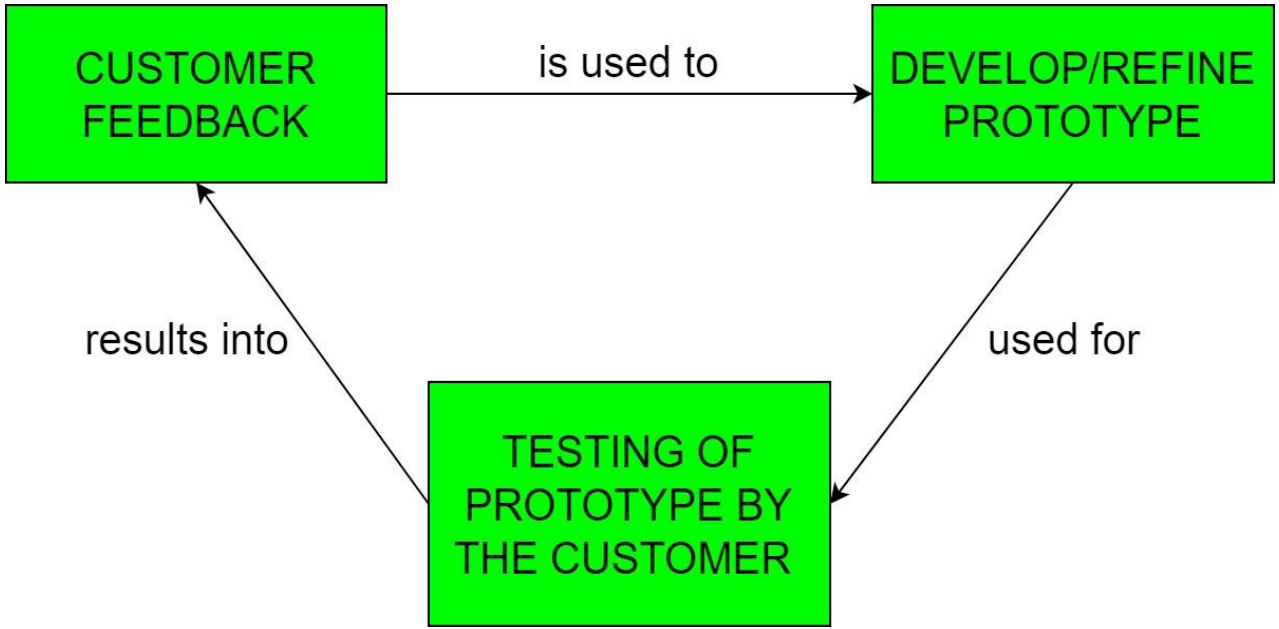
Spiral Model Methodology and its Phases

SDLC Models


V- Model
Developer's life Cycle
Tester's Life Cycle
Business req. Specification
System Req. Specification
High level Design
Low level Design
Coding
Unit Testing
Component Testing
System Intergration Testing
Acceptance Testing
Verfication Phase
Validation Phase

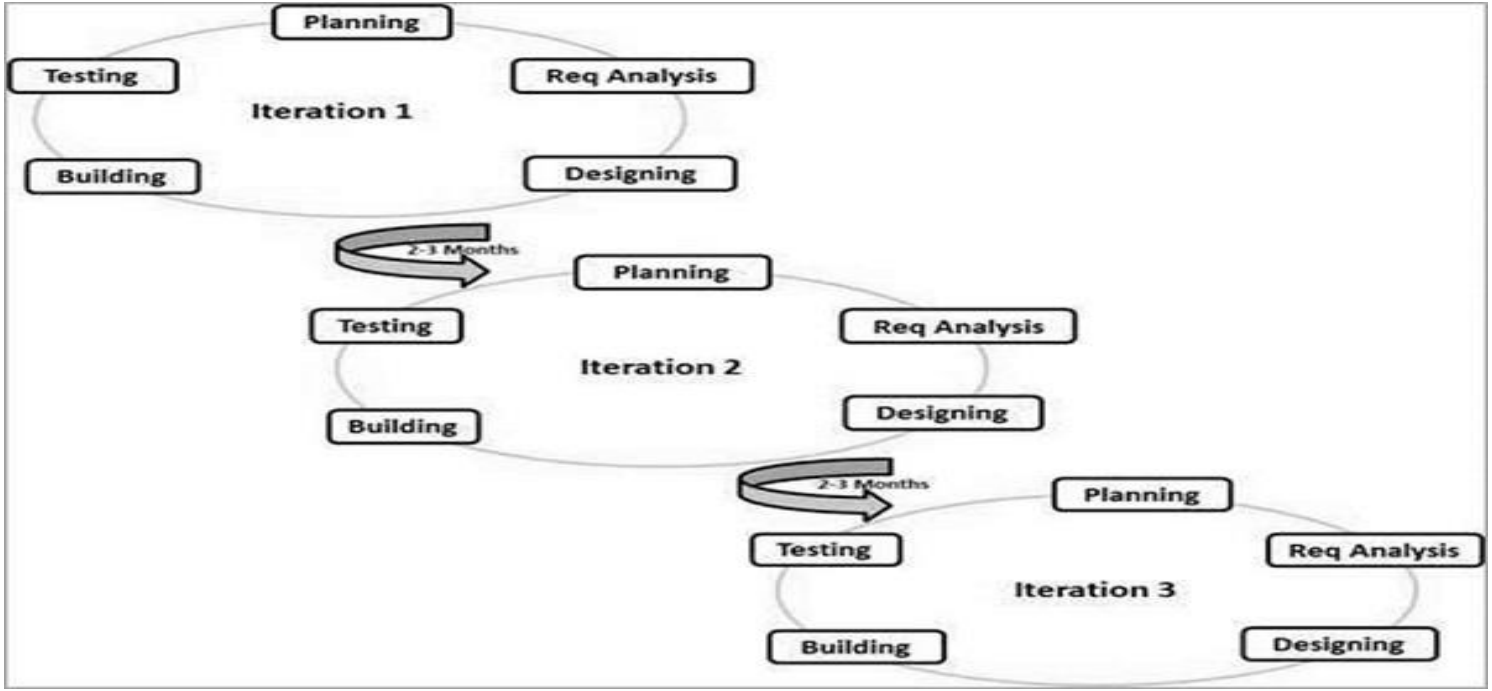
SDLC Models

Prototype Model



SDLC Models

Agile Model



SDLC Models

- DevOps Model:

SDLC Models

DevOps Model:

- Under a DevOps model, development and operations teams are no longer “siloed.”
- Sometimes, these two teams are merged into a single team where the engineers work across the entire application lifecycle, from development and test to deployment to operations, and develop a range of skills not limited to a single function.
- In some DevOps models, quality assurance and security teams may also become more tightly integrated with development and operations and throughout the application lifecycle. When security is the focus of everyone on a DevOps team, this is sometimes referred to as DevSecOps.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit I

Topics:

- 4+1 view model
- Elements of SA

4+1 View Model

- The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a multiple view model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.
- It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views.

4+1 View Model

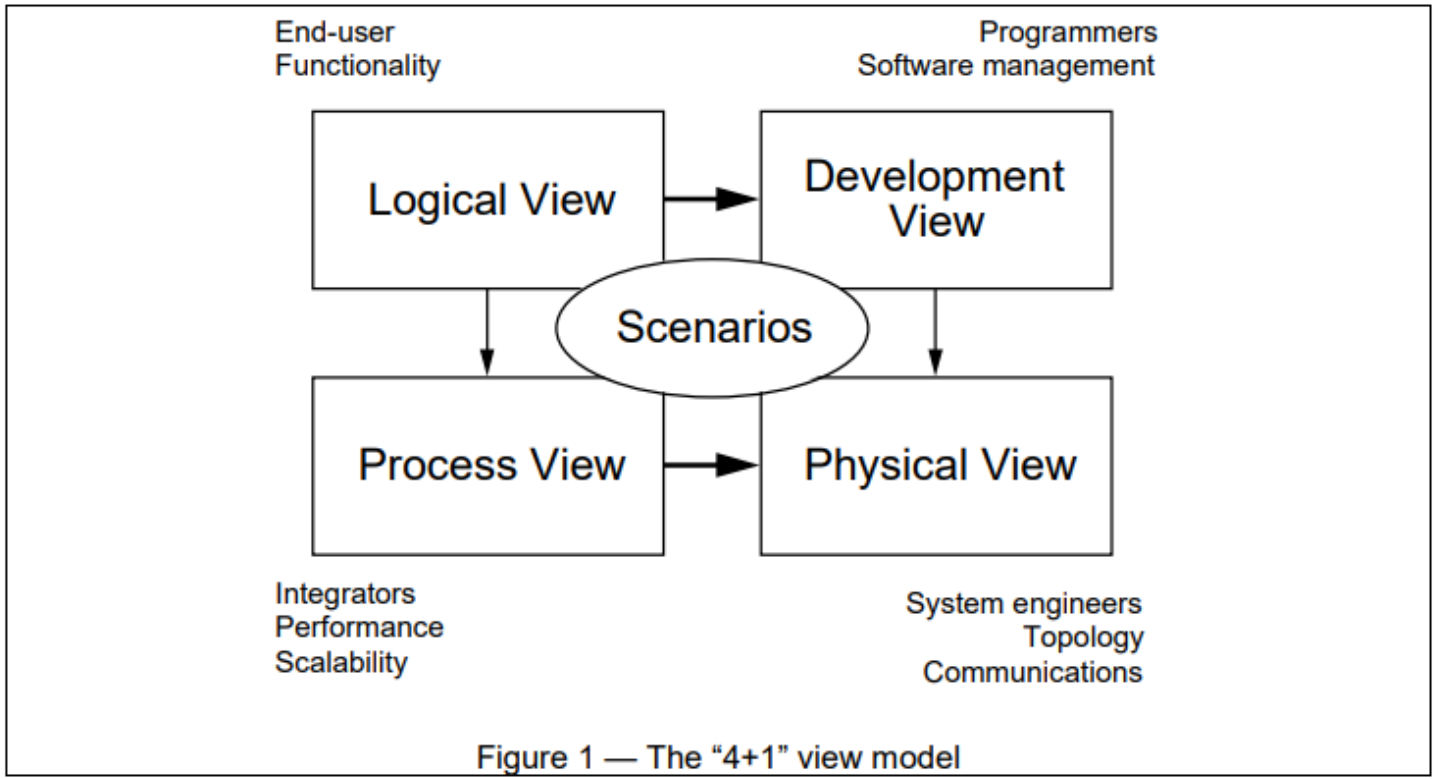


Figure 1 — The “4+1” view model

4+1 View Model

1. Logical View:

- The logical architecture primarily supports the functional requirements—what the system should provide in terms of services to its users.
- The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes. They exploit the principles of abstraction, encapsulation, and inheritance.
- Examples of Logical view models/diagrams: Sequence and Class diagrams.

4+1 View Model

2. Development View:

- The development architecture focuses on the actual software module organization on the software development environment.
- The software is packaged in small chunks—program libraries, or subsystems—that can be developed by one or a small number of developers.
- Provides a view from developers perspective which states where all code and its modules would be placed.
- Examples: Component and Package diagram.

4+1 View Model

3. Process View:

- The process architecture takes into account some non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, of system's integrity, of fault-tolerance.
- It provides a view from tasks' perspective.
- Checks the scalability and performance of system.
- Example: Activity Diagram

4+1 View Model

4. Physical View:

- It describes the mapping of software onto hardware and reflects its distributed aspect.
- It looks after the deployment of the system, tools and environment in which the product is installed.
- It takes a system engineer’s point of view in consideration.
- Example: Deployment diagram.

4+1 View Model

+1 Scenarios:

- This view model can be extended by adding one more view called scenario view or use case view for end-users or customers of software systems.
- It is coherent with other four views and are utilized to illustrate the architecture serving as “plus one” view, (4+1) view model.

Comparison of Views

	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Elements of SA

- **A software architecture is defined by a configuration of architectural elements--components, connectors, and configuration.**
- **The elements are:**

1. Components

2. Connectors

3. Configuration

Elements of SA

1. Components:

- **“A Software component is an architectural entity that -**
  - **1)encapsulates a subset of the system’s functionality**
  - **2)restricts access to that subset via an explicitly defined interface**
  - **3)has explicitly defined dependencies on itself”**
- A Component can be as simple as a single operation or as complex as an entire system.
- It can be “seen” by its end users completely (if the developer has made it public or it can be seen as a black box.
- It can be usable and reuseable, which is an important aspect.

Elements of SA

1. Components: (Contd..)

- One component can depend for its functionality on another component, and both can be linked by an ***interface***.
- The extent of the context captured by a component can include:

  - The interfaces it uses to link with other components
  - The availability of resources like the data files, directories
  - The required system software such as programming language run time environments, etc.
  - The hardware configurations.

Elements of SA

2. Connector:

- **“ A software connector is an architectural element tasked with effecting and regulating interaction among components”.**
- In box and line diagrams, the boxes represent the Components and lines represent the connectors.
- Most widely used connector is the Procedure call. They are directly implemented in programming languages where they directly enable the synchronous exchange of data between components.

Elements of SA

3. Configuration:

- **“An architectural configuration is a set of specific associations between the components and connectors of a software’s system architecture.”**
- A configuration may also be called as a graph where in nodes are the components and edges are the the connectors, the entire graph will be called as the configuration.

Notations

1. Component:

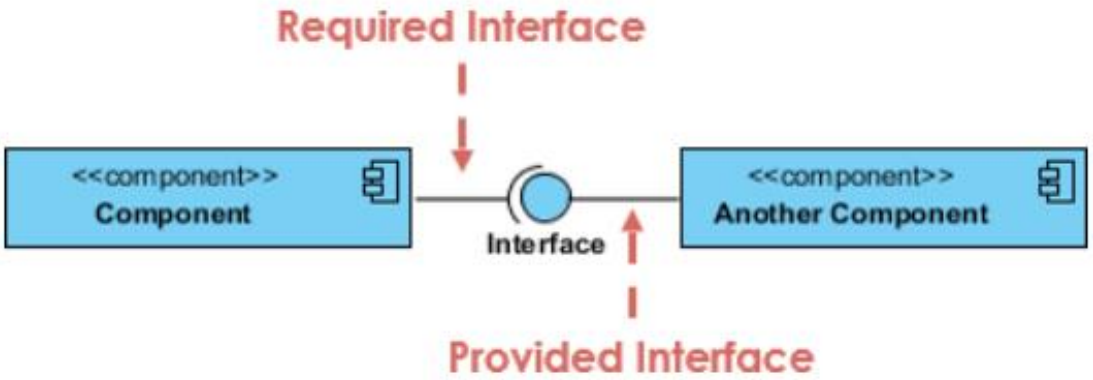
1. A rectangle with the component's name

2. A rectangle with the component icon

3. A rectangle with the stereotype text and/or icon

Notations

1. Interfaces:

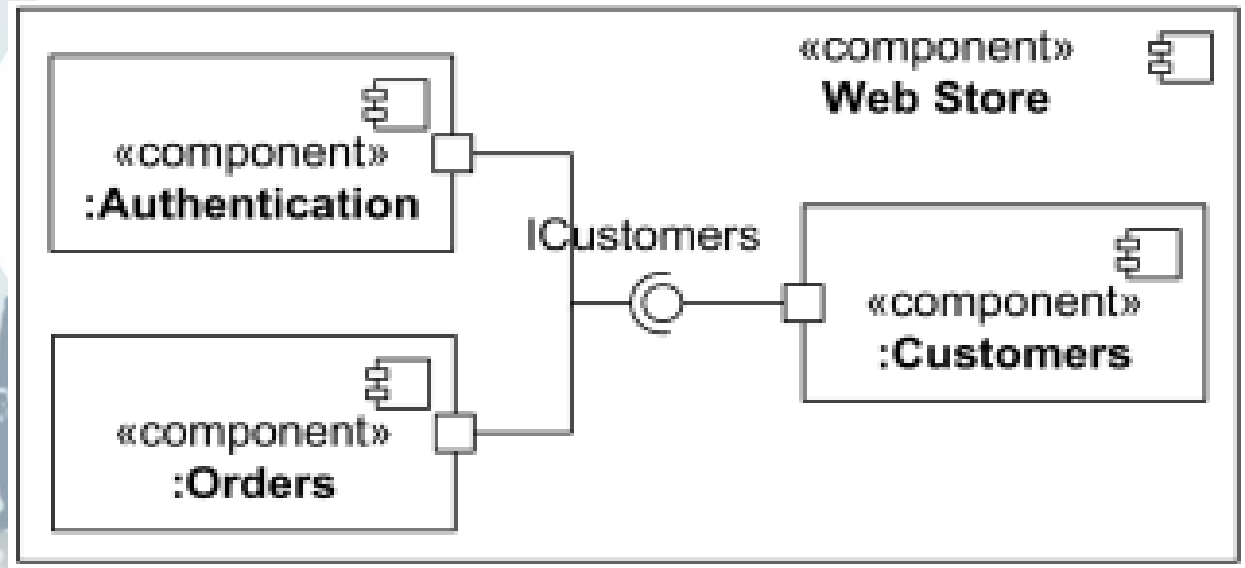


Notations

3. Ports

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.

Example: Notations



Lecture Takeaway

1. Define Software architecture and state needs.
2. Give advantages and disadvantages of Software Architecture.
3. Draw a Component Diagram for an Online Food Delivery App or Airline Management system. Assume necessary components for the entire flow of the application.
4. Define Software Architecture. Explain the need and significance of Software Architecture in detail.
5. Build the use case diagram “ Railway Management System”.
6. Compare and contrast the SDLC Models.
7. List the various challenges in Software Architectural Design.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit II

Topics:

- Challenges of SA
- Significance of SA
- Stakeholders and their influence
- Quality Attributes

Challenges of SA

1. Company’s Maturity:

Many organizations try to undertake the Software architecture program when they are not mature enough to do so. Big mistake, some immature organizations without a clear direction start creating code thinking the IT team will do miracles and fix or re-design everything and completely change their direction while actually doing the job resulting on weak architecture technology incurring in huge problems for architect managers and teams alike.

Challenges of SA

1. Company’s Maturity:

Typically, Software architects have no allocated budgets for which they have to account for. This is generally time consuming and it becomes a managing problem for the IT architect and the company as it is here where budget draws a limit on the efforts of the IT architect and the possibilities of solving potential issues.

Challenges of SA

3. Lack of Personnel

- IT architecture is always undertaking different projects to recognize issues and finding solutions. Being that, there is always a need for new for personnel (with new ideas and concepts) which entitles more financial resources and justification to the organization. Without enough team members, projects can be extended for longer periods resulting in a problem for both the IT department and the organization in the long run.
- Another issue an organization might face is the resource allocation of it's personnel. If there is no interest in a particular subject, productivity might become very low which will affect team flow and budget performance.

Challenges of SA

4. Communication:

Lack of communication with the internal IT team and stakeholders can become a huge barrier when achieving development and organizational goals. Finding a solution without effectively communicating an issue to the upper level of the organization creates problems and restricts the organization’s development.

Why is SA significant?

1. Enabling a System's Quality Attributes:

Whether a system will be able to enable its desired (or required) quality attributes is substantially determined by its architecture.

For eg. If your system requires high performance, then you need to pay attentionto managing the time-based behavior of elements, their use of sharedresources, and the frequency and volume of inter-element communication.

Why is SA significant?

2. Reasoning about and Managing Change:

- Modifiability—the ease with which changes can be made to a system—is a quality attribute.
- Virtually all software systems change over their lifetime, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But these changes are often fraught with difficulty.
- Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

Why is SA significant?

1. ***A local change*** can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.

2. ***A nonlocal change*** requires multiple element modifications but leaves the underlying architectural approach intact. For example, adding a new business rule to a pricing logic module, then adding new fields to the database that this new business rule requires, and then revealing the resultsof the rule in the user interface.

3. ***An architectural change*** affects the fundamental ways in which theelements interact with each other and will probably require changes allover the system. For example, changing a system from client-server topeer-to-peer.

Obviously, local changes are the most desirable, and so an effective architecture is one in which the most common changes are local, and hence easy to make.

Why is SA significant?

3. Enhancing Communication among Stakeholders:

Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different characteristics of the system that are affected by its architecture.

- The user is concerned that the system is fast, reliable, and available when needed.
- The customer is concerned that the architecture can be implemented on schedule and according to budget.
- The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
- The architect is worried about strategies to achieve all of those goals

Why is SA significant?

4. Carrying Early Design Decisions:

- Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life.
- It is also the earliest point at which these important design decisions affecting the systemcan be scrutinized.

Quality Attributes

1. Architecture and Requirements:

a. Functional Requirement

b. Quality Attribute Requirement

c. Constraints: A constraint is a design decision with zero degrees of freedom. That is, it’s a design decision that’s already been made. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a management fiat to make your system service oriented.

Quality Attributes

3. Availability: Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.

4. Interoperability: Interoperability is about the degree to which two or more systems can usefully

exchange meaningful information via interfaces in a particular context.

Quality Attributes

5. Modifiability—the ease with which changes can be made to a system—is a quality attribute.

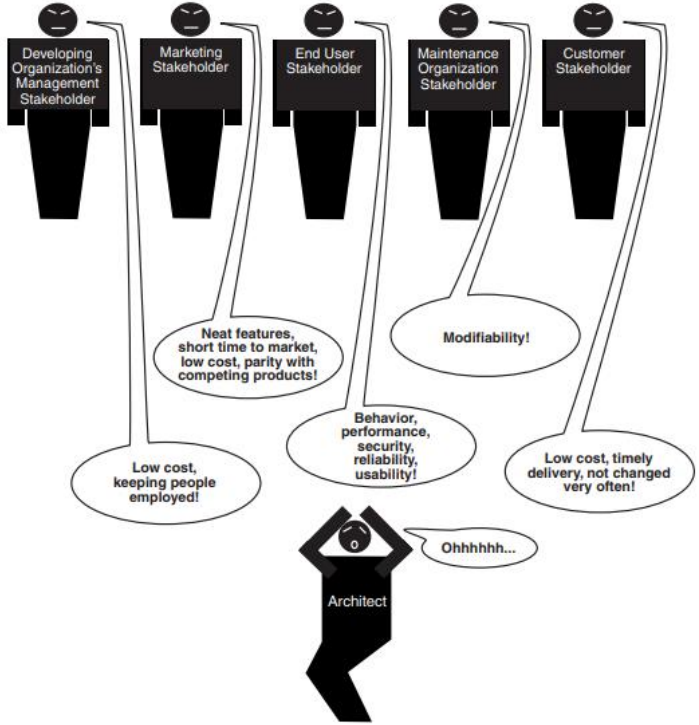
6. Performance:It’s about time and the software system’s ability to meet timing requirements. When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time.

7. Security: Security is a measure of the system’s ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.

Stakeholders of SA

- Many people and organizations are interested in a software system. We call these entities stakeholders.
- A stakeholder is anyone who has a stake in the success of the system: the customer, the end users, the developers, the project manager, the maintainers, and even those who market the system, for example.

Influence of Stakeholders



Stakeholders and their interest

Name	Description	Interest in Architecture
Analyst	Responsible for analyzing the architecture to make sure it meets certain critical quality attribute requirements. Analysts are often specialized; for instance, performance analysts, safety analysts, and security analysts may have well-defined positions in a project.	Analyzing satisfaction of quality attribute requirements of the system based on its architecture.
Architect	Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system.	Negotiating and making tradeoffs among competing requirements and design approaches. A vessel for recording design decisions. Providing evidence that the architecture satisfies its requirements.
Business Manager	Responsible for the functioning of the business/organizational entity that owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, etc.	Understanding the ability of the architecture to meet business goals.
Conformance Checker	Responsible for assuring conformance to standards and processes to provide confidence in a product's suitability.	Basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions.
Customer	Pays for the system and ensures its delivery. The customer often speaks for or represents the end user, especially in a government acquisition context.	Assuring required functionality and quality will be delivered; gauging progress; estimating cost; and setting expectations for what will be delivered, when, and for how much.
Database Administrator	Involved in many aspects of the data stores, including database design, data analysis, data modeling and optimization, installation of database software, and monitoring and administration of database security.	Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.
Deployer	Responsible for accepting the completed system from the development effort and deploying it, making it operational, and fulfilling its allocated business function.	Understanding the architectural elements that are delivered and to be installed at the customer or end user's site, and their overall responsibility toward system function.
Designer	Responsible for systems and/or software design downstream of the architecture, applying the architecture to meet specific requirements of the parts for which they are responsible.	Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understanding how their part will communicate and interact with other parts of the system.
Evaluator	Responsible for conducting a formal evaluation of the architecture (and its documentation) against some clearly defined criteria.	Evaluating the architecture's ability to deliver required behavior and quality attributes.

Stakeholders and their interest

Name	Description	Interest in Architecture
Implementer	Responsible for the development of specific elements according to designs, requirements, and the architecture.	Understanding inviolable constraints and exploitable freedoms on development activities.
Integrator	Responsible for taking individual components and integrating them, according to the architecture and system designs.	Producing integration plans and procedures, and locating the source of integration failures.
Maintainer	Responsible for fixing bugs and providing enhancements to the system throughout its life (including adaptation of the system for uses not originally envisioned).	Understanding the ramifications of a change.
Network Administrator	Responsible for the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment, configuration, maintenance, and monitoring of network components.	Determining network loads during various use profiles, understanding uses of the network.
Product-Line Manager	Responsible for development of an entire family of products, all built using the same core assets (including the architecture).	Determining whether a potential new member of a product family is in or out of scope and, if out, by how much.
Project Manager	Responsible for planning, sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities.	Helping to set budget and schedule, gauging progress against established budget and schedule, identifying and resolving development-time resource contention.
Representative of External Systems	Responsible for managing a system with which this one must interoperate, and its interface with our system.	Defining the set of agreement between the systems.
System Engineer	Responsible for design and development of systems or system components in which software plays a role.	Assuring that the system environment provided for the software is sufficient.
Tester	Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the architecture.	Creating tests based on the behavior and interaction of the software elements.
User	The actual end users of the system. There may be distinguished kinds of users, such as administrators, superusers, etc.	Users, in the role of reviewers, might use architecture documentation to check whether desired functionality is being delivered. Users might also use the documentation to understand what the major system elements are, which can aid them in emergency field maintenance.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit II

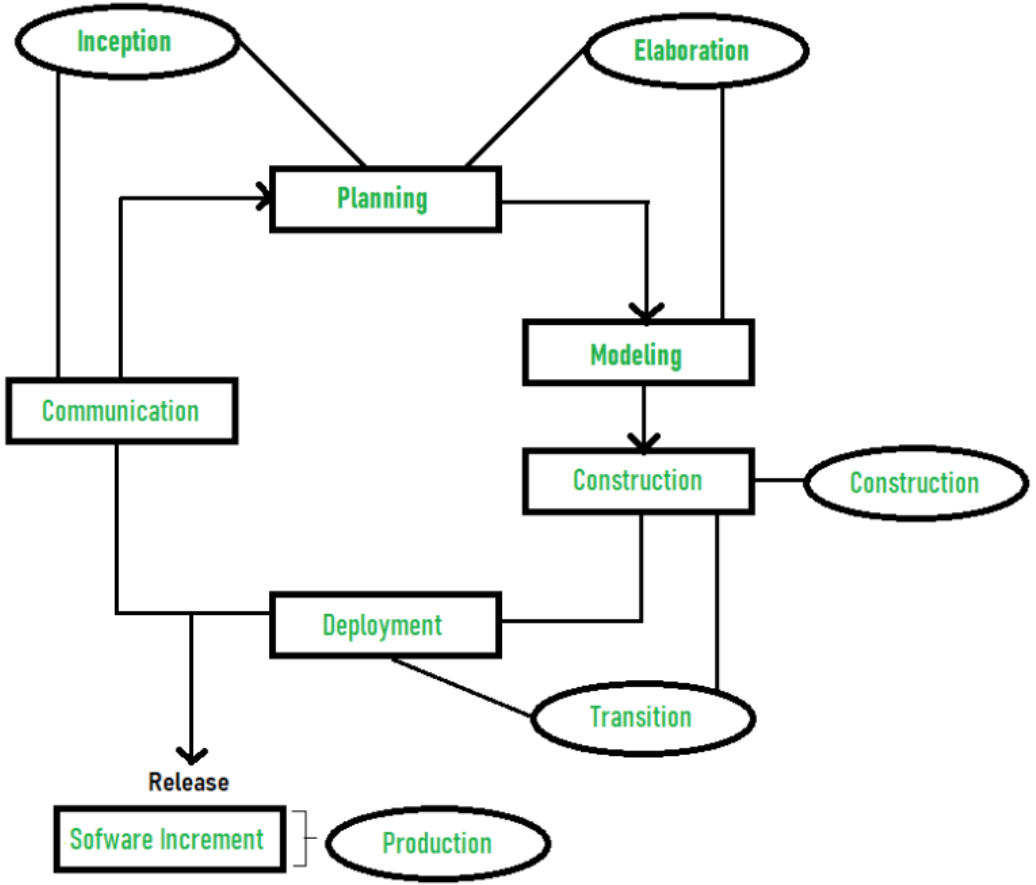
Topics:

- RUP by IBM
- Views in SA

RUP

- Rational Unified Process (RUP) is a software development process for object-oriented models.
- It is also known as the Unified Process Model. It is created by Rational corporation and is designed and documented using UML (Unified Modeling Language).
- RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rambaugh. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents wastage of resources.

Phases in RUP



Phases Description

1. Inception –

- Communication and planning are the main ones.
- Identifies the scope of the project using a use-case model allowing managers to estimate costs and time required.
- Customers' requirements are identified and then it becomes easy to make a plan for the project.
- The project plan, Project goal, risks, use-case model, and Project description, are made.

Phases Description

2. Elaboration –

- Planning and modeling are the main ones.
- A detailed evaluation and development plan is carried out and diminishes the risks.
- Revise or redefine the use-case model (approx. 80%), business case, and risks.
- Again, checked against milestone criteria and if it couldn’t pass these criteria then again project can be canceled or redesigned.

Phases Description

3. Construction –

- The project is developed and completed.
- System or source code is created and then testing is done.
- Coding takes place.

4. Transition –

- The final project is released to the public.
- Transit the project from development into production.
- Update project documentation.
- Beta testing is conducted.
- Defects are removed from the project based on feedback from the public.

Phases Description

5. Production –

- The final phase of the model.
- The project is maintained and updated accordingly.

Views: SA

- Views are ways of categorizing different perceptions of stakeholders using the **Rational Unified Process (RUP)** as a foundation.
- The views presented are:
  - Management
  - Software engineering
  - Engineering design
  - Architectural design
- Each view of the development process is comprised of **phases**, **activities**, **tasks**, and **steps**.

Management View

- Managers expect milestones in a software product development
- In modern software methodologies, deliverables are rarely completed before work begins on the next deliverable.
- Management view is based on the RUP life-cycle phases.
- The RUP defines four fundamental life-cycle phases:
  - Inception (*vision phase*)
  - Elaboration
  - Construction
  - Transition

Management View: Terminologies

- Lifecycle Objective Milestone (LCO)
- Acquirers: The stakeholders who desire the system are called the acquirers
- Builder: The software engineering organization that implements the system is sometimes referred to as the builder
- Lifecycle Architecture Milestone (LCA)
- The Initial Operational Capability Milestone (IOC)
- Product Release Milestone

Software Engineering View

- The main activities of software engineering are:
  - Requirements analysis and specification
  - Design
  - Implementation and testing
  - Deployment and maintenance
- Each software engineering activity maps to many phases of the management view.

Engineering Design View

- In this model, the design process is subdivided into four phases:
  - product planning (Specification of Information)
  - conceptual design (Specification of Principle)
  - embodiment design (Specification of Layout)
  - detailed design (Specification of Production)
- Each of these phases focuses on a different level of abstraction and a different set of design objectives.
- The design phases do not proceed in a pure serialized fashion.

Architectural View

- This view focuses on ***architecture-driven software construction***.
- The four phases of architecting are as follows:
  - Predesign phase
  - Domain analysis phase
  - Schematic design phase
  - Design development phase
- The four phases above when combined with the following build phases form an architecture-driven software construction method:
  - Project documents phase
  - Staffing or contracting phase
  - Construction phase
  - Postconstruction phase

Synthesizing the Views

- Each view contains some overlapping concepts and ways of visualizing the process.
- There is no one right way to design and construct software.
- You should choose the activities that make the most sense given the project at hand, finding a balance between risk and development speed.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit II

Topics:

- Architecture Design Process
- Types of Operators

Architecture Design Process

- Architecture design focuses on the decomposition of a system into components and the interactions between those components to satisfy functional and nonfunctional requirements.
- **Primary Output**: Architectural Description

Architecture Design Process

- The basic architecture design process is composed of these steps:
  - 1. Understand the problem.
  - 2. Identify design elements and their relationships.
  - 3. Evaluate the architecture design.
  - 4. Transform the architecture design.

Architecture Design Process

1. Understand the problem:

- 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).
- Also called as the implementation trap.
- Software engineers who are not given clear direction of the problem to solve may end up focusing their efforts on solving technical problems that ultimately do not address the original problem.
- Hence understanding the need for problem solution is something that needs to be addressed first.

Architecture Design Process

2. Identify design elements and their relationships:

- In this step, we establish a baseline decomposition that initially splits the system based on functional requirements.
- The decomposition can be modeled using a design structure matrix (DSM), which represents the dependencies between design elements without prescribing the granularity of the elements.
- A first draft of this model could be created during the predesign phase, and then revised during subsequent steps.

Architecture Design Process

3. Evaluate the architecture design:

- The third step involves evaluating the architectural design to determine if it satisfies the architectural requirements.
- The design is evaluated in order to gather qualitative measures or quantitative data, both of which are metrics. Each metric is then compared to the quality attribute requirements.
- If the measured or observed quality attribute does not meet its requirements, then a new design must be created.In order to evaluate an architecture design, you must have clearly articulated the quality attribute requirements that are affected by the architecture.
- It is not enough to say "the system must be fast" or "the system must be easy to modify or adapt to customer needs." These statements may be acceptable in an initial requirements list (or marketing requirements document), but they are not specific enough to evaluate the design later.

Architecture Design Process

4. Transforming the architecture:

- This step is performed after an evaluation or informal assessment of the architectural design. If the architectural design does not fully satisfy its quality attribute requirements, then it must be changed until it does satisfy them.
- However, instead of starting from scratch, we can take the existing design and apply design operators to it in order to transform it.
- The new version of the design is then assessed and the process continues until a satisfactory design is achieved.

Modular Operators

A design is transformed by applying operators. There are two types of operators- Modular and Design.

There are two types of design operators:

1. ***Splitting*** a design into two or more modules
2. ***Substituting*** one design module for another
3. ***Augmenting*** the system by adding a new module
4. ***Excluding*** a module from the system
5. ***Inverting*** a module to create new interfaces (design rules)
6. ***Porting*** a module to another system

Design Operators

1. ***Decomposition*** of a system into components.
2. ***Replication*** of components to improve reliability.
3. ***Compression*** of two or more components into a single component to improve performance.
4. ***Abstraction*** of a component to improve modifiability and adaptability.
5. ***Resource sharing***, or the sharing of a single component instance with other components to improve integrability (the ability to integrate applications and systems), portability, and modifiability.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit II

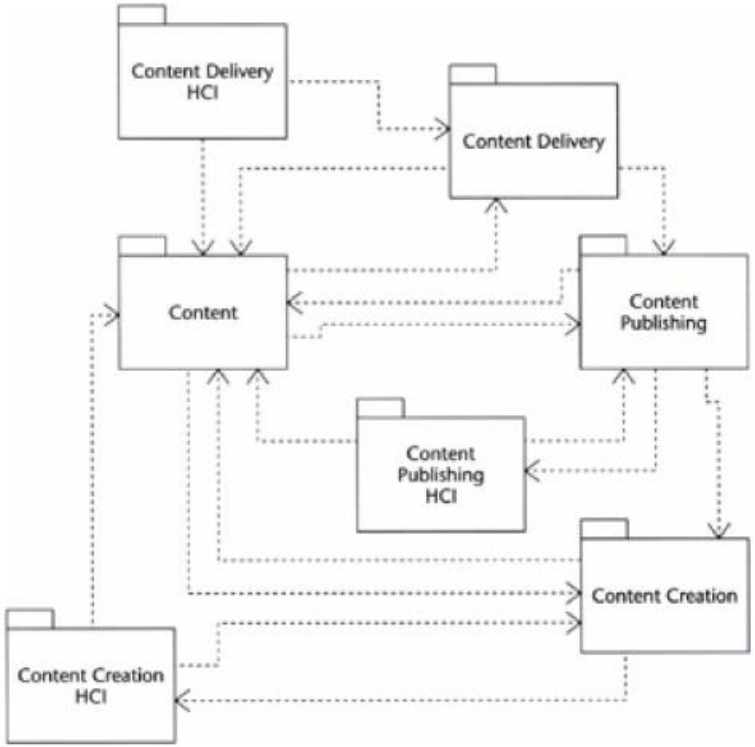
Topics:

- Design structure & matrix
- The Vitruvian Triad
- Function, form and Fabrication
- The scope of design
- The Psychology and Philosophy of Design

The DSM

- The architecture of a software application is often represented as a set of interconnected modules or subsystems, often called components. These modules are organizational constructs that structure source code, but often are not directly visible in the source code. Each component is also called as a Design element.
- The design structure matrix (DSM) is , in which each design element is represented as a design task and the dependencies between the design elements become dependencies between design tasks. The dependencies also represent the paths through the design that are affected by design decisions.

Design Structure using UML Package Diagram



Design Structure using matrix

	Content	Content Creation	Content Creation HCI	Content Publishing	Content Publishing HCI	Content Delivery	Content Delivery HCI
Content	O	X		X		X	
Content Creation	X	O	X				
Content Creation HCI	X	X	O				
Content Publishing	X	X		O	X		
Content Publishing HCI	X			X	O		
Content Delivery	X			X		O	X
Content Delivery HCI	X					X	O

DSM

The DSM is interpreted as follows:

- The dependencies of a given element are represented as an X in a cell along the row of that element.
- The Os along the diagonal are just a helpful visual marker; it is meaningless to state that an element is dependent upon itself.
- In this example, the design element Content is dependent on Content Creation, Content Publishing, and Content Delivery. Every element is dependent on the Content element. Therefore design decisions affecting the Content element affect design decisions in every other element.

The Vitruvian Triad

- Vitruvius wrote in his Ten Books on Architecture that an artifact should exhibit the principles of *firmitas, utilitas, and venustas- that is, stability, utility, and beauty.* These three principles are known as the Vitruvian triad.
- The Vitruvian triad is a useful anecdotal device for thinking about software architectures and the process of architecting.
- Utilitas includes the analysis of the purpose and need of the application (function).
- Venustas includes the application of design methods to balance many competing forces to produce a useful system that serves some application (form).
- Firmitas includes the principles of engineering and construction (fabrication).
- Thus we can think of the Vitruvian triad as the principles of function, form, and fabrication

Function & Product Planning

- The requirements establish the function of the application or system, not to be confused with the functional specification of the system (that is, part of the form).
- For the architectural metaphor to make sense, an application's architecture must address end-user needs.

Form & Interaction Design

- There is a distinction between design that directly affects the ultimate end user (**interaction design**) of the application and all other design (**program design**).
- Poor interaction design leads to *cognitive friction*.
- Cognitive friction is "the resistance encountered by a human intellect when it engages with a complex system of rules that change as the problem permutes“
- A good user interface can hide local complexities in an application

Fabrication

- An architecture must be realizable; it must be possible to build the system as described by the architectural description. That may sound obvious, but it is often not even considered as an architectural attribute.
- It must be possible to apply current software engineering practices and technologies toward the implementation of the application or system that is described and to satisfy the specified quality attributes.
- It must be possible to build the system given the available resources such as time, staff, budget, existing (legacy) systems, and components. If existing technologies are insufficient for the problem at hand, will the team be able to invent what is necessary?

The Scope of Design

Any design activity can be seen from many points of view:

- Psychological
- Systematic
- Organizational

From the *psychological view*, design is a creative process that requires knowledge in the appropriate disciplines such as software engineering, computer science, logic, cognitive science, linguistics, programming languages, and software design methodologies as well as application domain-specific knowledge.

The Scope of Design

From the *systematic* view, design is seen as an architecting or engineering activity that involves finding optimized solutions to a set of objectives or problems while balancing competing obstacles or forces.

The *organizational* view considers essential elements of the application or system life cycle

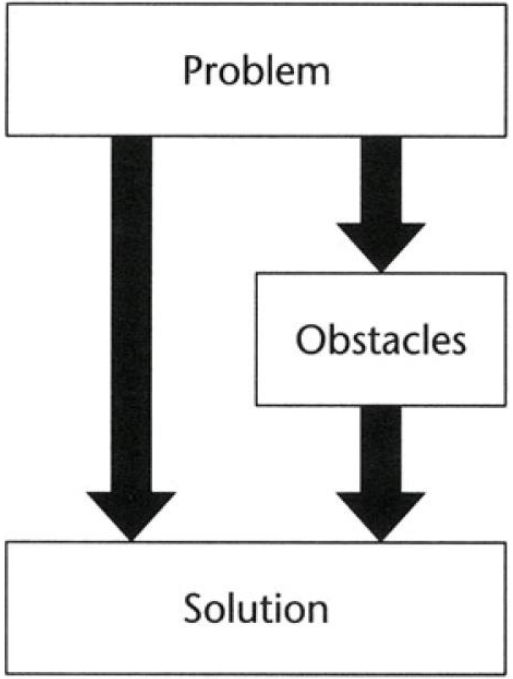


Figure 4.1: Design solution represented as a function of problems and obstacles.

The Psychology and Philosophy of Design

- All design methodologies consider design as an activity of finding or creating solutions to problems given a set of obstacles to overcome.
- Among the types of obstacles that exist between a problem and its solution is a lack of understanding of the problem. Fundamental to good design is a grasp of the problem being solved; without this understanding it is hard to create a good design because it cannot be assessed based on its effectiveness in satisfying the problem.
- This is not to say that designs created without a complete understanding of the problem areinherently bad, but the likelihood of having stumbled onto the best solution is diminished.

Lecture Takeaway

1. List and explain Quality attributes.
2. List and explain modular operators.
3. What is scope of design?
4. RUP model.
5. List 5 stakeholders with their interest in SA.
6. Explain the architecture design process.
7. Draw the Design Structure Matrix for “Content Management System”.
8. Explain the various Views of SA.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit III

Topics:

- Complexity & Modularity
- Cohesion & Coupling
- Difference between Software Architecture and Design.
- What are Models?

Complexity

- The term complexity stands for state of events or things, which have multiple interconnected links and highly complicated structures.
- In software programming, as the design of software is realized, the number of elements and their interconnections gradually emerge to be huge, which becomes too difficult to understand at once.
- Complexity is one of the primary problems that we attempt to address with software development tools and methods.

Complexity

- Complexity, if not managed, can cause a product to be low quality or delivered late, over budget, or the project could be cancelled completely.
- There are no easy solutions to removing complexity and improving productivity and quality.
- The best we can do is reduce or manage complexity.

Complexity

- Complexity in designs and processes can be measured by the interconnectedness of things.
- In order for a system or process to exhibit complexity, it must be composed of multiple parts that are interconnected.
- We refer to these connections as dependencies.
- If some task or design element B is dependent upon A, then performing task A (or modifying design A) must occur before performing task B (or modifying design B).

Complexity

- In the design context, if we have already created design elements A and B and we change A, then we must change B.
- If A and B are interdependent, then changing A and B forms a loop and making a change in one requires making a change in the other.
- Complex systems can be represented as a graph where the nodes correspond to design elements or tasks and the directed edges correspond to dependencies. An example of such a graph is depicted in further slide , where elements A and B are interdependent.

Complexity

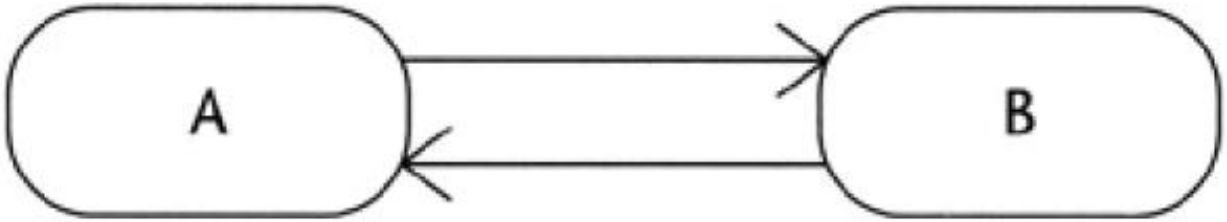


Figure 5.1: Interdependency between tasks A and B.

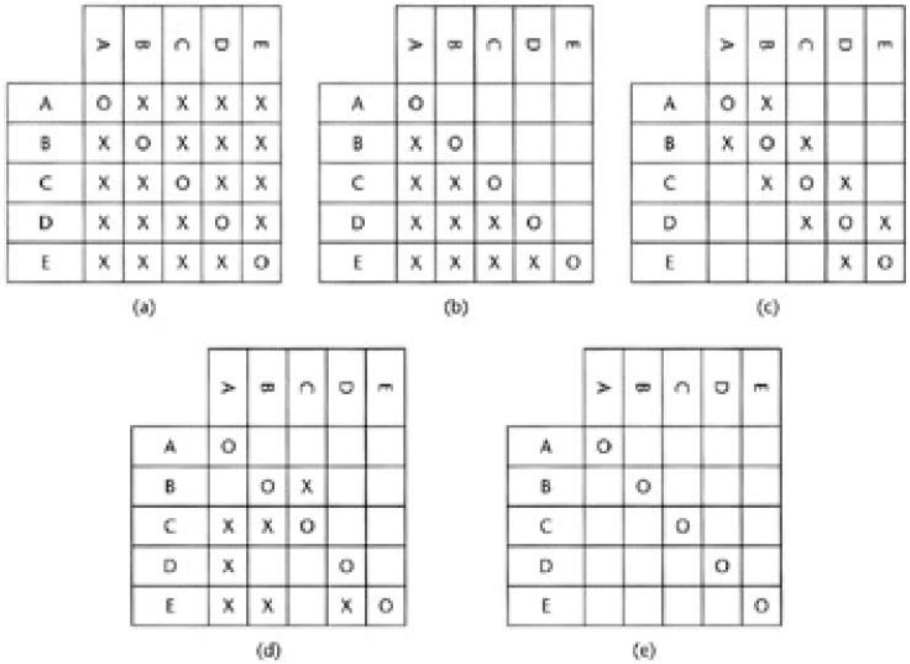
Complexity

Complexity arises in many aspects of software design, including:

- Requirements
- User interface
- "High-level" design
- "Low-level" design
- Source code

- Murray Gell-Mann says, "When defining complexity it is always necessary to specify a level of detail up to which the system is described, with finer details being ignored. This is termed as Granularity.

Varying degrees of Complexity



	A	B	C	D	E
A	O	X	X	X	X
B	X	O	X	X	X
C	X	X	O	X	X
D	X	X	X	O	X
E	X	X	X	X	O

(a)

	A	B	C	D	E
A	O				
B	X	O			
C	X	X	O		
D	X	X	X	O	
E	X	X	X	X	O

(b)

	A	B	C	D	E
A	O	X			
B	X	O	X		
C		X	O	X	
D			X	O	X
E				X	O

(c)

	A	B	C	D	E
A	O				
B		O	X		
C	X	X	O		
D	X			O	
E	X	X		X	O

(d)

	A	B	C	D	E
A	O				
B		O			
C			O		
D				O	
E					O

(e)

Figure 5.3: Varying degrees of complexity.

Modularity

- Modularity is a fundamental principle for achieving simplicity of design and development effort and thus reducing and managing complexity.
- Modularity is the primary principle by which we manage complexity of designs and design tasks by identifying and isolating those connections or relationships that are the most complex.
- Modularity specifies the separation of concerned ‘components’ of the software which can be addressed and named separately. These separated components are referred to as ‘modules’.

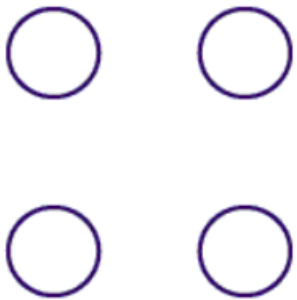
Modularity

- The module simply means the architectural components that are been created by dividing the software architecture. The architecture is divided into various components that work together to form a single functioning item. This process of creating software modules is known as Modularity.
- The basic principle of Modularity is that “Systems should be built from cohesive, loosely coupled components (modules)” which means s system should be made up of different components that are united and work together in an efficient way and such components have a well-defined function.

Coupling

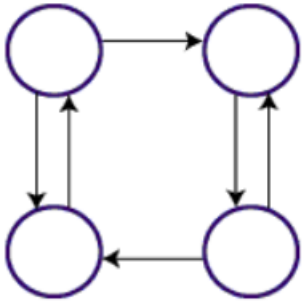
- Coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. Uncoupled modules have no interdependence at all within them.
- A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Coupling Techniques



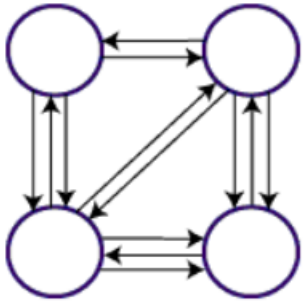
Uncoupled: no dependencies

(a)



Loosely Coupled: Some dependencies

(b)



Highly Coupled: Many dependencies

(c)

Cohesion

- Cohesion is a measure of the degree to which the elements of the module are functionally related.
- It is the degree to which all elements directed towards performing a single task are contained in the component.
- Basically, cohesion is the internal glue that keeps the module together.
- A good software design will have high cohesion.

Cohesion Types

1. Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
2. Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
3. Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.

Cohesion Types

4. Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span.

5. Logical Cohesion: The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

Software Design and Arch

Software Architecture	Software Design
1. Software architecture is about the complete architecture of the overall system.	1. Software design is about designing individual modules/components.
2. In general it refers to the process of creating high level structure of a software system.	2. In general it refers to the process of creating a specification of software artifact which will help to developers to implement the software.
3. Software architecture is more about the design of entire system.	3. Software design is more about on individual module/component.
4. It is a plan which constrains software design to avoid known mistakes and it achieves one organizations business and technology strategy.	4. It is considered as one initial phase of Software Development Cycle (SSDLC) and it gives detailed idea to developers to implement consistent software.
5. Software architecture defines the fundamental properties.	5. Software design defines the detailed properties.

What are Models?

- Models are realized as diagrams, formulae, textual descriptions, or combinations of these.
- Models may be grouped into views of the system, where each view represents some aspect (or dimension) of a system
- Models are specified in modeling languages or notations and textual descriptions. E.g. UML class models, UML package diagrams
- There are three parts to interpreting system representation models:
  - Syntax : tells us how to use the elements of the modeling notation
  - Semantics : is the meaning that a particular model has
  - Pragmatics : is the broader context in
  - which a model is related and the constraints and assumptions affecting the model

Models

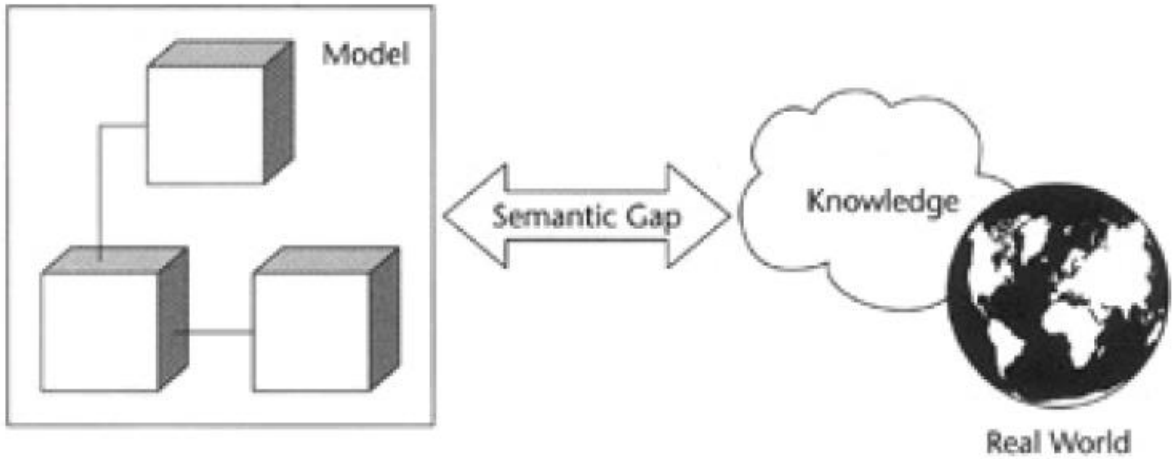


Figure 6.1: Models are representations of knowledge.

Semantic gap

- The term semantic gap refers to the discontinuity between a thing being modeled and the model's own representation of that thing. Semantic gaps occur in all parts of the architectural model and system implementation.
- The larger the semantic gap between a model and reality, the harder it is to judge the correctness of the model.
- One way we can address the problem of the semantic gap is by using more models to help reduce the gap.

Uses of Models

- They can be used to represent systems knowledge.
- They are used to simulate existing systems.
- They also provide a guide to systems analysis and design.
- Models for simulation are primarily used to discover new behaviors of a system that is being studied.
- Design models assist architects in making design decisions before the actual system is constructed, when such decisions are more cost-effective.

Lecture Takeaway

1. Define Coupling and Cohesion.
2. Define Complexity and Modularity.
3. Differentiate between Software architecture and design.
4. What are models? State uses.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit III

Topics:

- What are Models?
- Use of models?
- Semantic gap.
- Problem and solution domain
- Views

What are Models?

- Models are realized as diagrams, formulae, textual descriptions, or combinations of these.
- Models may be grouped into views of the system, where each view represents some aspect (or dimension) of a system
- Models are specified in modeling languages or notations and textual descriptions. E.g. UML class models, UML package diagrams
- There are three parts to interpreting system representation models:
  - Syntax : tells us how to use the elements of the modeling notation
  - Semantics : is the meaning that a particular model has
  - Pragmatics : is the broader context in
  - which a model is related and the constraints and assumptions affecting the model

Models

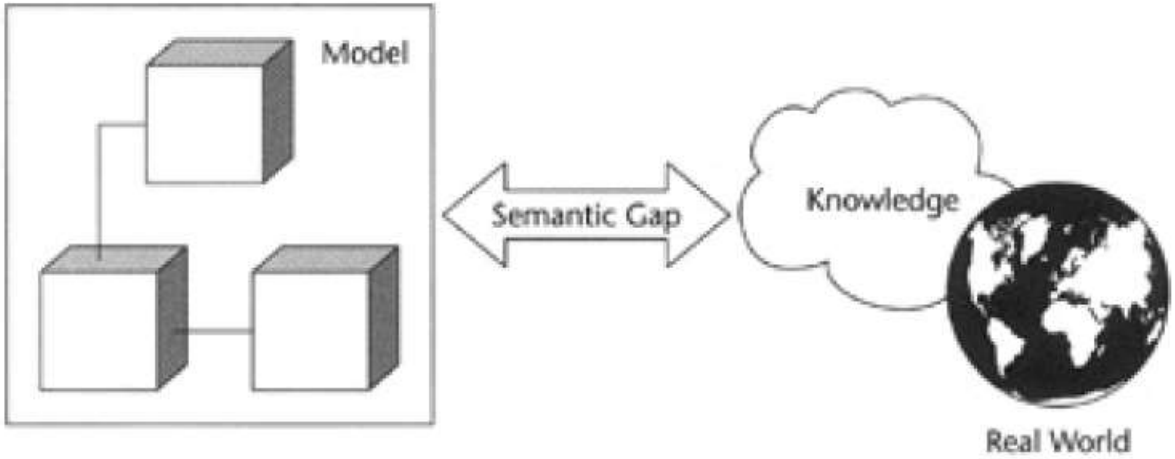


Figure 6.1: Models are representations of knowledge.

Semantic gap

- The term semantic gap refers to the discontinuity between a thing being modeled and the model's own representation of that thing. Semantic gaps occur in all parts of the architectural model and system implementation.
- The larger the semantic gap between a model and reality, the harder it is to judge the correctness of the model.
- One way we can address the problem of the semantic gap is by using more models to help reduce the gap.

Uses of Models

- They can be used to represent systems knowledge.
- They are used to simulate existing systems.
- They also provide a guide to systems analysis and design.
- Models for simulation are primarily used to discover new behaviors of a system that is being studied.
- Design models assist architects in making design decisions before the actual system is constructed, when such decisions are more cost-effective.

Problem and Solution domain

- Models can be divided into two groups:
  - those that model the problem domain
  - those that model the solution domain.
- Solution domain models can be divided into:
  - platform- or technology-independent models
  - platform- or technology-specific models.
- Problem domain models can be divided into:
  - systems analysis models
  - requirements analysis models.

Problem and Solution domain

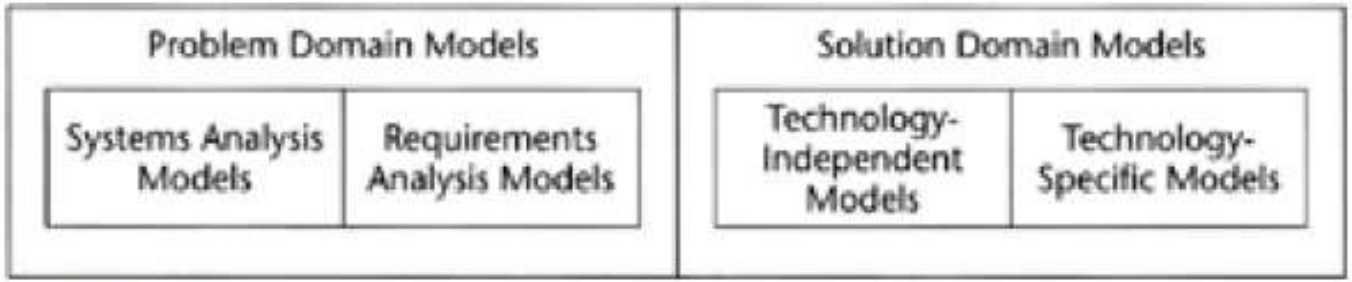


Figure 6.2: Problem and solution domain models.

Problem domain

- Commonly, the activity of creating problem domain models is called analysis.
- Methodologies for systems analysis include structured analysis, information modeling, and object-oriented analysis.
- A goal of problem domain modeling methodologies is to minimize the semantic gap between the real system (for example, an enterprise) and its representation models.

Solution Domain

- Commonly, the activity of creating solution domain models is called design.
- A goal of solution domain modeling methodologies is to reduce the semantic gap between the various models that form a chain of models from the problem domain models to the implementation.

Views

- Models may be classified by or organized into views.
- Views are aspects or dimensions of an architectural model.
- A view is a "representation of a whole system from the perspective of a related set of concerns." A view can contain one or more models.

Common Views

- Objectives/purpose. Describes what is needed.
- Behavior/function. Describes what the system does (to satisfy the objectives).
- Information/data. Describes the information created by and retained in the system.
- Form/structure. Describes the physical structure of the software (for example, modules and
- components).
- Performance. Describes how effectively the system performs its functions.

Examples of models

- Objectives and Purpose Models
  - Use-case model
- Behavioral/Functional Models
  - Scenarios and threads (sequence diagrams, activity diagrams)
  - State transition diagrams
- Information/Data Models
  - DFD
- Models of Form
  - Components and Connectors
  - Source code
- Nonfunctional/Performance Models
  - Analytical
  - Simulation
  - Judgmental

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit IV

Topics:

- What is ADL?
- Goals of ADL
- Elements of ADL
- Characteristics of ADL
- Architecting with Design Operators
- Functional Design Strategies

ADL

- A type of model for representing the form (the component structure) of a software system, which can be used to model a software architecture in terms of components and connectors and even generate a system or parts of a system.
- This representation model is known as an architectural description language.
- An architectural description language (ADL) has a formal, usually textual, syntax and can be used to describe actual system architectures in a machine-readable way.

Goals of ADL

Goals of architecture representation are:

- Prescribe the architectural constraints of components and connectors to any desired level of granularity
- Separate aesthetics from engineering
- Express different aspects of the architecture in an appropriate view or manner
- Perform dependency and consistency analysis
- Support system generation or instantiation

Elements of ADL

Common architectural description elements include:

1. **Computation (processing elements)**- Computation elements represent simple input/output relations and do not have retained state. These are processing elements.
2. **Memory (data element)** - Memory elements represent shared collections of persistent structured data such as databases, file systems, and parser symbol tables. These are data elements. Manager elements represent elements that manage state and closely related operations.
3. **Manager** - A manager provides a higher-level semantic view on top of primitive processing, data, and connecting elements.

Elements of ADL

4. **Controller** - A controller governs time sequences of events, such as a scheduler or resource synchronizer. The controller is responsible in managing the entire time line of the ADL with respect to various events that occur.
5. **Link** - Links are elements that transmit information between other elements. A link may be a communication channel between distributed processes or it may represent a user interface (HCI element)

Characteristics of ADL

1. **Composition**: An ADL should allow for the description of a system as a composition of components and connectors. As we have already seen, composition (describing a system as a hierarchy of simpler subsystems) helps us manage the complexity of a design or a design process. An ADL must support the ability to split a system or module into two modules.
2. **Abstraction**: A programming language provides an abstract view of the underlying hardware. A programmer does not need to think in terms of registers and binary machine instructions, for example. This abstract view allows a programmer to focus on higher-level concerns without having to think in terms of low-level implementation details. A programming language is considered an abstraction because it removes nonessential details for solving a problem at a particular level of granularity.

Characteristics of ADL

3. **Reusability**: An architectural module is not a reusable executable module like a reusable programming language library. Rather, it is a reusable pattern of component compositions.

4. **Configuration**: Configuration is related to composition. It should be possible with an ADL to describe a composite structure separately from its elements so that the composition can be reasoned about as an atomic element and support the dynamic reconfiguration of a system in terms of restructuring compositions without knowing about their internal structure.

Characteristics of ADL

5. **Heterogeneity**: Heterogeneity refers to the ability to mix architectural styles within a single architecture specification. At one level, the architecture may exhibit a particular pattern of compositions but the structure of each composition may follow a different pattern.
6. **Architecture Analysis:** An ADL should support the ability to analyze an architecture. Analysis of this sort goes beyond type checking such as may be performed by a programming language compiler. Analysis of architectures includes automated and nonautomated reasoning about quality attributes of a system. The difficulty in validating a program statically (by parsing it) applies to an architecture specification as well.

Design Operators

- Design operators are a fundamental design tool for creating an architectural design.
- The design operators can be used together with the modular operators.
- Common software design operators are:
  1. **Decomposition**
  2. **Replication**
  3. **Compression**
  4. **Abstraction**
  5. **Resource Sharing**
- There are various categories of design principles, such as for graphical *user interface (GUI) design, user interaction design, object-oriented design etc*.
- These statements of first principles form the design goals or objectives

Decomposition

- Decomposition is the operation of separating distinct functionality into distinct components that have well-defined interfaces.
  - part/whole
  - generalization/ specialization.
- The choice of where to draw the line between components is driven by what quality attributes you are trying to improve or emphasize.
- The following are the component decomposition techniques:
  - Identifying Functional Component
  - Composition/Aggregation
  - Component Communication

Replication

- Replication, also known as redundancy, is the operation of duplicating a component in order to enhance reliability and performance.
- There are two flavors of runtime replication:
  - Redundancy, where there are several identical copies of a component executing simultaneously.
  - N-version programming, where there are several different implementations of the same functionality

Compression

- Compression involves merging components into a single component or removing layers or interfaces between components.
- Composition involves coupling or combining two components to form a new system.
- Compression is intended to improve performance by eliminating a level of indirection.
- This may involve removing an interface between two components (effectively merging the two components into one), or it may involve removing some middle layer between two components so that the two components interact directly instead of through another layer.

Abstraction & Resource sharing

Abstraction:

- Abstraction hides information by introducing a semantically rich layer of services while simultaneously hiding the implementation details.

Resource sharing:

- Resource sharing is encapsulating data or services in order to share them among multiple independent client components.
- The result is enhanced integrability, portability, and modifiability.
- Resource sharing is useful when the resource itself may be scarce, such as during processing or threading. Persistent data is a common shared resource such as that stored in databases or directories.

Functional Design Strategies

- FDS suggest a functional decomposition strategy to achieve specific quality attributes.
- These strategies can help guide the architect as he or she decomposes a system based on functional and nonfunctional needs. Two strategies are presented here:
  - self-monitoring
  - recovery

Self Monitoring

- A system that is self-monitoring is able to detect certain types of failures and react to them appropriately, possibly without involving an operator or by notifying an operator about a specific condition (Bosch, 2000).
- The functionality added to the system is not in the application domain, and should be documented as such.
- There are two basic approaches to self-monitoring:
  - Process monitoring: A process monitor is a layer "above" the application or system that watches over the system.
  - Component monitoring: each component monitor is responsible for monitoring its own component and reporting issues to the next higher level component monitor.

Recovery

- Recovery functions are related to the quality attribute of recoverability.
- Recoverability is usually a quality that is introduced based on certain design decisions; it is typically not a product requirement based on the application domain.
- Recovery is related to reliability because it can affect the mean-time-to-repair (MTTR), which is a factor of the mean-time-between-failures (MTBF), a common measure of reliability.
- It may be necessary for a system or application to restore itself to some stable prior state by resetting some flags in a database or possibly restoring a database from a backup.
- Both are functions introduced to help the reliability of the application. Each component may need to address recovery differently and some not at all.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit V

Topics:

- Defining Architectural Styles
- Common Architectural Styles

Architectural Style

- The architectural style shows how do we organize our code, or how the system will look like from 10000 feet helicopter view to show the highest level of abstraction of our system design.
- Furthermore, when building the architectural style of our system we focus on layers and modules and how they are communicating with each other.

Common Arch. Styles

1. Dataflow systems
   - Pipes and filters
2. Repositories/ Data centered architecture
   - Databases
3. Call-and-return systems
   - Main program and subroutine
4. Object-oriented systems
5. Layered Architecture

Dataflow Systems

- Dataflow systems are characterized by how data moves through the system.
- Dataflow architectures have two or more data processing components that each transform input data into output data.
- The data processing components transform data in a sequential fashion where the output of an upstream processing component becomes the input of the next processing component.
- This example is called a pipeline because it is limited to a linear sequence of filters.

Dataflow Systems

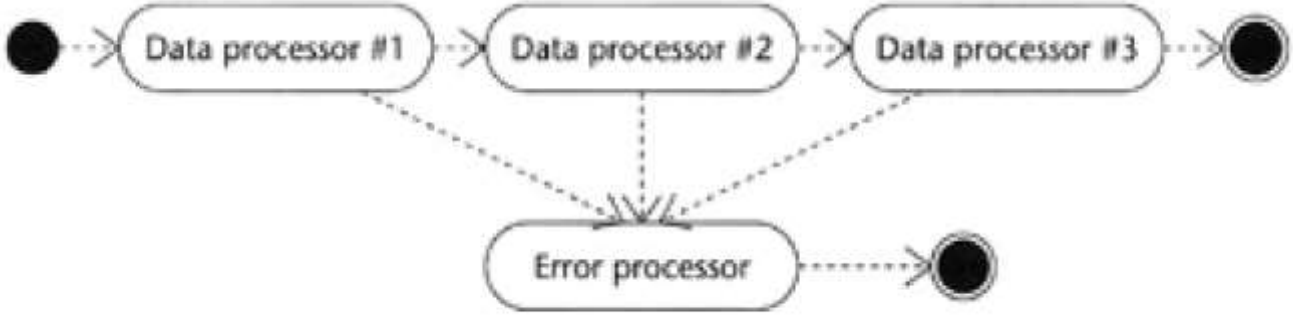


Figure 10.2: Pipes-and-filters architectural style.

Pipes and Filters

- It is common in a pipes-and-filters architecture that a processing component has two outputs, a standard output and an error output and a single input called standard input.
- Generically, the input and output mechanisms for a given processing element are called ports.
- Thus a typical filter has three ports.

Data Centered

- In data-centered architecture, the data is centralized and accessed frequently by other components, which modify data. The main purpose of this style is to achieve integrality of data. Data-centered architecture consists of different components that communicate through shared data repositories. The components access a shared data structure and are relatively independent, in that, they interact only through the data store.
- The most well-known examples of the data-centered architecture is a database architecture, in which the common database schema is created with data definition protocol – for example, a set of related tables with fields and data types in an RDBMS.

Data Centered types

The flow of control differentiates the architecture into two categories −

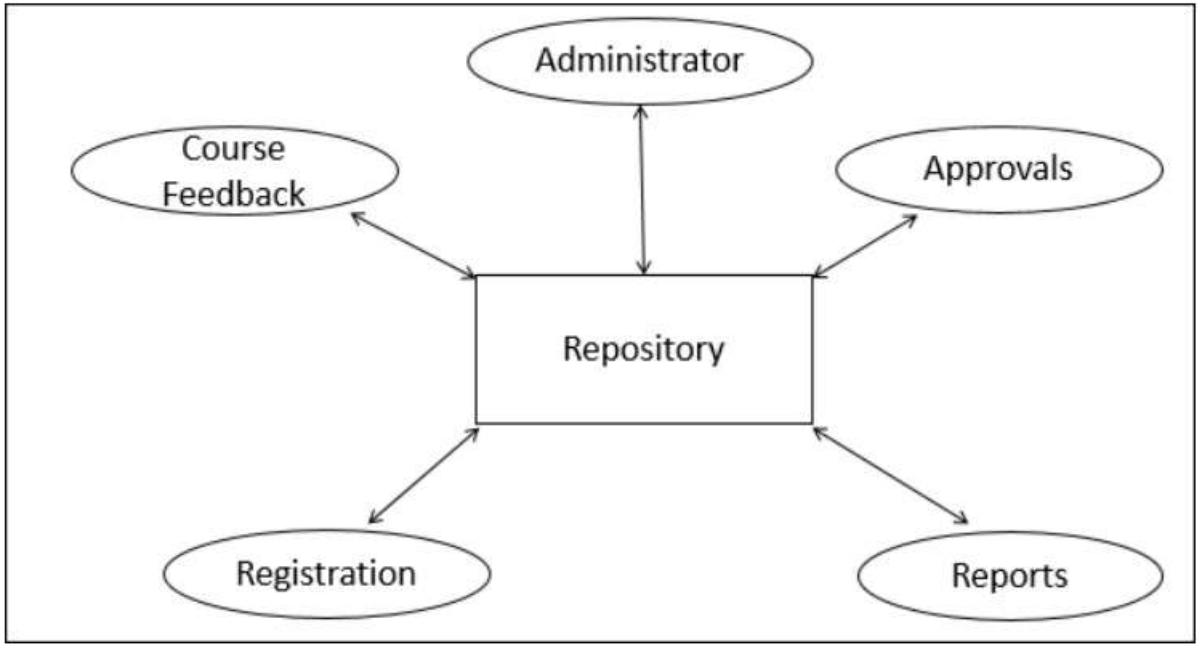
- Repository Architecture Style

In Repository Architecture Style, the data store is passive and the clients (software components or agents) of the data store are active, which control the logic flow. The participating components check the data-store for changes.

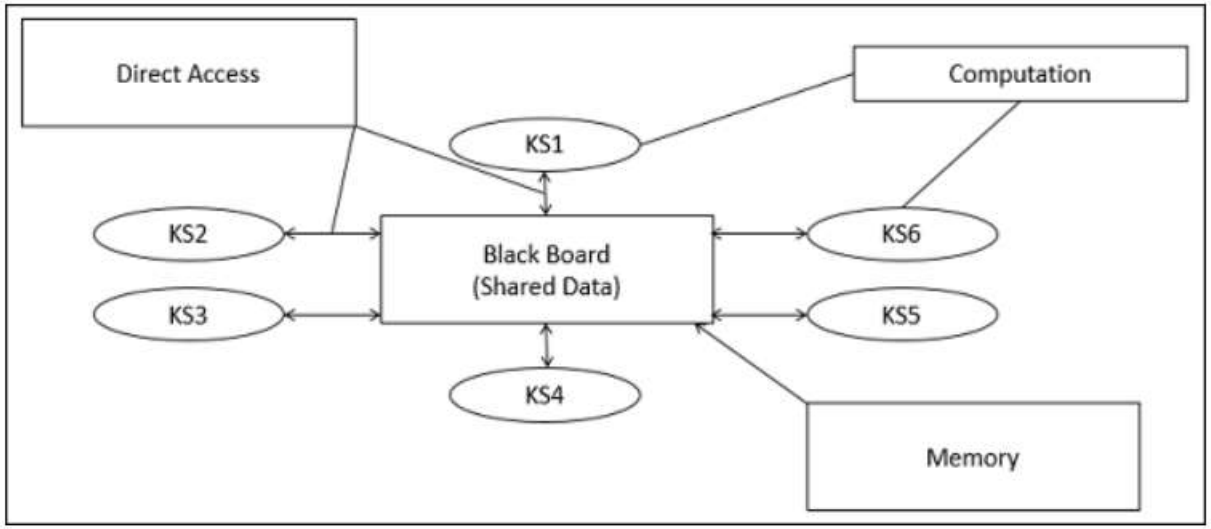
- Blackboard Architecture Style

In Blackboard Architecture Style, the data store is active and its clients are passive. Therefore the logical flow is determined by the current data status in data store. It has a blackboard component, acting as a central data repository, and an internal representation is built and acted upon by different computational elements.

Repository


Administrator
Course Feedback
Approvals
Repository
Registration
Reports

Blackboard

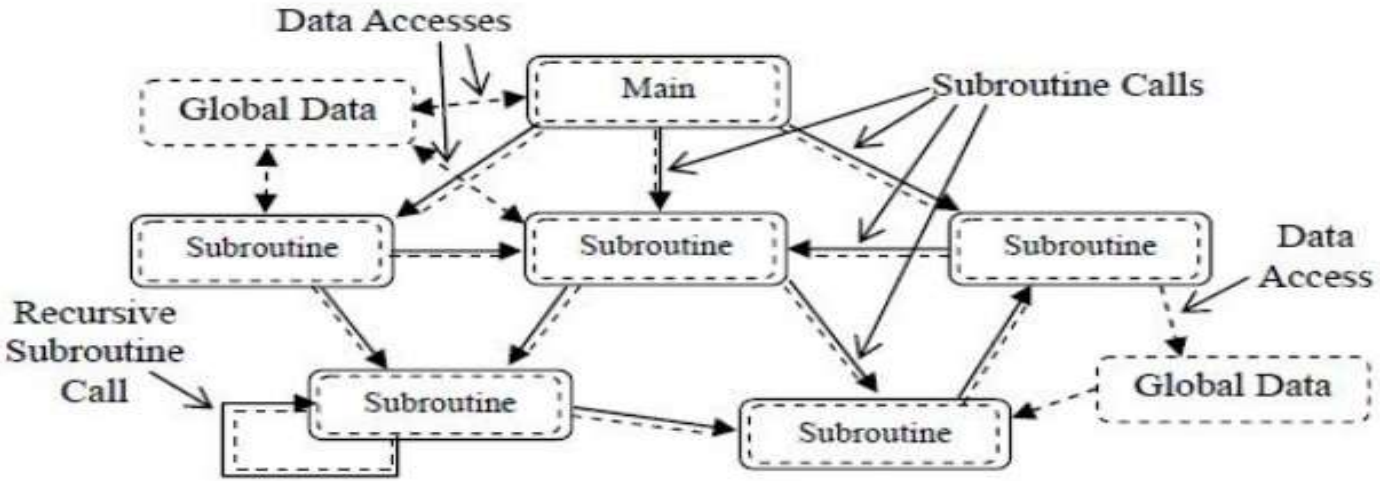

Direct Access
Computation
KS1
KS2
Black Board
(Shared Data)
KS6
KS3
KS5
KS4
Memory

Call-and-return systems

- Call-and-return systems are characterized by an activation model that involves a main thread of control that performs operation invocations.
- The classic system architecture is the main program and subroutine.
- This architectural style enables a software designer to achieve a program structure that is relatively easy to modify and scale.

Call-and-return systems

Structure of call and return architectures



Call-and-return systems

1. Main program or subprogram architecture

- The program is divided into smaller pieces hierarchically.
- The main program invokes many of program components in the hierarchy that program components are divided into subprogram.

2. Remote procedure call architecture

- The main program or subprogram components are distributed in network of multiple computers.
- The main aim is to increase the performance.

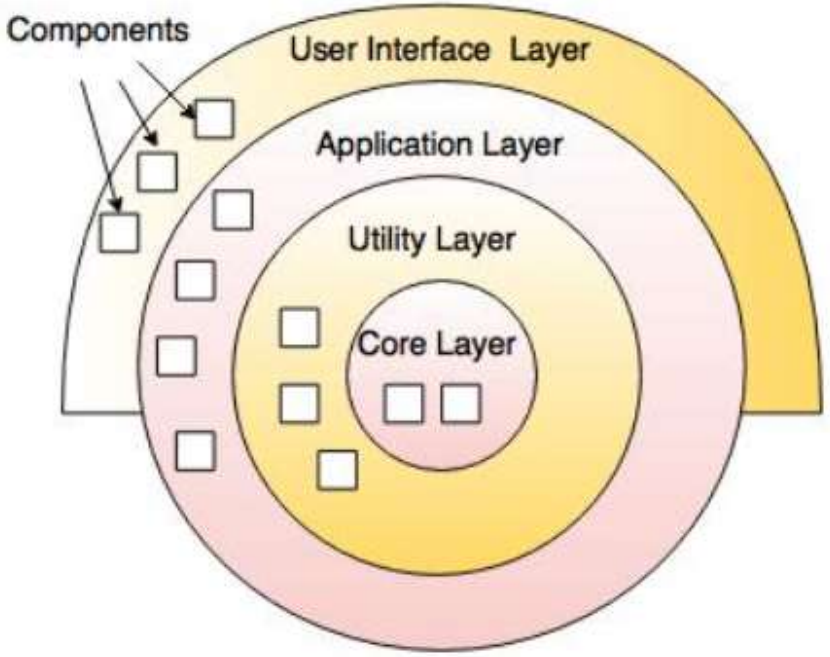
Object Oriented Systems

- The components of a system encapsulate data and the operations that must be applied to manipulation the data. Communication and coordination between components is accomplished via message passing.
- Basic features include:
  - Encapsulation
  - Information hiding
  - Inheritance
  - Polymorphism
  - Message passing

Layered Architecture

- The layered architecture style is one ofthe most common architectural styles. The idea behind Layered Architecture is that modules or components with similarfunctionalities are organized into horizontal layers. As a result, each layer performs a specific role within the application.
- The layered architecture style does not have a restriction on the number oflayers that the application can have, as the purpose is to have layers that promote the concept ofseparation of concerns. The layered architecture style abstracts the view ofthe system as a whole while providing enough detail to understand the roles and responsibilities ofindividual layers and the relationship between them.

Layered Architecture



Layered Architecture

Presentation tier

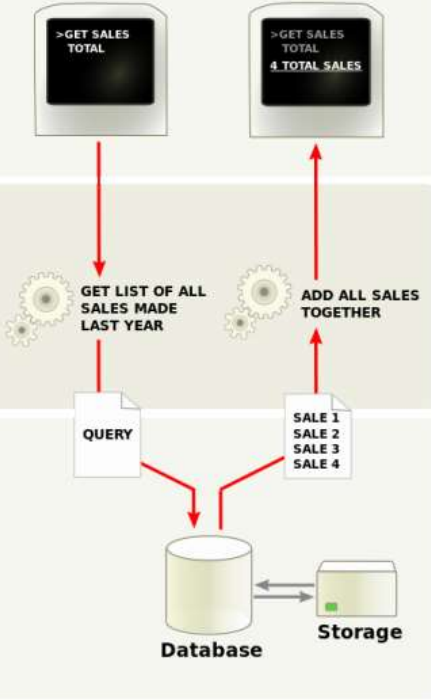
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

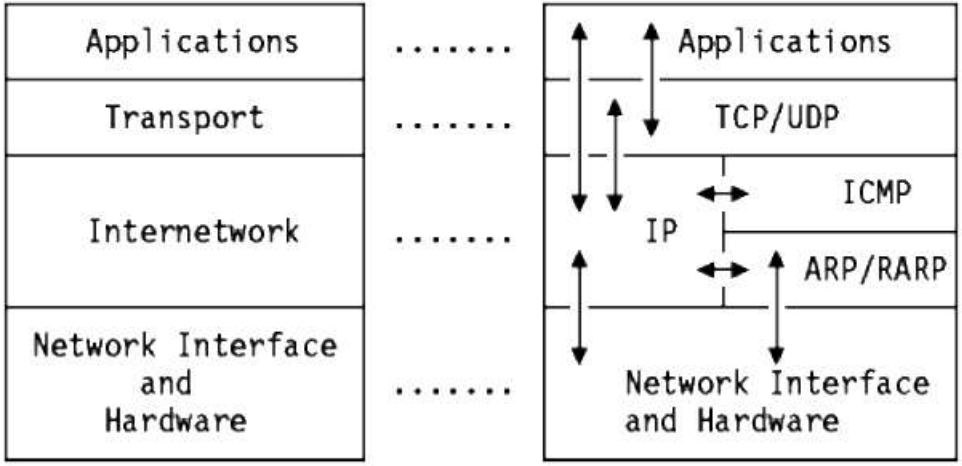
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



Ex: Protocol Stack



THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit V

Topics:

- Defining Architectural Patterns
- Common Architectural Patterns

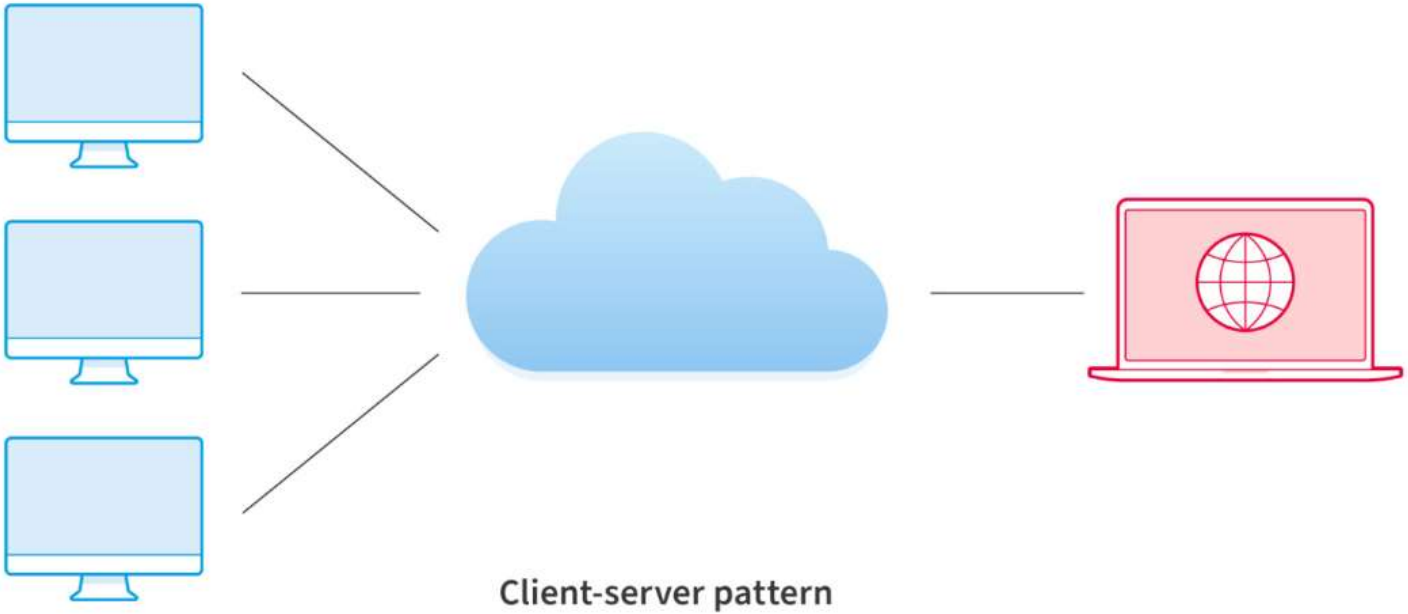
Architectural Patterns

- An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.

Common Arch. Patterns

1. Client-Server
2. Master-Slave
3. Broker
4. Peer to Peer
5. Model View Controller
6. Event-Bus

Client Server



Client-server pattern

Client Server

“Client-server software architecture pattern” is the one, where there are 2 entities. It has a set of clients and a server. The following are key characteristics of this pattern:

- Client components send requests to the server, which processes them and responds back.
- When a server accepts a request from a client, it opens a connection with the client over a specific protocol.
- Servers can be stateful or stateless. A stateful server can receive multiple requests from clients. It maintains a record of requests from the client, and this record is called a ‘session’.
- Email applications are good examples of this pattern.

Client Server

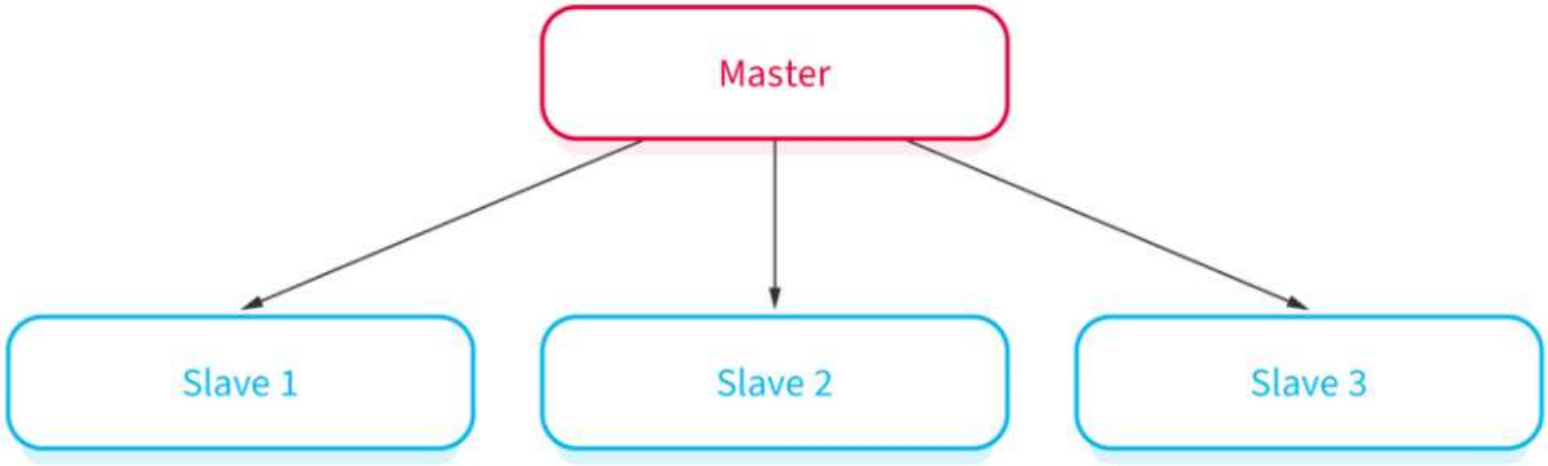
Advantages:

- Clients access data from a server using authorized access, which improves the sharing of data.
- Accessing a service is via a ‘user interface’ (UI), therefore, there’s no need to run terminal sessions or command prompts.
- Client-server applications can be built irrespective of the platform or technology stack.
- This is a distributed model with specific responsibilities for each component, which makes maintenance easier.

Disadvantages:

- The server can be overloaded when there are too many requests.
- A central server to support multiple clients represents a ‘single point of failure’.

Master-Slave



Master-slave pattern

Master-Slave

- “Master-slave architecture pattern” is useful when clients make multiple instances of the same request. The requests need simultaneous handling. Following are its’ key characteristics:
- The master launches slaves when it receives simultaneous requests.
- The slaves work in parallel, and the operation is complete only when all slaves complete processing their respective requests.

Master-Slave

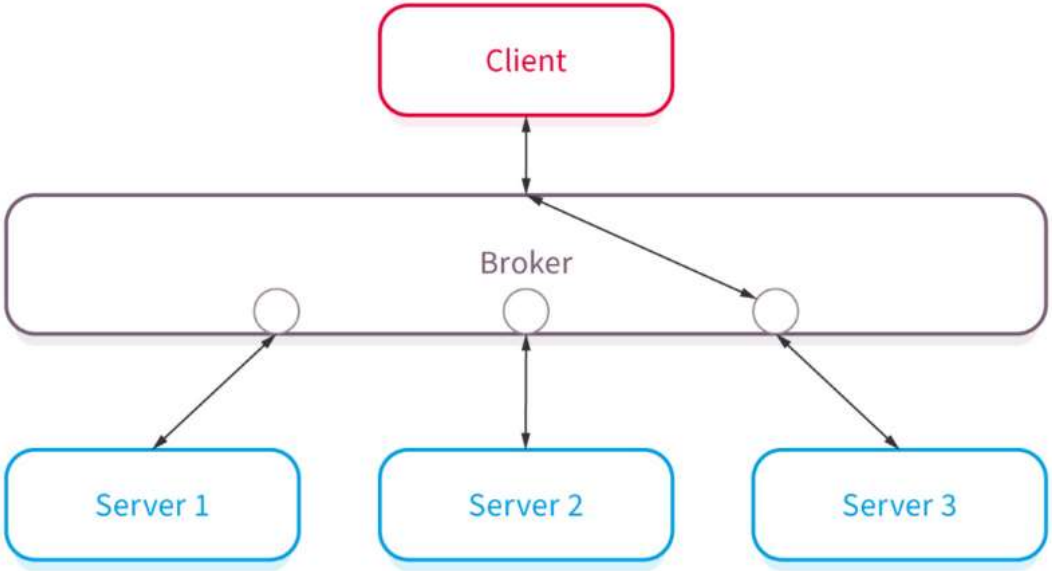
Advantages:

- Applications read from slaves without any impact on the master.
- Taking a slave offline and the later synchronization with the master requires no downtime.

Disadvantage:

- This pattern doesn’t support automated fail-over systems since a slave needs to be manually promoted to a master if the original master fails.

Broker

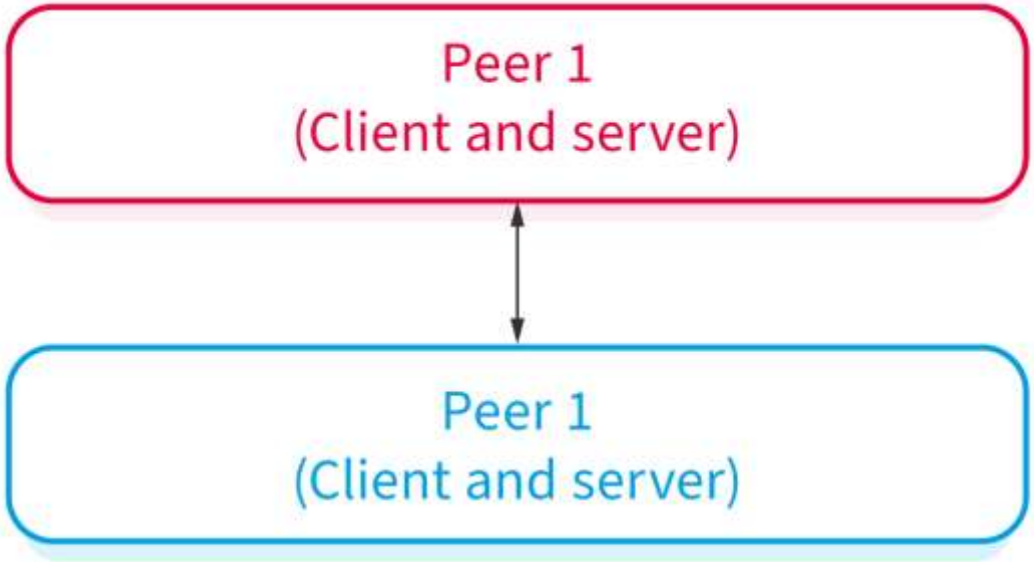


Broker pattern

Broker

- Client and servers can be totally of different architectures.
- Broker helps to coordinate between such heterogenous architectures.
- A broker component coordinates requests and responses between clients and servers.
- The broker has the details of the servers and the individual services they provide.
- The main components of the broker architectural pattern are clients, servers, and brokers. It also has bridges and proxies for clients and servers.
- Clients send requests, and the broker finds the right server to route the request to.
- It also sends the responses back to the clients.

Peer to Peer

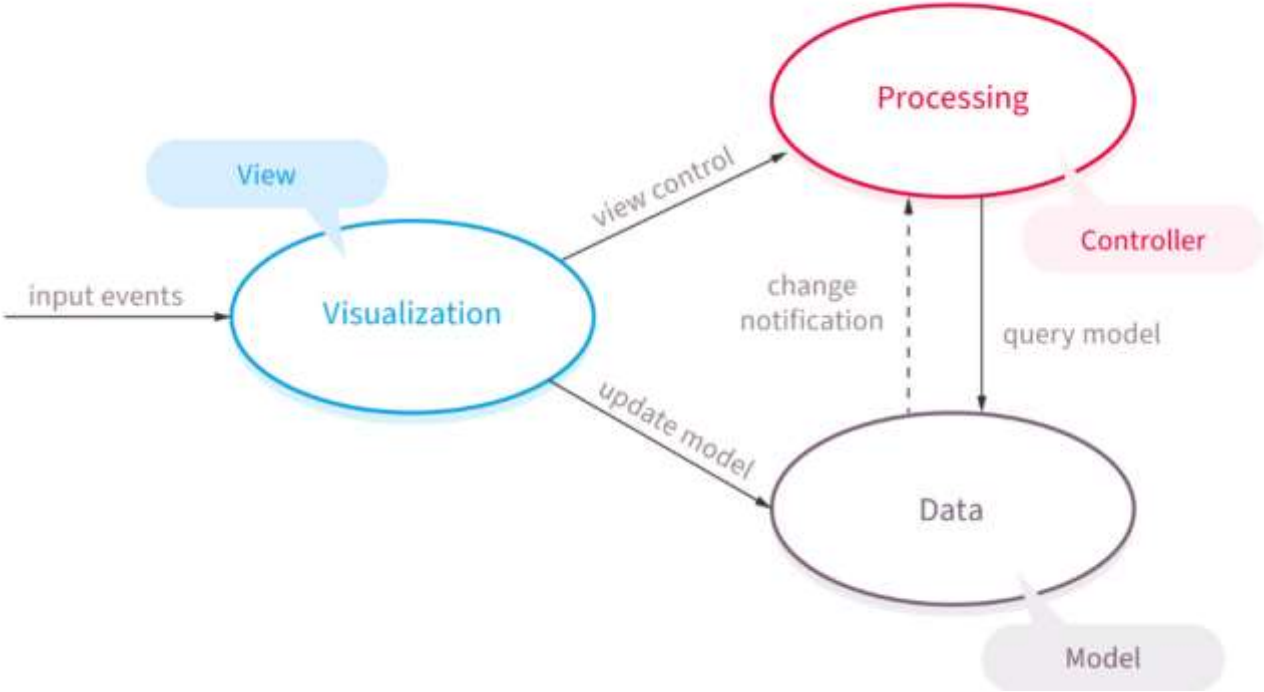


Peer-to-peer pattern

Peer to Peer

- “Peer-to-peer (P2P) pattern” is markedly different from the client-server pattern since each computer on the network has the same authority. Key characteristics of the P2P pattern are as follows:
- There isn’t one central server, with each node having equal capabilities.
- Each computer can function as a client or a server.
- When more computers join the network, the overall capacity of the network increases.

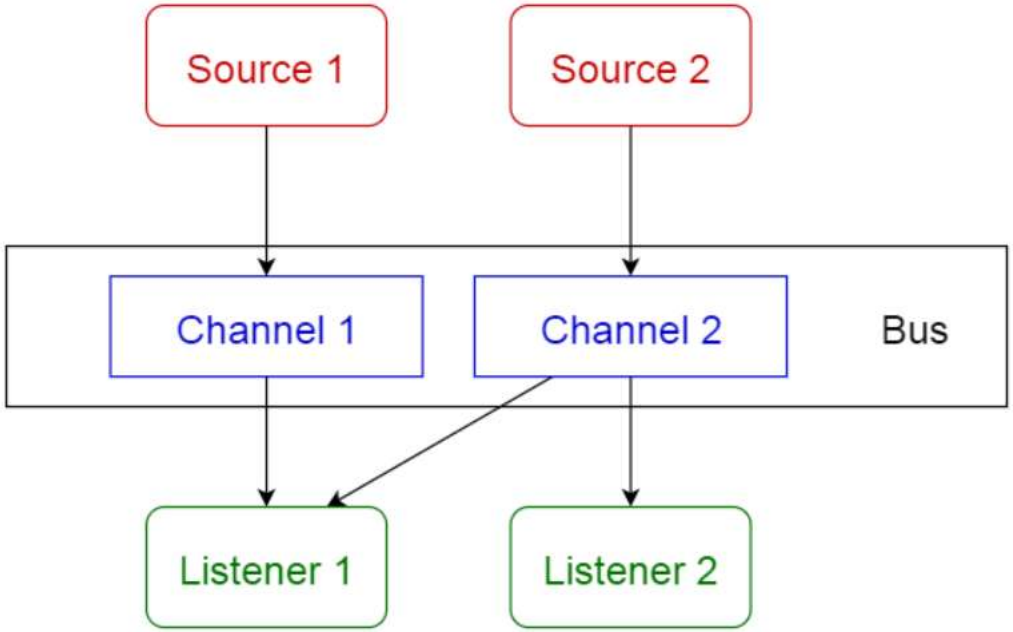
MVC


Processing
View
view control
Controller
input events
Visualization
change
notification
query model
update model
Data
Model

MVC

- “Model-View-Controller (MVC) architecture pattern” involves separating an applications’ data model, presentation layer, and control aspects. Following are its’ characteristics:
- There are three building blocks here, namely, model, view, and controller.
- The application data resides in the model.
- Users see the application data through the view, however, the view can’t influence what the user will do with the data.
- The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action is typically a method call to the model. The response is shown in the view.

Event-Bus



Event-bus pattern

Event-Bus

- This pattern primarily deals with events and has 4 major components; event source, event listener, channel and event bus.
- Sources publish messages to particular channels on an event bus.
- Listeners subscribe to particular channels.
- Listeners are notified of messages that are published to a channel to which they have subscribed before.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit V

Topics:

- Introduction to Metamodels
- Understanding Metamodels
- Applying Reference Models
- Seeheim Model
- Arch/ Slinky Model

Metamodels

- Metamodels are literally "models of models.“
- A metamodel is like the grammar of a formal language.
- Models and metamodels form a hierarchy of models. Each higher-level layer (the meta layer) describes the structure (syntax) of the next lower-level layer.
- Metamodels are fairly abstract tools for creating, understanding and evaluating models.
- The relationship between metamodels, models, and data corresponds to the three layers of knowledge representation (**ontology layer, domain layer, and technology layer**).

Example

❖ **For example,**

- The Unified Modeling Language (UML) metamodel describes the syntax of diagrams expressed in UML.
- Every model has a metamodel that describes it, although the metamodel may be implicit.
- It is like the grammar of a formal language, and a model expressed in that grammar is a sentence or set of sentences expressed in the language of the metamodel.

Understanding Metamodels

- Models and metamodels form a **hierarchy of models**.
- Each higher-level layer (the meta layer) describes the **structure (syntax) of the next lower-level layer.**
- The term metamodel can be considered a **role** that a model assumes with respect to other models.
- Another way to think of a metamodel is as a **set of instructions for creating an instance of class models**.
- Metamodels are like **design languages** they are a domain-specific, self-contained design ontology.

3 Layer Model of Knowledge Representation

1. Ontology layer
2. Domain layer
3. Technology layer

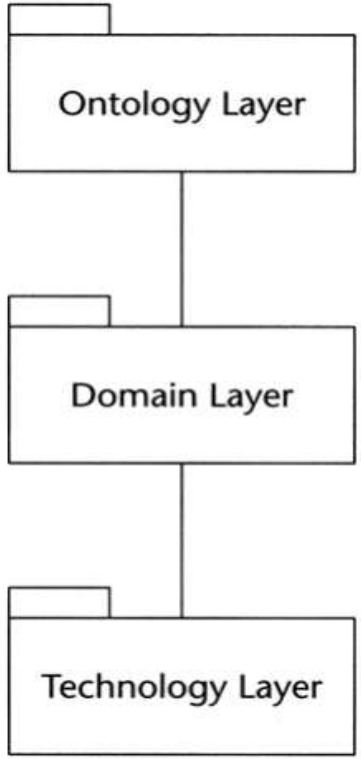


Figure 11.2: Three-layer model of knowledge representation.

Ontology Layer

- Ontology means something that exists for a fact. Here, it means the objects that are existing at the core.
- The ontology layer contains core concepts or abstractions and their relationships.
- An example of an ontology for software design is three core business abstractions representing this ontological model using UML meta classes.

Domain Layer

- The domain layer contains domain models, such as models of specific business domains. These domain models are described in terms of ontology abstractions.
- The domain layer contains domain models, such as models of specific business domains. These domain models are described in terms of ontology abstractions.
- A domain model that conforms to the OPR **i.e organization, process, and resource** describe any business application (and those that are not traditionally considered to be business applications)

Example

In this model, Enterprise X (an organization) manages the product development process, which consumes money (a resource) and uses engineers (a resource) to produce Product Y (a resource).

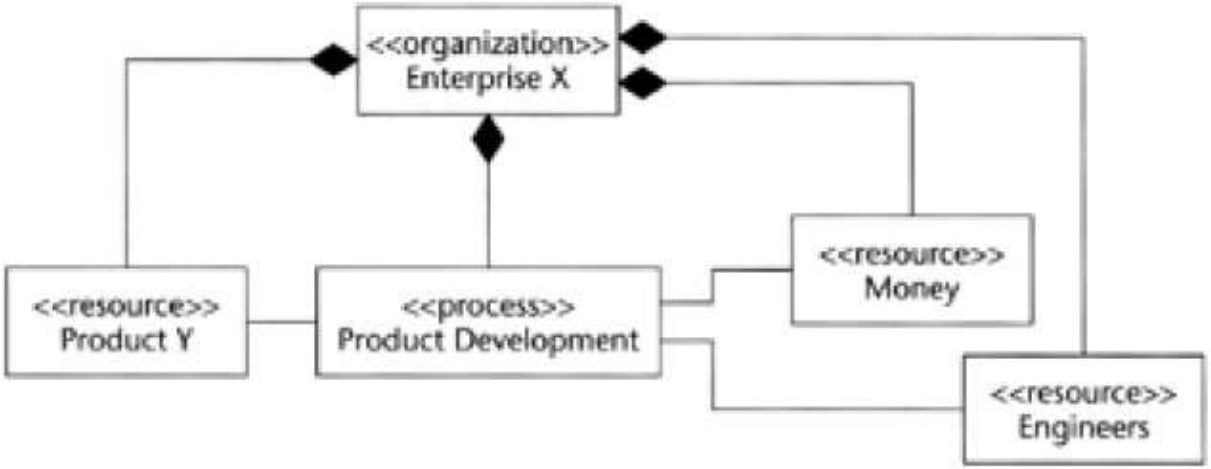


Figure 11.4: Domain model of a business application.

Technology Layer

- The technology layer is composed of models that are the technology projections of the domain layer.
- An example of a technology model is represented in Figure 11.5. This example is a simplified model relying on the class names to suggest the technology involved in each object, as well as the domain object being represented.
- The mapping of ontology abstractions is represented using additional boxes and text around groups of classes.

Technology Layer

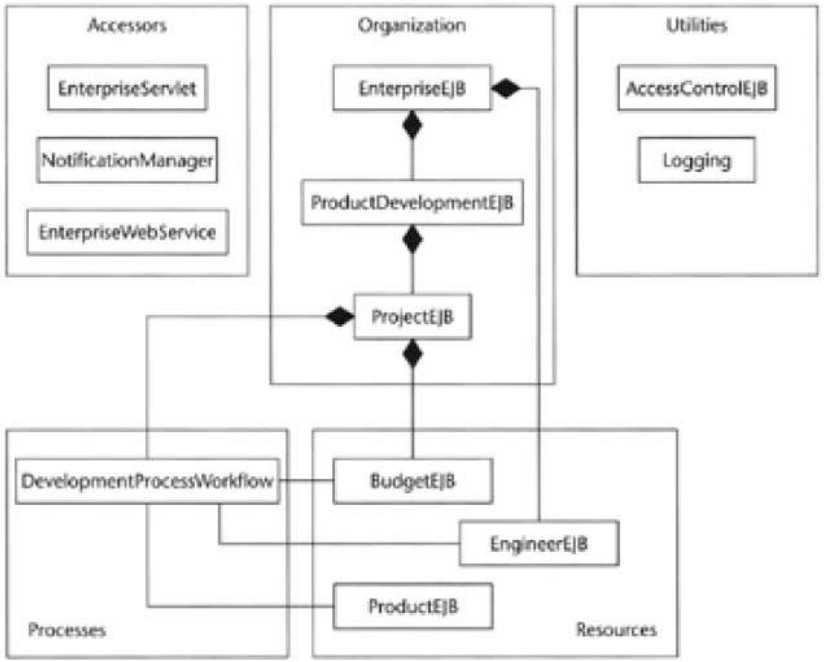


Figure 11.5: Technology model of a business application.

Applying Reference Models

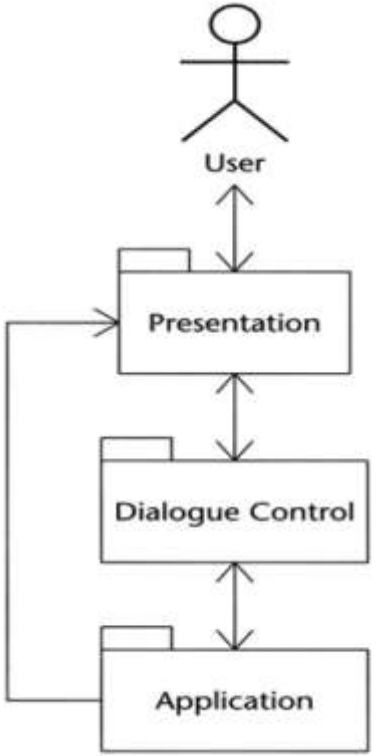
- A reference model abstracts software components and expresses the system as connectors and components.
- Reference models are common in mature domains such as HCI, compiler design, and database system design.
- A reference model may be created as a result of domain analysis, whereby a problem domain is analyzed and as a solution, a model is created to solve the issues which might occur later as well.

Types of reference models

- The Seeheim metamodel and the arch/slinky metamodel are reference models for the architecture of interactive software applications that have a graphical user interface (GUI) element.
- Both models specify a form for designing an application and are based on several years of research in the field of HCI. In order to understand these models, you must mentally separate the concept of a user interface from an application.

Type 1: Seeheim Model

- First formulated in 1985 at Seeheim, Germany.
- The presentation part specifies the layout of the input and output, basically the GUI of the application.
- The dialogue part specifies the logics and functionality of these input- and output-element, basically communication between Presentation and Application.
- Application provides a linking between the backend and the other layers.



Arch/Slinky Model

- Arch/slinky is another metamodel for interactive applications that evolved from the Seeheim model. Arch/slinky, like the Seeheim model, separates the user interface (presentation) and user interaction logic (dialogue control) from the application functional core. The arch/slinky reference model is composed of five elements:
- Presentation
- Virtual toolkit
- Dialogue control
- Virtual application
- Application

Arch/slinky model

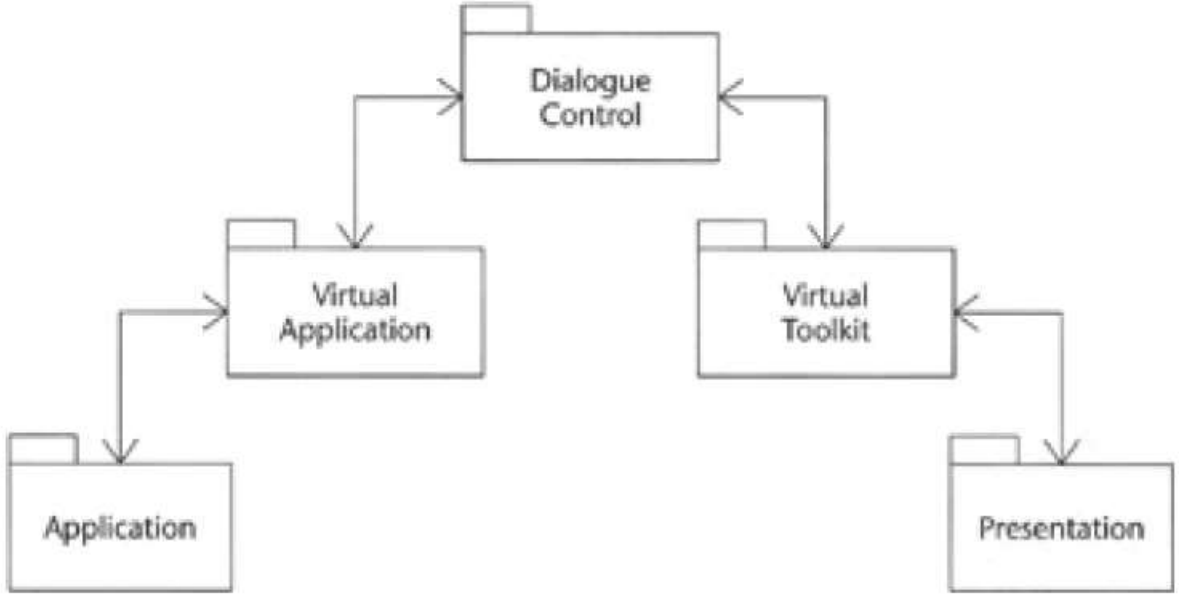


Figure 11.9: Arch/slinky metamodel.

Arch/Slinky Model

- The term arch in arch/slinky is based on a common view of the model as an arch.
- The term slinky comes from the fact that an architecture that is based on this model does not necessarily explicitly contain the five elements. Some elements may be compressed into a single module.

THANK YOU

Software Architecture (SEM VII)

Presented by: Ms. Drashti Shrimal

Unit VI

Topics:

- Intro to Framework
- Framework Viewpoints
- Architectural Framework Goals
- Methodology and Framework
- 4+1 View model

Intro to Framework

- The term framework is used in many contexts in software development. A framework, in general, is a structure composed of parts that together support a structure.
- Architecture frameworks are frameworks for architecture specifications.You use it as a template for creating an architecture specification.
- It is possible, however, to reuse portions of architecture specifications.

Arch Design specification

What is the architecture design specification?

- An architectural design specification is a technical document that describes how a software system is to be developed to achieve the goals described in the requirements.
- It's analogous to the house plans.

Framework Viewpoints

Frameworks typically include the following types of viewpoints:

- *Processing* (for example, functional or behavioral requirements and use cases)
- *Information* (for example, object models, entity relationship diagrams, and data flow diagrams)
- *Structure* (for example, component diagrams depicting clients, servers, applications, and databases and their interconnections)

Architectural Framework Goals

- No software architecture framework currently satisfies the IEEE 1471 recommendations, which makes comparing them difficult.

The general goals are:

1. Codify best practices for architectural description (to improve the state of the practice).
2. Ensure that the framework sponsors receive architectural information in the format they want. (Basically, client needs to be satisfied)
3. Facilitate architecture assessment. (Good evaluation of arch.)
4. Improve the productivity of software development teams by using standardized means for design representation.
5. Improve interoperability of information systems.

Methodology and Framework

- A methodology is a way to systematically solve a problem. It is a combination of two things together – the methods you've chosen to get to a desired outcome and the logic behind those methods.
- On the other hand, a framework is a structured approach to problem solving. Frameworks provide the structural components you need to implement a model. It is a skeletal structure around which something can be built.
- A framework is a collection of reusable components that offer a consultant shortcuts to avoid developing a structure from scratch, each time they start an engagement.

4+1 View Model

- The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a multiple view model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.
- It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views.

4+1 View Model

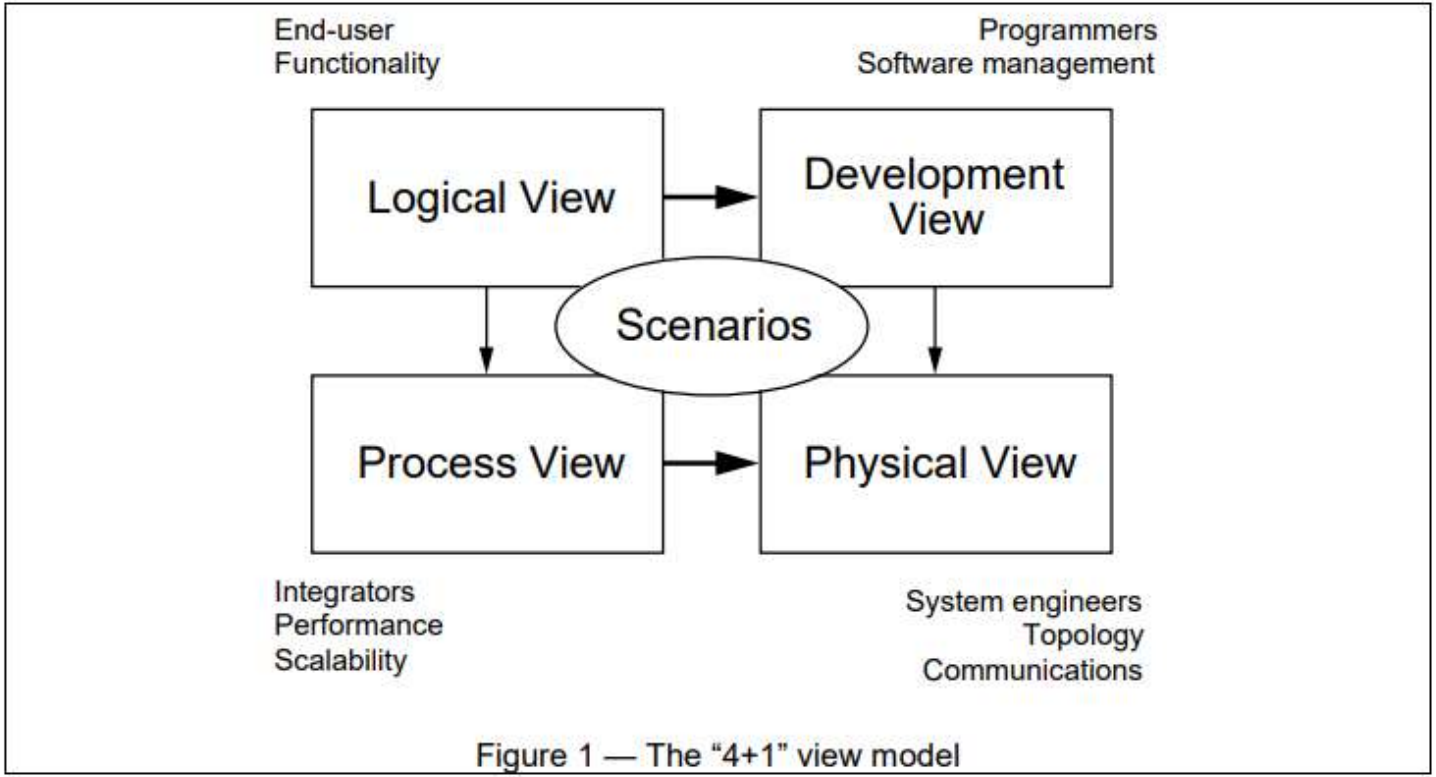


Figure 1 — The “4+1” view model

4+1 View Model

1. Logical View:

- The logical architecture primarily supports the functional requirements—what the system should provide in terms of services to its users.
- The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of objects or object classes. They exploit the principles of abstraction, encapsulation, and inheritance.
- Examples of Logical view models/diagrams: Sequence and Class diagrams.

4+1 View Model

2. Development View:

- The development architecture focuses on the actual software module organization on the software development environment.
- The software is packaged in small chunks—program libraries, or subsystems—that can be developed by one or a small number of developers.
- Provides a view from developers perspective which states where all code and its modules would be placed.
- Examples: Component and Package diagram.

4+1 View Model

3. Process View:

- The process architecture takes into account some non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, of system's integrity, of fault-tolerance.
- It provides a view from tasks' perspective.
- Checks the scalability and performance of system.
- Example: Activity Diagram

4+1 View Model

4. Physical View:

- It describes the mapping of software onto hardware and reflects its distributed aspect.
- It looks after the deployment of the system, tools and environment in which the product is installed.
- It takes a system engineer’s point of view in consideration.
- Example: Deployment diagram.

4+1 View Model

+1 Scenarios:

- This view model can be extended by adding one more view called scenario view or use case view for end-users or customers of software systems.
- It is coherent with other four views and are utilized to illustrate the architecture serving as “plus one” view, (4+1) view model.

THANK YOU

Software Architecture (SEM VII)

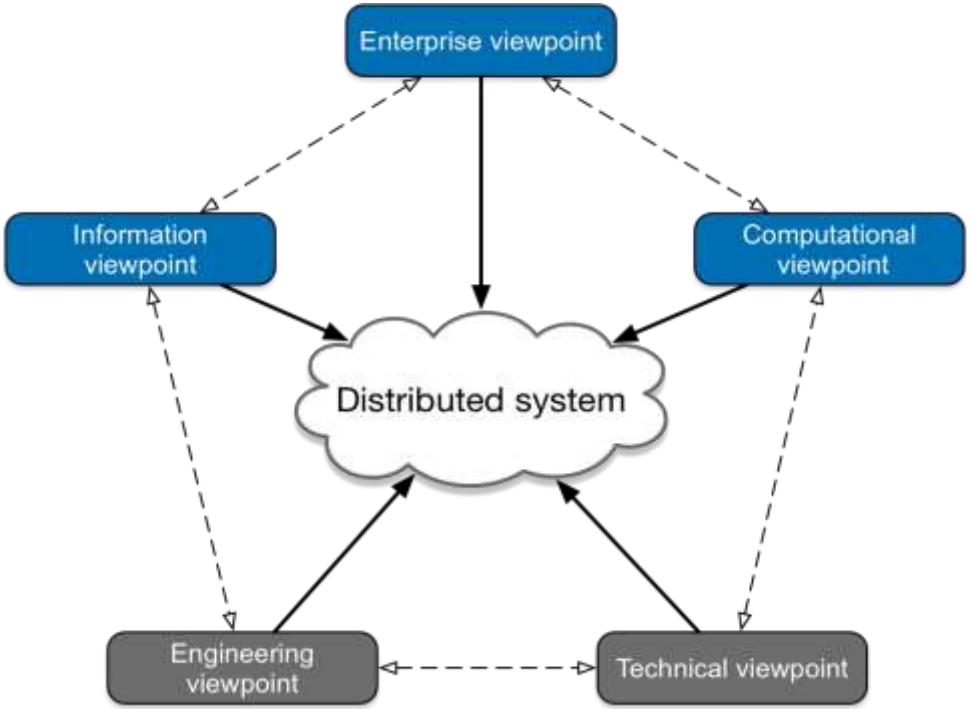
Presented by: Ms. Drashti Shrimal

Unit VI

Topics:

- Reference Model for Open Distributed Processing
- Importance of Assessing QA
- How to improve Quality?

RM-ODP


Enterprise viewpoint
Information viewpoint
Computational viewpoint
Distributed system
Engineering viewpoint
Technical viewpoint

RM-ODP

- The main goal of RM-ODP is to provide mechanisms for architecting distributed processing and information systems and to support enterprise application integration. RM-ODP supports these goals through five viewpoints, each of which has a precise viewpoint language and rules for mapping models across views.
- The views of a system form a succession of models from the abstract function and purpose of the system to the concrete implementation of the system. RM-ODP viewpoints separate the distributed nature of an application from its implementation, allowing the application to be designed without prematurely restricting the design to a particular middleware platform. Each of the five viewpoints addresses specific aspects of a distributed system

RM-ODP

The five viewpoints of RM-ODP are:

- Enterprise
- Information
- Computational
- Engineering
- Technology

Viewpoints

1. Enterprise Viewpoint:

- The enterprise viewpoint focuses on the purpose, scope, and policies of the system and provides a means of capturing system requirements.
- The enterprise viewpoint language is very expressive and can be used for creating business specifications as well as expressing software requirements.

Viewpoints

2. Information Viewpoint:

- The information viewpoint focuses on the semantics of information and information processing within the system.
- This viewpoint specifies a metamodel for representing functional requirements in terms of information objects.
- It resembles the 4+1 logical viewpoint in that it is used to specify a business information model that is implementation independent.

Viewpoints

2. Information Viewpoint:

- An information view defines the universe of discourse of the system: the information content and the information about the processing of the system, a logical representation of the data in the system, and the rules to be followed in the system, such as policies specified by the stakeholders.
- The information viewpoint is central to all the other viewpoints. Changes in an information view necessarily ripple through the other views.

Viewpoints

3. Computational Viewpoint:

- The computational viewpoint specifies a metamodel for representing the functional decomposition of the system as platform-independent distributed objects that interact at interfaces.
- A computational view specifies the system in terms of computational objects and their interactions. A computational view partitions the system into logical objects that perform the capabilities of the system and are capable of being distributed throughout the enterprise but does not specify how they are distributed.

Viewpoints

4. Engg. Viewpoint:

- The engineering viewpoint addresses the mechanisms and functions for supporting distributed object interactions and distribution transparency.
- An engineering view specifies the mechanisms for physical distribution to support the logical processing model of the computational view without specifying a particular technology or middleware platform.

Viewpoints

5. Technology Viewpoints:

- The technology viewpoint specifies a language for representing the implementation of a system.

Separating the technology and engineering viewpoints allows the architect to focus on distribution aspects without prematurely committing to a particular middleware platform or influencing the architecture of the application by assuming technologies. All of the views prior to the technology view transcend implementation and are reusable even when technologies change.

Importance of Assessing QA

- A key software engineering principle is that quality cannot be tested into the product; it has to be designed into it.
- Modifying existing code is much more time-intensive and expensive than modifying design specifications.
- Therefore, as part of a systematic design process, you should perform assessments of the design.

How to improve Quality?

How do we assess the quality of a system's architecture? There are two aspects to this question:

- How to evaluate an architectural description (architectural assessment or analysis)?
- How to conduct an architectural assessment (an architectural evaluation)?

How to improve Quality?

- The following are some of the activities and techniques:
  - Systematic Design Process
  - Understand the Right Problem
  - Differentiating Design and Requirements
  - Assessing Software Architectures (*questioning, measuring*)
  - Scenarios: Reifying Nonfunctional Requirements (*Utility tree*)
  - The Role of the Architectural Description

Systematic Design Process

- In a systematic design process, a designer is searching for a solution. The designer can go down many paths.
- Each branch in the path increases the number of potential solutions that may be discovered.
- These solutions are referred to as a solution field. By simultaneously considering multiple design variations, we increase our chances of discovering a suitable solution.
- However, a solution field can become prohibitively large producing a negative effect on the process. Thus, combining design assessment with solution searching helps a designer manage the size of the solution field.

Understand the Right Problem

- The software architect, in many cases, performs the role traditionally assumed by a systems analyst.
- As a software architect, you should not assume that the list of requirements you have been given is the best possible list. Most of the time the requirements are vague, complex, and not stated in a way that identifies the true problem being solved.
- You must analyze these requirements not only to understand their interdependencies and hidden structures, but also to understand the problems that really need to be solved.

Differentiating Design and Requirements

- Recall that design begins earlier than we usually think. Whenever a feature is specified in a requirements document, there is design.
- Most people don't handle abstraction well, even writers of requirements.
- The problem is that when these people are writing up requirements, they tend to shift focus from abstract problem statements to concrete features. Once these feature requirements make it into a system, they become hard to eradicate later.

Assessing Software Architectures

- Analyzing a model is more difficult than analyzing a working system. It is not always possible to evaluate a software design to understand a single quality attribute. Some quality attributes interact with each other, such as modifiability and performance.
- A common technique for making a system modifiable is to introduce layers of modules. However, layers can result in computational overhead. If each layer is responsible for wrapping and unwrapping information structures, then this can result in a lot of additional data creation and parsing.
- If a layer can be removed, then there are fewer transformations occurring. Some systems allow layers to be bridged in order to provide some performance improvements, but at the expense of modifiability

Scenarios: Reifying Nonfunctional Requirements

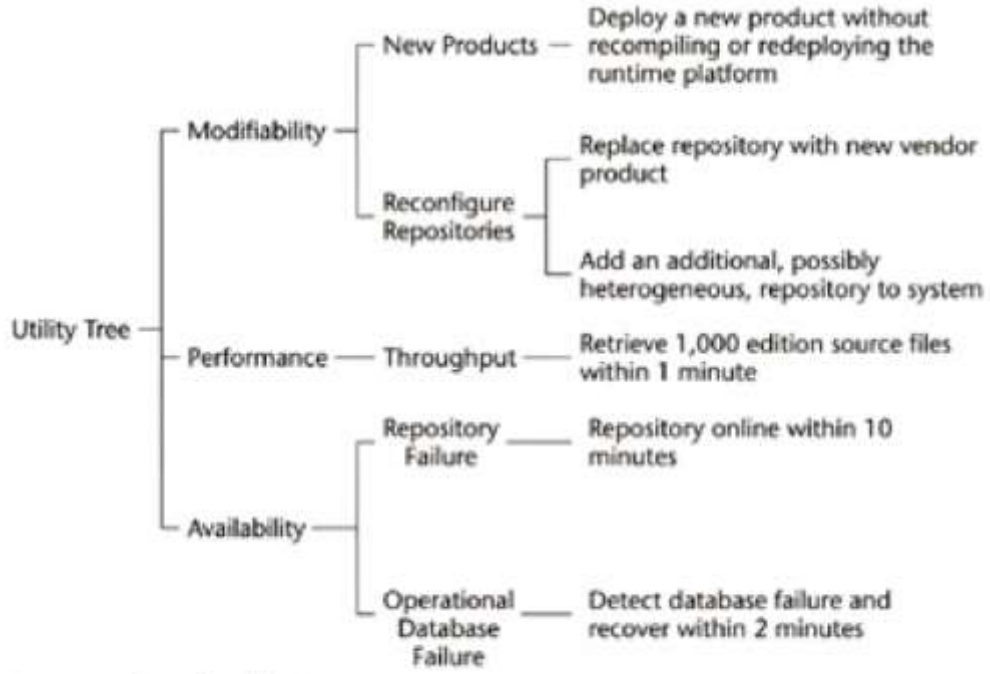


Figure 14.2: Example utility tree.

The Role of the Architectural Description

- The architectural design models that comprise the architectural description can have a significant impact on the ability of an architect to analyze an architecture. It is very difficult to analyze an architecture that is not actually written down.
- The types of models used are relevant in the analysis. In some cases, specific model need to be created for a single quality attribute. For example, in order to assess the architecture for performance, various execution models need to be created. Recall that the architectural description is composed of views that address different concerns. These concerns include specific quality requirements.

THANK YOU