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

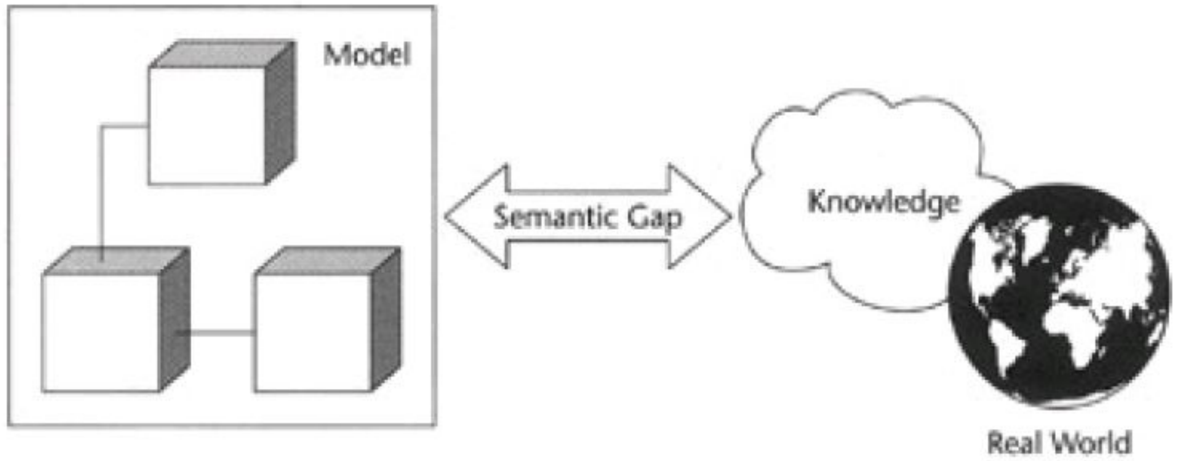


Figure 6.1: Models are representations of knowledge.

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.

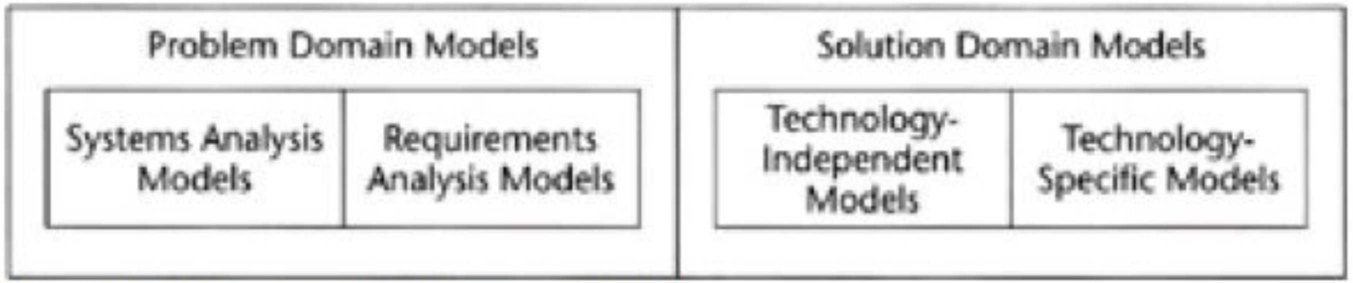
Uses of Models

- 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



Problem Domain Models
Systems Analysis Models
Requirements Analysis Models
Solution Domain Models
Technology-Independent Models
Technology-Specific Models

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

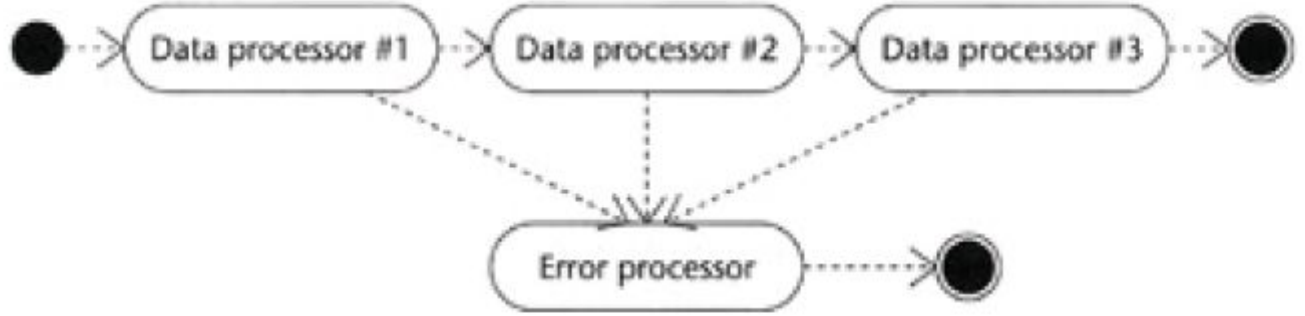


Figure 10.2: Pipes-and-filters architectural style.

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 −

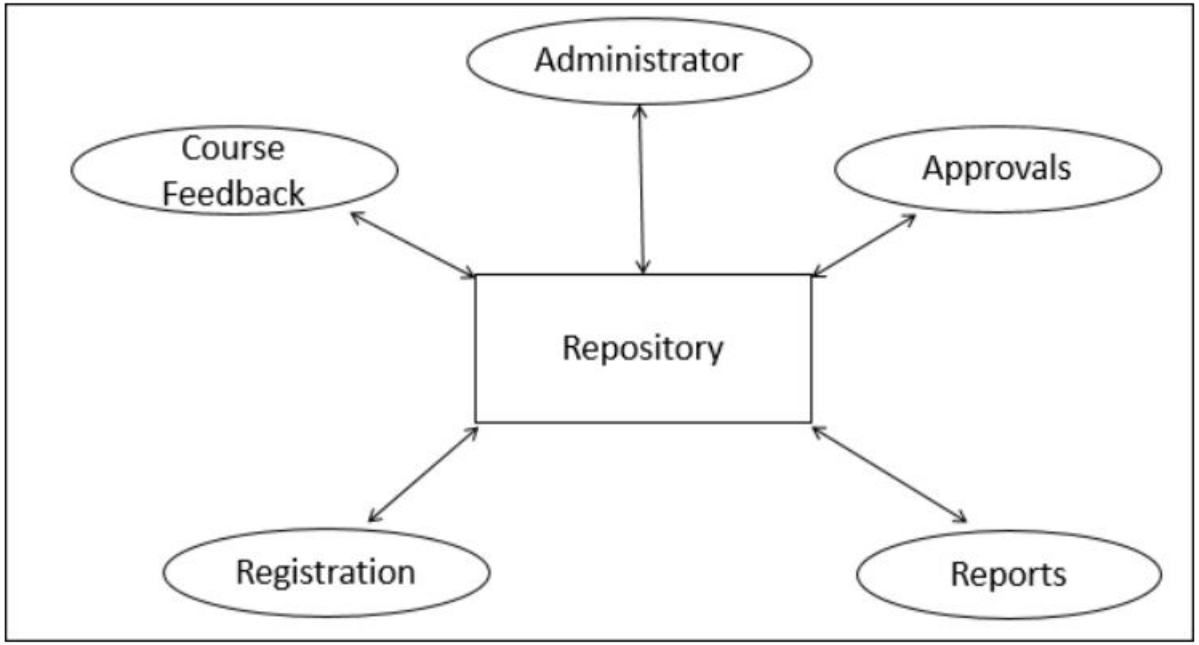
- Repository Architecture Style

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.

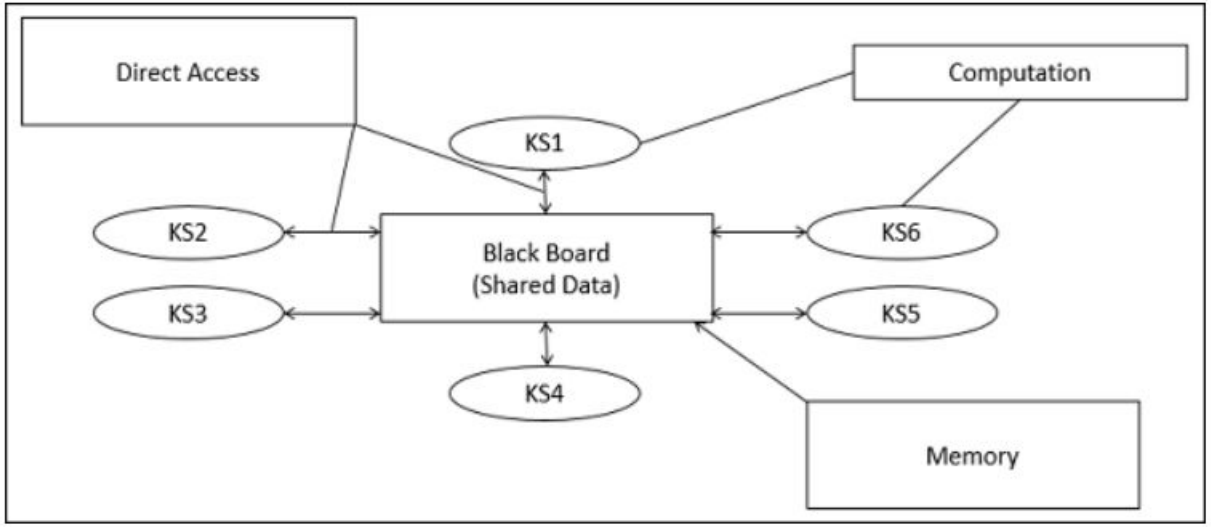
- Blackboard Architecture Style

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

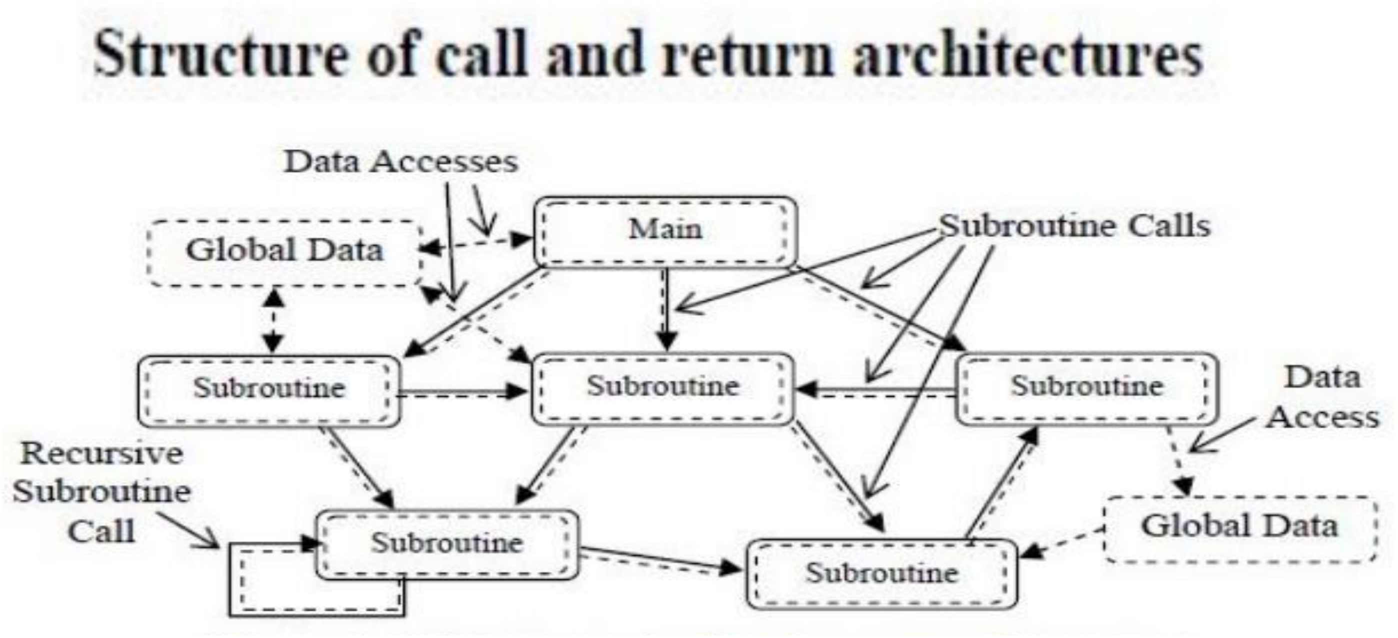
Blackboard


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



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.

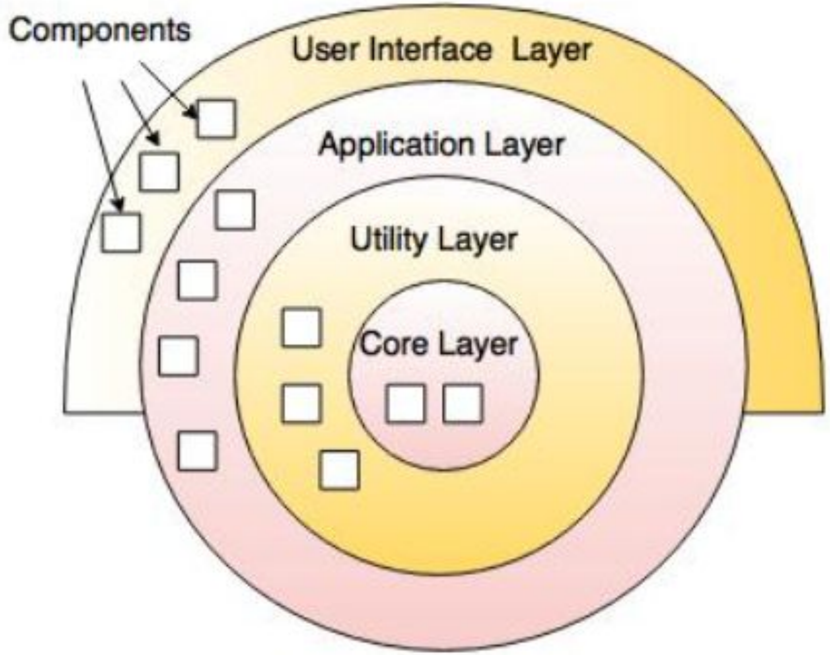
Object Oriented Systems

- 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

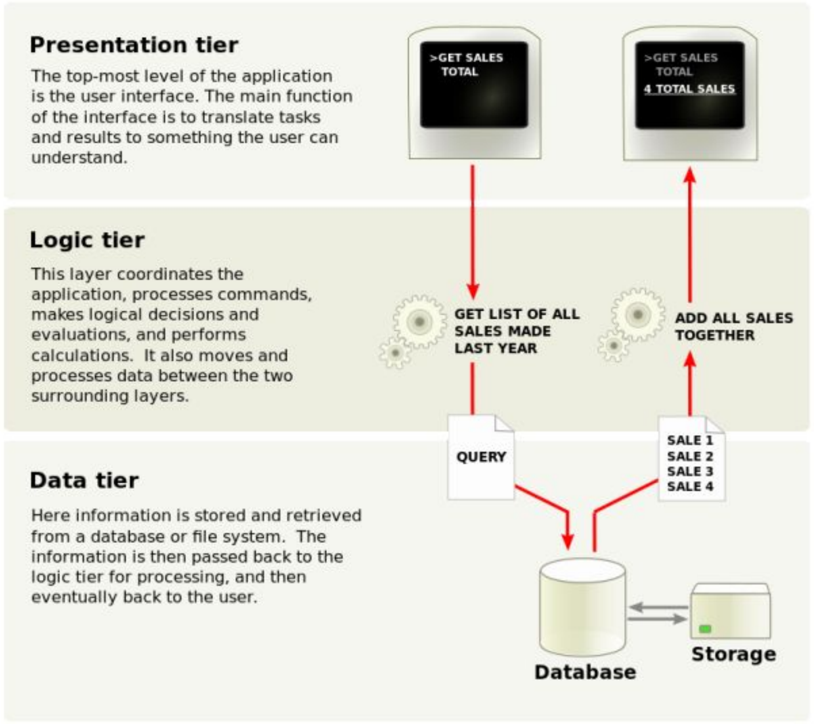
Layered Architecture

- 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.

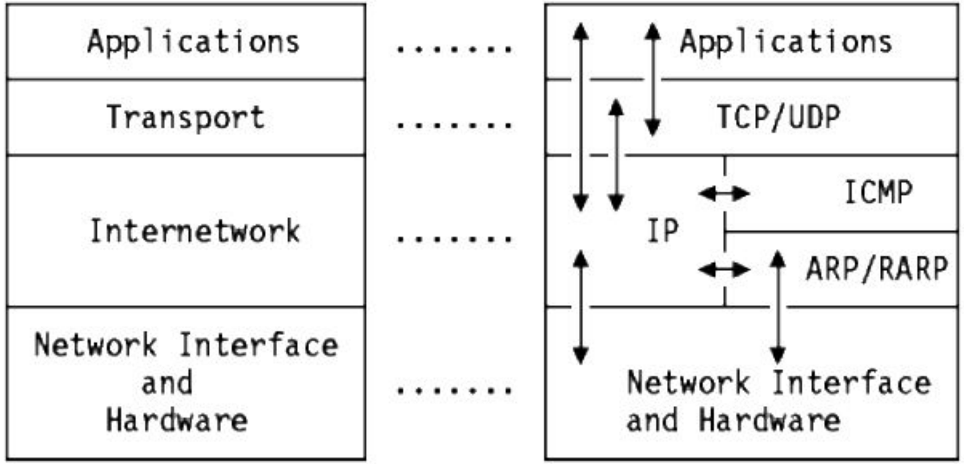
Logic tier

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

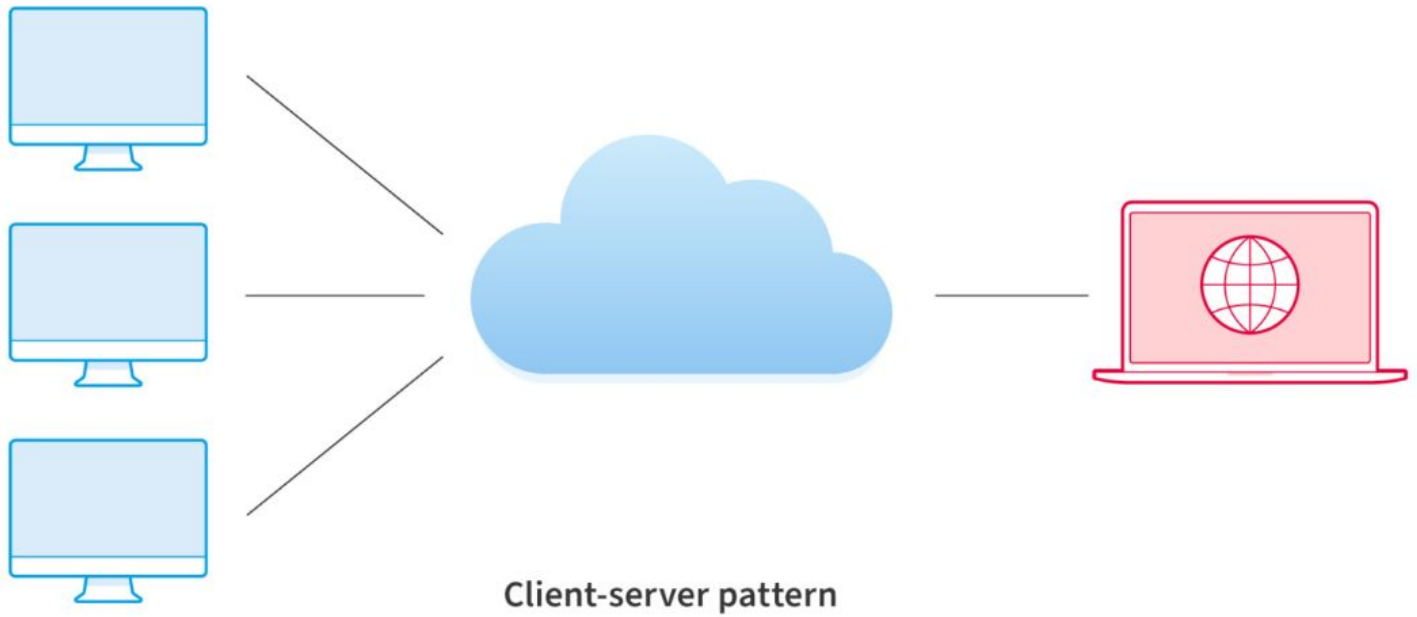
Architectural Patterns

- 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

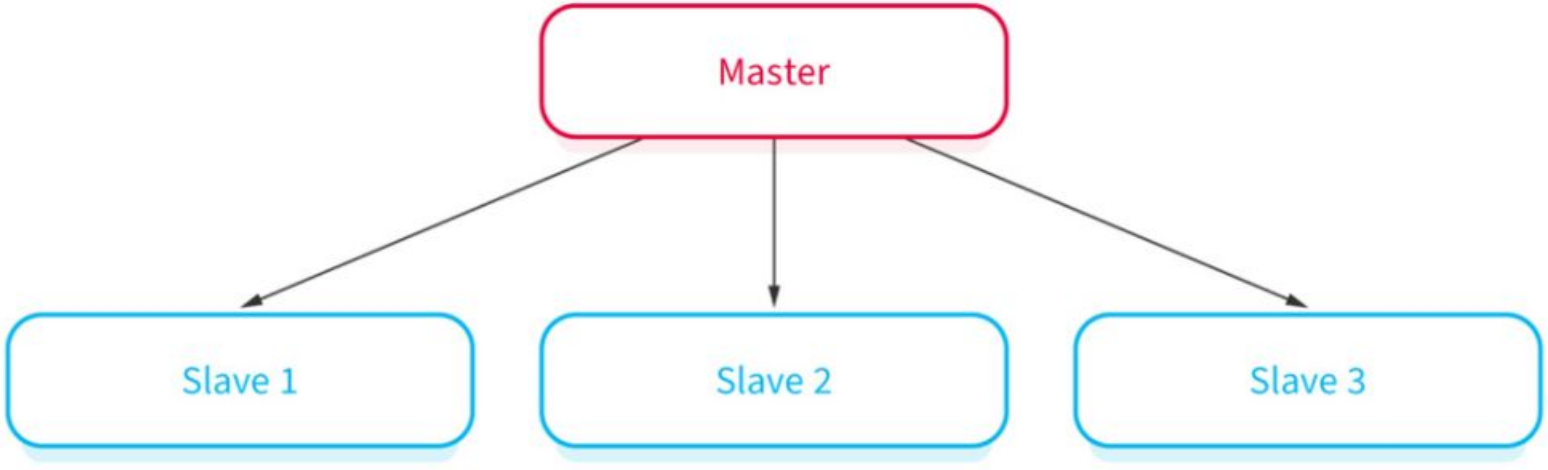
Advantages:

- 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

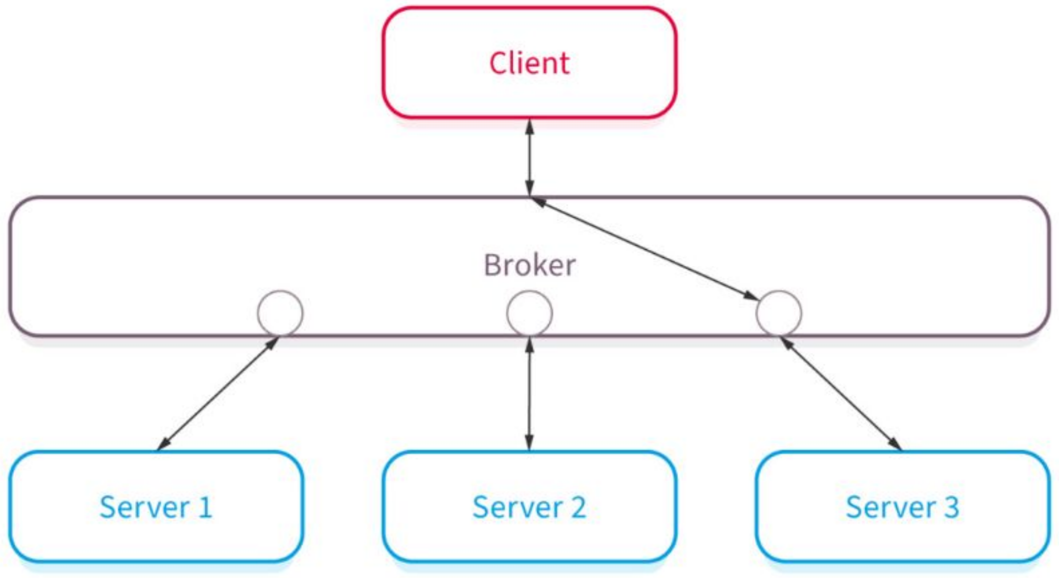
Advantages:

- 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

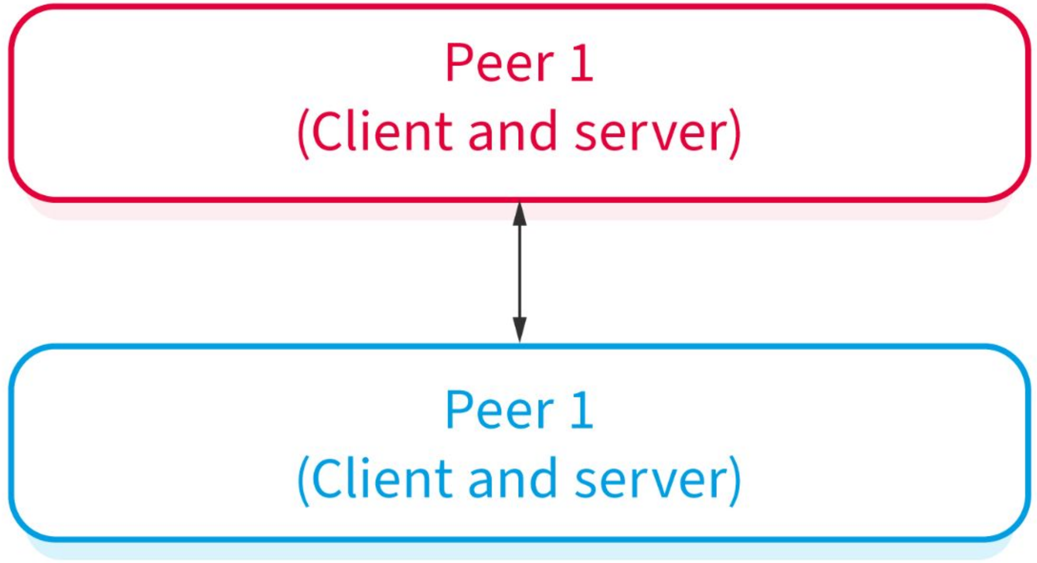


Broker pattern

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

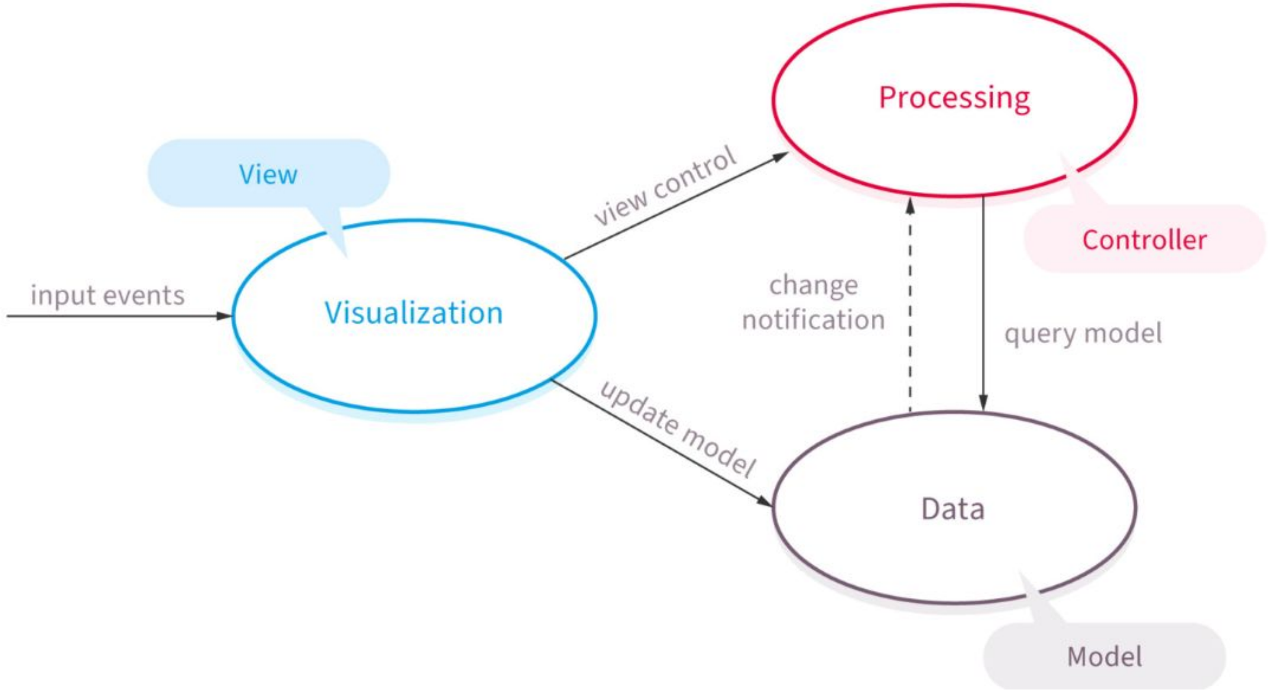


Peer-to-peer pattern

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.

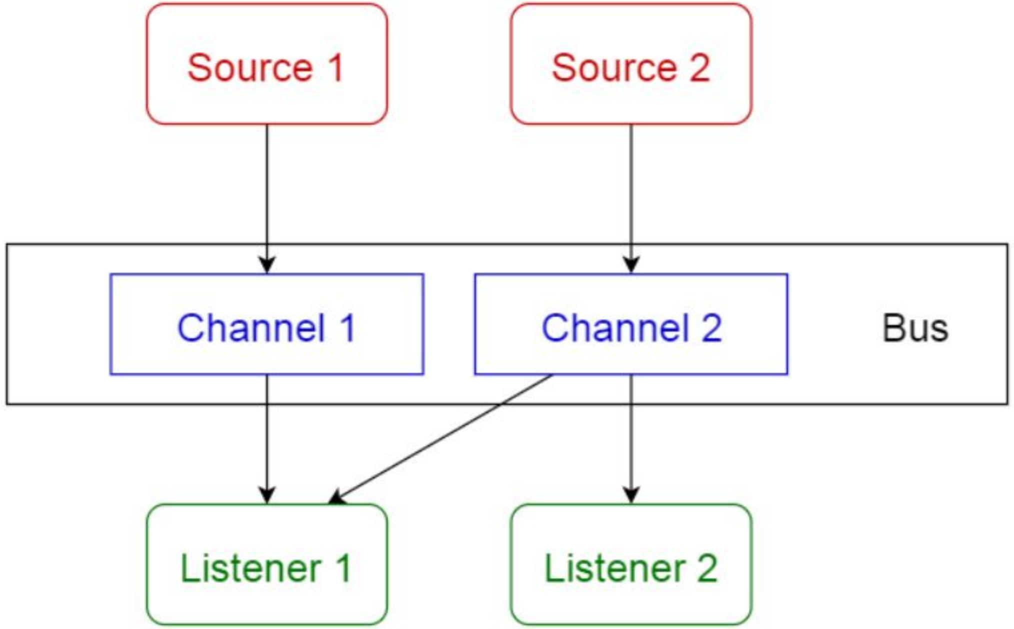
MVC


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

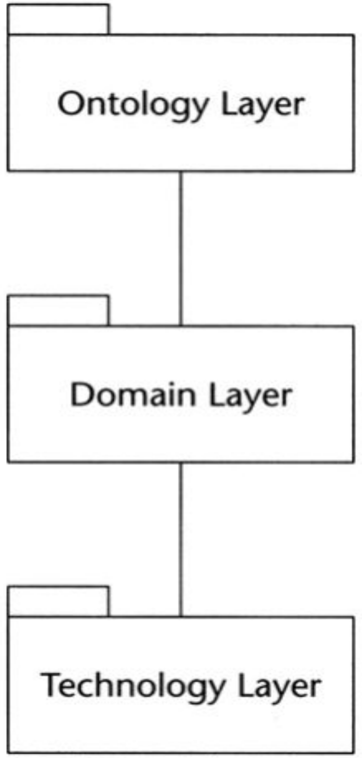


Figure 11.2: Three-layer model of knowledge representation.

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).

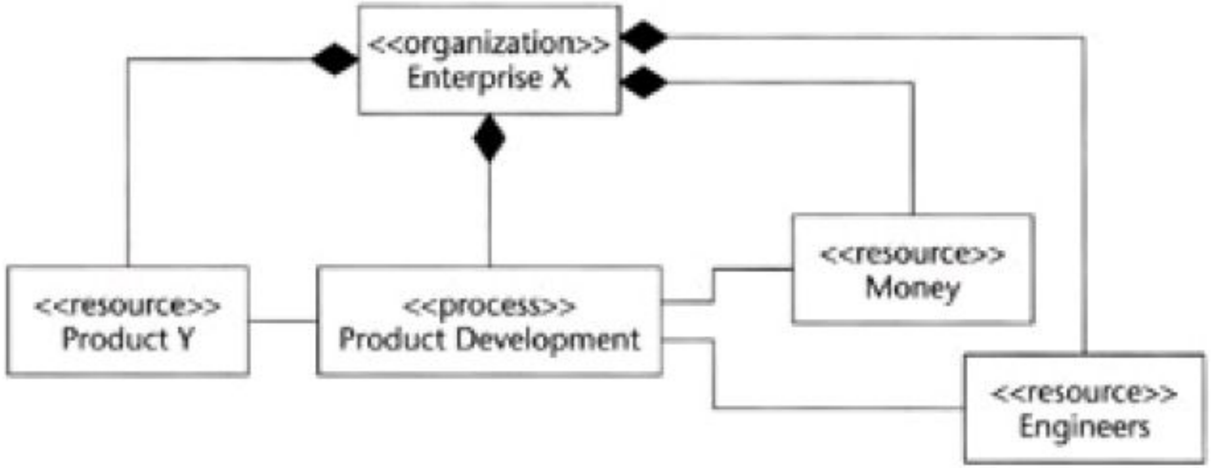


Figure 11.4: Domain model of a business application.

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

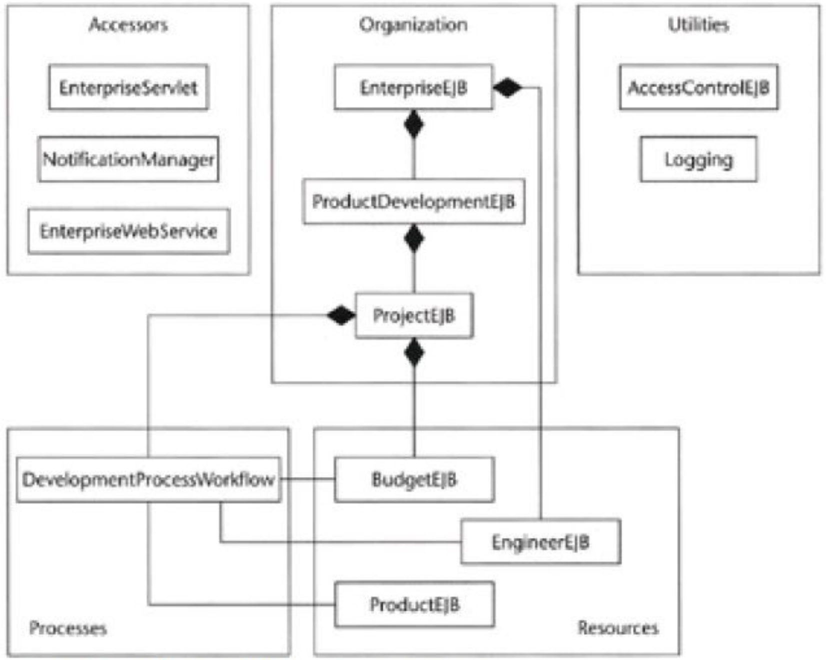


Figure 11.5: Technology model of a business application.

Applying Reference Models

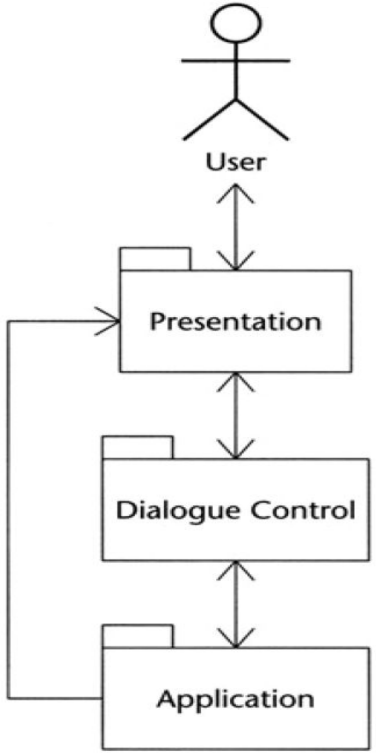
- A reference model abstracts software components and expresses the system as connectors and components.
- Reference models are common in mature domains such as HCI, compiler design, and database system design.
- A reference model may be created as a result of domain analysis, whereby a problem domain 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

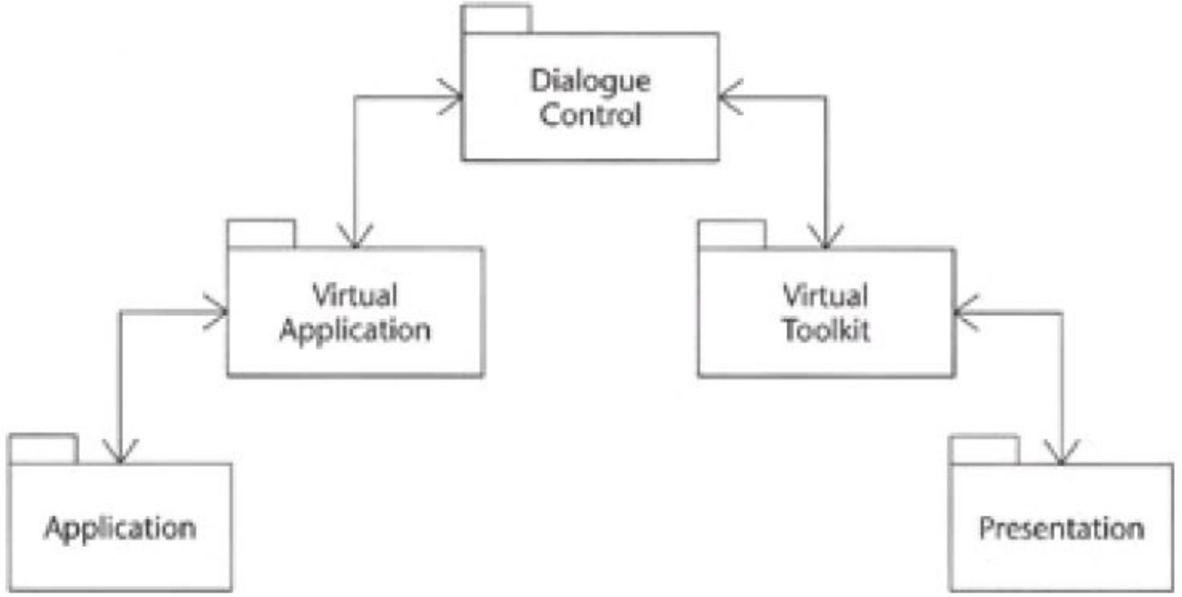


Figure 11.9: Arch/slinky metamodel.

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

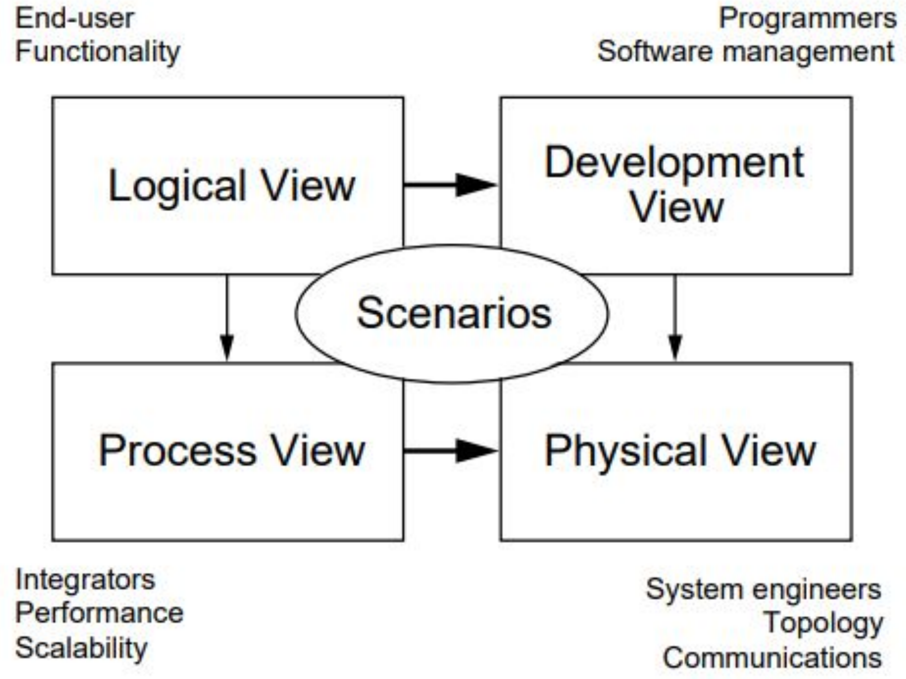


Figure 1 — The “4+1” view model

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)

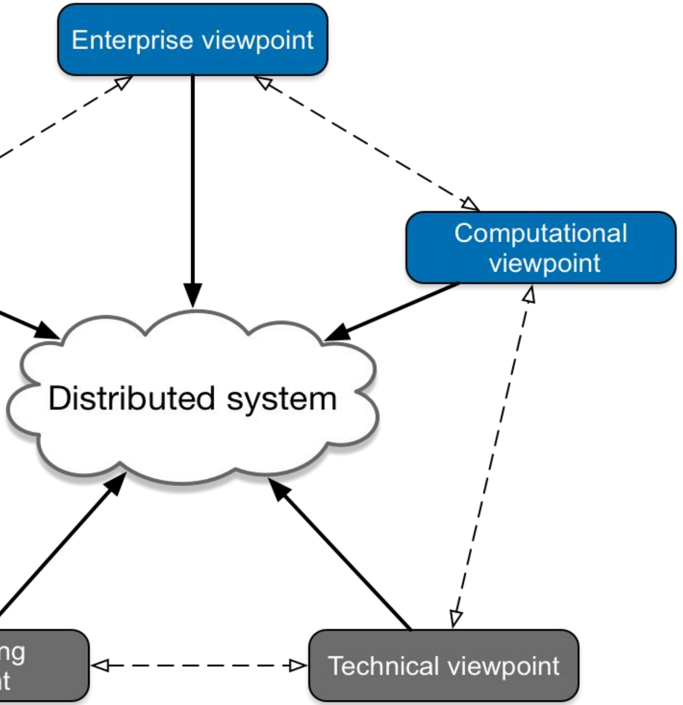
Presented by: Ms. Drashti Shrimal

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

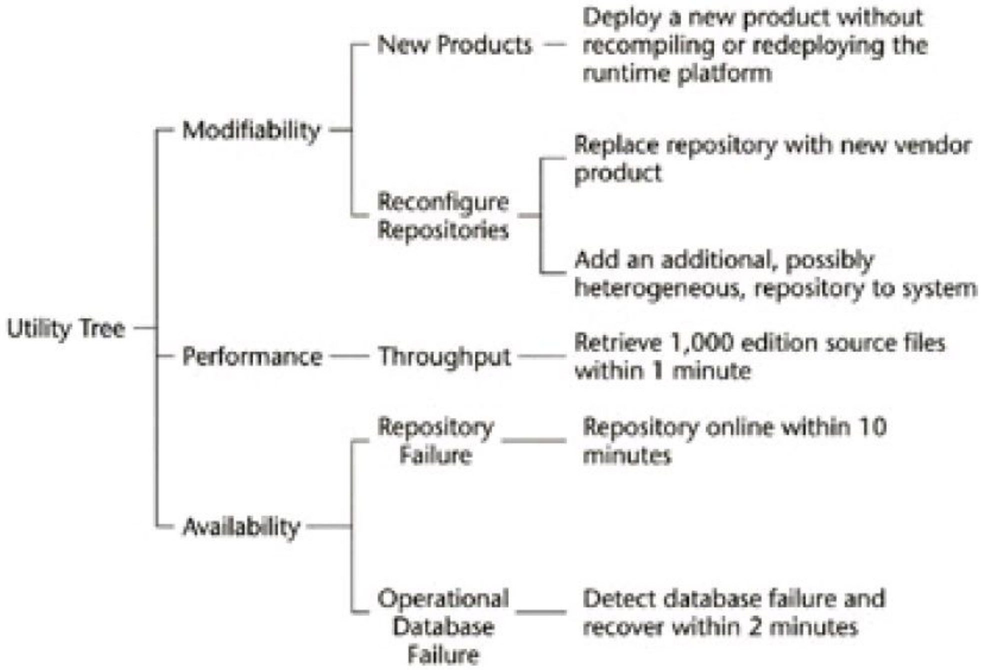

Utility Tree
Modifiability
New Products
Deploy a new product without recompiling or redeploying the runtime platform
Reconfigure Repositories
Replace repository with new vendor product
Add an additional, possibly heterogeneous, repository to system
Performance
Throughput
Retrieve 1,000 edition source files within 1 minute
Availability
Repository Failure
Repository online within 10 minutes
Operational Database Failure
Detect database failure and recover within 2 minutes

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