RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, VII-Semester

CS701 Software Architectures

Pre-Requisite: Software Engineering

Course Outcomes:
After completing the course student should be able to:

1. Describe the Fundamentals of software architecture, qualities and terminologies.
2. Understand the fundamental principles and guidelines for software architecture design, architectural styles, patterns, and frameworks.
3. Use implementation techniques of Software architecture for effective software development.
4. Apply core values and principles of software architectures for enterprise application development.

Course Contents:

Unit 1. Overview of Software development methodology and software quality model, different models of software development and their issues. Introduction to software architecture, evolution of software architecture, software components and connectors, common software architecture frameworks, Architecture business cycle – architectural patterns – reference model.

Unit 2. Software architecture models: structural models, framework models, dynamic models, process models. Architectures styles: dataflow architecture, pipes and filters architecture, call-and return architecture, data-centered architecture, layered architecture, agent based architecture, Micro-services architecture, Reactive Architecture, Representational state transfer architecture etc.

Unit 3. Software architecture implementation technologies: Software Architecture Description Languages (ADLs), Struts, Hibernate, Node JS, Angular JS, J2EE – JSP, Servlets, EJBs; middleware: JDBC, JNDI, JMS, RMI and CORBA etc. Role of UML in software architecture.

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

Unit 5. Software Architecture documentation: principles of sound documentation, refinement, context diagrams, variability, software interfaces. Documenting the behavior of software elements and software systems, documentation package using a seven-part template.

Text Books

1. Bass, L., P. Clements, and R. Kazman, “Software Architecture in Practice”, Second Edition, Prentice-Hall.

2. Jim Keogh, “J2EE – Complete Reference”, Tata McGraw Hill.

3. Dikel, David, D. Kane, and J. Wilson, “Software Architecture: Organizational Principles and Practices”, Prentice-Hall.

Reference Books

1. Bennett, Douglas, “Designing Hard Software: The Essential Tasks”, Prentice-Hall, 1997.

2. Clements, Paul, R. Kazman, M. Klein, “Evaluating Software Architectures: Methods and Case Studies”, Addison Wesley, 2001.

3. Albin, S. “The Art of Software Architecture”, Indiana: Wiley, 2003.

4. Robert Mee, and Randy Stafford, “Patterns of Enterprise Application Architecture”, Addison-Wesley, 2002.

5. Witt, B., T. Baker and E. Meritt, “Software Architecture and Design: Principles, Models and Methods”, Nostrand Reinhold, 1994.

Course Outcomes:

After completing the course student should be able to:

1. Describe the Fundamentals of software architecture, qualities and terminologies.
2. Understand the fundamental principles and guidelines for software architecture design, architectural styles, patterns, and frameworks.
3. Use implementation techniques of Software architecture for effective software development.
4. Apply core values and principles of software architectures for enterprise application development.

Course Contents:

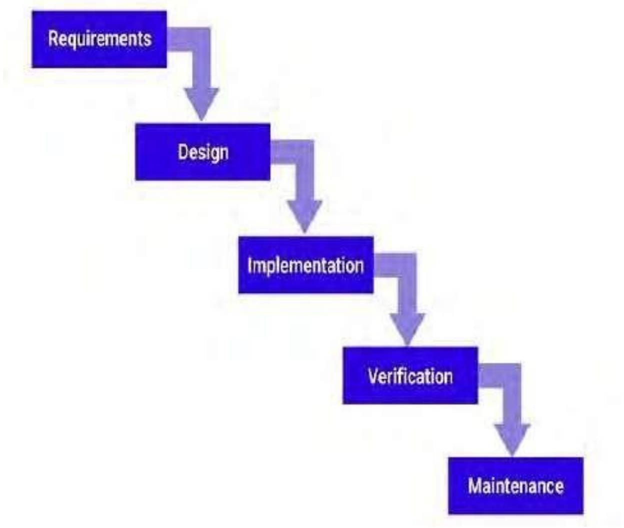
Unit 1. Overview of Software development methodology and software quality model, different models of software development and their issues. Introduction to software architecture, evolution of software architecture, software components and connectors, common software architecture frameworks, Architecture business cycle – architectural patterns – reference model.

Overview of Software Development Methodology:

Software development methodology is a framework that is used to structure, plan, and control the process of developing an information system. This kind of development methodologies are only concerned with the software development process, so it does not involve any technical aspect of, but only concern with proper planning for the software development.

There are Various Software development Methodology:

1. Waterfall Development Model:



The waterfall model is one of the most traditional and commonly used software development methodologies for software development. This life cycle model is often considered as the classic style of the software development. This model clarifies the software development process in a linear sequential flow that means that any phase in the development process begins only if the earlier phase is completed. This development approach does not define the process to go back to the previous phase to handle changes in requirements.

Figure 1.1 Waterfall Model

Advantages of Waterfall Model:

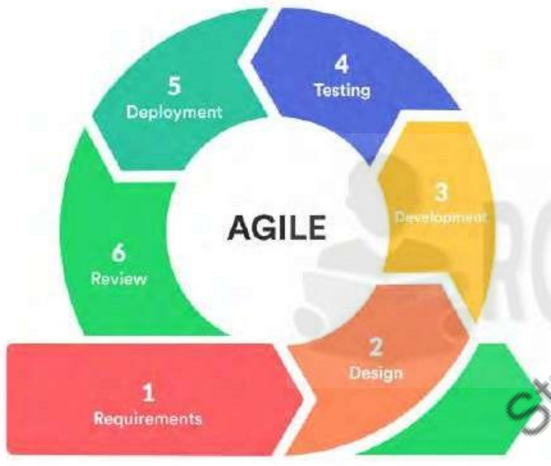
1. Waterfall model is very simple and easy to understand and use a method that is why it is really beneficial for the beginner or novice developer.

2. It is easy to manage, because of the rigidity of the model. Moreover, each phase has specific deliverables and individual review process.
3. In this model phases are processed and completed are at once in a time thus it saves a significant amount of time.
4. This type of development model works more effectively in the smaller projects where requirements are very well understood.
5. The testing is easier as it can be done by reference to the scenarios defined in the earlier functional specification.

Disadvantages of Waterfall Model:

1. This model can only be used when very precise up-front requirements are available.
2. This model is not applicable for maintenance type of projects.
3. The main drawback of this method is that once an application is in the testing stage, it is not possible to go back and edit something.
4. There is no possibility to produce any working software until it reaches the last stage of the cycle.
5. In this model, there is no option to know the end result of the entire project.

2. Agile Software Development:



Agile Software Development is an approach that is used to design a disciplined software management process which also allows some frequent alteration in the development project. This is a type of software development methodologies which is one conceptual framework for undertaking various software engineering projects. It is used to minimize risk by developing software in short time boxes which are called iterations that generally last for one week to one month.

Figure 1.2 Agile Software Development

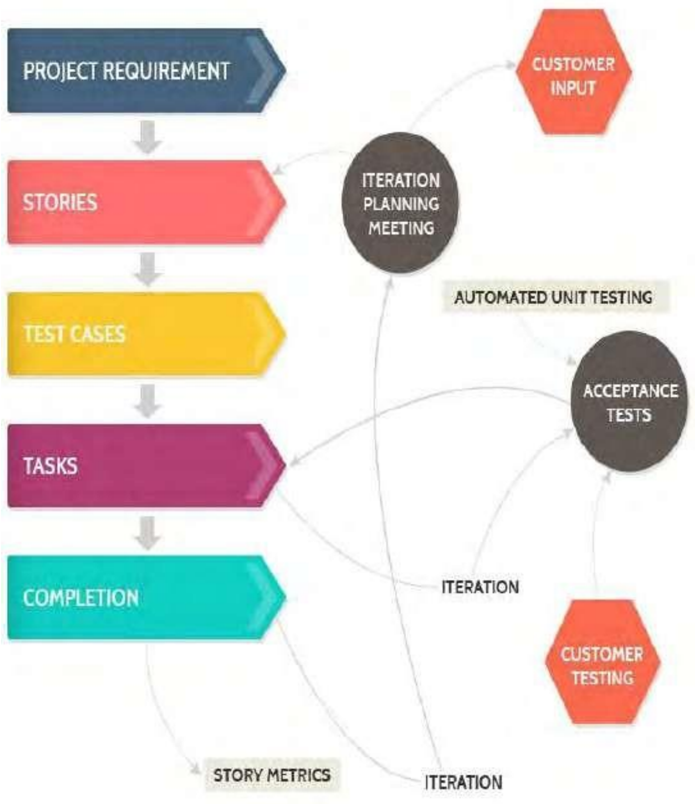
Advantages of Agile Development Methodology:

1. Agile methodology has an adaptive approach which is able to respond to the changing requirements of the clients.
2. Direct communication and constant feedback from customer representative leave no space for any guesswork in the system.

Disadvantage of Agile Development Methodology:

1. This methodology focuses on working software rather than documentation, hence it may result in a lack of documentation.
2. The software development project can get off track if the customer is not very clear about the outcome of his project.

3. **Extreme Programming:** Extreme Programming is an agile software engineering methodology. This methodology, which is shortly known as XP methodology is mainly used for creating software within a very unstable environment. It allows greater flexibility within the modelling process. The main goal of this XP model is to lower the cost of software requirements. It is quite common in the XP model that the cost of changing the requirements on later stage in the project can be very high.



Advantages of Extreme Programming:

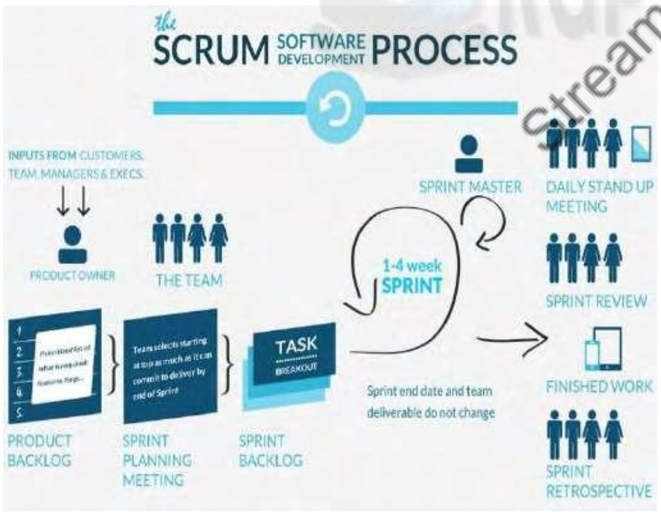
1. It emphasis on customer involvement.
2. This model helps to establish rational plans and schedules and to get the developers personally committed to their schedules which are surely a big advantage in the XP model.
3. This model is consistent with most modern development methods so, developers are able to produce quality software.

Disadvantages of Extreme Programming:

1. It requires meetings at frequent intervals at enormous expense to customers.
2. It requires too much development changes which are really very difficult to adopt every time for the software developer.
3. It tends to impossible to be known exact estimates of work effort needed to provide a quote, because at the starting of the project nobody aware about the entire scope and requirements of the project

Figure 1.3 Extreme Programming Software Development Methodology

4. **Scrum Development Methodology**:



The Scrum Development Methodology can be applied to nearly any project. This process is suited for development projects that are rapidly changing or highly emergent requirements. The Scrum software development model begins with a brief planning, meeting and concludes with a final review. This development methodology is used for speedy development of software which includes a series of iterations to create required software. It is an ideal methodology because it easily brings on track even the slowest progressing projects.

Figure 1.4 Scrum Software Development Methodology

Advantages of Scrum Development:

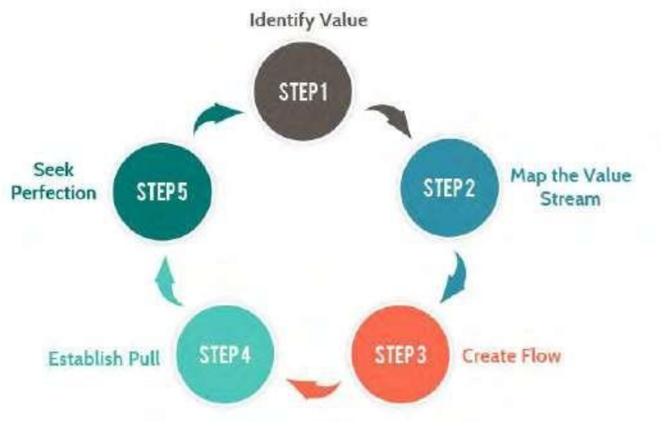
1. In this methodology, decision-making is entirely in the hands of the teams.
2. This methodology enables projects where the business requirements documentation is not considered very significant for the successful development.
3. It is a lightly controlled method which totally empathizes on frequent updating of the progress, therefore, project development steps are visible in this method.

Disadvantages of Scrum Development:

1. This kind of development model is suffered if the estimating project costs and time will not be accurate.
2. It is good for small, fast moving projects but not suitable for large size projects.

3. This methodology needs experienced team members only. If the team consists of people who are novices, the project cannot be completed within exact time frame.

5. Lean Development Methodology:



Lean Development Methodology focuses on the creation of easily changeable software. This Software Development model is more strategically focused than any other type of agile methodology. The goal of this methodology is to develop software in one-third of the time, with very limited budget, and very less amount of required workflow.

Figure 1.5 Lean Software Development Methodology

Advantages of Lean Development Methodology:

1. The early elimination of the overall efficiency of the development process certainly helps to speeds up the process of entire software development which surely reduces the cost of the project.
2. Delivering the product early is a definite advantage. It means that development team can deliver more functionality in a shorter period of time, hence enabling more projects to be delivered.
3. Empowerment of the development team helps in developing the decision-making ability of the team members which created more motivation among team members.

Disadvantages of Lean Development Methodology:

1. Success in the software development depends on how disciplined the team members are and how to advance their technical skills.
2. The role of a business analyst is vital to ensure the business requirements documentation is understood properly. If any organization doesn't have a person with the right business analyst, then this method may not be useful for them.
3. In this development model, great flexibility is given to developer, which is surely great, but too much of it will quickly lead to a development team who lost focus on its original objectives thus, it hearts the flow of entire project development work.

6. **Rapid Application Development Methodology:** Rapid Application Development (RAD) is an effective methodology to provide much quicker development and higher-quality results than those achieved with the other software development methodologies. It is designed in such a way that, it easily take the maximum advantages of the software development. The main objective of this methodology is to accelerate the entire software development process. The goal is easily achievable because it allows active user participation in the development process.

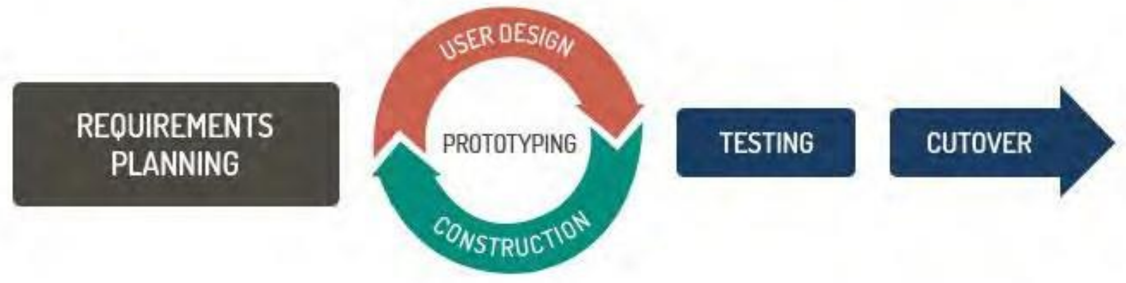


Figure 1.6 Rapid Application Development Methodology

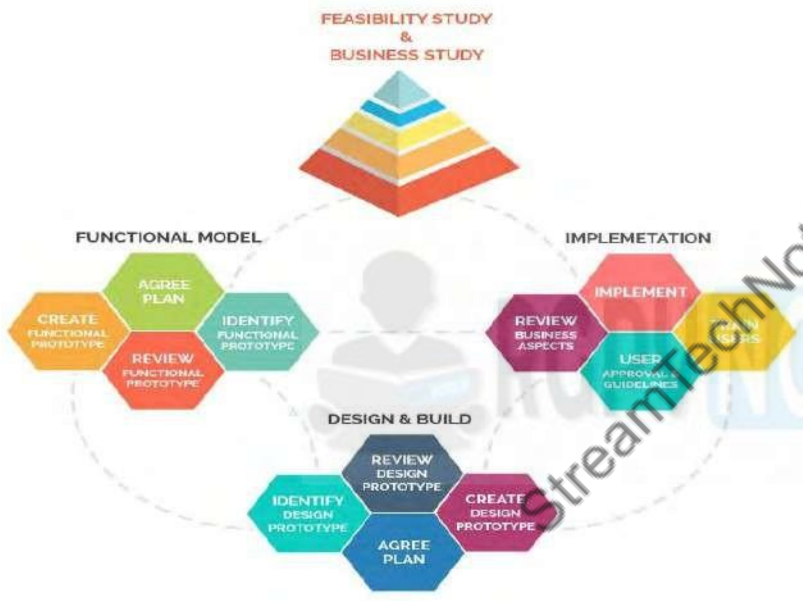
Advantages of the RAD model:

1. Rapid Application development model helps to reduce the risk and required efforts on the part of the software developer.
2. This model also helps clients to take quick reviews for the project.
3. This methodology encourages customer feedback which always provides improvement scope for any software development project.

Disadvantages RAD model:

1. This model depends on the strong team and individual performances for clearly identifying the exact requirement of the business.
2. This approach demands highly skilled developers and designer's team which may not be possible for every organization.
3. This method is not applicable for the developer to use in small budget projects as a cost of modelling and automated code generation is very high.

7. Dynamic Systems Development Model Methodology:



Dynamic Systems Development Model is a software development methodology originally based on the Rapid Application Development methodology. This is an iterative and incremental approach that emphasizes continuous user involvement. Its main aim is to deliver software systems on time and on the budget. This model simply works on the philosophy that nothing is developed perfectly in the first attempt and considers as an ever-changing process.

Figure 1.7 Dynamic System Development Methodology

Advantages of Dynamic Systems Development Model:

1. Users are highly involved in the development of the system so, they are more likely to get a grip on the software development project.
2. In this model, basic functionality is delivered quickly, with more functionality being delivered at frequent intervals.

Disadvantages of Dynamic Systems Development Model:

1. The first thing is DSDM is costly to implement, as it requires users and developers both to be trained to employ it effectively. It may not be suitable for small organizations or one-time projects.
2. It is a relatively new model, therefore, it is not very common and easy to understand.

Software Quality Model

Software Quality Models are a standardised way of measuring a software product. With the increasing trend in software industry, new applications are planned and developed every day. This eventually gives rise to the need for reassuring that the product so built meets at least the expected standards.

Following are few models that explains what kind of quality criteria is to be followed.

1. **Mc Call's Model -**

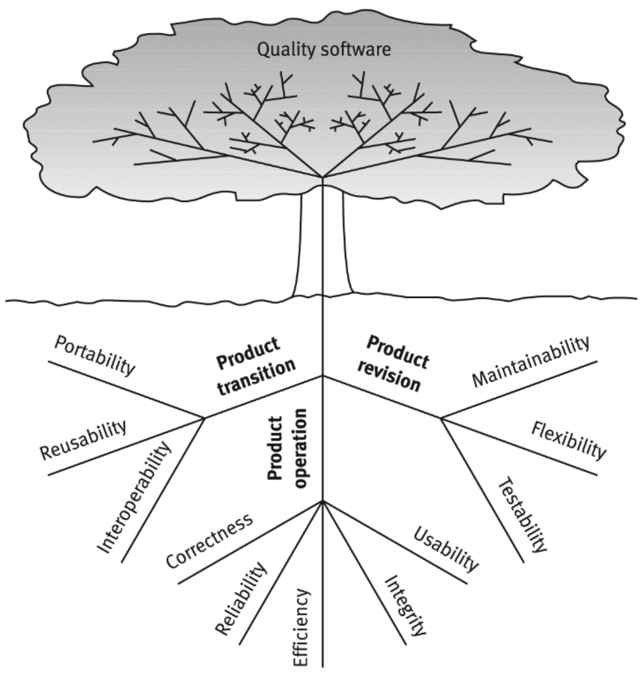


Figure 1.8 Mc Call's Model

It is the first quality model developed, which defines a layout of the various aspects that define the product's quality. It defines the product quality in the following manner – Product Revision, Product Operation, Product Transition. Product revision deals with maintainability, flexibility and testability, product operation is about correctness, reliability, efficiency and integrity.

(i) Product Revision - The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are:-**(a) Maintainability -** The ability to find and fix a defect. **(b) Flexibility -** The ability to make changes required as dictated by the business. **(c) Testability-** The ability to Validate the software requirements.

(ii) Product Transition -The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments: - **(a) Portability** -The ability to transfer the software from one environment to another. **(b) Reusability** - The ease of using existing software components in a different context. **(c) Interoperability** - The extent, or ease, to which software components work together.

(iii) Product Operations - The product operations perspective identifies quality factors that influence the extent to which the software fulfils its specification -**(a)Correctness** - The functionality matches the specification.**(b)Reliability** - The extent to which the system fails.**(c) Efficiency -** System resource (including CPU, disk, memory, network) usage.**(d) Integrity** - Protection from unauthorized access.**(e) Usability -**ease of use.

2. Boehm Model -

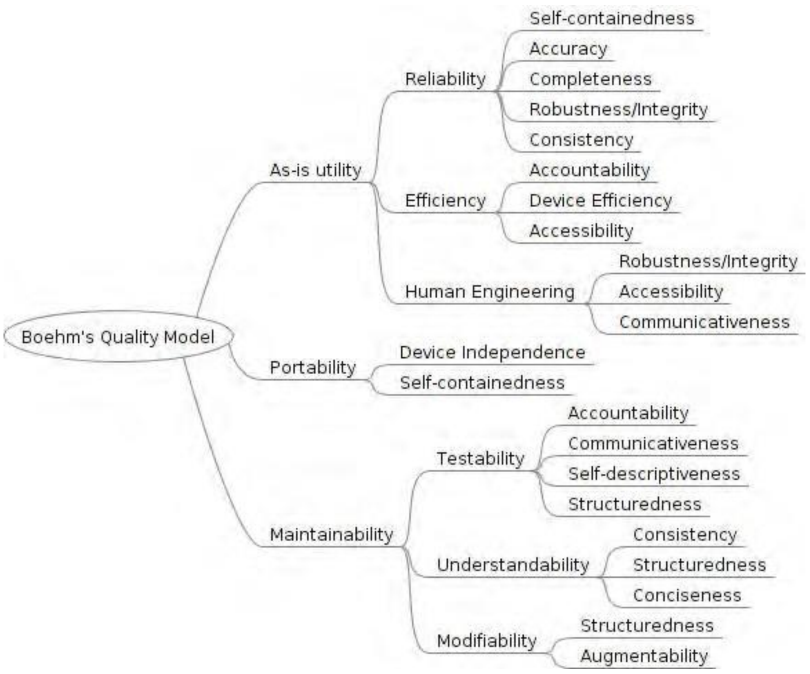
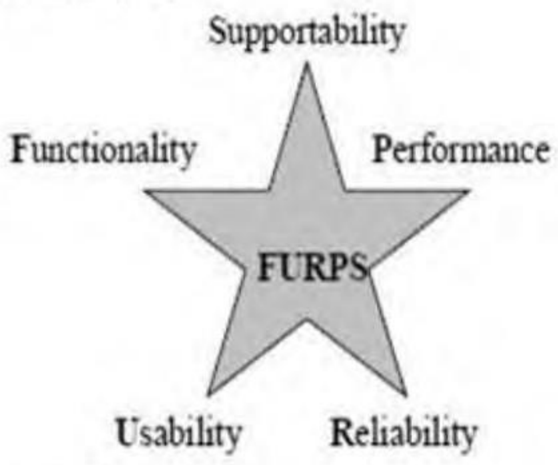


Figure 1.9 Bohem Model

It describes how easily and reliably a software product can be used. This model actually elaborates the aspects of McCall model in detail. It begins with the characteristics that resorts to higher level requirements. The model's general utility is divided into various factors - portability, efficiency and human engineering, which are the refinement of factors like portability and utility. Further maintainability is refined into testability, understand ability and modifiability.

3. FURPS Model -



This model categorises requirements into functional and non-functional requirements. The term FURPS is an acronym for Functional requirement(F) which relies on expected input and output, and in non-functional requirements (U) stands for Usability which includes human factors, aesthetic, documentation of user material of training, (R) stands for reliability(frequency and severity of failure, time among failure), (P) stands for Performance that includes functional requirements, and finally (S) stands for supportability that includes backup, requirement of design and implementation etc.

Figure 1.10 FURPS Model

FURPS is a checklist for requirements, which help maintain a SQ Standard. It compromises:- (a) Functional (features, capabilities, security), (b) Usability (human factors, help, documentation) (c) Reliability (frequency of failure, recoverability, predictability), (d) Performance (response time, throughput, accuracy, availability, resource usage), (e) Supportability (adaptability, maintainability, internationalization, configurability)

4. ISO 9126 Quality Model -

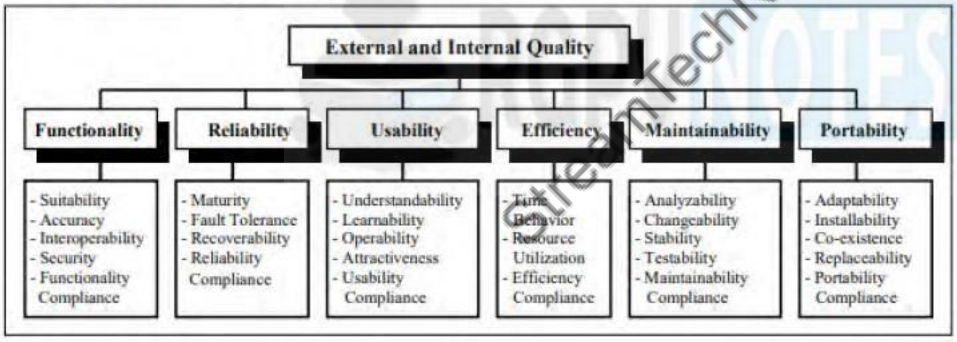


Figure 1.11 ISO 9126 Quality Model for External and Internal Quality

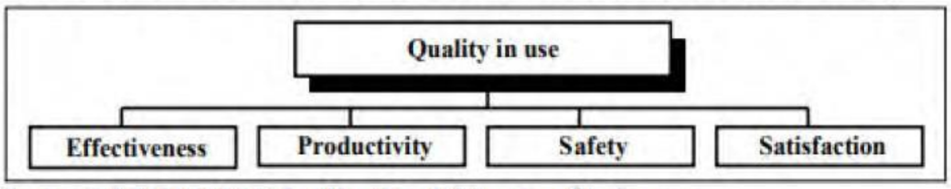


Figure 1.12 ISO 9126 Quality Model for Quality in use

It has two primary categories – internal and external quality attributes and quality in use attributes. The internal quality attributes are the properties of the system the evaluation of which can be done without executing it. Whereas the external quality attributes are those that are evaluated by observing the system during execution.

Different Models of Software Development and their issues

1. **Waterfall Model** - The waterfall model is one of the most traditional and commonly used software development methodologies for software development. This life cycle model is often considered as the classic style of the software development. This model clarifies the software development process in a linear sequential flow that means that any phase in the development process begins only if the earlier phase is completed.

Issues

- This model can only be used when very precise up-front requirements are available.
- The main issue of this method is that once an application is in the testing stage, it is not possible to go back and edit something.

- There is no possibility to produce any working software until it reaches the last stage of the cycle.
- In this model, there is no option to know the end result of the entire project.
- This model is good for a small project but not ideally suitable for long and ongoing projects.

2. **Iteration Model**-

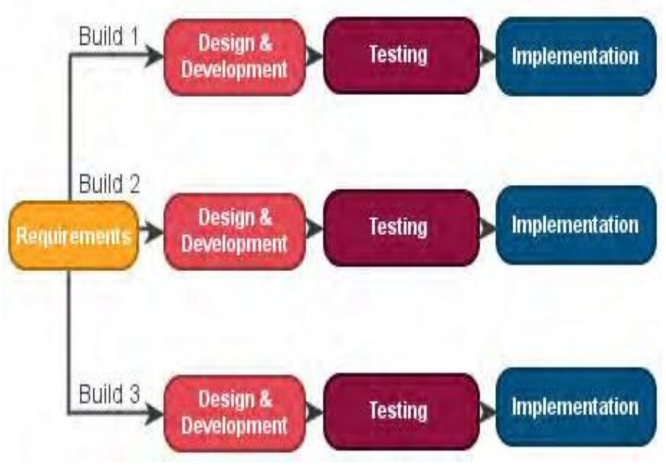


Figure 1.13 Iterative Model

An iterative life cycle model does not start with a full specification of requirements. In this model, the development begins by specifying and implementing just part of the software, which is then reviewed in order to identify further requirements. Each release of Iterative Model is developed in a specific and fixed time period, which is called iteration. Furthermore, this iteration focuses on a certain set of requirements. Each cycle ends with a usable system i.e., a particular iteration results in an executable release.

Issues

- More resources may be required, and more management attention is required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- It is not suitable for smaller projects, and highly skilled resources are required for skill analysis.
- Project progress is highly dependent upon the risk analysis phase.
- Defining increments may require definition of the complete system.

3. **V-Shaped Model-**

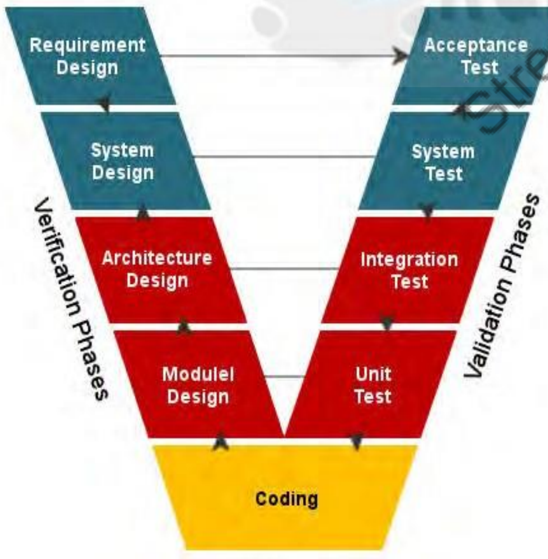


Figure 1.14 V- Model

V-Model is an SDLC model that has a testing phase corresponding to every development stage in the waterfall model. It is pronounced as the "vee" model. The V-model is an extension of the waterfall model. V model Testing is done in parallel to development. It is also called a Validation and Verification Model.

Issues

- Very rigid and least flexible.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- If any changes happen in midway, then the test documents along with requirement documents has to be updated.

4. **Rapid Application Development Methodology:** Rapid Application Development (RAD) is an effective methodology to provide much quicker development and higher-quality results than those achieved with the other software development methodologies. It is designed in such a way that, it easily takes the maximum advantages of the software development. The main objective of this methodology is to accelerate the entire software development process. The goal is easily achievable because it allows active user participation in the development process.

Issues

- This model depends on the strong team and individual performances for clearly identifying the exact requirement of the business.

- It only works on systems that can be modularized can be built using this methodology.
- This approach demands highly skilled developers and designer's team which may not be possible for every organization.
- This method is not applicable for the developer to use in small budget projects as a cost of modelling and automated code generation is very high.

5. Spiral Model-

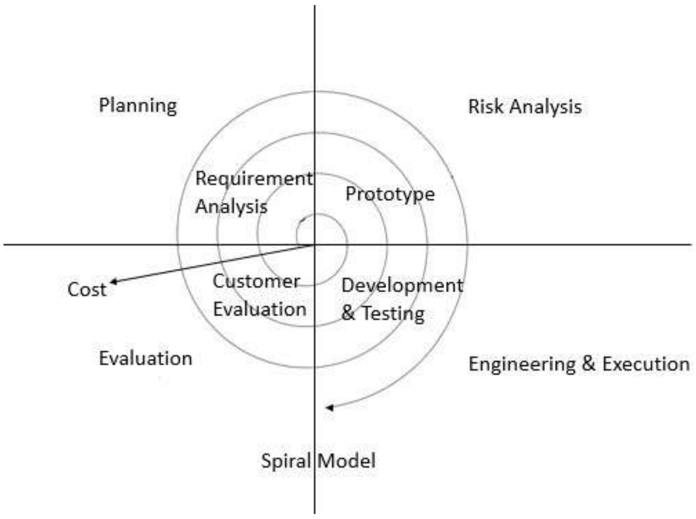


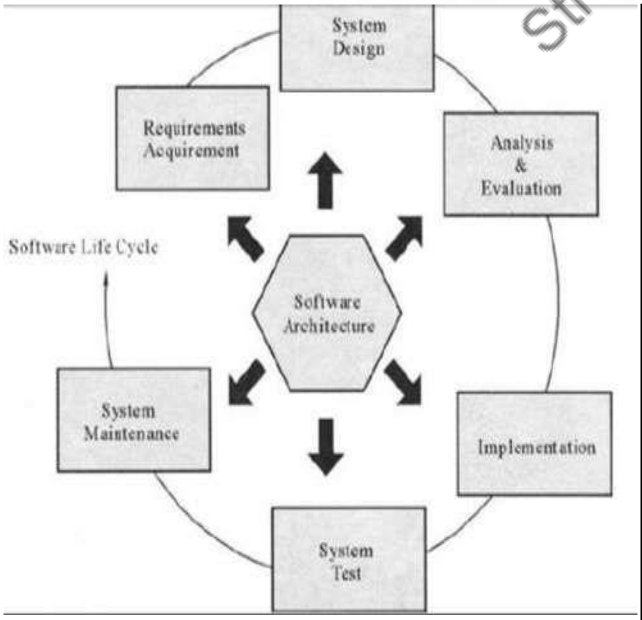
Figure 1.15 Spiral Model

Spiral Model is a combination of a waterfall model and iterative model. Each phase in spiral model begins with a design goal and ends with the client reviewing the progress. The spiral model was first mentioned by Barry Boehm in his 1986 paper. The development team in Spiral-SDLC model starts with a small set of requirement and goes through each development phase for those set of requirements. The software engineering team adds functionality for the additional requirement in every-increasing spirals until the application is ready for the production phase.

Issues

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

Software Architecture



"The software architecture of a program or computing system is the structure or structures of the system, which comprise- Software components, The externally visible properties of those components, The relationships among them. "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." It encompasses the set of significant decisions about the organization of a software system.

- Selection of the structural elements and their interfaces by which a system is composed.
- Behaviour as specified in collaborations among those elements.

Figure 1.16 Software Architecture

There are fundamentally three reasons for software architecture's importance from a technical perspective.

- Communication among stakeholders
- Early design decisions
- Transferable abstraction of a system

Evolution of Software Architecture

Software architecture has been an evolutionary discipline starting with monolithic mainframes to recent micro services.

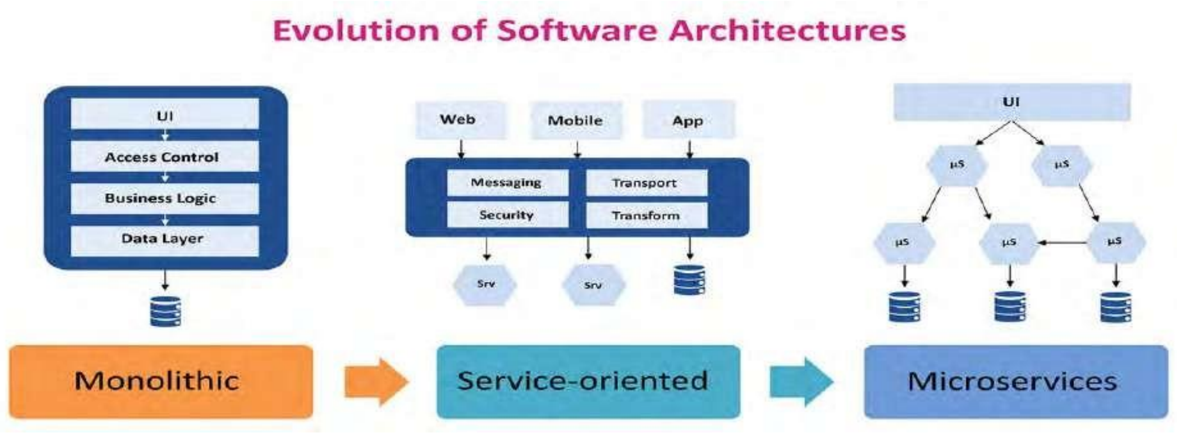


Figure 1.17 Evolution of Software Architecture

The mainframe era was of expensive hardware, having powerful server capable of processing large amount instructions and client connecting to it via dumb terminals. This evolved as hardware became cheaper and dumb terminals paved way to smart terminals. These smart terminals had reasonable processing power leading to a client server models. There are many variations of client server models around how much processing a client should do versus the server. For instance, client could all the processing having server just act as centralized data repository. The primary challenge with that approach was maintenance and pushing client-side updates to all the users. This led to using browser clients where the UI is essentially rendered from server in response to a HTTP request from browser.

As server started having multiple responsibilities in this new world like serving UI, processing transactions, storing data and others, architects broke the complexity by grouping these responsibilities into logical layers – UI Layer, Business Layer, Data Layer, etc. Specific products emerged to support these layers like Web Servers, Database servers, etc. Depending on the complexity these layers were physically separated into tier. The word tier indicates a physical separation where Web Server, Database server and business processing components run on their own machines.

Software Component and Connectors

Component - A Component is a unit of behaviour. Its description defines what the component can do and what it requires to do that job.

Connector - A Connector is an indication that there is a mechanism that relates one component to another usually through relationships such as data flow or control flow.

Changing the grouping of behaviours in components or changing which components are connected changes the value of certain quality attributes.

Component-and-connector structures help answer questions such as-

- What are the major executing components and how do they interact?
- What are the major shared data stores?
- Which parts of the system are replicated? How does data progress through the system?
- What parts of the system can run in parallel?
- How can the system's structure change as it executes?

Process, or communicating processes - Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations. The relation in this (and in all component-and-connector structures) is *attachment*, showing how the components and connectors are hooked together. The process structure is important in helping to engineer a system's execution performance and availability.

Concurrency - This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are "logical threads." A logical thread is a sequence of computation that can be allocated to a separate physical thread later in the design process. The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.

Shared data, or repository - This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate. It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.

Client-server - If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work. This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

Common Software Architecture Framework

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

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

1. Layered Architecture: This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer. The most commonly found 4 layers of a general information system are as follows.

- Presentation layer (also known as UI layer)
- Application layer (also known as service layer)
- Business logic layer (also known as domain layer)
- Data access layer (also known as persistence layer)

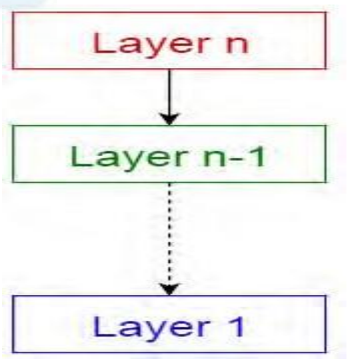


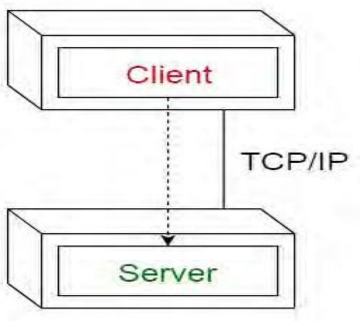
Figure 1.18 Layered Architecture

Usage

1. General Desktop application.

2. E- commerce application.

2. Client Server Architecture - This pattern consists of two parties; a server and multiple clients. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.



Usage

1. Online applications such as email, document sharing and banking.

Figure 1.19 Client Server Architecture

3. Master Slave - This pattern consists of two parties; master and slaves. The master component distributes the work among identical slave components and computes a final result from the results which the slaves return.

Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).

4. Model View Controller-

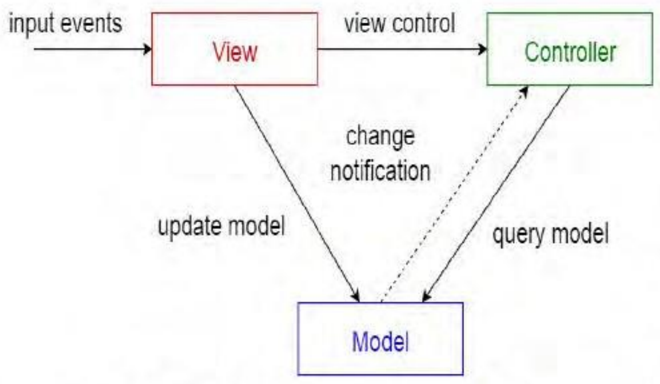


Figure 1.20 Model View Controllers

This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

model — contains the core functionality and data

view — displays the information to the user (more than one view may be defined)

controller — handles the input from the user

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as Django and Rails.

Architecture Business Cycle

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Software architecture is a result of technical, business and social influences. Its existence in turn affects the technical, business and social environments that subsequently influence future architectures. We call this cycle of influences, from environment to the architecture and back to the environment, the **Architecture Business Cycle (ABC).**

Working of architecture business cycle:

1) The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system it particularly prescribes the units of software that must be implemented and integrated to form the system. Teams are formed for individual software units; and the development, test, and integration activities around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.

2) The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of the similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.

3) The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system in a more reliable, timely and economical manner than if the subsequent system were to be built from scratch.

4) The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base.

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.
For ex: client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.
A reference model is a division of functionality together with data flow between the pieces.
A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem.
A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them. Whereas a reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition.

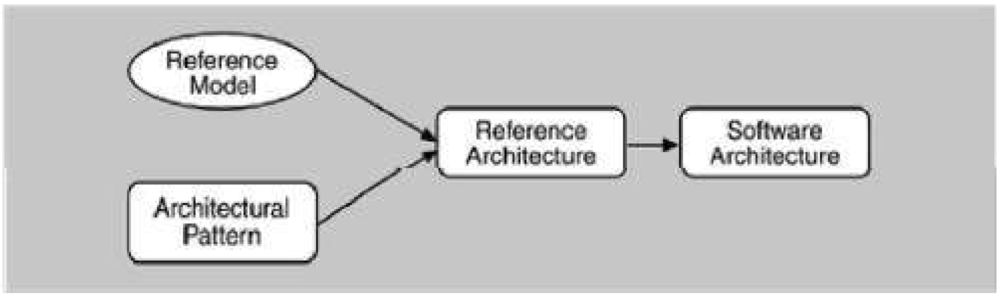


Figure 1.21 Relationship of Reference models, architectural patterns, and reference architectures

Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architecture. Each is the outcome of early design decisions. A software architect must design a system that provides concurrency, portability, modifiability, usability, security, and the like, and that reflects consideration of the trade-offs among these needs.

<u>**UNIT-II**</u>

Course Contents:

Unit 2. Software architecture models: structural models, framework models, dynamic models, process models. Architectures styles: dataflow architecture, pipes and filters architecture, call-and return architecture, data-centered architecture, layered architecture, agent based architecture, Micro-services architecture, Reactive Architecture, Representational state transfer architecture etc.

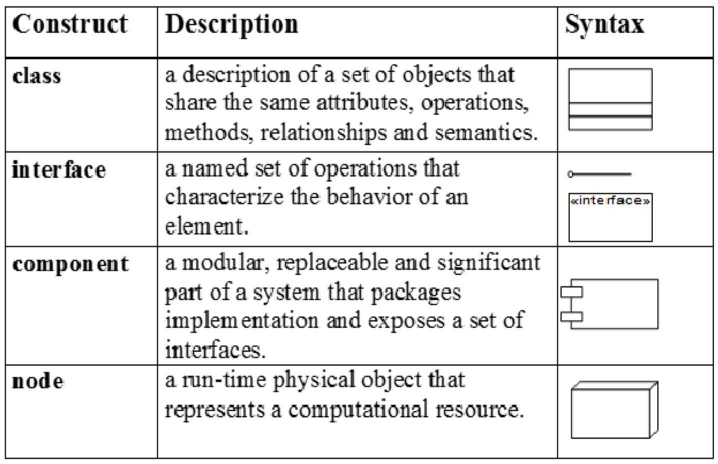
Software Architecture Models:

Structural Models: Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design. Structural models of a system required during the discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. Structural model represents the framework for the system and this framework is the place where all other components exist.

Structural modelling captures the static features of a system. They consist of the following −

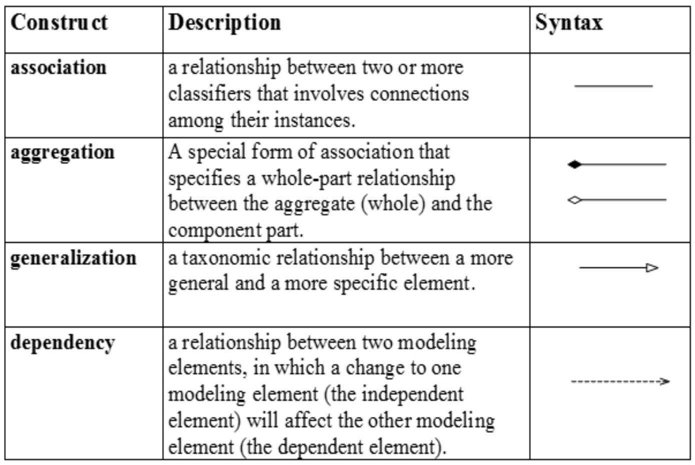
- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural Modelling: Core Elements-



Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
interface	a named set of operations that characterize the behavior of an element.	«interface»
component	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
node	a run-time physical object that represents a computational resource.	

Figure 2.1 (a): Structural Modelling Core Elements



Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances.	
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	
generalization	a taxonomic relationship between a more general and a more specific element.	
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	

Figure 2.1 (b): Structural Modelling Core Relationships

Framework Models: An architecture framework is an encapsulation of a minimum set of practices and requirements for artifacts that describe a system's architecture. Models are representations of how objects fit in system and behave as part of the system. An architecture framework captures the "conventions, principles and practices for the description of architectures, established within a specific domain of application and/or community of stakeholders". A framework is usually implemented in terms of one or more viewpoints. Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Dynamic Models: The dynamic model is used to express and model the behavior of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modelling. After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modeling.

Dynamic Modeling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world."

The process of dynamic modeling can be visualized in the following steps −

- Identify states of each object.
- Identify events and analyze the applicability of actions.
- Construct a dynamic model diagram, comprising of state transition diagrams.
- Express each state in terms of object attributes.
- Validate the state–transition diagrams drawn.

Dynamic model is represented graphically with the help of state diagrams. It is also known as state modelling. State model consist of multiple state diagrams, one for each class with temporal behavior that is important to an application. State diagram relates with events and states. Events represents external functional activity and states represents values objects.

Events: An event is something that happen at a point in particular time such as a person press button or train 12345 departs from Indore. Event conveys information from one object to another.

The events are of three types: Signal event, Change event, and Time event.

Signal Event: A signal event is a particular occurrence in time. A signal is an explicit one-way transmission of information from one object to another such as sending or receiving signal. When an object send signal to another object it waits for acknowledgement, but acknowledgement signal is the separate signal under the control of second object, which may or may not choose to send it.

Change Event: It is caused by the satisfaction of a boolean expression. The intent of the change event is that the expression is tested continually whenever the expression changes from false to true.

Example: when (battery power < lower limit)
When (room temperature < heating/cooling point)

Time event: It is caused by occurrence of an absolute or the elapse of time interval. The UML notation for absolute time is the keyword when followed by a parenthesized expression involving time and for the time interval is keyword after followed by a parenthesized expression that evaluates time duration.

Example: when (Date = mar 2, 2005)
after (50 seconds)

State: A state is an abstraction of attribute values and links of an object. Values and links are combined into a state according to their entire behaviour. The response of object according to input event is called state. A state corresponds to the interval between two events received by an object. The state of the event depends on the past event. So basically, state represents intervals of time. The UML notation for the state is a round box containing an optional state name list, list the name in boldface, center the name near the top of the box, capitalize the first letter.

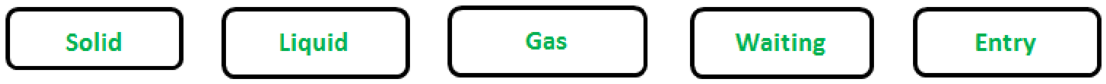
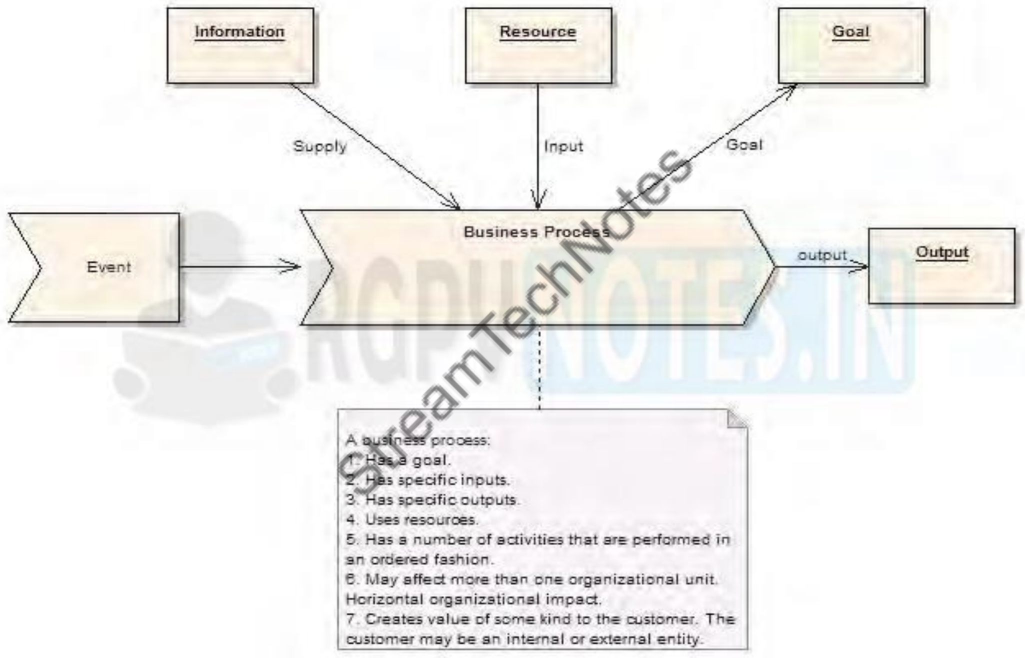


Figure 2.2: Example & representation of State

The following are the important points needs to be remembered about state.

1. Ignore attributes that do not affect the behaviour of object.
2. The objects in the class have finite number of possible states. Each object can be in one state at a time.
3. All events are ignored in a state, except those for which behaviour is explicitly prescribed.
4. Both events and states depend upon level of abstraction.

Process Models: A process model is a UML extension of an activity diagram used to model a business process - this diagram shows what goal the process has, the inputs, outputs, events and information that are involved in the process.



A business process:
1. Has a goal.
2. Has specific inputs.
3. Has specific outputs.
4. Uses resources.
5. Has a number of activities that are performed in an ordered fashion.
6. May affect more than one organizational unit. Horizontal organizational impact.
7. Creates value of some kind to the customer. The customer may be an internal or external entity.

Figure 2.3: Process Model

Architectures Styles: Architectural styles tell, how to organise our code. It's the highest level of granularity and it specifies layers, high-level modules of the application and how those modules and layers interact with each other, the relations between them. An Architectural Style can be implemented in various ways, with a specific technical environment, specific policies, frameworks or practices, or can say - The architectural styles that are used while designing the software.

Dataflow Architecture: In data flow architecture, the whole software system is seen as a series of transformations on consecutive pieces or set of input data, where data and operations are independent of each other. In this approach, the data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output or a data store).

- Data Flow Architecture is transformed input data by a series of computational or manipulative components into output data. The data can be flow in the graph topology with cycles or in a linear structure without cycles.

- It is a part of Von-Neumann model of computation which consists of a single program counter, sequential execution and control flow which determines fetch, execution, commit order.
- Its main objective is to achieve the qualities of reuse and modifiability. And suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications.

There are three types of execution sequences between modules−

1. Batch sequential
2. Pipe and filter or non-sequential pipeline mode
3. Process control

1. Batch Sequential:

- Batch sequential compilation was regarded as a sequential process in 1970. It is a classical data processing model.
- In Batch sequential, separate programs are executed in order and the data is passed as an aggregate from one program to the next.
- It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.
- The main disadvantage of batch sequential architecture is that, it does not provide concurrency and interactive interface. It provides high latency and low throughput.

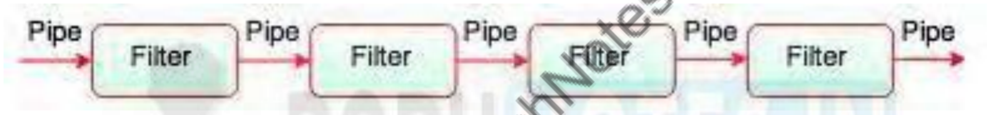


Figure 2.4: Flow of Batch Sequential Architecture

The above diagram shows the flow of batch sequential architecture. It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.

Advantages

- Provides simpler divisions on subsystems.
- Each subsystem can be an independent program working on input data and producing output data.

Disadvantages

- Provides high latency and low throughput.
- Does not provide concurrency and interactive interface.
- External control is required for implementation.

2. Pipe and Filter Architecture:

This approach lays emphasis on the incremental transformation of data by successive component. In this approach, the flow of data is driven by data and the whole system is decomposed into components of data source, filters, pipes, and data sinks. The connections between modules are data stream which is first-in/first-out buffer that can be stream of bytes, characters, or any other type of such kind. The main feature of this architecture is its concurrent and incremented execution.

Pipe represents-

- Pipe is a connector which passes the data from one filter to the next.
- Pipe is a directional stream of data implemented by a data buffer to store all data, until the next filter has time to process it.
- It transfers the data from one data source to one data sink.
- Pipes are the stateless data stream.

Filter represents-

- A filter is a component and an independent entity or independent data stream transformer or stream transducers.
- It transforms and refines input data or input data stream, processes it, and writes the transformed data stream over a pipe for the next filter to process.
- It works in an incremental mode, in which it starts working as soon as data arrives through connected pipe.
- It has interfaces from which a set of inputs can flow in and a set of outputs can flow out.

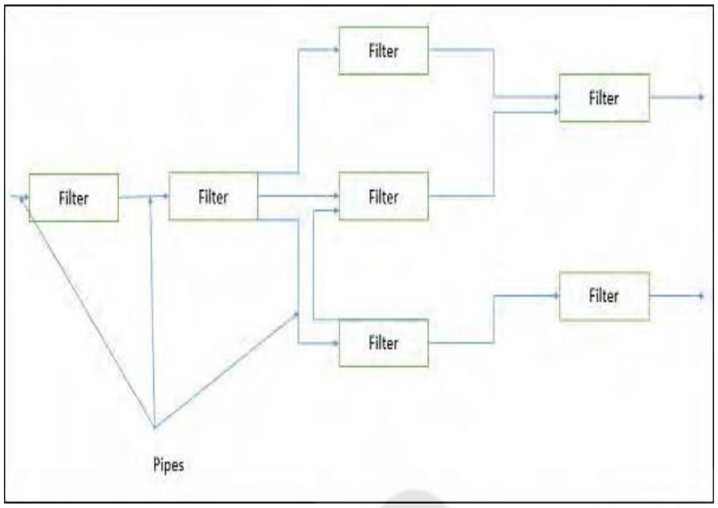


Figure 2.5: Pipes and Filter

There are two types of filters − active filter and passive filter.

Active filter: Active filter lets connected pipes to pull data in and push out the transformed data. It operates with passive pipe, which provides read/write mechanisms for pulling and pushing. This mode is used in UNIX pipe and filter mechanism.

Passive filter: Passive filter lets connected pipes to push data in and pull data out. It operates with active pipe, which pulls data from a filter and pushes data into the next filter. It must provide read/write mechanism.

All filters are the processes that run at the same time, it means that they can run as different threads, coroutines or be located on different machines entirely. Each pipe is connected to a filter and has its own role in the function of the filter. The filters are robust where pipes can be added and removed at runtime. Filter reads the data from its input pipes and performs its function on this data and places the result on all output pipes. If there is insufficient data in the input pipes, the filter simply waits.

Advantages:

- Provides concurrency and high throughput for excessive data processing.
- Provides reusability and simplifies system maintenance.
- Provides modifiability and low coupling between filters.
- Provides flexibility by supporting both sequential and parallel execution.

Disadvantages:

- Not suitable for dynamic interactions.
- Overhead of data transformation between filters.
- Does not provide a way for filters to cooperatively interact to solve a problem.
- Difficult to configure this architecture dynamically.

3. Process Control: Process Control Architecture is a type of Data Flow Architecture, where data is neither batch sequential nor pipe stream. In process control architecture, the flow of data comes from a set of variables which controls the execution of process.

This architecture decomposes the entire system into subsystems or modules and connects them. Types of Subsystems- A process control architecture would have a processing unit for changing the process control variables and a controller unit for calculating the amount of changes.

A controller unit must have the following elements −

- **Controlled Variable** − Controlled Variable provides values for the underlying system and should be measured by sensors. For example, speed in cruise control system.
- **Input Variable** − Measures an input to the process. For example, temperature of return air in temperature control system
- **Manipulated Variable** − Manipulated Variable value is adjusted or changed by the controller.
- **Process Definition** − It includes mechanisms for manipulating some process variables.
- **Sensor** − Obtains values of process variables pertinent to control and can be used as a feedback reference to recalculate manipulated variables.
- **Set Point** − It is the desired value for a controlled variable.
- **Control Algorithm** − It is used for deciding how to manipulate process variables.

Application Areas:

Process control architecture is suitable in the following domains −

- Embedded system software design, where the system is manipulated by process control variable data.
- Applications, which aim is to maintain specified properties of the outputs of the process at given reference values.
- Applicable for car-cruise control and building temperature control systems.
- Real-time system software to control automobile anti-lock brakes, nuclear power plants, etc.

Call-and Return Architecture: A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles.

1. Main program/subprogram architecture: In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components.
2. Remote procedure call architecture: In this, components of the main or subprogram architecture are distributed over a network across multiple computers.

Call and Return (Functional):

- Routines correspond to units of the task to be performed.
- Combined through control structures.
- Routines known through interfaces (argument list)

Advantages:	Disadvantages:
• Architecture based on well-identified parts of the task. • Change implementation of routine without affecting clients. • Reuse of individual operations.	• Must know which exact routine to change. • Hides role of data structure. • Bad support for extendibility.

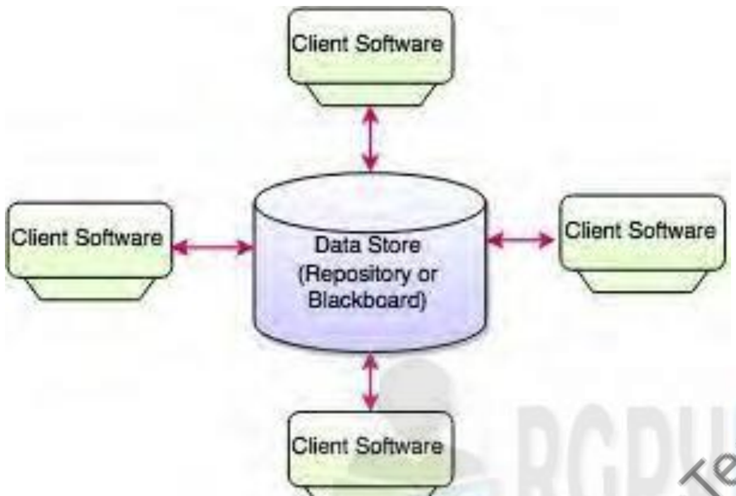
Call and Return (Object-Oriented):

- A class describes a type of resource and all accesses to it (encapsulation).
- Representation hidden from client classes.

Advantages:	Disadvantages:
• Change implementation without affecting clients. • Can break problems into interacting agents. • Can distribute across multiple machines or networks.	• Objects must know their interaction partners; when partner changes, clients must change. • Side effects: if A uses B and C uses B, then C's effects on B can be unexpected to A.

Data-Centered Architecture: Data Centered Architecture is also known as Database Centric Architecture. It is a layered process which provides architectural guidelines in data center development. This architecture is the physical and logical layout of the resources and equipment within a data center facility.

- In data-centred architecture, the data is centralized and accessed frequently by other components, which modify data.
- It 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.
- This architecture specifies how these devices will be interconnected and how physical and logical security workflows are arranged.



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.

Another example of data-centered architectures is the web architecture which has a common data schema (i.e. meta-structure of the Web) and follows hypermedia data model and processes communicate through the use of shared web-based data services.

Figure 2.6: Data-Centered Architecture

The above figure shows the architecture of Data-centred Architecture. In this architecture, the data is centralized and accessed frequently by other components which modify the data. The main purpose of data centred architecture is to achieve integrity of data.

There are two types of Components: Central Data and Data Accessor

Central Data:

- Central data provides permanent data storage.
- Central data represents the current state.

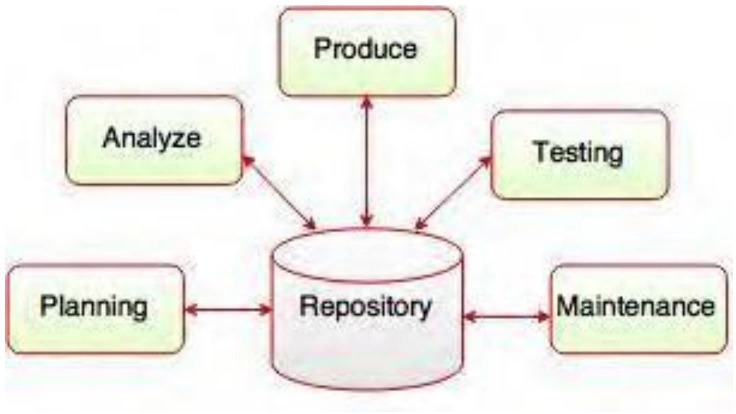
Data Accessor:

- Data accessor is a collection of independent components.
- It operates on the central data store, performs computations and displays the results.
- Communication can be done between the data accessors is only through the data store.

There are two categories which differentiates the architecture flow of control: Repository Architecture Style and Blackboard Architecture Style.

Repository Architecture Style:

- Repository architecture is a collection of independent components which operate on central data structure. It includes central data structure.
- Information System, Programming Environments, Graphical Editors, AI Knowledge Bases, Reverse Engineering System are the examples of Repository Architecture Style.



Repository architecture style is very important for data integration introduced in a variety of applications including software development, CAD etc.

In this architecture, the data store is passive, and the software clients or components of the data store are active which controls the logic flow and checks the data store for changes.

Figure 2.7: Repository Architecture style

Advantages:

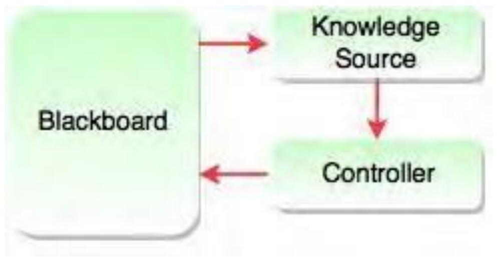
- Repository Architecture Style provides data integrity, backup and restore features.
- It reduces overhead of transient data between software components.
- It has an efficient way to store large amount of data.
- It has a centralized management which consists of backup, security and concurrency control.

Disadvantages:

- In repository architecture style, evolution of data is difficult and expensive.
- It has high dependency between data structure of data store and its software components or clients.

Blackboard Architecture Style:

- Blackboard architecture style is an artificial intelligence approach which handles complex problem, where the solution is the sum of its parts.
- Blackboard architecture style has a blackboard component which acts as a central data repository.
- It is used in location-locomotion, data interpretation and environmental changes for solving the problem.
- It is an approach to processing agent communication centrally.



There are major three components:

1. Knowledge Source
2. Blackboard
3. Control Shell

Figure 2.8: Blackboard Architecture Style

1. Knowledge Source:

- Knowledge source is also known as Listeners or Subscribers. They are distinct and independent units.
- Knowledge source solves the problem and aggregate partial results.

2. Blackboard:

- Blackboard is a shared repository of problems, solutions, suggestions and contributed information.
- It can be thought of as a dynamic library of an information to the current problem which have been published by other knowledge sources.

3. Control Shell:

- In Control shell, it controls the flow of problem-solving activity in the system.

- It manages the task and checks the work state.

Advantages:

- Blackboard architecture style provides concurrency which allows knowledge sources to work in parallel.
- This architecture supports experimentation for hypotheses and reusability of knowledge source components.
- It allows blackboard applications to adapt to changing requirements.
- It allows the new knowledge sources which can be developed and applied to the system without affecting on the existing system.

Disadvantages:

- Blackboard architecture style has the provision of tight dependency between the blackboard and knowledge source.
- It has difficulty in deciding for reasoning termination.
- It has an issue in synchronization of multiple agents.

Layered Architecture:

A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.

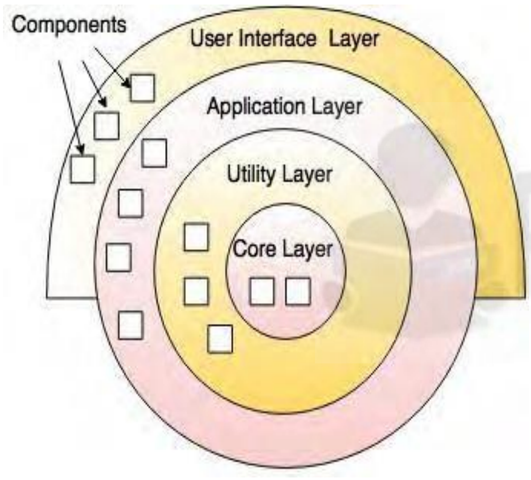


Figure 2.9: Layered architecture

- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS).
- Intermediate layers to utility services and application software functions.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer. The inner layers are application layer, utility layer and the core layer.

Agent Based Architecture: Now a days an increasing number of software projects are revised, restructured, and reconstructed in terms of software agents. Agent based software development a new way of analysis and synthesis of software system. Here software agents are new experimental embodiment of computer program.

- An agent in computer science refers to a software or other computational entities which has intelligence characteristics and can decide, and act based on its intelligence and other information taken from its environment. An agent usually acts on behalf of computer user.
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

Application of Agent based System:

- An agent become a part of distributed system, as a processing node.
- Agents for distributed sensing, and information retrieval and management.
- Agents for e-commerce, Agents for human-computer interfaces.
- Agents for virtual environments, Agents for social simulation.
- Agents for industrial systems management.

Micro-services Architecture: "Microservices" - yet another new term on the crowded streets of software architecture. The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms.

- The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.
- A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.
- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.

Benefits:

- Agility. Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process.
- Small, focused teams. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- Small code base. In a monolithic application, there is a tendency over time for code dependencies to become tangled, adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- Fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application.
- Scalability. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application.
- Data isolation. It is much easier to perform schema updates, because only a single microservice is affected.

Challenges: The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- Complexity: A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Development and testing: Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies.
- Network congestion and latency: The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem.
- Data integrity: With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- Versioning: Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so design carefully.

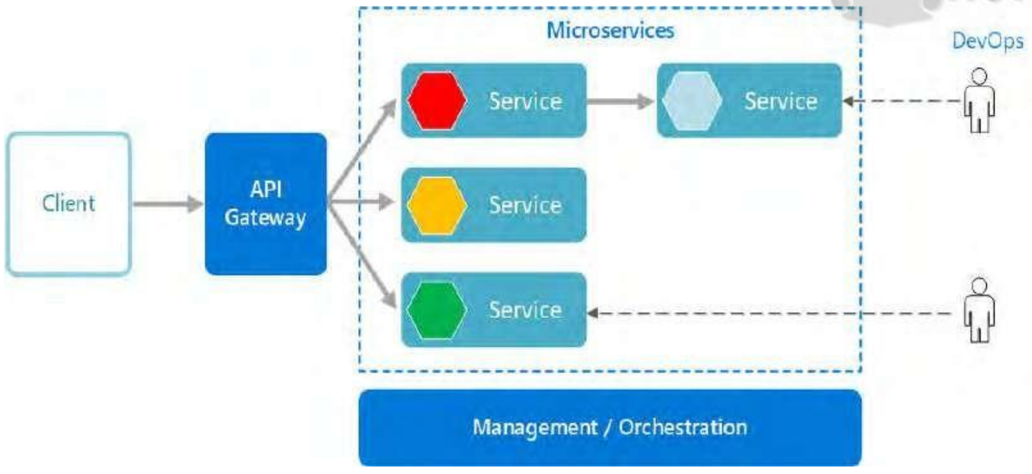


Figure 2.10: Micro-services architecture

Reactive Architecture: Reactive programming is an asynchronous programming paradigm, concerned with streams of information and the propagation of changes. Reactive Architecture is nothing more than the combination of reactive programming and software architectures. Also known as reactive systems, the goal is to make the system responsive, resilient, elastic, and message driven.

A Reactive system is an architectural style that allows multiple individual applications to merge as a single unit, reacting to its surroundings while aware of each other, and enable automatic scale up and down, load balancing, responsiveness under failure, and more.

Reactive Architecture can elastically scale in the face of varying incoming traffic. Scaling usually serves one of two purposes: either we need to scale out (by adding more machines) and up (by adding beefier machines), or we need to scale down, reducing the number of resources occupied by our application.

Reactive Architecture Benefits:

- Be responsive to interactions with its users.
- Handle failure and remain available during outages.
- Strive under varying load conditions.
- Be able to send, receive, and route messages in varying network conditions.
- Systems built as Reactive Systems are more flexible, loosely-coupled and scalable.

Representational State Transfer Architecture: Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed. These principles were first described in 2000 by Roy Fielding as part of his doctoral dissertation. REST is an alternative to SOAP and JavaScript Object Notation (JSON).

It is important to note that REST is a style of software architecture as opposed to a set of standards. As a result, such applications or architectures are sometimes referred to as *RESTful* or *REST-style* applications or architectures. Interaction in REST based systems happen through Internet's Hypertext Transfer Protocol (HTTP).

An application or architecture considered RESTful or REST-style is characterized by:

- State and functionality are divided into distributed resources.
- Every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet).
- The protocol is client/server, stateless, layered, and supports caching.

A Restful system consists of a:

- client who requests for the resources.
- server who has the resources.

Architectural Constraints of RESTful API: There are six architectural constraints which makes any web service are listed below:

1. Client-server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand (optional)

Client Server: Separation of concerns is the principle behind the client-server constraints.

Stateless: Statelessness means communication must be stateless in nature as in the client stateless server style, i.e. each request from client to server must contain all of the information necessary to understand the request and cannot take advantage of any stored context on the server.

Cacheable: In order to improve network efficiency, cache constraints are added to the REST style.

Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Uniform interface: The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components.

Layered system: The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.

Code on demand: The final addition to our constraint set for REST comes from the code-on-demand style.

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.

Steps Creating a RESTful service:

1. Define the domain and data.
2. Organize the data into groups.
3. Create URI to resource mapping.
4. Define the representations to the client (XML, HTML, CSS, …).
5. Link data across resources (connectedness or hypermedia).
6. Create use cases to map events/usage.
7. Plan for things going wrong.

Course Contents:

Unit 3. Software architecture implementation technologies: Software Architecture Description Languages (ADLs), Struts, Hibernate, Node JS, Angular JS, J2EE – JSP, Servlets, EJBs; middleware: JDBC, JNDI, JMS, RMI and CORBA etc. Role of UML in software architecture.

Unit-3

Software Architecture Description Languages (ADLs)

Architecture description languages (ADLs) are widely used to describe system architectures. Components and connectors are the main elements of ADLs and include rules and guidelines for well-formed architectures. The output of the architecture description process is the system architecture documents. It should describe how the system is structured into sub-systems and how each sub-system is structured into components. Each component is described based on many properties such as behaviour. The generated architecture description includes a static structure of components, their dynamic processes, interface modules, and the relationships among components. ADLs are important in system component design, since they affect system performance, robustness, disreputability and maintainability.

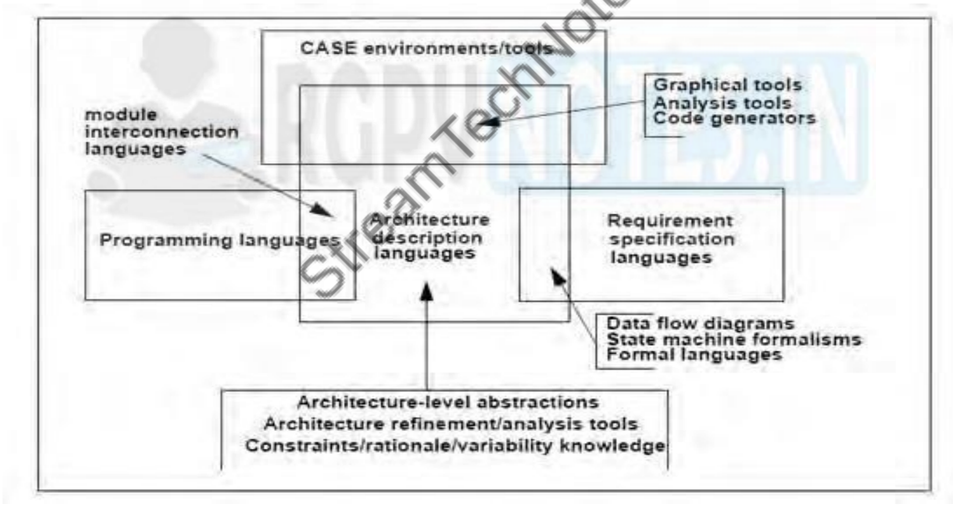

CASE environments/tools
Graphical tools
Analysis tools
Code generators
module interconnection languages
Programming languages
Architecture description languages
Requirement specification languages
Data flow diagrams
State machine formalisms
Formal languages
Architecture-level abstractions
Architecture refinement/analysis tools
Constraints/rationale/variability knowledge

Figure 3.1: Relation of ADL's to other notations and Tools

Figure shows the relation of ADLs to other, more familiar software engineering notations and tools.

1. Parts of traditional programming languages (e.g., Ada package specifications) represent module interconnection, which is also important for ADLs.
2. Requirements specification languages share some notations with ADLs.
3. Computer Aided Software Engineering (CASE) environments significantly overlap with ADLs, but there is much Computer Aided Software Engineering (CASE) functionality that is not part of architecture (e.g. test tools).

Characteristics:

- ADLs support the routine use of existing designs and components in new application systems.
- ADLs support the evaluation of an application system before it is built.

Elements of ADL:

1. **Component -** Primitive building block.
2. **Connector -** Mechanisms of combining components.

3. **Abstraction-** Rules for referring to the combination of components and connectors.

Important Properties of ADL

- **Composition –** composition of independent components and connections.
- **Abstraction –** need to describe design elements clearly and explicitly.
- **Reusability –** ability to reuse components, connectors in different architectural descriptions.
- **Configuration –** ability to understand and change the architectural structure.
- **Heterogeneity –** ability to combine multiple, heterogeneous architectural descriptions.
- **Analysis -** possible to perform rich and varied analysis of architectural description.

Advantage

- ADLs represent a formal way of representing architecture.
- ADLs are intended to be both human and machine readable.
- It support describing a system at a higher level than previously possible.
- ADLs permit analysis of architectures – completeness, consistency, ambiguity, and performance.
- It can support automatic generation of software systems.

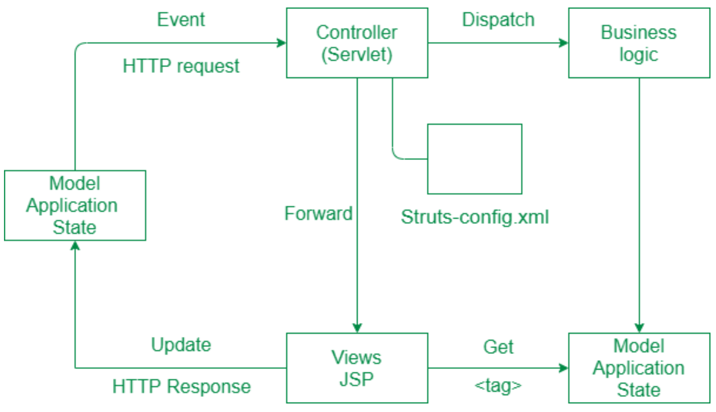
Disadvantage

- There is not universal agreement on what ADLs should represent, particularly as regards the behaviour of the architecture.
- Representations currently in use are relatively difficult to parse and are not supported by commercial tools.
- Most ADL work today has been undertaken with academic rather than commercial goals in mind.
- Most ADLs tend to be very vertically optimized toward a particular kind of analysis.

Struts

Struts is used to create a web applications based on servlet and JSP. It depend on the MVC (Model View Controller) framework and its application is a genuine web application. Struts are thoroughly useful in building J2EE (Java 2 Platform, Enterprise Edition) applications because struts takes advantage of J2EE design patterns. Struts follows these J2EE design patterns including MVC.
Struts consists of a set of own custom tag libraries. Struts are based on MVC framework which is pattern oriented and includes JSP custom tag libraries. Struts also supports utility classes.

Working of Struts



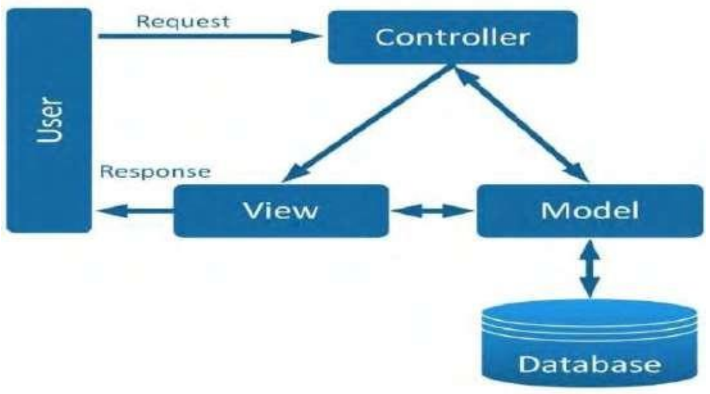
In the initialization phase, the controller rectifies a configuration file and used it to deploy other control layer objects. Struts configuration is form by these objects combined. The struts configuration defines among other things the action mappings for an application. Struts controller servlet considers the action mappings and routes the HTTP requests to other components in the framework. Request is first delivered to an action and then to JSP.

Figure 3.2: Working of Struts

The mapping helps the controller to change HTTP requests into application actions. The action objects can handle the request from and responds to the client (generally a web browser). Action objects have access to the applications controller servlet and also access to the servlet’s methods. When delivering the control, an action objects can indirectly forward one or more share objects, including java-beans by establish them in the typical situation shared by java servlets.

Model View Controller Architecture

The MVC design pattern consists of three modules model, view and controller.



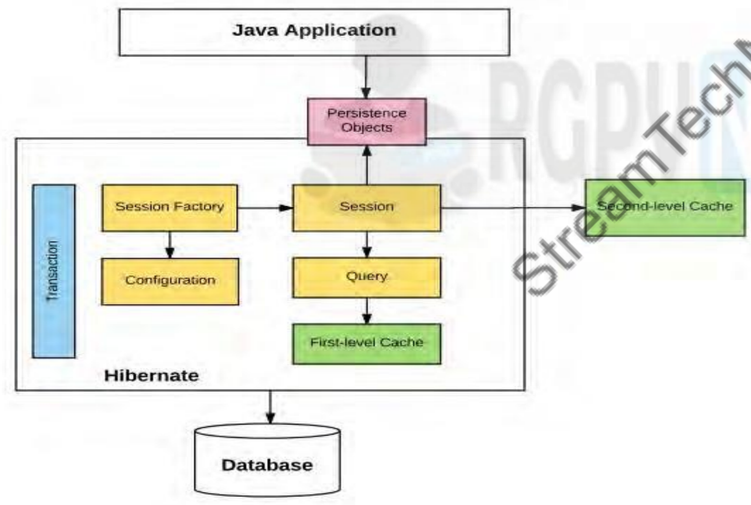
1. **Model** The model represents the state (data) and business logic of the application.
2. **View** The view module is responsible to display data i.e. it represents the presentation.
3. **Controller** The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.

Figure 3.3: MVC Architecture

Advantage of MVC architectures	**Disadvantage of MVC Architecture**
• Navigation control is centralized only controller contains the logic to determine the next page. • Easy to maintain, extend and test. • Better separation of concerns.	• We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

Hibernate

Hibernate Architecture summarizes the main building blocks in hibernate architecture.



Hibernate is an open source Java persistence framework project. It performs powerful object-relational mapping and query databases using HQL and SQL. Hibernate is a great tool for ORM mappings in Java. Hibernate's primary feature is mapping from Java classes to database tables, and mapping from Java data types to SQL data types. Hibernate also provides data query and retrieval facilities. It generates SQL calls and relieves the developer from the manual handling and object conversion of the result set.

Figure 3.4: Hibernate Architecture

1. **Configuration**: Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.
2. **Session Factory**: Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.
3. **Session**: This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the SessionFactory bean**.**
4. **Query**: It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including NamedQuery and CriteriaAPI**.**
5. **First-level cache:** It represent the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All

requests from the session object to the database must pass through the first level cache or session cache.

6. **Transaction**: enables you to achieve data consistency, and rollback in case something goes unexpected.
7. **Persistent objects**: These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate. They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.
8. **Second-level cache**: It is used to store objects across sessions. This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is EhCache.

Node.js

Node.js is an open-source server-side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.

Traditional Web Server Model

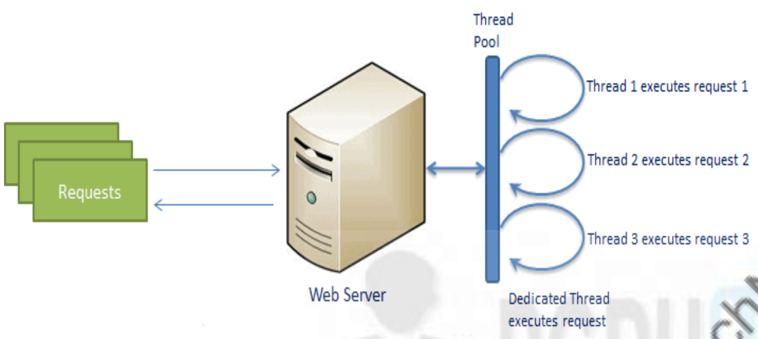

Thread Pool
Thread 1 executes request 1
Thread 2 executes request 2
Thread 3 executes request 3
Requests
Web Server
Dedicated Thread executes request

Figure 3.5: Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

Node.js Process Model

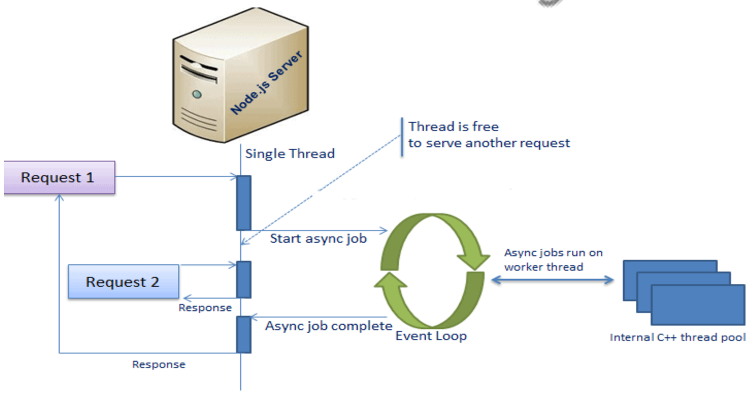


Figure 3.6: Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model. It runs in a single process and the application code runs in a single thread. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

Angular JS

AngularJS is a client side JavaScript MVC framework to develop a dynamic web application. It was originally started as a project in Google but now, it is open source framework. AngularJS is entirely based on HTML and JavaScript. It changes static HTML to dynamic HTML. It extends the ability of HTML by adding built-in attributes and components and also provides an ability to create custom attributes using simple JavaScript.

Example

```
<html>
<head>
<title>AngularJS First Application</title>
</head>
<body>
<h1>Sample Application</h1>
   <div ng-app = "">
      <p>Enter your Name: <input type = "text" ng-model = "name"></p>
      <p>Hello <span ng-bind = "name"></span>!</p>
    </div>
<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
    </script>
</body>
</html>
```

Output: Enter your Name:

Hello !

Advantages of AngularJS

- **Dependency Injection**: Dependency Injection specifies a design pattern in which components are given their dependencies instead of hard coding them within the component.
- **Two way data binding:** AngularJS creates a two way data-binding between the select element and the orderProp model. orderProp is then used as the input for the orderBy filter.
- **Testing:** Angular JS is designed in a way that we can test right from the start. So, it is very easy to test any of its components through unit testing and end-to-end testing.
- **Model View Controller:** In Angular JS, it is very easy to develop application in a clean MVC way. You just have to split your application code into MVC components i.e. Model, View and the Controller.

JSP (Java Server Page)

JSP technology is used to create dynamic web applications. JSP pages are easier to maintain then a Servlet, as servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags. Everything a Servlet can do, a JSP page can also do it. JSP enables us to write HTML pages containing tags, inside which we can include powerful Java programs.

Lifecycle of JSP

A JSP page is converted into Servlet in order to service requests. The translation of a JSP page to a Servlet is called Lifecycle of JSP. JSP Lifecycle is exactly same as the Servlet Lifecycle, with one additional first step, which is, translation of JSP code to Servlet code.

Following are the JSP Lifecycle steps:

1. Translation of JSP to Servlet code.
2. Compilation of Servlet to byte code.
3. Loading Servlet class.
4. Creating servlet instance.
5. Initialization by calling jspInit() method.
6. Request Processing by calling jspService() method Destroying by calling jspDestroy() method.

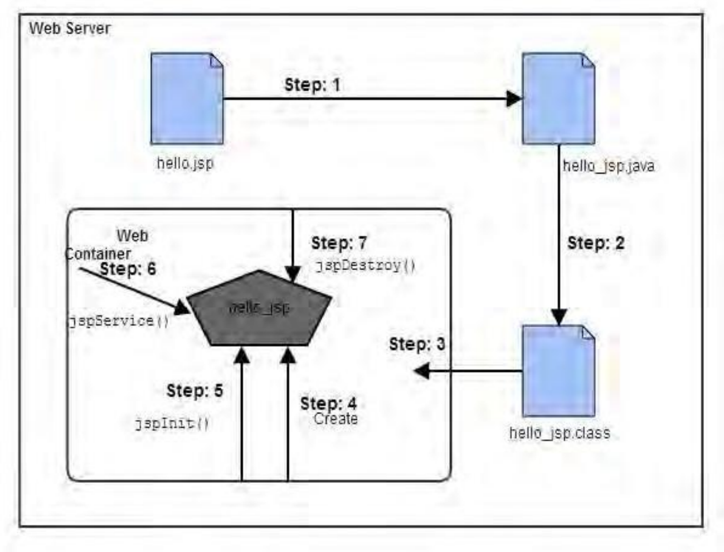


Figure 3.7: Lifecycle of JSP

Web Container translates JSP code into a servlet class source(.java) file, then compiles that into a java servlet class. In the third step, the servlet class bytecode is loaded using classloader. The Container then creates an instance of that servlet class.

The initialized servlet can now service request. For each request the Web Container call the jspService() method. When the Container removes the servlet instance from service, it calls the jspDestroy() method to perform any required clean up.

TAGS in JSP

Writing a program in JSP is nothing but making use of various tags which are available in JSP. In JSP we have three categories of tags- scripting elements, directives and standard actions.

SCRIPTING ELEMENTS: Scripting elements are basically used to develop preliminary programming in JSP such as, declaration of variables, expressions and writing the java code. Scripting elements are divided into three types- declaration tag, expression tag and scriptlet.

1. Declaration tag: Whenever we use any variables as a part of JSP we have to use those variables in the form of declaration tag i.e., declaration tag is used for declaring the variables in JSP page.

Syntax: <%! Variable declaration or method definition %>

When we declare any variable as a part of declaration tag these variables will be available as data members in the servlet and they can be accessed throughout the entire servlet. When we use any methods definition as a part of declaration tag they will be available as member methods in servlet and it will be called automatically by the servlet container as a part of service method.

For example-1:

<%! int a=10,b=30,c; %>

Example 2: <%! Int count () { return(a+b); } %>

2. Expression tag: Expression tags are used for writing the java valid expressions as a part of JSP page.

Syntax: <%=java valid expression %>

Whatever the expression we write as a part of expression tags that will be given as a response to client by the servlet container. All the expression we write in expression tag they will be placed automatically in out.println () method and this method is available as a part of service method. Expressions in the expression tag should not be terminated by semi-colon (;).

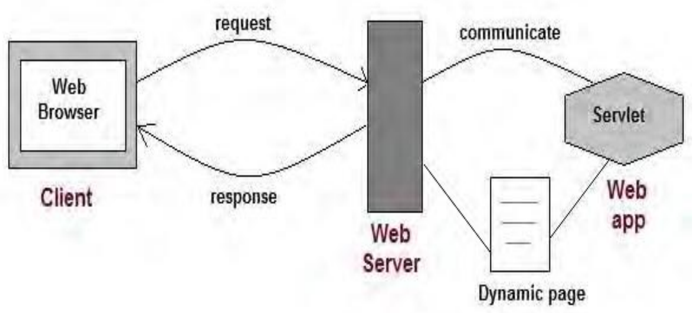
Example-1 <%! Int a=10, b=20 %> <%=a+b%>

The equivalent servlet code for the above expression tag is out.println (a+b); out is implicit object of JSPWriter class.

3. scriplet tag: scriplets are basically used to write a pure java code. Whatever the java code we write as a part of scriplet, that code will be available as a part of service () method of servlet.

Syntax: <%pure java code% >

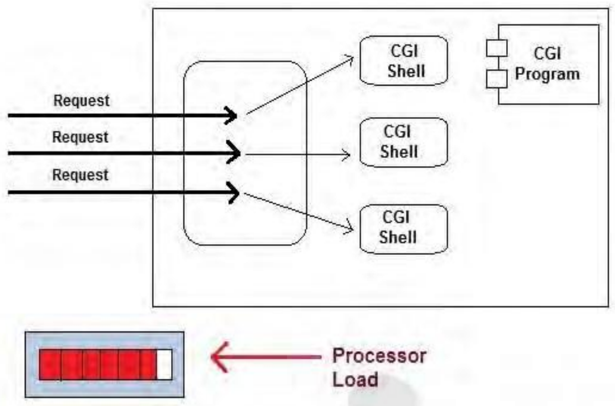
Servlet



Servlet Technology is used to create web applications. Servlet technology uses Java language to create web applications. Web applications are helper applications that resides at web server and build dynamic web pages. A dynamic page could be anything like a page that randomly chooses picture to display or even a page that displays the current time.

Figure 3.8: Servlet

CGI (Common Gateway Interface) – CGI programming was used to create web applications.



Here's how a CGI program works:

- User clicks a link that has URL to a dynamic page instead of a static page.
- The URL decides which CGI program to execute.
- Web Servers run the CGI program in separate OS shell. The shell includes OS environment and the process to execute code of the CGI program.
- The CGI response is sent back to the Web Server, which wraps the response in an HTTP response and send it back to the web browser.

Figure 3.9: Common Gateway Interface

Drawbacks of CGI programs

- High response time because CGI programs execute in their own OS shell.
- CGI is not scalable.
- CGI programs are not always secure or object-oriented.
- It is Platform dependent.

Because of these disadvantages, developers started looking for better CGI solutions. And then Sun Microsystems developed **Servlet** as a solution over traditional CGI technology.

Advantages of using Servlets

- Less response time because each request runs in a separate thread.
- Servlets are scalable.
- Servlets are robust and object oriented.
- Servlets are platform independent.

How a Servlet Application works

Web container is responsible for managing execution of servlets and JSP pages for Java EE application. When a request comes in for a servlet, the server hands the request to the Web Container. **Web Container** is responsible for instantiating the servlet or creating a new thread to handle the request. It is the job of Web Container to get the request and response to the servlet. The container creates multiple threads to process multiple requests to a single servlet.

Servlets don't have a main() method. Web Container manages the life cycle of a Servlet instance.

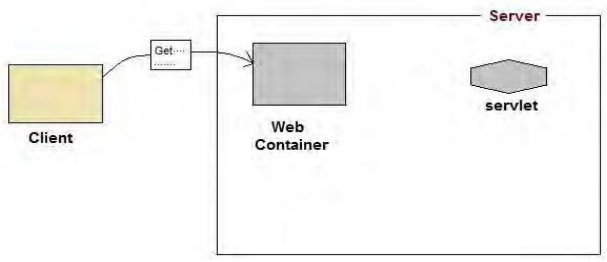


Figure 3.10 (a): User sends request

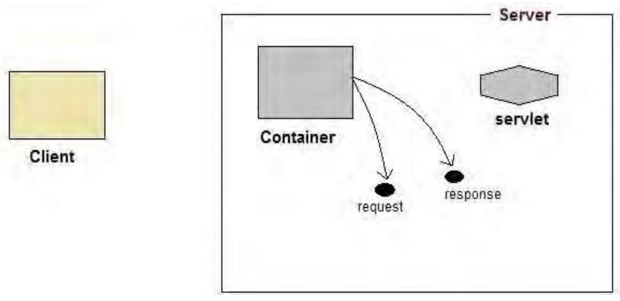


Figure 3.10 (b): Container find servlet

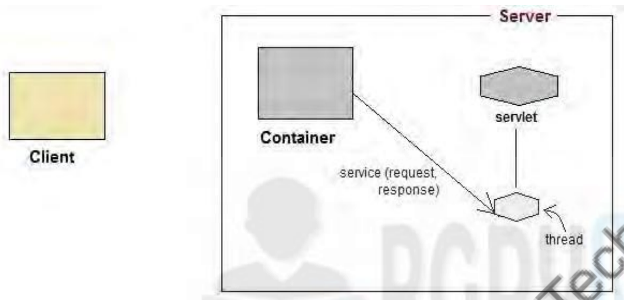


Figure 3.10 (c): Container allocates thread

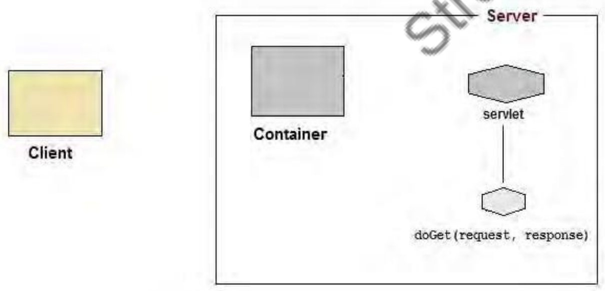


Figure 3.10 (d): Service method used

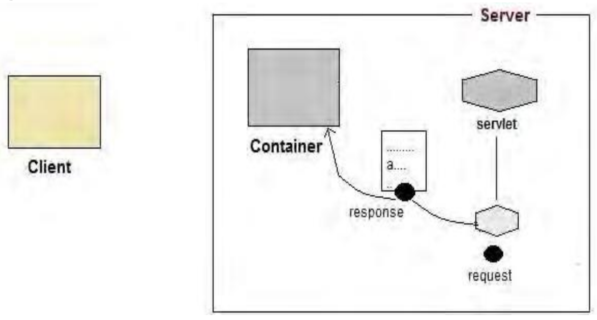


Figure 3.10 (e): Servlet Response

1. User sends request for a servlet by clicking a link that has URL to a servlet (Figure 3.10 (a)).
2. The container finds the servlet using **deployment descriptor** and creates two objects - **HttpServletRequest, HttpServletResponse** (Figure 3.10 (b)).
3. Then the container creates or allocates a thread for that request and calls the Servlet's service() method and passes the **request, response** objects as arguments (Figure 3.10 (c)).
4. The service () method, then decides which servlet method, doGet()or doPost() to call, based on **HTTP Request Method**(Get, Post etc) sent by the client (Figure 3.10 (d).
5. Then the Servlet uses response object to write the response back to the client(Figure 3.10 (e).
6. After the service() method is completed the thread dies. And the request and response objects are ready for garbage collection(Figure 3.10 (f).

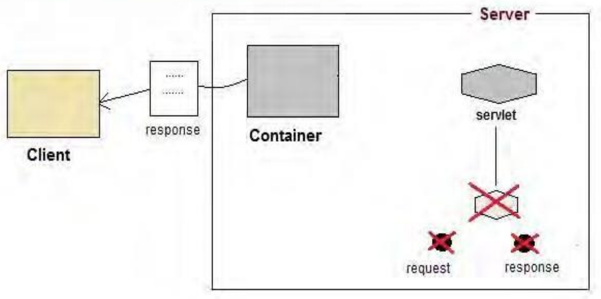
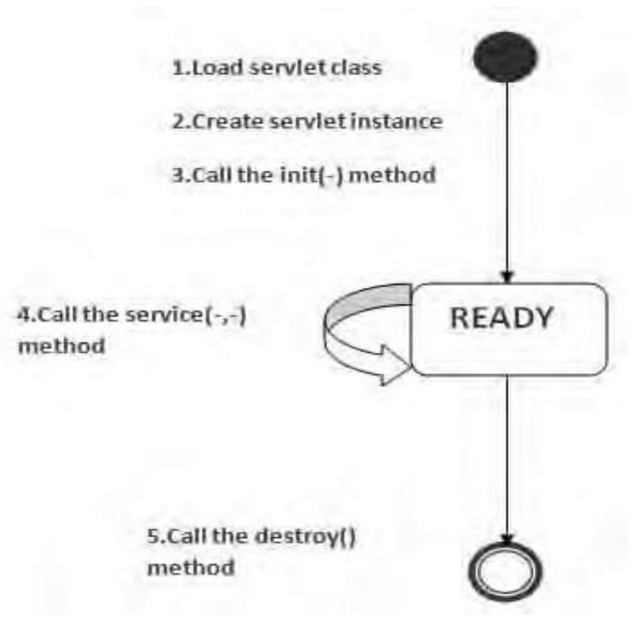


Figure 3.10 (f): Servlet Destroy

Servlet Life Cycle



1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

Figure 3.11: Lifecycle of Servlet

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

- The servlet is initialized by calling the **init()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

There are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy () method, it shifts to the end state.

1. **Servlet class is loaded** - The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.
2. **Servlet instance is created -** The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.
3. **init method is invoked-** The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:
   public void init(ServletConfig config) **throws** ServletException
4. **service method is invoked -** The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:
   public void service (ServletRequest request, ServletResponse response)
   throws ServletException, IOException
5. **Destroy method is invoked** - The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:
   public void destroy()

EJB (Enterprise Java Beans)

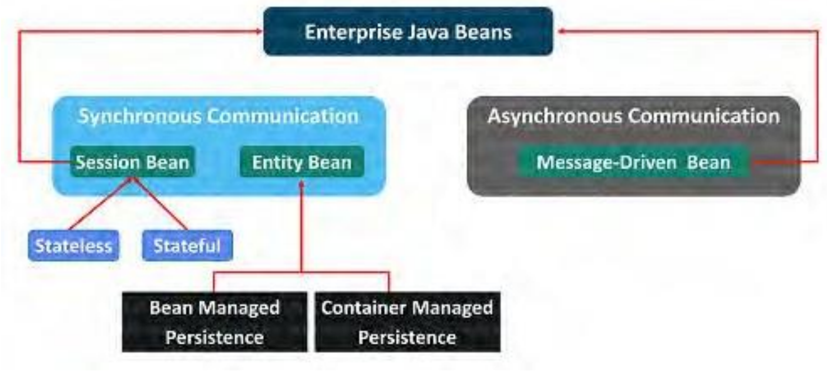


Figure 3.12: Enterprise Java Beans

EJB is server-side software that helps to summarize the business logic of a certain application. EJB was provided by sun micro-systems in order to develop robust, secure applications. The EJB enumeration is a subset of the Java EE enumeration.

Types of EJB - There are several types of enterprise Java beans-

1. Session bean
2. Entity bean
3. Message-driven beans

Advantages of EJB-

- EJB is an API, hence the application's build on EJB can run on Java EE web application server.
- The EJB developer focuses on solving business problems and business logic.
- Java beans are portable components that help the JAVA application assembler to formulate new applications for the already existing JavaBeans.
- EJB container helps in providing system-level services to enterprise Java beans.
- EJB contains business logic hence the front end developer can focus on the presentation of the client interface.

Disadvantages of EJB-

- The specification of EJB is pretty complicated and large.
- It creates costly and complex solutions.
- It takes time for development.
- Continuous revision of the specifications takes place.
- There are more complex cities than straight Java classes.

JDBC (Java Database Connectivity)

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational database. It is part of Java Standard Edition platform, from Oracle Corporation. It acts as a middle layer interface between java applications and database.

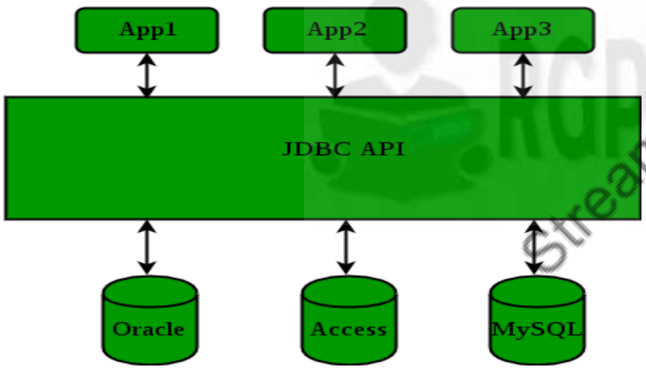


Figure 3.13: JDBC

The JDBC classes are contained in the Java Package java.sql and javax.sql.

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database.
3. Retrieve and process the results received from the database in answer to your query.

DBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver.
2. Type-2 driver or Native-API driver.
3. Type-3 driver or Network Protocol driver.
4. Type-4 driver or Thin driver.

Type-1 driver or JDBC-ODBC bridge driver- It uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

Advantage

- Easy to use.
- Can be easily connected to any database.

Disadvantage

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver nee.ds to be installed on the client machine.

Type 2 driver or Native-API driver - The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.

- Driver needs to be installed separately in individual client machines.
- Type-2 driver isn't written in java, that's why it isn't a portable driver.
- It is a database dependent driver.

Advantage

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Type-3 driver - The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- Switch facility to switch over from one database to another database.

Advantage

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantage

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.

Type-4 driver native protocol driver - This driver interact directly with database. It does not require any native database library that's why it is also known as Thin Driver.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

Advantage

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage

- Drivers depend on the Database.

Connectivity - There are 5 steps to connect any java application with the database using JDBC.

- Register the Driver class.
- Create connection.
- Create statement.
- Execute queries.
- Close connection.

JNDI

JNDI provides a common-denominator interface to many existing naming services, such as LDAP (Lightweight Directory Access Protocol) and DNS (Domain Name System). These naming services maintain a set of bindings, which relate names to objects and provide the ability to look up objects by name. JNDI allows the components in distributed applications to locate each other.

WebLogic Server JNDI - The WebLogic Server implementation of JNDI supplies methods that:

- Give clients access to the Web Logic Server naming services.
- Make objects available in the Web Logic namespace.
- Retrieve objects from the Web Logic namespace.

Each Web Logic Server cluster is supported by a replicated cluster wide JNDI tree that provides access to both replicated and pinned RMI and EJB objects. While the JNDI tree representing the cluster appears to the client as a single global tree, the tree containing the cluster-wide services is actually replicated across each Web Logic Server instance in the cluster.

JNDI in a Clustered Environment. Other Web Logic services can use the integrated naming service provided by Web Logic Server JNDI. For example, Web Logic RMI can bind and access remote objects by both standard RMI methods and JNDI methods.

JMS (Java Message Service)

JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled, reliable and asynchronous communication. It is also known as a messaging service. Messaging is a technique to communicate applications or software components. JMS is mainly used to send and receive message from one application to another.

Requirement of JMS - Generally, user sends message to application. But, if we want to send message from one application to another, we need to use JMS API. Consider a scenario, one application A is running in INDIA and another application B is running in USA. To send message from A application to B, we need to use JMS.

Advantage of JMS

- **Asynchronous:** To receive the message, client is not required to send request. Message will arrive automatically to the client.
- **Reliable:** It provides assurance that message is delivered.

Messaging Domains - There are two types of messaging domains in JMS-

1. Point-to-Point Messaging Domain- In PTP model, one message is delivered to one receiver only. Here, Queue is used as a message oriented middleware (MOM).The Queue is responsible to hold the message until receiver is ready. In PTP model, there is no timing dependency between sender and receiver.

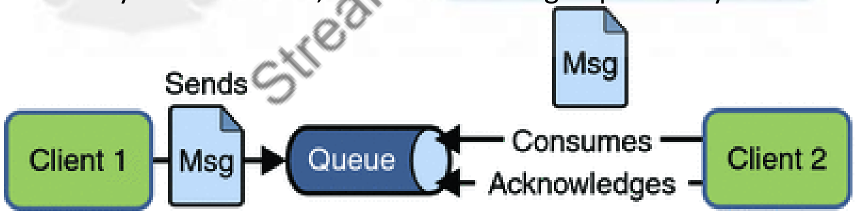


Figure 3.14: Point to Point Messaging Domain

2. Publisher Messaging Domain - In Pub/Sub model, one message is delivered to all the subscribers. It is like broadcasting. Here, Topic is used as a message-oriented middleware that is responsible to hold and deliver messages.

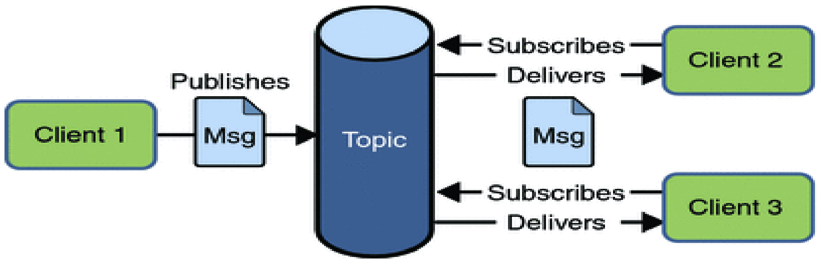


Figure 3.15: Publisher Messaging Domain

RMI (Remote Method Invocation)

Remote Method Invocation (RMI) is the standard for distributed object computing in Java. RMI enables an application to obtain a reference to an object that exists elsewhere in the network, and then invoke methods on that object as though it existed locally in the client's virtual machine. RMI specifies how distributed Java applications should operate over multiple Java virtual machines.

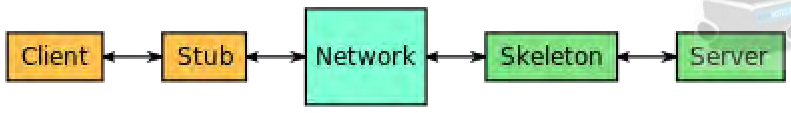


Figure 3.16: Implementation model of Java RMI

Features of Web Logic RMI

Table 3.1 Features of Web Logic implementation of RMI

S.No.	Features	Web logic RMI
1.	Overall performance	Enhanced by Web Logic RMI integration into the Web Logic Server framework, which provides underlying support for communications, scalability, management of threads and sockets, efficient garbage collection, and server-related support.
2.	Standards compliant	Compliance with the Java Platform Standard Edition 6.0 API Specification.
3.	Failover and Load balancing	Web Logic Server support for failover and load balancing of RMI objects.

CORBA (Common Object Request Broker Architecture)

CORBA is based on the Request-Response architecture. There is an object implementation on the server, which client requests to execute. The client and the server object implementation do not have any restrictions on the address space, for example the client and the server can exist in the same address space or can be located in separate address spaces on the same node or can be located on separate nodes altogether.

Object IDL interface and the Remote references are called Object References. There is a specification of the CORBA IDL Language and how it is mapped with other languages. It essentially provides a means for the interface definitions. The Proxy or a local representative for the client side is called the IDL stub; the server-side proxy is the IDL skeleton.

The proxy represents an object created on the client side, which is used for more functionality like support for Dynamic invocation. For marshalling the request and the response, the information is delivered in a canonical format defined by the IIOP protocol used for CORBA interoperability on the Internet.

IDL stub makes use of dynamic invocation interface for marshalling on the client side. Similarly on the server side, IDL Skeletons use the Dynamic Skeleton Interface for marshalling the information. The request (response) can also contain Object Reference as parameters; remote object can be passed by reference.

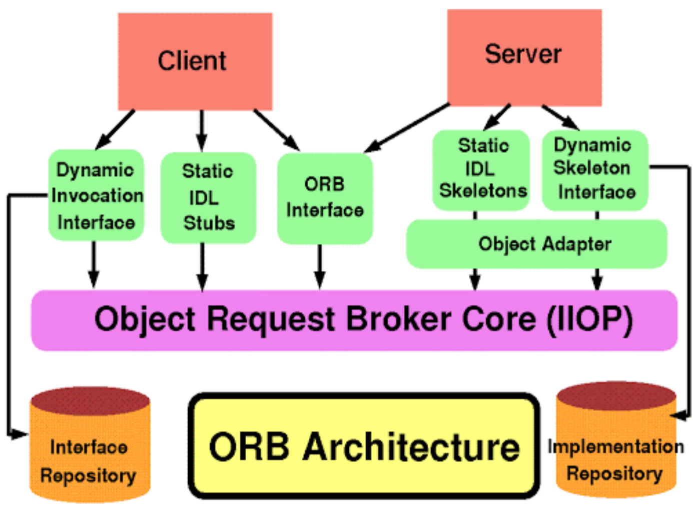

Client
Server
Dynamic Invocation Interface
Static IDL Stubs
ORB Interface
Static IDL Skeletons
Dynamic Skeleton Interface
Object Adapter
Object Request Broker Core (IIOP)
Interface Repository
ORB Architecture
Implementation Repository

Figure 3.17: CORBA Architecture

CORBA architecture follows a Broker pattern. The ORB is used for connecting the client and the server and it acts as a broker object. The IDL stub along with DII stub performs the client-side proxy and the POA IDL skeleton (along with the DSI skeleton used for dynamic delivery) does the Server-Side proxy. Many CORBA implementations use a direct communication between the client stubs and the Server Skeleton. The Interface Repository is used for introspection (particularly in case of a dynamic invocation and dynamic delivery). The Implementation Repository is used for reactivation of servers.

Role of UML in Software Architecture

A model is a simplified representation of the system. To visualize a system, we will build various models. The subset of these models is a view. Architecture is the collection of several views. The stakeholders (end users, analysts, developers, system integrators, testers, technical writers and project managers) of a project will be interested in different views.

Architecture can be best represented as a collection five views:

Table 3.2: Description of Different View in Software Architecture

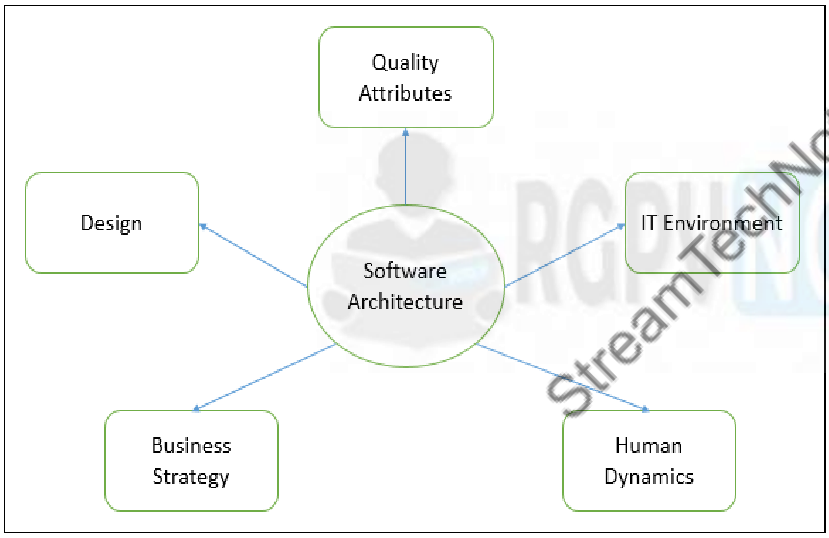
View	**Stakeholder**	**Static Aspects**	**Dynamic Aspects**
Use Case View	End users Analysts Tester	Use case Diagrams	Interaction Diagrams State chart Diagrams Activity Diagrams
Design View	End users	Class Diagrams	Interaction Diagrams State chart Diagrams Activity Diagrams
Implementation View	Programmers Configuration Managers	Component Diagrams	Interaction Diagrams State chart Diagrams Activity Diagrams
Process View	Integrator	Class Diagram (Active Classes)	Interaction Diagrams State chart Diagrams Activity Diagrams
Deployment View	System Engineer	Deployment Diagrams	Interaction Diagrams State chart Diagrams Activity Diagrams

Course Contents:

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

Unit-4

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



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

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

Figure 4.1: Software Architecture Design factors

Software Architecture

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

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

Software Design

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

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

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

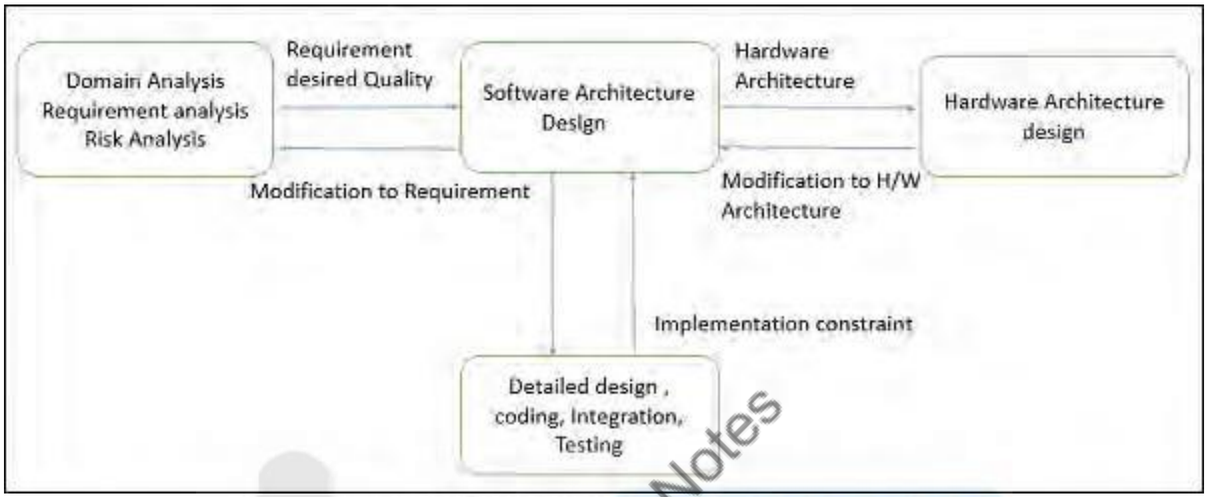


Figure 4.2: Software Architecture Design Task

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

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

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

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

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

Table 4.1: Common Software Architectural Design

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

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

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

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

Understand the Problem

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

Identify Design Elements and their Relationships

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

Evaluate the Architecture Design

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

Transform the Architecture Design

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

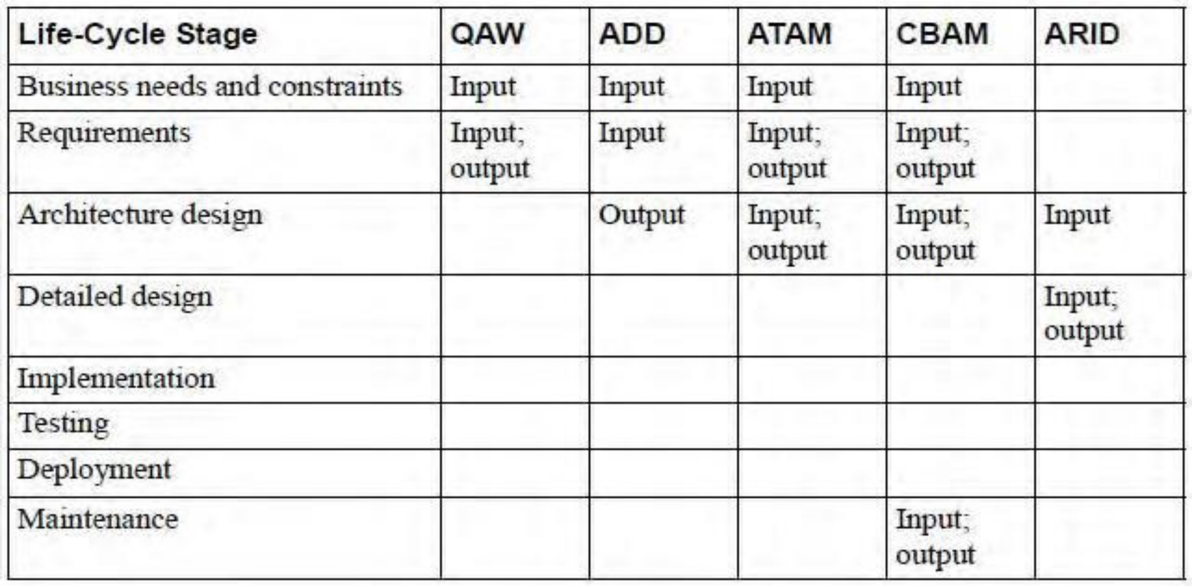
Key Design Principles: Following are the design principles to be considered for minimizing cost, maintenance requirements, and maximizing extendibility, usability of architecture –

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

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

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

Table 4.2: Methods and Life-Cycle Stages



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

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

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

Inputs to the CBAM: The inputs include-

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

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

1. Collate scenarios: Collate the scenarios elicited during the ATAM exercise and give the stakeholders the chance to contribute new ones. Prioritize these scenarios based on satisfying the business goals of the system and choose the top one-third for further study.
2. Refine scenarios: Refine the scenarios, focusing on their stimulus/response measures. Elicit the worst, current, desired, and best-case quality-attribute-response level for each scenario.
3. Prioritize scenarios: Allocate 100 votes to each stakeholder to be distributed among the scenarios, where the stakeholder's voting is based on considering the desired response value for each scenario. Total the votes and choose the top 50% of the scenarios for further analysis. Assign a weight of 1.0 to the highest rated scenario. Relative to that scenario, assign the other scenarios a weight that becomes the number used in calculating the architectural strategy's overall benefit. Make a list of the quality attributes that concern the stakeholders.

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

Outputs of the CBAM: Outputs include

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

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

Inputs to the ATAM: Inputs include the-

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

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

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

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

Outputs of the ATAM: Outputs include a-

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

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

Inputs to ARID: Inputs include-

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

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

Output of ARID: The output includes a list of "issues and problems" preventing successful use of the design.

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

Inputs to ADD: Inputs include a-

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

Steps of ADD: This method includes the following steps:

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

2. Refine the module according to the following steps:

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

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

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

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

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

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

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

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

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

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

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

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

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

Domain specific software architecture comprises:

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

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

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

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

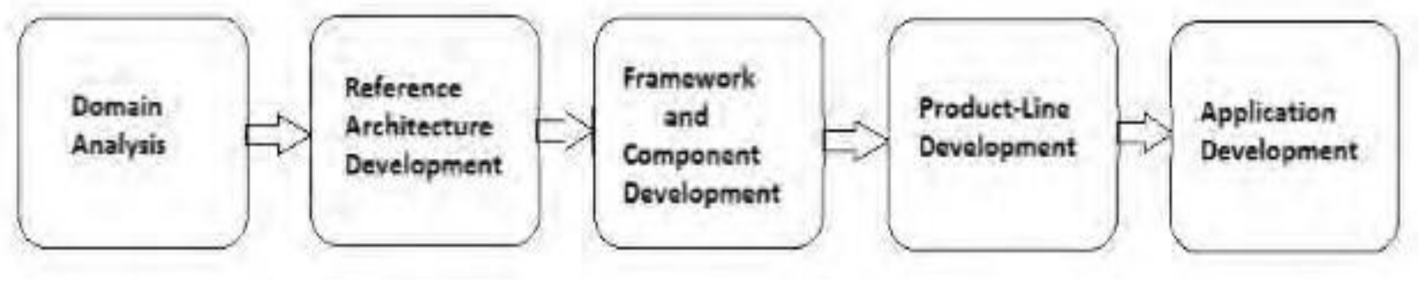

Domain Analysis
Reference Architecture Development
Framework and Component Development
Product-Line Development
Application Development

Figure 4.3: The overview of the DSSA process

Course Contents:

Unit 5. Software Architecture documentation: principles of sound documentation, refinement, context diagrams, variability, software interfaces. Documenting the behavior of software elements and softwaresystems, documentation package using a seven-part template.

Unit-5

Software Architecture Documentation

The software architecture document provides a comprehensive overview of the architecture of the software system or may be consider a map of the software. We can use it to see, how the software is structured. It mainly helps to understand the software's modules and components without digging into the code. It serves as a tool to communicate with others like developers and non-developers—about the software.

Below are the three primary goals for architectural documentation:

- **Knowledge sharing-** It is suitable to transfer knowledge between people working in different functional areas of the project, as well as for knowledge transfer to new participants.
- **Communication-** Documentation is the starting point for interaction between different stakeholders. It helps to share the ideas of the architect to the developers.
- **Analyses-** Documentation is also a starting point for future architectural reviews of the project.

Two approaches to create software architecture

There are two well-known approaches- top-down and bottom-up to create software and its architecture. When we start from general idea and iteratively decompose system into smaller components, this is called the top-down approach.

Alternatively, we can start with defining project goals and iteratively add more and more small pieces of functionality to the system. All those small elements and their relations will compose system with its architecture, this is called the bottom-up approach.

Principles of Sound Documentation

Principle 1: Write from the point of view of the reader

The Web-based design includes a component to support role-based user login and access control. Within the context of a particular document, once a stakeholder logs in, the system generates a view tailored to that stakeholder's needs and access rights. The view might provide a part of the document as a Web form to the architect and that same part as a Portable Document Format (PDF) file for a developer; a junior architect is likely to benefit from guidance at a level that would cause a senior architect frustration. Clickable pop-up windows show tips relevant to the role of the currently logged-in stakeholder. For instance, if a junior architect is creating a primary presentation, the tip might provide guidance on organizational standards or notations, while the tip for a developer would provide the semantics of the notation used by the architect.

Principle 2: Avoid unnecessary repetition

The database back end of the system allows each piece of information to be created and maintained as individual files and accessed by multiple users in a variety of contexts. Hyperlinking allows the illusion of redundancy while completely removing its need. A glossary captures definitions in a single place and provides a readily accessible and consistent reference for all stakeholders. Similarly, an acronym list is created and is readily accessible at any time. Diagrams and other files can be created once and referenced from multiple locations throughout the document.

Principle 3: Avoid ambiguity

While the use of natural language always allows for varied interpretation, the design reinforces the need for precision and clarity. In addition, the use of Unified Modelling Language (UML) and other more formal languages is supported by way of file upload.

Principle 4: Use a standard organization

A template, by nature, enforces the use of a standard organization. With this system's design, we use Web forms as templates that provide rigid structure to the document's various elements.

Principle 5: Record rationale

The design provides entries specific to recording rationale in appropriate places and encourages their use by querying the author if this section of the form is not used.

Principle 6: Keep documentation current but not too current

Keeping the documentation current is simple in Web-based documentation; however, issues associated with "too current" are more pressing in this type of environment. The Web-based documentation system is best backed with a configuration management system that makes available various levels of currency based on the user's role.

Principle 7: Review documentation for fitness of purpose

The Web-based design supports easy and immediate feedback on both the documentation's form and content as email links on every Web page.

Refinement

Refinement is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed.

At the early steps of the refinement process the software engineer does not necessarily know how the software will perform what it needs to do. This is determined at each successive refinement step, as the design and the software is elaborated upon.

Refinement can be seen as the compliment of abstraction. Abstraction is concerned with hiding lower levels of detail; it moves from lower to higher levels. Refinement is the movement from higher levels of detail to lower levels. Both concepts are necessary in developing software.

Context Diagram

The system context diagram (also known as a level 0 DFD) is the highest level in a data flow diagram and contains only one process, representing the entire system, which establishes the context and boundaries of the system to be modelled. It identifies the flows of information between the system and external entities (i.e. actors). A context diagram is typically included in a requirements document. It must be read by all project stakeholders and thus should be written in plain language, so the stakeholders can understand items.

Purpose of a System Context Diagram

The objective of the system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of systems requirements and constraints. A system context diagram is often used early in a project to determine the scope under investigation. Thus, within the document. A system context diagram represents all external entities that may interact with a system. The entire software system is shown as a single process. Such a diagram pictures the system at the centre, with no details of its interior structure, surrounded by all its External entities, interacting systems, and environments.

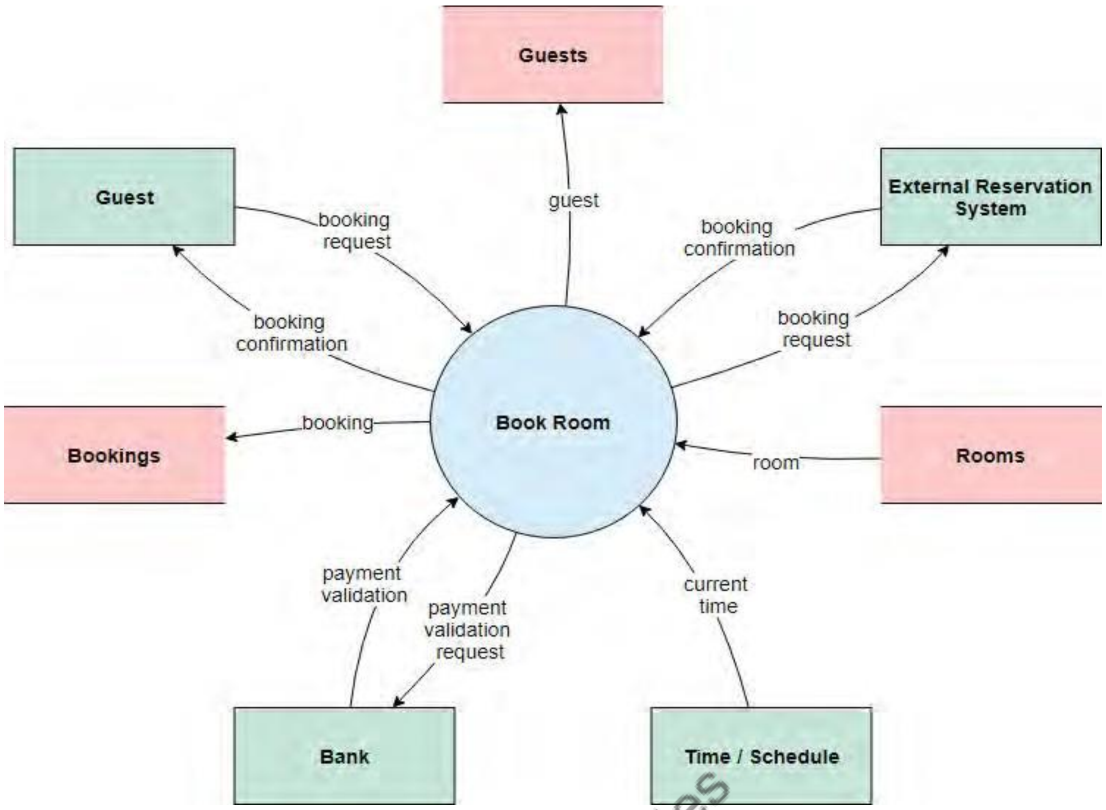


Figure 5.1: Example of context diagram for hotel reservation system

Variability

Variability is the ability of a software artifact to be changed (e.g., configured, customized, extended, adapted) for a specific context, in a pre-planned manner. This means, variability can be understood as "anticipated change".

- It helps manage commonalities and differences between systems.
- It supports the development of different versions of software by allowing the implementation of variants within systems.
- It facilitates planned reuse of software artifacts in multiple products,
- It allows the delay of design decisions to the latest point that is economically feasible.
- It supports runtime adaptations of deployed systems.

Reasons for variability include the deferral of design decisions, multiple deployment / maintenance scenarios.

Software Interfaces

Software interfaces (programming interfaces) are the languages, codes, and messages that programs use to communicate with each other and to the hardware. An interface is a boundary across which two independent entities meet and interact or communicate with each other. The characteristics of an interface depend on the view type of its element. If the element is a component, the interface represents a specific point of its potential interaction with its environment. If the element is a module, the interface is a definition of services. There is a relation between these two kinds of interfaces, just as there is a relation between components and modules.

By the element's environment, we mean the set of other entities with which it interacts. We call those other entities actors: An element's actors are the other elements, users, or systems with which it interacts. In general, an actor is an abstraction for external entities that interact with the system. Interaction is part of the element's interface. Interactions can take a variety of forms. Most involve the transfer of control

and/or data. Some are supported by standard programming-language constructs, such as local or remote procedure calls (RPCs), data streams, shared memory, and message passing.
Some principles about interfaces:

- All elements have interfaces. All elements interact with their environment.
- An element's interface contains view-specific information.
- Interfaces are two ways.
- An element can have multiple interfaces.
- An element can interact with more than one actor through the same interface.

An interface is documented with an interface specification: An interface specification is a statement of what an architect chooses to make known about an element in order for other entities to interact or communicate with it.

Documenting the behaviour of software elements and software systems

A software component simply cannot be differentiated from other software elements by the programming language used to implement the component. The difference must be in how software components are used. Software comprises many abstract, quality features, that is, the degree to which a component or process meets specified requirement for example, an efficient component will receive more use than a similar, inefficient component. It would be inappropriate, however, to define a software component as "an efficient unit of functionality." Elements that comprise the following definition of the term software component are described in the "Terms" sidebar. A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model. A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

The main goal of effective documentation is to ensure that developers and stakeholders are headed in the same direction to accomplish the objectives of the project. To achieve them, plenty of documentation types exist.

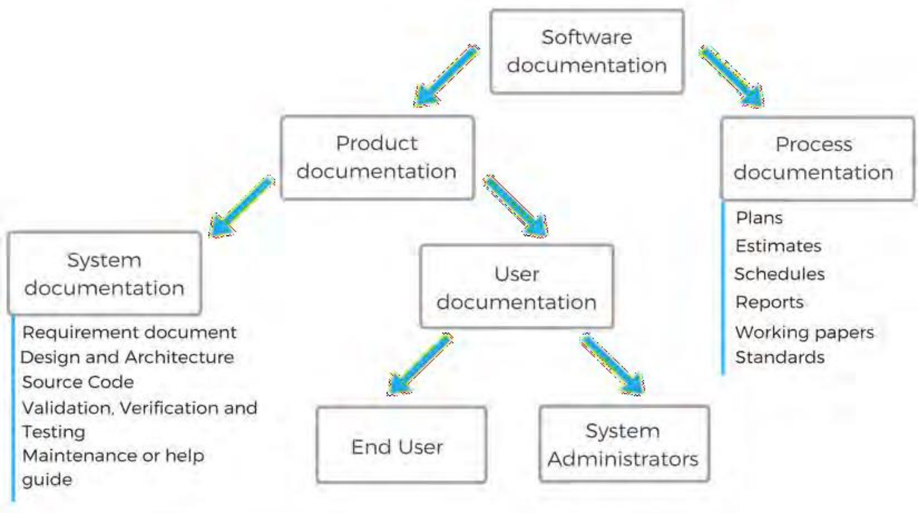


Figure 5.2: Types of Documentation

All software documentation can be divided into two main categories:

- Product documentation
- Process documentation

Product documentation describes the product that is being developed and provides instructions on how to perform various tasks with it. Product documentation can be broken down into:

- System documentation and
- User documentation

System documentation represents documents that describe the system itself and its parts. It includes requirements documents, design decisions, architecture descriptions, program source code, and help guides.

User documentation covers manuals that are mainly prepared for end-users of the product and system administrators. User documentation includes tutorials, user guides, troubleshooting manuals, installation, and reference manuals.

Process documentation represents all documents produced during development and maintenance that describe well, process. The common examples of process documentation are project plans, test schedules, reports, standards, meeting notes, or even business correspondence.

The main difference between process and product documentation is that the first one record the process of development and the second one describes the product that is being developed.

Documentation Package using a seven-part template

Each ECS view is presented as a number of related view packets. A view packet is a small, relatively self-contained bundle of information about the system or a particular part of the system, rendered in the languageelement and relation typesof the view to which it belongs. Two view packets are related to each other as either parent/childbecause one shows a refinement of the information in the otheror as siblingsbecause both are children of another view packet.

1. A primary presentation that shows the elements and their relationships that populate the view packet. The primary presentation contains the information important to convey about the system, in the vocabulary of that view, first.
   The primary presentation is usually graphical. If so, the presentation will include a key that explains the meaning of every symbol used. The first part of the key identifies the notation, If a defined notation is being used, the key will name it and cite the document that defines it or defines the version of it being used. If the notation is informal, the key will say so and proceed to define the symbols and the meaning, if any, of colours, position, or other information-carrying aspects of the diagram.
2. Element catalog detailing at least those elements depicted in the primary presentation and others that were omitted from the primary presentation. Specific parts of the catalog include-
   - Elements and their properties- This section names each element in the view packet and lists the properties of that element. For example, elements in a module decomposition view have the property of "responsibility," an explanation of each module's role in the system, and elements in a communicating-process view have timing parameters, among other things, as properties.
   - Relations and their properties -Each view has a specific type of relation that it depicts among the elements in that view. However, if the primary presentation does not show all the relations or if there are exceptions to what is depicted in the primary presentation, this section will record that information.
   - Element interface - An interface is a boundary across which elements interact or communicate with each other. This section is where element interfaces are documented.
   - Element behaviour- Some elements have complex interactions with their environment and for purposes of understanding or analysis, the element's behaviour is documented.
3. Context diagram showing how the system depicted in the view packet relates to its environment.
4. Variability guide showing how to exercise any variation points that are a part of the architecture shown in this view packet.
5. Architecture background explaining why the design reflected in the view packet came to be. Itexplains why the design is as it is and to provide a convincing argument that it is sound. Architecture background includes-

- Rationale- It explains why the design decisions reflected in the view packet were made and gives a list of rejected alternatives and why they were rejected. This will prevent future architects from pursuing dead ends in the face of required changes.
- Analysis results -This document the results of analyses that have been conducted, such as the results of performance or security analyses, or a list of what would have to change in the face of a particular kind of system modification.
- Assumptions - This document any assumptions the architect made when crafting the design. Assumptions are generally about either environment or need.
- Environmental assumptions document what the architect assumes is available in the environment that can be used by the system being designed. They also include assumptions about invariants in the environment.

6. Other information – It includesno architectural and organization-specific information. "Other information" will usually include management or configuration control information, change histories, bibliographic references or lists of useful companion documents, mapping to requirements, and the like.
7. Related view packets - It will name other view packets that are related to the one being described in a parent/child or sibling capacity.