RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Software Engineering**

Subject Code: **CS-403**

Semester: **4th**

UNIT-III

SOFTWARE DESIGN PROCESS:

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

The architectural design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system.

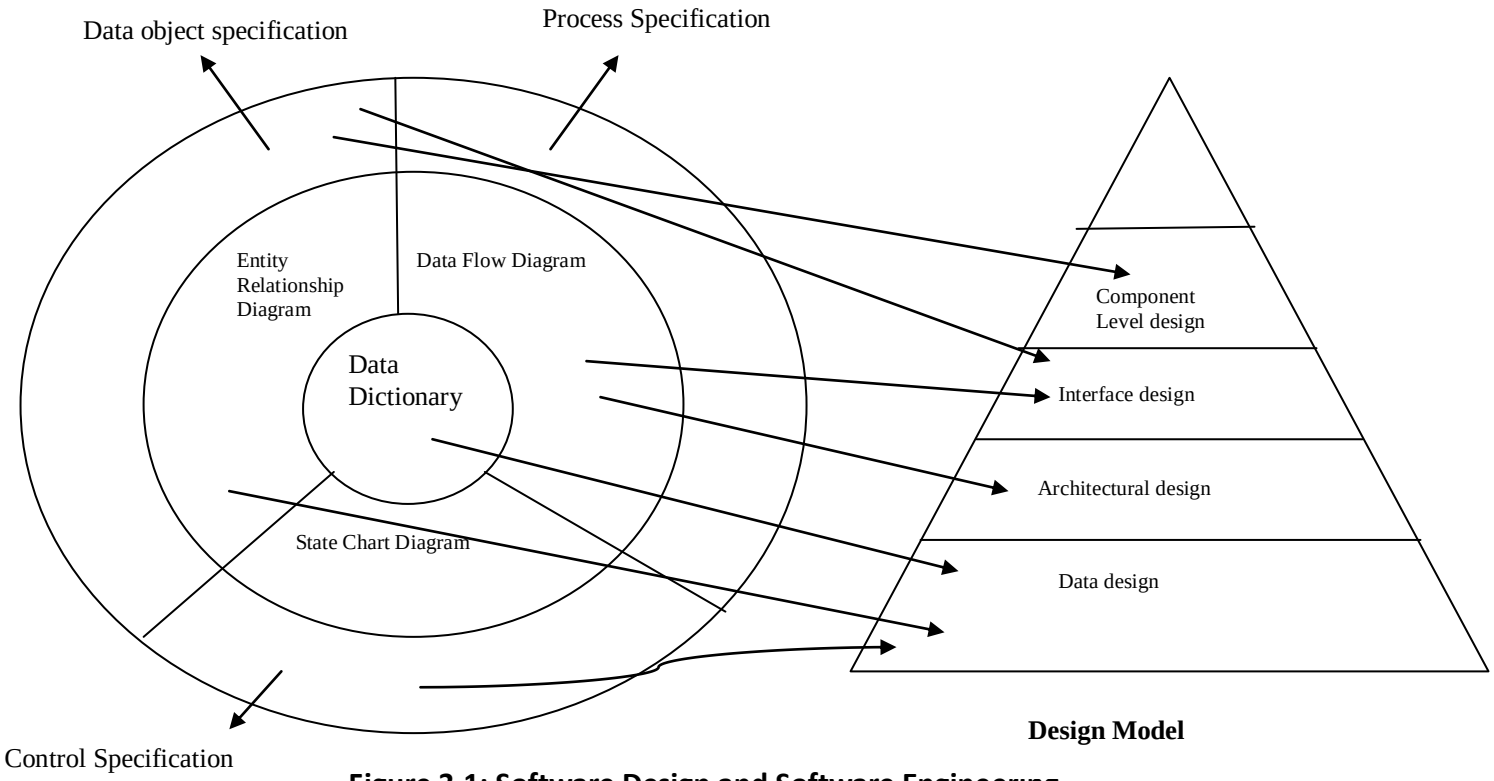


Figure 3.1: Software Design and Software Engineering

The Design Process:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction at level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower level of abstraction. These can still be trace or requirements, but the connection is subtler.

DESIGN CONCEPTS AND PRINCIPLES:

Design Principles: -

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide

guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process.

Principles for software design, which have been adapted and extended in the following list:

- The design process should not suffer from "tunnel vision."
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
- That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Design concepts:

Following issues are considered while designing the software.

- **Abstraction:** "Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level detail. At the highest level of abstraction a solution is stated in broad terms using the language of the problem environment. At lower level, a procedural orientation is taken. At the lowest level of abstraction the solution is stated in a manner that can be directly implemented.
  Types of abstraction: 1. Procedural Abstraction 2. Data Abstraction
- **Refinement:** Process of elaboration. Refinement function defined at the abstract level, decompose the statement of function in a stepwise fashion until programming language statements are reached.
- **Modularity:** software is divided into separately named and addressable components called modules. Follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces.

SOFTWARE MODELING AND UML:

Software modeling:

Software models are ways of expressing a software design. Usually some sort of abstract language or pictures are used to express the software design. For object-oriented software, an object modeling language such as UML is used to develop and express the software design.

Unified Modeling Language (UML):

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the Unified Modeling Language (UML), has become widely used throughout the industry. UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules. In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams. The following views are present in UML:

- **User model view.** This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective.

- **Structural model view.** Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.
- **Behavioral model view.** This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the inter actions or collaborations between various structural elements described in the user model and structural model views.
- **Implementation model view.** The structural and behavioral aspects of the system are represented as they are to be built.
- **Environment model view.** The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

UML Diagram Types:

There are several types of UML diagrams:

- **User model view represent through**:
  Use-case Diagram: Shows actors, use-cases, and the relationships between them.
- **Structural model view represents through**
  Class Diagram: Shows relationships between classes and pertinent information about classes themselves.

  Object Diagram: Shows a configuration of objects at an instant in time.
- **Behavioral model view represents through**
  Interaction Diagrams: Show an interaction between groups of collaborating objects.

  Two types: **Collaboration diagram** and **sequence diagram**

  Package Diagrams: Shows system structure at the library/package level.

  State Diagram: Describes behavior of instances of a class in terms of states, stimuli, and transitions.

  Activity Diagram: Very similar to a flowchart— shows actions and decision points, but with the ability to accommodate concurrency.
- **Environment model view represent through**
  Deployment Diagram: Shows configuration of hardware and software in a distributed system.
- **Implementation model view represent through**
  Component Diagram: It shows code modules of a system. This code module includes application program, ActiveX control, Java beans and back end databases. It representing interfaces and dependencies among software architect.

ARCHITECTURAL DESIGN:

Establishing the overall structure of a software system.

Objectives:

- To introduce architectural design and to discuss its importance
- To explain why multiple models are required to document software architecture to describe types of architectural model that may be used.
- A high-level model of a thing: -
- Describes critical aspects of the thing Understandable to many stakeholders
- Allows evaluation of the thing's properties before it is built
- Provides well understood tools and techniques for constructing the thing from its blueprint.

ARCHITECTURAL VIEWS AND STYLES:

Architectural Styles:

The builder has used an architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a pattern for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the pattern—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles.

Each style describes a system category that encompasses

- A set of components (e.g., a database, computational modules) that perform a function required by a system;
- A set of connectors that enable "communication, co-ordinations and cooperation" among components.
- Constraints that define how components can be integrated to form the system; and
- Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts. In the section that follows, we consider commonly use d architectural patterns for software.

The commonly used architectural styles are:

- **Data-centered Architectures:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A typical Data-centered style. Clients of the are accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.

  Data-centered architectures promote integrity. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

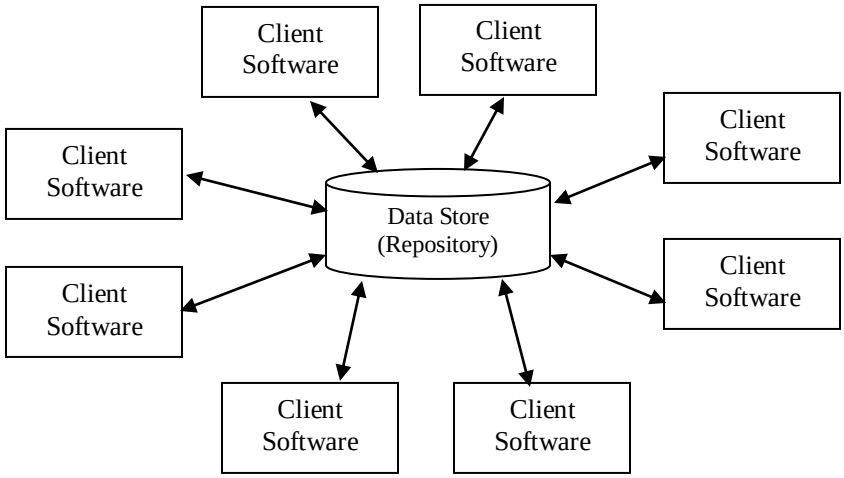
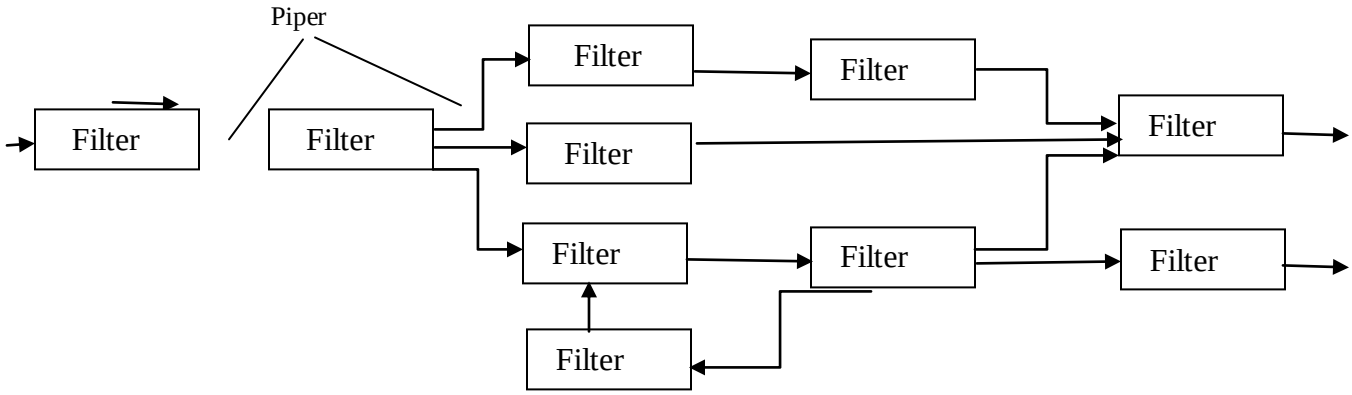


Figure 3.2 Data centered architecture

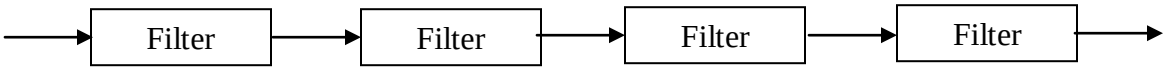
- **Data-flow Architectures:** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter pattern (Figure 3.3 a) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its

neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern Figure (3.3 b) accepts a batch of data and then applies a series of sequential components (filters) to transform it.



(a) Pipes and Filters



(b) Batch Sequential

Figure 3.3: Data flow architecture

- **Call and return Architectures:** The program structure can be easily modified or scaled. The program structure is organized into modules within the program. In this architecture how modules call each other. The program structure decomposes the function into control hierarchy where a main program invokes number of program components.

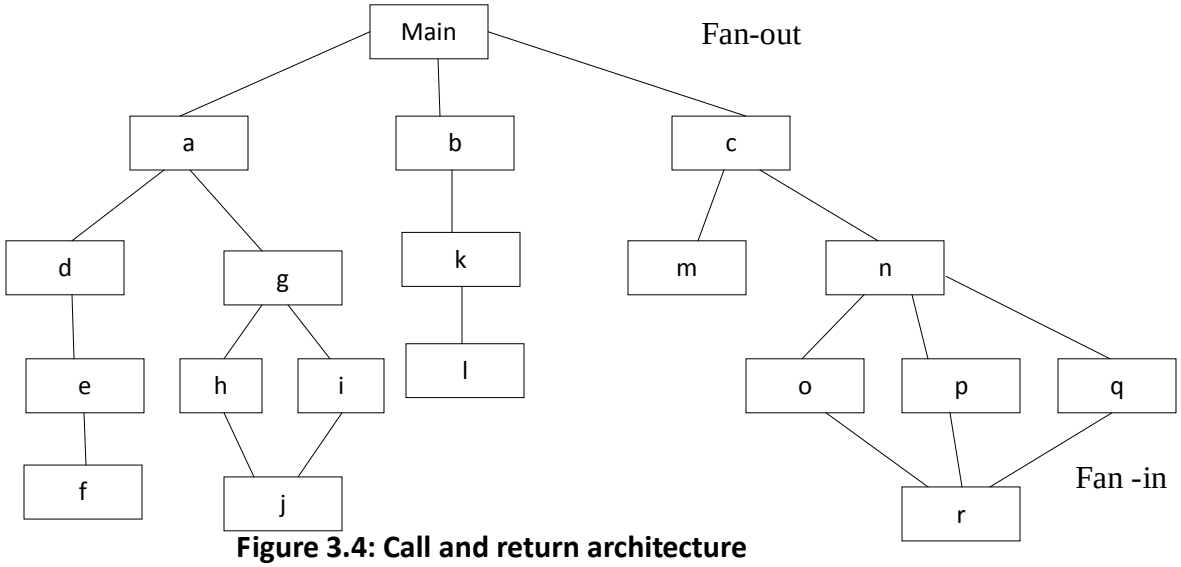


Figure 3.4: Call and return architecture

- **Object-oriented Architecture:** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

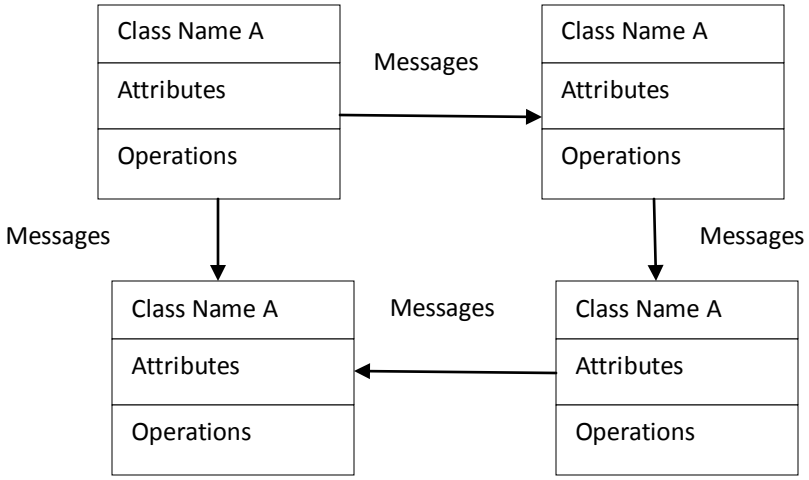


Figure 3.5: Object-oriented Architecture

- **Layered Architectures:** The basic structure of a layered architecture is illustrated in Figure 3.6. A number of different layers are defined, each accomplishing operation that progressively become closer to the machine instruction set.

  At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

  These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

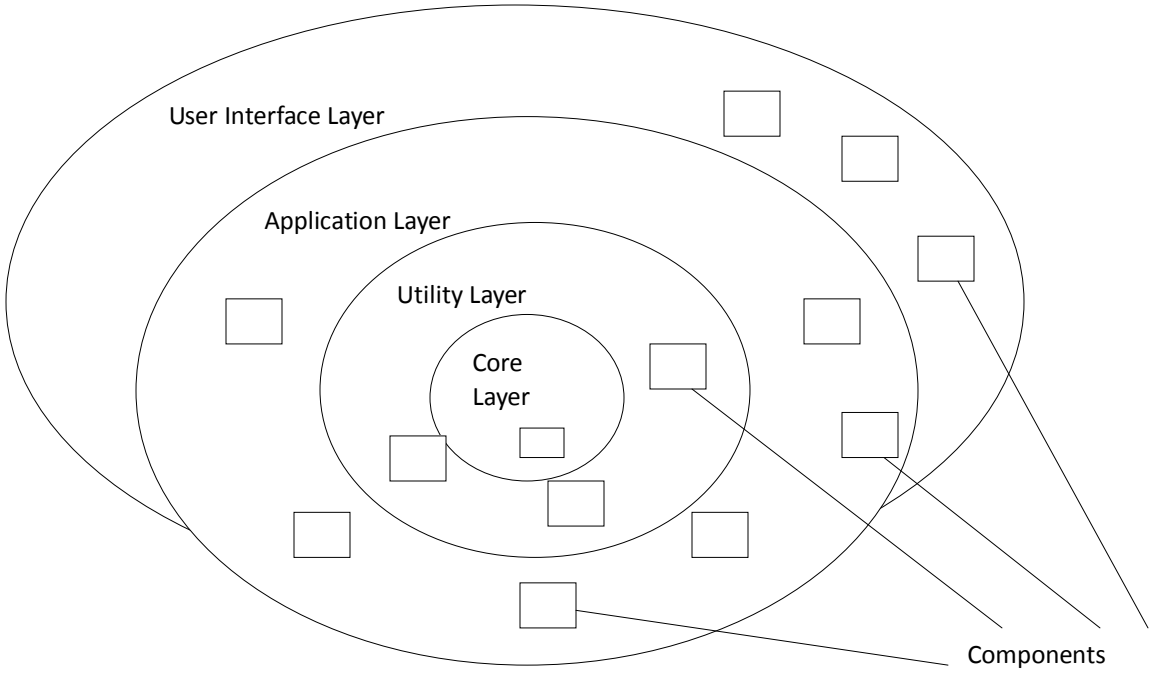


Figure 3.6: Layered Architecture Components

Architectural Views:

4+1 is a architectural view model used for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are

logical, development, process and physical view. In addition selected use cases or scenarios are used to illustrate the architecture serving as the 'plus one' view. Hence the model contains 4+1 views:

- **Development view:** The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.
- **Logical view:** The logical view is concerned with the functionality that the system provides to end-users. UML diagrams used to represent the logical view include, class diagrams, and state diagrams.
- **Physical view:** The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. This view is also known as the deployment view. UML diagrams used to represent the physical view include the deployment diagram.
- **Process view:** The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML diagrams to represent process view include the activity diagram.
- **Scenarios:** The description of architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. The scenarios describe sequences of interactions between objects and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as the use case view.

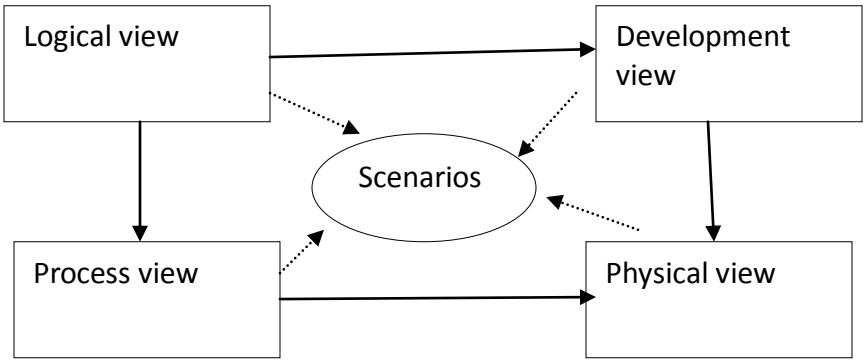


Figure 3.7: 4+1 Architectural View Model

USER INTERFACE DESIGN:

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. User interface provides fundamental platform for human-computer interaction.

User interface can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

User interface is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

User Interface Design Principles:

The principles of user interface design are intended to improve the quality of user interface design.

- **The structure principle:** Design should organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.
- **The simplicity principle:** The design should make simple, common tasks easy, communicating clearly and simply in the user's own language, and providing good shortcuts that are meaningfully related to longer procedures.
- **The visibility principle:** The design should make all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with alternatives or confuse with unneeded information.
- **The feedback principle:** The design should keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
- **The tolerance principle:** The design should be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing, while also preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.
- **The reuse principle:** The design should reuse internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency, thus reducing the need for users to rethink and remember.

User Interface Analysis and Design:

User interface analysis and design can be done with the help of following steps:

- Create different models for the system functions.
- In order to perform these functions identify the human-computer interface tasks.
- Prepare all interface designs by solving various design issues.
- Apply modern tools and techniques to prototype the design.
- Implement design model.
- Evaluate the design from end user to bring quality in it.

These steps can be broadly categorized in two classes.

1. Interface analysis and design models
2. The process

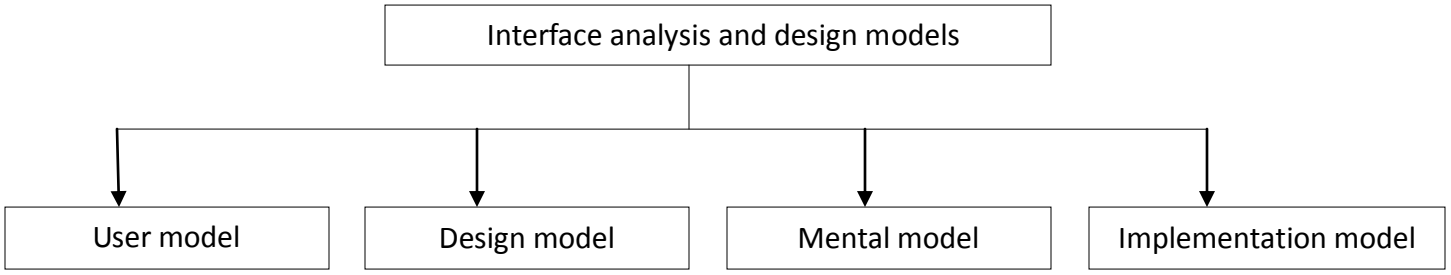


Figure 3.8: Interface Analysis and Design Model

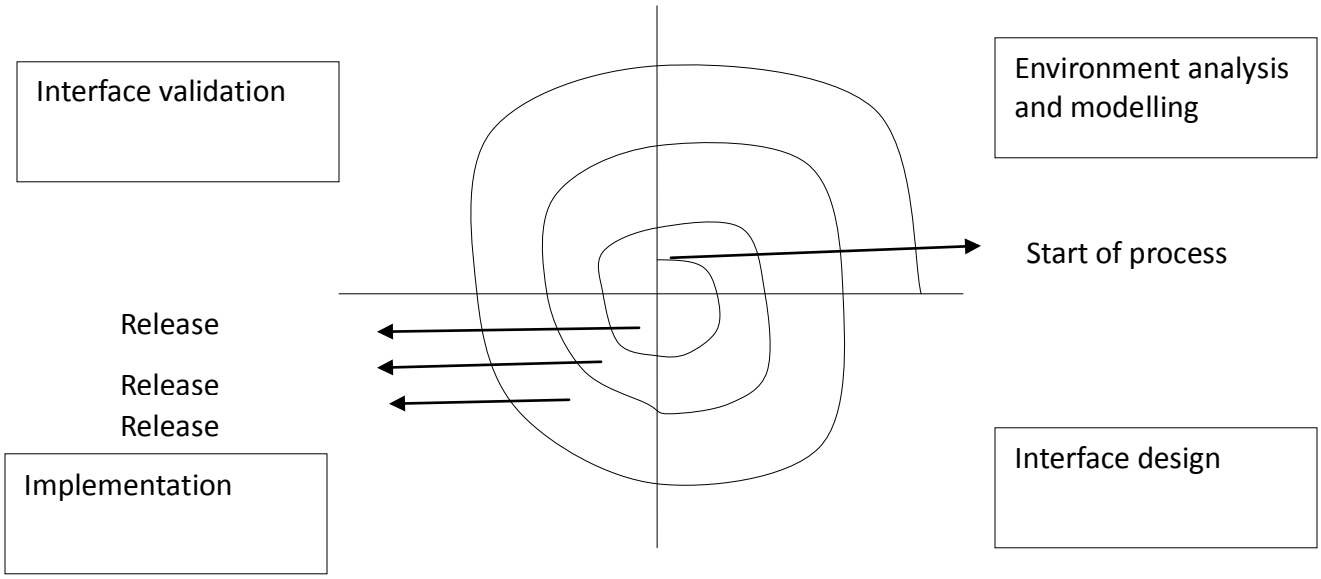


Figure 3.9: Interface Analysis and Design Process

FUNCTION-ORIENTED DESIGN:

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Functional design process:

- **Data-flow design:** Model the data processing in the system using data-flow diagrams.
- **Structural decomposition:** Model how functions are decomposed to sub-functions using graphical structure charts.
- **Detailed design:** The entities in the design and their interfaces are described in detail. These may be recorded in a data dictionary and the design expressed using a PDL.

SA/SD COMPONENT BASED DESIGN:

Structured Analysis and Structured Design: Structured analysis is a set of techniques and graphical tools that allow the analyst to develop a new kind of system specification that are easily understandable to the user.

Goals of SASD

- Improve Quality and reduce the risk of system failure
- Establish concrete requirements specifications and complete requirements documentation
- Focus on Reliability, Flexibility, and Maintainability of system

Component Based Design:

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure component reusability. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. Component-oriented software design has many advantages over the traditional object-oriented approaches such as −

- Reduced time in market and the development cost by reusing existing components.
- Increased reliability with the reuse of the existing components.

Principles of Component–Based Design

A component-level design can be represented by using some intermediary representation (e.g. graphical, tabular, or text-based) that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.

It has following salient features −

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plug-in to offer another plug-in API.

Component-Level Design Guidelines

It creates naming conventions for components that are specified as part of the architectural model and then refines or elaborates as part of the component-level model.

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.

- Recognizes and discover these independent entities as new components.
- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

DESIGN METRICS

In software development, a metric is the measurement of a particular characteristic of a program's performance or efficiency. Design metric measure the efficiency of design aspect of the software. Design model considering three aspects:

- Architectural design
- Object oriented design
- User interface design

Architectural Design Metrics:

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Object Oriented Design Metrics:

There are nine measurable characteristics of object oriented design and those are:

- **Size:** It can be measured using following factors:
  - Population: means total number of classes and operations.
  - Volume: means total number of classes or operation that is collected dynamically.
  - Length: means total number of interconnected design elements.
  - Functionality: is a measure of output delivered to the customer.
- **Complexity:** It is measured representing the characteristics that how the classes are interrelated with each other.
- **Coupling:** It is a measure stating the collaboration between classes or number of messages that can be passed between the objects.
- **Completeness:** It is a measure representing all the requirements of the design component.
- **Cohesion:** It is a degree by which we can identified the set of properties that are working together to solve particular problem.
- **Sufficiency:** It is a measure representing the necessary requirements of the design component.
- **Primitiveness:** The degree by which the operations are simple, i.e. number of operations independent from other.
- **Similarity:** the degree by which we measure that two or more classes are similar with respect to their functionality and behavior.
- **Volatility:** Is the measure that represents the probability of changes that will occur.

User Interface Design Metrics:

Although there is significant literature on the design of human/computer interfaces, relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

- **Layout appropriateness (LA)** is a worthwhile design metric for human/computer interfaces. A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next.
- **Cohesion metrics** can be defined as the relative connection of on screen content to other screen contents. UI cohesion for screen is high.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in