UNIT- IV

Design Concepts: Design with Context of Software Engineering, The Design Process, Design Concepts, The Design Model.
Architectural Design: Software Architecture, Architecture Styles, Architectural Design, Architectural Mapping Using Data Flow.

Design Concepts

Introduction: Software design **encompasses the set of principles**, **concepts, and practices** that lead to the development of a **high-quality system or product**.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight Software design changes continually as new methods, better analysis, and broader understanding evolve.

Design Within The Context Of Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, **software design is the last software engineering action within the modeling activity** and sets the stage for **construction** (code generation and testing).

Each of the elements of the requirements model provides information that is necessary to create the **four design models** required for a complete specification of design.

The requirements model, manifested by **scenario-based, class-based, flow-oriented, and behavioral elements,** feed the design task.

The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software.

The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer- based system—is derived from the requirements model.

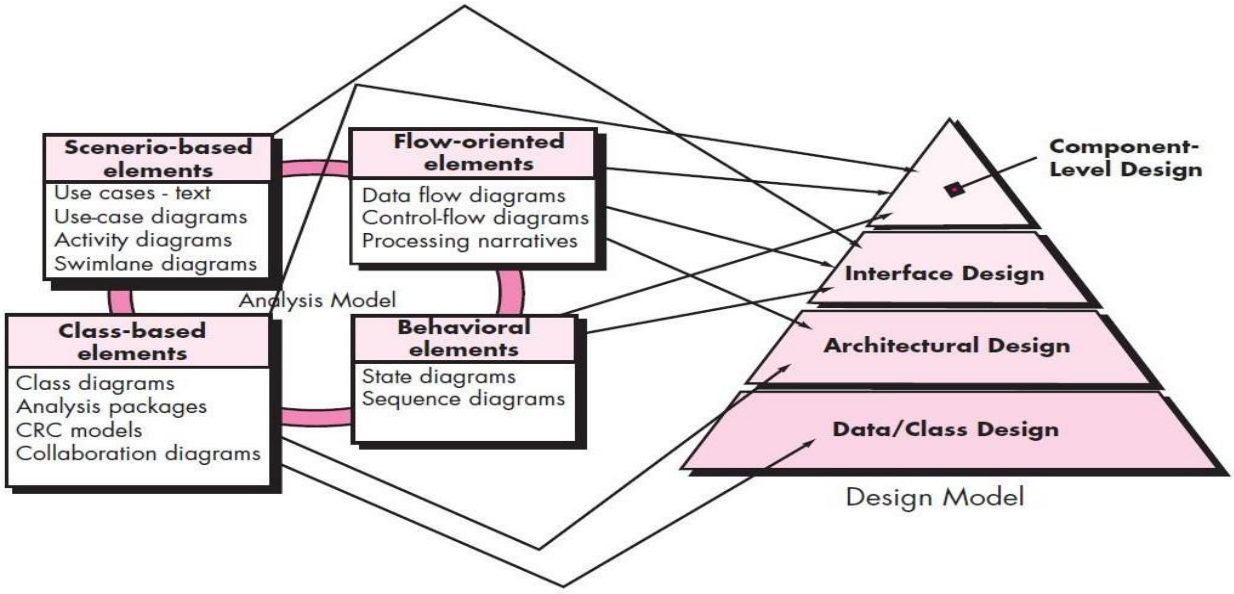

Fig : Translating the requirements model into the design model

The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word—***quality*.** Design provides you with representations of software that can be assessed for quality. Software design serves as the foundation for all the software engineering and software support activities that follow.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "**blueprint**" for constructing the software. Initially, the blueprint shows a holistic view of software. That is, the design is represented at a **high level of abstraction.**

Software Quality Guidelines and Attributes

Mc Glaughlin suggests **three** characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit

requirements desired by stakeholders.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines. In order to evaluate the quality of a design representation, consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes. Hewlett-Packard developed a set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**. The **FURPS** quality attributes represent a target for all software design:

• ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system..

• ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.

• ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of theprogram.

• ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, andefficiency.

• ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, ***maintainability***— and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can belocalized.

The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called ***structured programming***.

Design methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

DESIGN CONCEPTS

The following software design concepts that span both traditional and object-oriented software development.

Abstraction

Abstraction is the act of representing essential features without including the background details or explanations. the *abstraction* is used to reduce complexity and allow efficient design and implementation of complex *software* systems. Many levels of abstraction can be posed. At the **highest level** of abstraction, a solution is stated in broad terms using the language of the problem environment. At **lower levels** of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create both **procedural** and data abstractions.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A ***data abstraction*** is a named collection of data that describes a data object.

Architecture

Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Shaw and Garlan describe a set of properties that should be specified as part of an

architectural design:

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

The architectural design can be represented using one or more of a number of different models. ***Structural models:*** *R*epresent architecture as an organized collection of program components. ***Framework models:*** Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models *:*Address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models *:*Focus on the design of the business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

A number of different ***architectural description languages*** **(ADLs)** have been developed to represent these models.

Patterns

Brad Appleton defines a ***design pattern*** in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns"

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can bereused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the

requirements model for the software.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called ***module.***

Modularity is the single attribute of software that allows a program to be intellectually manageable.

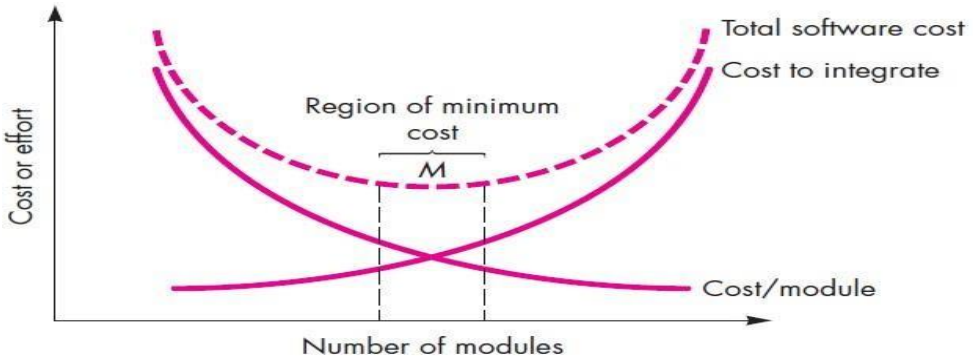

Fig : Modularity and software cost

Information Hiding

The principle of information hiding suggests that modules be "characterized by design decisions that hides from all others." In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "**single minded**" function and an "aversion" to excessive interaction with other modules.

Independence is assessed using **two** qualitative criteria: **cohesion** and **coupling**. ***Cohesion*** is an indication of the relative functional strength of a module. ***Coupling*** is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing. Although you should always strive for **high cohesion** (i.e.,single-mindedness).

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the **lowest possible coupling**.

Refinement

Refinement is actually a process of ***elaboration***. You begin with a statement of function that is defined at a high level of abstraction.

Abstraction and **refinement** are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

Aspects

An ***aspect*** is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements.

Refactoring

"**Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."**

Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

Design Classes

The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. A set of ***design classes*** that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Five different types of design classes, each representing a different layer of the design architecture, can be developed:

- ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). The design classes for the interface may be visual representations of the elements of the metaphor.
- ***Business domain classes*** are often refinements of the analysis classes defined

earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

• ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.

• ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.

• ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "**well-formed**." They define **four** characteristics of a well-formed design class:

- **Complete and sufficient.** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
- **Primitiveness.** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
- **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain overtime.

THE DESIGN MODEL

The design model can be viewed in **two** different dimensions. The ***process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process. The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. The design model has **four** major elements: data, architecture, components, and interface.

3.4.1. Data Design Elements

Data design (sometimes referred to as *data architecting*) creates a model of data

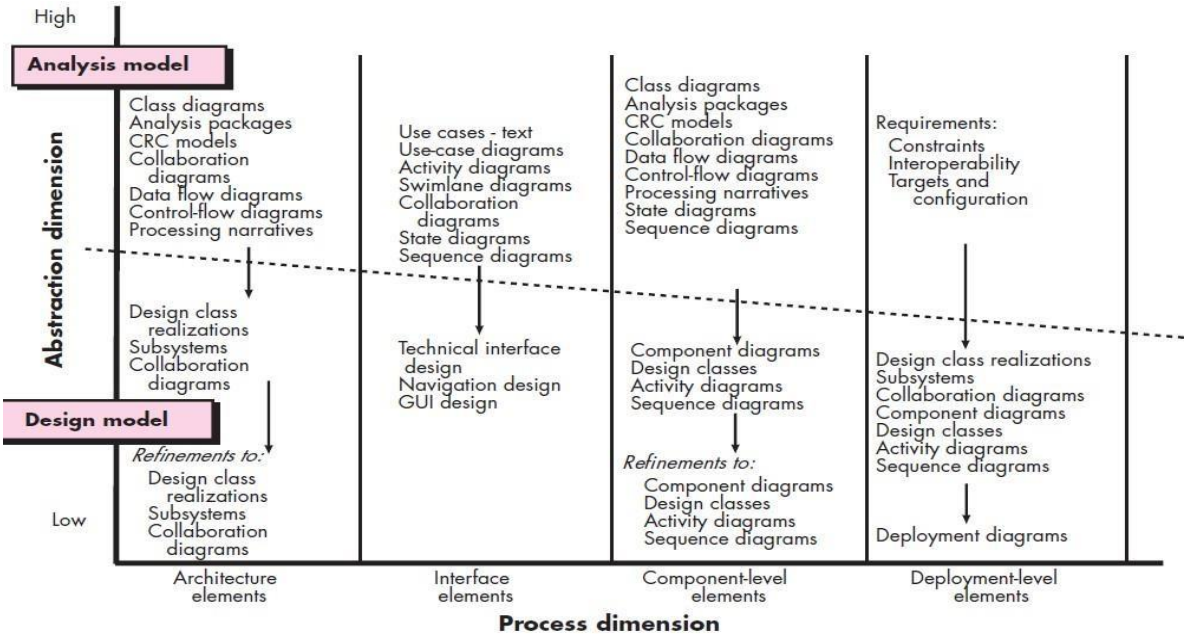

Fig : Dimensions of the design model

and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. The structure of data has always been an important part of software design. At the program **component level**, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high- quality applications. At the **application level**, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the **business level**, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

3.4.2 Architectural Design Elements

The ***architectural design*** for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. Architectural design elements give us an overall view of the software.

The architectural model is derived from **three** sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.

Interface Design Elements

The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house.

There are **three** important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components.

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, sinks, showers, tubs, drains, cabinets, and closets.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.

Deployment-Level Design Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.