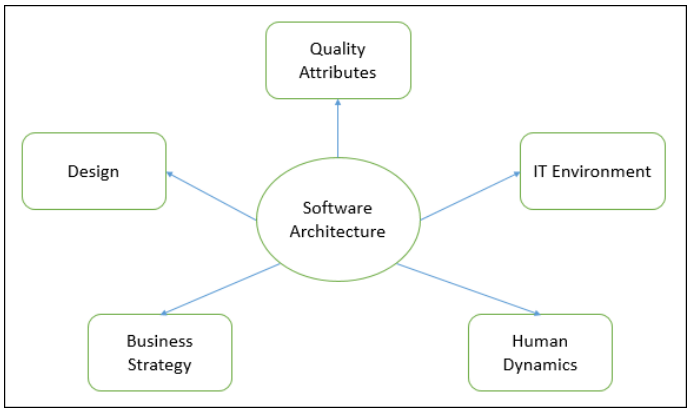
Unit-4

Creating and Modeling the Design: Software architecture, Architectural design, Nature of component, Designing class-based components: Principles, Guidelines, Cohesion, Coupling, Conducting component level design.

➢ Software Architecture

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.



In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements.

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.

Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

Some of the other 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.

Limitations

Software architecture is still an emerging discipline within software engineering. It has the following limitations −

- Lack of tools and standardized ways to represent architecture.
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development.
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design.

➢ Architectural Design

Introduction :

As architectural design begins, the software to be developed must be put into context— .

Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

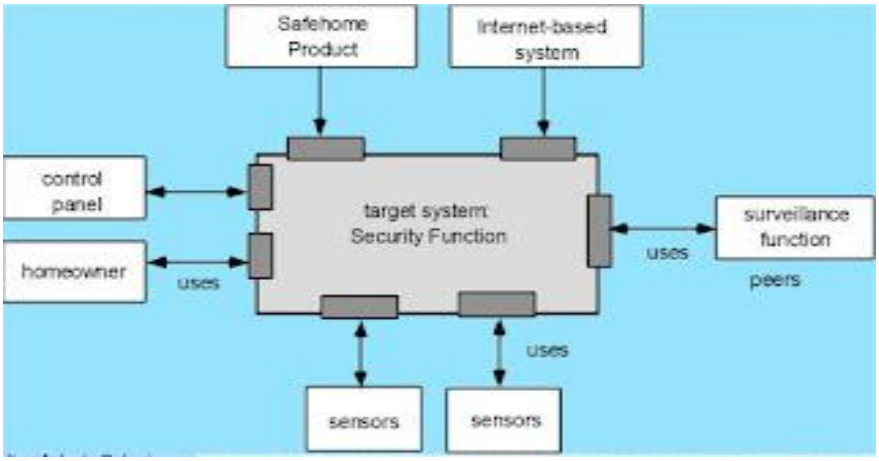
An archetype is an abstraction (similar to a class) that represents one element of system behavior.

This process continues iteratively until a complete architectural structure has been derived..

1.Representing the System in Context

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries.

The generic structure of the architectural context diagram is illustrated in Figure.



Super ordinate systems : those systems that use the target system as part of some higher-level processing scheme.
Subordinate systems—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
Peer-level systems—those systems that interact on a peer-to- peer basis (i.e., information is either produced or consumed by the peers and the target system.
Actors—entities (people, devices) that interact with the target system by producing or consuming information.
Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

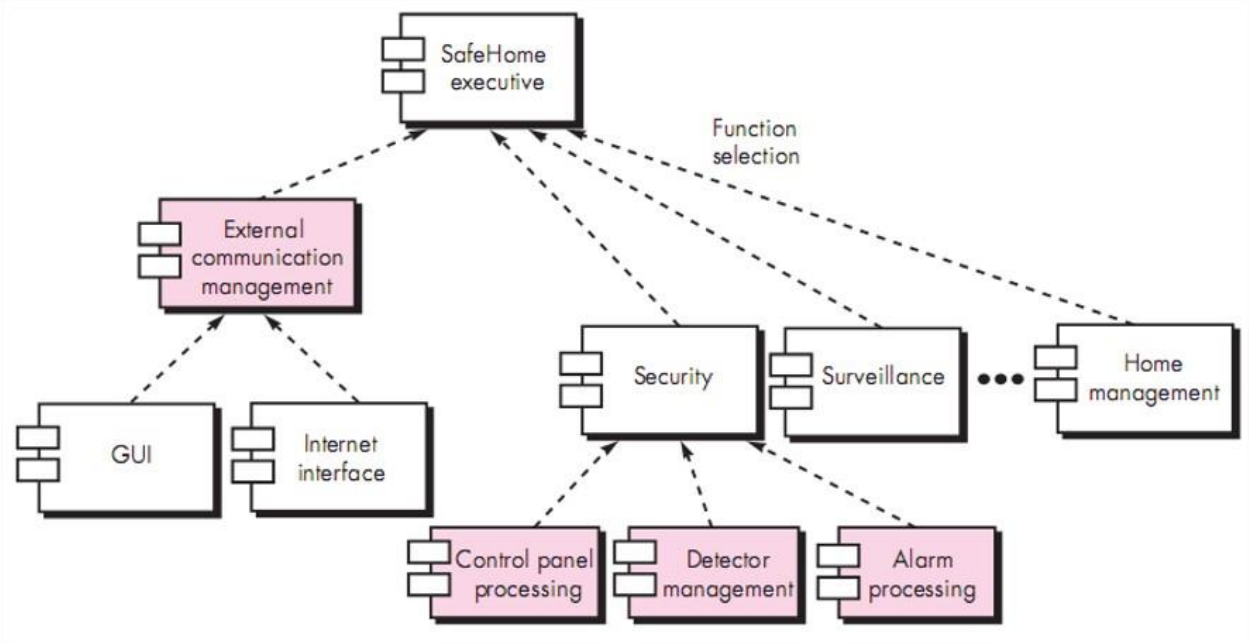
2.Defining Archetypes

An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.
For example, an archetype for a car: wheels, doors, seats, engine In software engineering
As per in previous figure : The SafeHome home security function, you might define the following archetypes :
Node : Represents a cohesive collection of input and output elements of the home security function.
For example a node might be included of (1) various sensors and (2) a variety of alarm (output) indicators.
Detector : An abstraction that covers all sensing equipment that feeds information into the target system.
Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
Controller. An abstraction that describes the mechanism that allows the arming (Supporting) or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

3.Refining the Architecture into Components

As the software architecture is refined into components.
In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

For Example : The SafeHome home security function example, you might define the set of top-level components that address the following functionality:
External communication management — coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
Control panel processing— manages all control panel functionality.
Detector management — coordinates access to all detectors attached to the system.
Alarm processing — verifies and acts on all alarm conditions
The overall architectural structure (represented as a UML component diagram) is in the following Figure.



4.Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level.
The context of the system has been represented
Archetypes that indicate the important abstractions within the problem domain have been defined.

➢ What is a Component?

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

Views of a Component

A component can have three different views − object-oriented view, conventional view, and process-related view.

Object-oriented view

A component is viewed as a set of one or more cooperating classes. Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

Conventional view

It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

Process-related view

In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

Characteristics of Components

- **Reusability** − Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.
- **Replaceable** − Components may be freely substituted with other similar components.
- **Not context specific** − Components are designed to operate in different environments and contexts.
- **Extensible** − A component can be extended from existing components to provide new behavior.
- **Encapsulated** − A A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.
- **Independent** − Components are designed to have minimal dependencies on other components.

DESIGNING CLASS-BASED COMPONENTS:

Principles,
Guidelines,
Cohesion,
Coupling,
Conducting component level design

➢ Basic Design Principles

Introduction : Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.

The Open-Closed Principle (OCP): "A module [component] should be open for extension but closed for modification.

The Liskov Substitution Principle (LSP): "Subclasses should be substitutable for their base classes.

Dependency Inversion Principle (DIP) :"Depend on abstractions. Do not depend on concretions."

The Interface Segregation (Separation) Principle (ISP): "Many client-specific interfaces are better than one general purpose interface.

The Release Reuse Equivalency Principle (REP): "The granule of reuse is the granule of release."

The Common Closure Principle (CCP): "Classes that change together belong together."

The Common Reuse Principle (CRP): "Classes that aren't reused together should not be grouped together."

➢ Guidelines:

In addition to the principles , a set of pragmatic (Practical) design guidelines can be applied as component-level design proceeds.

These guidelines apply to

- Components,
- Interfaces
- Dependencies and inheritance

Components

Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model

Interfaces

Interfaces provide important information about communication and collaboration.

Dependencies and Inheritance

it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion

Cohesion refers to the degree to which the responsibilities of a single module are related & focussed.

It measures how strongly the internal elements of a module are connected & work together to achieve a common purpose.

High cohesion promotes reusability, maintainability, and understandability of the code.

The higher the degree of cohesion, better the quality of the software.

Coupling

Coupling refers to the degree of interdependence between two or more modules in a system.

It measures how closely the modules are connected & how much they rely on each other.

Low Coupling promotes independence, modularity, and flexibility of the code.

The lower the degree of coupling, better the quality of the software.

➢ What is cohesion?

Cohesion in software engineering refers to the degree to which the elements inside a module/components belong together, directed towards performing a single task.

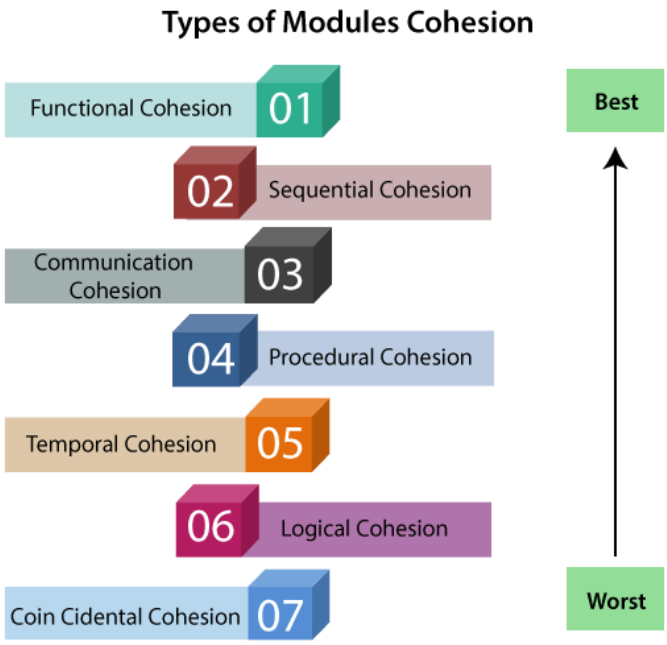
In simple words, it is the degree to which the elements of the module are functionally related.

A good software design will have high cohesion.

The concept of cohesion is closely related to the Single Responsibility Principle, where a class should only have one responsibility or should perform a single task.

The modules following the SRP are likely to have high cohesion as they are meant to perform a single task in the software.

Types of Modules Cohesion



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

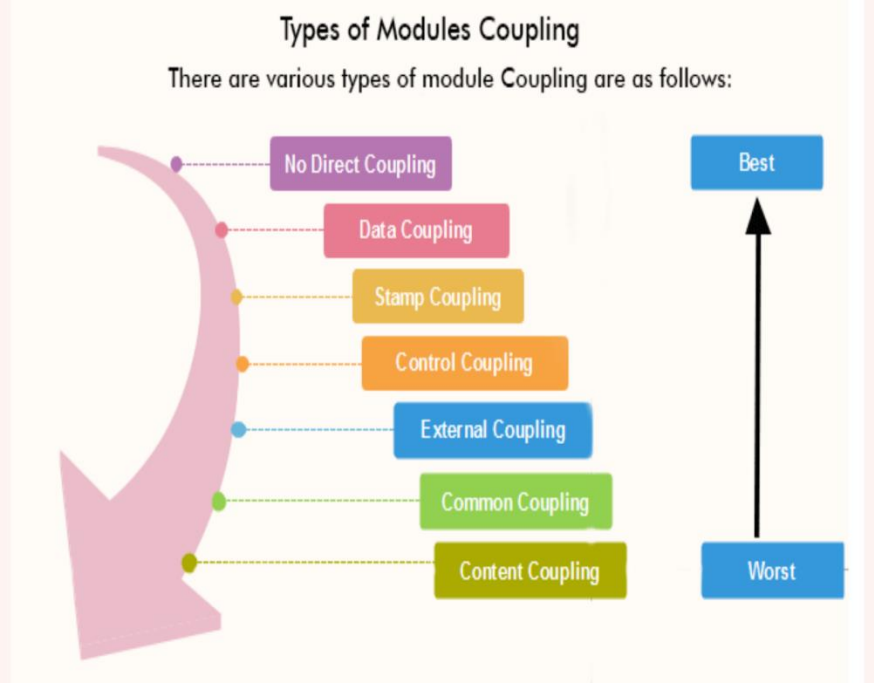
➢ What is Coupling?

Coupling in software engineering refers to the degree of interdependence & connection between modules or components within a software system.

Two modules are said to have high coupling if they are closely connected.

In simple words, coupling is not just about modules, but the connection between modules and the degree of interaction or interdependence between the two modules. If two modules contain a good amount of data, then they are highly interdependent.

If the connection between components is strong, we speak about strongly coupled modules; when the connection is weak, we speak about the loosely coupled modules.



What are the different types of Coupling in software engineering?

1. Data Coupling

Data coupling occurs when modules share data through parameters or arguments.

Each module maintains its own data and does not exactly access or modify the data of other modules.

This type of coupling promotes encapsulation & module interdependence.

2. Stamp Coupling

Stamp coupling is a weaker form of coupling where modules share a composite data structure, but not all the elements are used by each module.

As the data and elements are pre-organized and well-placed beforehand, no junk or unused data is shared or passed between the two coupling modules which improves the efficiency of the modules.

3. Control Coupling

Control coupling occurs when one module controls the behavior of another module.

This type of coupling implies that one module has knowledge of internal workings & decisions or another module, that makes the code more difficult to maintain.

4. External Coupling

External coupling measures the degree to which the system relies on external entities to fulfill its functionality or interact with the external environment.

Low external coupling - Changes in the external entities have little impact on internal implementation of the system.

Medium external coupling - Changes in the external entities require some modifications within the system to accomodate new interfaces.

High external coupling - Changes in the external entities have a substantial impact on internal implementation of the system, requiring extensive modifications.

5. Common Coupling

Common coupling occurs when two or more modules in the system share global data.

The modules can access & manipulate the same global variables & the data structures.

6. Content Coupling

Content coupling occurs when one module directly accesses or modifies the content of another module.

➢ Conducting Component-level design

It transforms information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity.

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain.

Step 2. Identify all design classes that correspond to the infrastructure domain.

Step 3. Elaborate all design classes that are not acquired (Obtain) as reusable components.

Step 3 (a) . Specify message details when classes or components collaborate.

The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another.

Step 3 (b). Identify appropriate interfaces for each component.

Step 3 (c). Elaborate attributes and define data types and data structures required to implement them.

Step 3 (d). Describe processing flow within each operation in detail.

Step 4.Describe persistent data sources (databases and files) and identify the classes required to manage them.

Step 5.Develop and elaborate behavioral representations for a class or component.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural.

Step 7.Refactor every component-level design representation and always consider alternatives.