Unit-5
Software Design

- Introduction to software Design
- Characteristics of a good software design
- Design Principle
- Design Concepts
- Design strategy
- Design Process and design Quality
- Software architecture and its types

Introduction to Software Design Process

- Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.
- The software design phase is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the solution domain.
- In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

Levels or phases of Software Design

- The software design process can be divided into the following three levels or phases of design:
  - Interface Design
  - High level design
  - Architectural Design
  - Detailed Design

Interface Design

- Interface design is the specification of the interaction between a system and its environment.
- This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored, and the system is treated as a black box.
- The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

High level design

- High-Level Design (HLD), also known as Architectural Design, is the first level of the software design process.
- It focuses on defining the overall structure of the system, its major components, and how they interact with each other.
- HLD provides a blueprint for the system, ensuring that it meets the functional and non-functional requirements while aligning with the project's goals and constraints.

Architectural Design

- Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them.
- In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.
- Architectural Design is the highest level of the software design process.
- It focuses on defining the overall structure of the system, its major components, and how they interact with each other.

Detailed Design

- Detailed design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.
- Detailed Design (Low-Level Design) is the final level of the software design process.
- It focuses on specifying the internal logic, structure, and implementation details of each component or module identified during the architectural and component design phases.
- This phase bridges the gap between high-level design and actual coding, ensuring that developers have a clear and precise plan for implementation.

Characteristics of Good Software Design

- The characteristics of software design define the qualities and attributes that make a design effective, efficient, and maintainable. A well-designed software system exhibits the following key characteristics:

1. Correctness

- The design must accurately implement all the specified requirements.
- It should fulfill both functional and non-functional requirements.

2. Modularity

- The system is divided into smaller, independent, and interchangeable modules.

3. Abstraction

- Focuses on essential features while hiding unnecessary details.

4. Encapsulation

- Hides the internal details of a module or component.
- Exposes only the necessary interfaces for interaction.
- Protects data integrity and reduces dependencies.

5. Scalability

- The design should support growth in terms of users, data, or functionality.

6. Performance

- The design should optimize resource usage (e.g., CPU, memory, network).

7. Maintainability

- The design should be easy to understand, modify, and extend.

8. Reusability

- Components or modules should be designed for reuse in other systems or projects.

9. Security

- The design should incorporate security measures to protect against vulnerabilities.

10. Portability

- The system should be able to run on different platforms or environments with minimal changes.

11. Reliability

- The design should ensure the system operates correctly under specified conditions.

12. Flexibility

- The design should accommodate future changes or enhancements.

13. Simplicity

- The design should avoid unnecessary complexity.

14. Traceability

- The design should clearly map to the requirements.

Software Design Principles

- Software Design Principles are recommendations that help you write code that is clear, manageable, and scalable.
- These principles give a foundation for developing software systems that are both robust and responsive to change.
- Software design principles are essential recommendations that assist developers in writing effective and maintainable code.

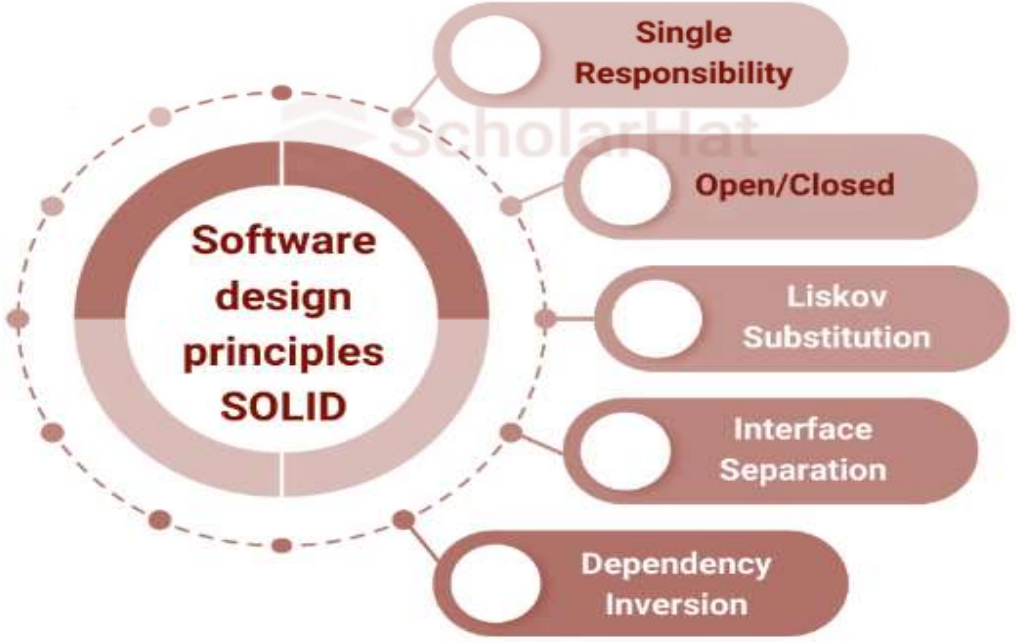
Different Types of Software Design Principles

- Software design principles are a set of criteria that assist developers in creating good system designs.
- The many kinds of software design principles are as follows:

1. SOLID

- The **SOLID principles** are a set of five design principles in object-oriented programming and software engineering that aim to make software designs more understandable, flexible, and maintainable.
- These principles were introduced by Robert C. Martin (also known as Uncle Bob) and have become foundational guidelines for writing clean and robust code.

- ❑Single Responsibility Principle (SRP)
- ❑Open/Closed Principle (OCP)
- ❑Liscov Substitution Principle (LSP)
- ❑Interface Segregation Principle (ISP)
- ❑Dependency Inversion Principle (DIP)



- **Single Responsibility Principle (SRP)**
  - This principle states that there should never be more than one reason for a class to change.
  - This means that you should design your classes in such a way that each class should have a single purpose.
- **Open/Closed Principle (OCP)**
  - This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
  - The "closed" part of the rule states that once a module has been developed and tested, the code should only be changed to correct bugs. The "open" part says that you should be able to extend existing code to introduce new functionality.
- **Liscov Substitution Principle (LSP)**
  - Subclasses should be able to replace their parent class without breaking the program.
  - Subclasses must behave in a way that doesn't change the expected results from the parent class.
- **Interface Segregation Principle (ISP)**
  - This principle states that Clients should not be forced to depend upon interfaces that they don't use.
  - This means minimizing the number of members in the interface visible to the dependent class.
- **Dependency Inversion Principle (DIP)**
  - **The Dependency Inversion Principle states that:**
    - The main parts of your code shouldn't rely directly on the details; they should depend on general concepts.
    - Details should follow these general concepts, not the other way around.
    - This makes your code more flexible and easier to update without breaking other parts.

```python
# Bad: One class doing multiple things
class UserManager:
    def authenticate(self, username, password):
        # Authentication logic
        pass

    def send_email(self, user, message):
        # Email sending logic
        pass

# Good: Separate responsibilities
class Authenticator:
    def authenticate(self, username, password):
        # Authentication logic
        pass

class EmailSender:
    def send_email(self, user, message):
        # Email sending logic
        pass
```

```python
# Bad: Modifying existing class to add new shapes
class AreaCalculator:
    def calculate_area(self, shape):
        if shape.type == "circle":
            return 3.14 * shape.radius ** 2
        elif shape.type == "square":
            return shape.side ** 2
        # Adding a new shape requires modifying this class

# Good: Open for extension, closed for modification
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2
```

```python
# Bad: Subclass violates LSP
class Bird:
    def fly(self):
        pass

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches can't fly")  # Violates LSP

# Good: Subclass adheres to LSP
class Bird:
    pass

class FlyingBird(Bird):
    def fly(self):
        pass

class Ostrich(Bird):
    pass  # Ostriches don't fly, but they don't break the behavior of Bird
```

```python
# Bad: One large interface
class Printer:
    def print(self):
        pass

    def scan(self):
        pass

    def fax(self):
        pass

# Forces all printers to implement all methods, even if they don't support them

# Good: Segregated interfaces
class Printer:
    def print(self):
        pass

class Scanner:
    def scan(self):
        pass

class FaxMachine:
    def fax(self):
        pass
```

```python
# Bad: High-level module depends on low-level module
class LightBulb:
    def turn_on(self):
        pass

class Switch:
    def __init__(self, bulb):
        self.bulb = bulb

    def operate(self):
        self.bulb.turn_on()

# Good: Both depend on abstraction
from abc import ABC, abstractmethod

class Switchable(ABC):
    @abstractmethod
    def turn_on(self):
        pass

class LightBulb(Switchable):
    def turn_on(self):
        pass

class Switch:
    def __init__(self, device: Switchable):
        self.device = device

    def operate(self):
        self.device.turn_on()
```

Advantages of SOLID Principle

- Avoid Duplicate code
- Easy to maintain
- Easy to understand
- Flexible software
- Reduce Complexity

2. DRY (Don't Repeat Yourself)

- The DRY principle states that every piece of knowledge or logic in a system should have a single, unambiguous representation. In other words, avoid duplicating code or logic across your codebase.
- It underlines the importance of having a single, unambiguous representation of all information or logic within a system.
- The DRY concept helps to eliminate errors by reducing code duplication and consolidating comparable code into reusable functions or components.
- It also simplifies maintenance and makes the codebase more cohesive and manageable.

Without DRY:

```
area1 = 3.14 * radius1 * radius1
area2 = 3.14 * radius2 * radius2
```

With DRY:

```
def calculate_area(radius):
    return 3.14 * radius * radius
area1 = calculate_area(radius1)
area2 = calculate_area(radius2)
```

3. KISS (Keep it simple, Stupid!)

- The **KISS principle** stands for **"Keep It Simple, Stupid"** and is a design philosophy that emphasizes simplicity in software development and problem-solving.
- The idea is to avoid unnecessary complexity and keep systems, code, and designs as simple as possible. This principle is widely applicable across various fields, including software engineering, product design, and even communication.
- Keep It Simple, Stupid (KISS) is a software design guideline that encourages simplicity in code and design.
- It highlights the importance of keeping systems as simple as feasible while avoiding unnecessary complexity and over-engineering.

 # Without KISS

```
def calculate_discount(prices, discount_rate):
    discounted_prices = []
    for i in range(len(prices)):
        discounted_prices.append(prices[i] * (1 - discount_rate))
    return discounted_prices
```

 # With KISS

```
def calculate_discount(prices, discount_rate):
    return [price * (1 - discount_rate) for price in prices]
```

4. YAGNI (You aren't going to need it)

- You Aren't Gonna Need It (YAGNI) is a software design theory that recommends not implementing capability until it is genuinely needed.
- It highlights that developers should avoid adding features or capabilities based on fictional demands or future requirements.
- By following **YAGNI**, developers can simplify their codebase, prevent over-engineering, and focus on solving existing problems, resulting in more manageable and efficient code.

5. Design by Contract

- Specify clear preconditions, postconditions, and invariants for components.
- Components must fulfill their contract obligations as specified.
- Both callers and implementers must meet their contractual obligations.
- Detect and handle contract violations to ensure reliable behavior.

6. Principle of Least Astonishment

- Ensure that software behaves in a way that users or developers expect, minimizing surprises or confusion.
- Follow established conventions and patterns to make the system predictable and easy to understand.
- Provide clear explanations and documentation to help users understand how the system should work and what to expect.
- Design features and interfaces to align with standard practices and user experiences to avoid unexpected behavior.

7. Composition Over Inheritance

- Build functionality by composing objects from other objects rather than using a rigid inheritance hierarchy. This allows for more flexible and reusable code.
- Reduces the problems associated with deep inheritance chains, such as tight coupling and inflexibility, by favoring composition.
- It simplifies changes and extensions by allowing objects to be composed in different ways rather than modifying or extending existing classes.

8. Fail Fast

- Identify and address errors as soon as they occur rather than allowing issues to propagate through the system.
- Validate inputs and conditions at the earliest possible point to prevent invalid data from causing problems later.
- By catching errors early, you simplify the debugging process and make it easier to locate and fix issues.
- Ensure the system behaves reliably by handling errors promptly, preventing unexpected behaviors, and improving overall stability.

9. Law of Demeter (LoD)

- Limit the knowledge that a class has about the internal details of other classes to reduce dependencies and promote encapsulation.
- A class should only interact with its direct collaborators and avoid reaching into the internals of other objects.
- Prevent method calls from being chained together (e.g., objectA.getB().getC().doSomething()) to reduce coupling and improve code maintainability.
- Design classes to have clear and simple interfaces, making the system easier to understand and less prone to errors.

10. Separation of Concerns

- Break down a system into distinct sections, each handling a specific aspect of functionality. This helps manage complexity by keeping related tasks together.
- Ensure that each section or module works independently and does not overlap with others. Changes in one part should not affect others.
- Write code so that each part of the system does one job well, making it easier to understand, test, and maintain.
- By keeping different concerns separate, you can make updates or fixes to one part without worrying about unintended effects on other parts.

Importance Software Design Principles

- Software design principles are important because they guide developers in creating high-quality software that is efficient, maintainable, and scalable:

- **Improves Code Quality:** Following principles ensures the software is well-structured, making it easier to read, understand, and debug.
- **Enhances Maintainability:** Good design principles reduce complexity and improve code organization, making future updates and maintenance easier.
- **Promotes Reusability:** Principles like DRY (Don't Repeat Yourself) encourage reusing code, saving time, and reducing redundancy.
- **Increases Flexibility:** Adopting principles like SOLID allows the software to adapt to changes with minimal impact, improving its lifespan.
- **Encourages Collaboration:** Well-designed software is easier for teams to work on, as it is modular and follows standard patterns.
- **Prevents Errors and Bugs:** Consistent design reduces the likelihood of introducing errors, leading to fewer bugs and a more stable system.

Design Concepts

- Design concepts in software engineering refer to the fundamental principles and guidelines that guide the process of creating software solutions.
- These concepts help ensure that the software is well-structured, maintainable, and meets the desired requirements.

Some design concepts:

- **Modularity**: Breaking down the software into smaller, independent modules or components that can be developed, tested, and maintained separately. This promotes reusability, flexibility, and easier collaboration among developers.
- **Abstraction**: Hiding unnecessary details and complexity of the system behind simplified interfaces. This allows developers to focus on high-level concepts and makes the system easier to understand and modify.
- **Encapsulation**: Bundling data and related functions into a single unit called a class or object. Encapsulation protects data from unauthorized access and allows for better control and organization of code.
- **Inheritance**: Creating new classes based on existing classes, inheriting their attributes and behaviors. Inheritance promotes code reuse, reduces redundancy, and allows for the creation of specialized classes.

- **Polymorphism**: The ability of objects to take on different forms or behaviors based on their context. Polymorphism allows for code flexibility, extensibility, and the implementation of generic algorithms.
- **Separation of Concerns**: Dividing the software into distinct modules or layers, each responsible for a specific aspect or concern. This promotes code organization, maintainability, and allows for easier debugging and testing.
- **High Cohesion**: Ensuring that each module or class has a single, well-defined responsibility. High cohesion reduces dependencies, improves code readability, and makes the system easier to understand and modify.
- **Low Coupling**: Minimizing the dependencies between modules or classes. Low coupling promotes code reusability, flexibility, and reduces the impact of changes in one module on other modules.
- **Design Patterns**: Reusable solutions to common software design problems. Design patterns provide proven approaches for solving specific design challenges and promote best practices in software development.

Software Design Strategies

- Software Design Strategies are systematic approaches or methodologies used to design software systems.
- These strategies help developers organize and structure their code, ensuring that the system is scalable, maintainable, and efficient.
- Different strategies are suited for different types of problems, and choosing the right one depends on the project's requirements, complexity, and constraints.
- The choice of system design strategy will depend on the particular requirements of the software system, the size and complexity of the system, and the development methodology being used.
- A well-designed system can simplify the development process, improve the quality of the software, and make the software easier to maintain.

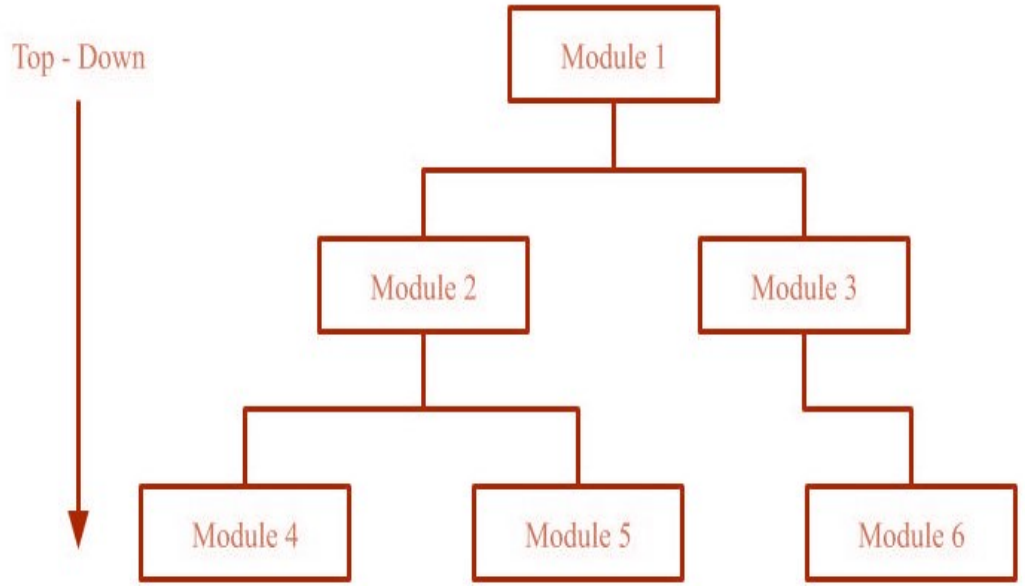
Strategies used to design software systems

- There are two generic approaches for software designing:
  - Top-Down Design
  - Bottom-Up Design

Top-down approach:

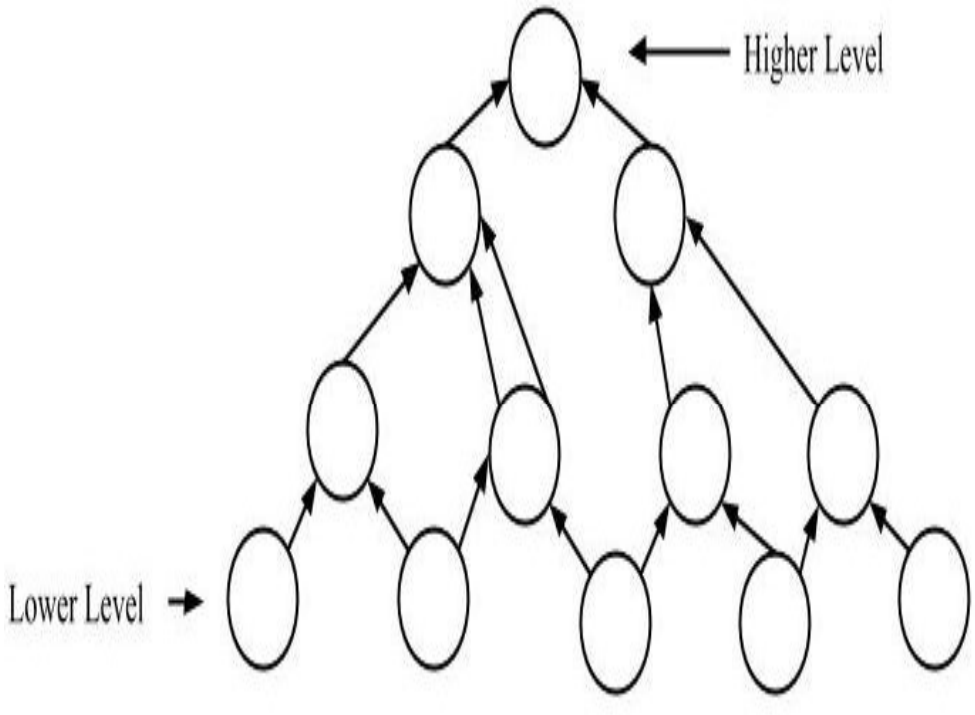
- Each system is divided into several subsystems and components.
- Each of the subsystems is further divided into a set of subsystems and components.
- This process of division facilitates forming a system hierarchy structure.

- The complete software system is considered a single entity and in relation to the characteristics, the system is split into sub-systems and components.



Bottom-Up Design

- The design starts with the lowest level components and subsystems.
- By using these components, the next immediate higher-level components and subsystems are created or composed.
- The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system.
- The amount of abstraction grows high as the design moves to more high levels.



Software Design Process

- Software design process can be perceived as series of well-defined steps.
- Though it varies according to design approach

1. Structured Design

- Structured design is a data-flow based methodology that helps in identifying the input and output of the developing system.
- The main objective of structured design is to minimize the complexity and increase the modularity of a program. Structured design also helps in describing the functional aspects of the system.
- These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely

❑ Cohesion :

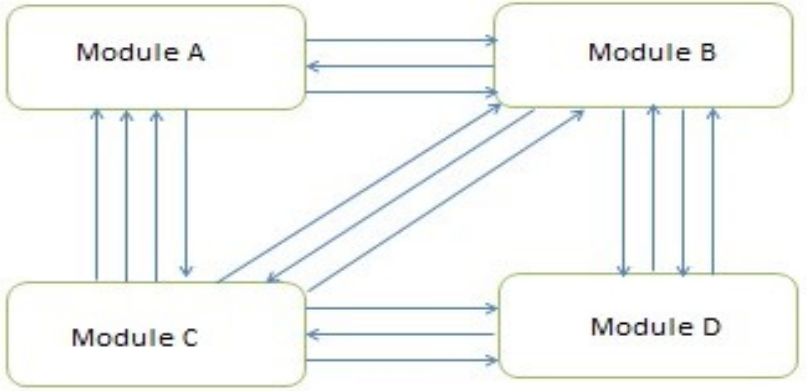
- Cohesion is the measure of closeness of the relationship between its components. It defines the amount of dependency of the components of a module on one another.
- In practice, this means the systems designer must ensure that :
  - They do not split essential processes into fragmented modules.
  - They do not gather together unrelated processes represented as processes on the DFD into meaningless modules.

❑ Coupling :

- Coupling is the measure of the independence of components.
- It defines the degree of dependency of each module of system development on the other.
- In practice, this means the stronger the coupling between the modules in a system, the more difficult it is to implement and maintain the system.
- Each module should have simple, clean interface with other modules, and that the minimum number of data elements should be shared between modules.

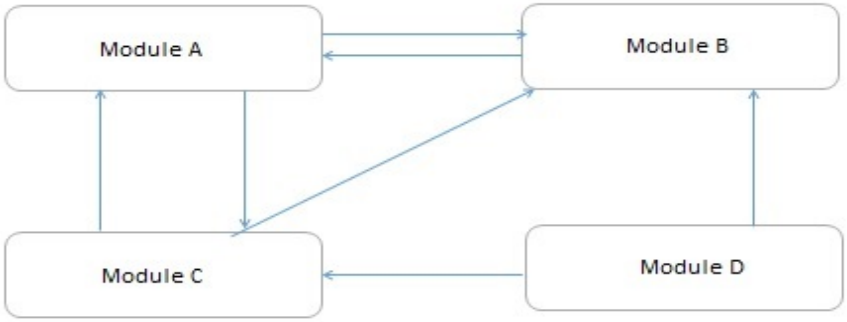
❑ High coupling:

- ➢ These type of systems have interconnections with program units dependent on each other.
- ➢ Changes to one subsystem leads to high impact on the other subsystem.



❑ Low Coupling

- These type of systems are made up of components which are independent or almost independent.
- A change in one subsystem does not affect any other subsystem.



2. 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.

3. Object Oriented Design

- Object oriented design works around the entities and their characteristics instead of functions involved in the software system.
- This design strategy focuses on entities and its characteristics.
- The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

Objects :

- All entities involved in the solution design are known as objects.
- For example, person, banks, company and customers are treated as objects.
- Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes :

- A class is a generalized description of an object.
- An object is an instance of a class.
- Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.
- In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Encapsulation:

- In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation.
- Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world.
- This is called information hiding.

Inheritance:

- OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes.
- This property of OOD is known as inheritance.
- This makes it easier to define specific class and to create generalized classes from specific ones.

Polymorphism

- OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name.
- This is called polymorphism, which allows a single interface performing tasks for different types.
- Depending upon how the function is invoked, respective portion of the code gets executed.

Software Quality

- Software Quality shows how good and reliable a product is.
- To convey an associate degree example, think about functionally correct software.
- It performs all functions as laid out in the SRS document.
- But, it has an associate degree virtually unusable program, even though it should be functionally correct, we tend not to think about it to be a high-quality product.

- **Portability:** A software is claimed to be transportable, if it may be simply created to figure in several package environments, in several machines, with alternative code merchandise, etc.
- **Usability:** A software has smart usability if completely different classes of users (i.e. knowledgeable and novice users) will simply invoke the functions of the merchandise.
- **Reusability:** A software has smart reusability if completely different modules of the merchandise will simply be reused to develop new merchandise.
- **Correctness:** Software is correct if completely different needs as laid out in the SRS document are properly enforced.
- **Maintainability:** A software is reparable, if errors may be simply corrected as and once they show up, new functions may be simply added to the merchandise, and therefore the functionalities of the merchandise may be simply changed, etc
- **Reliability:** Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make.
- **Efficiency.** The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth, and other resources.

Software Architecture

- Software architecture supports analysis of system qualities when teams are making decisions about the system rather than after implementation, integration, or deployment.
- Whether designing a new system, evolving a successful system, or modernizing a legacy system, this timely analysis enables teams to determine whether the approaches they've chosen will yield an acceptable solution.
- An effective architecture serves as the conceptual glue that holds every phase of the project together for all of its stakeholders, enabling agility, time and cost savings, and early identification of design risks.
- Building an effective architecture that enables rapid product delivery for today's needs while also addressing long-term goals can prove challenging.
- Failing to identify, prioritize, and manage trade-offs among architecturally significant qualities often leads to project delays, costly rework, or worse.

Types of Software Architecture

- Software Architecture Design considers the system structure and requirements to get a successful system architecture.
- It is important to focus on those things which will help you to create an architecture.

The important features of a good architecture are as follows:

- An architecture should try to address the requirements of several stakeholders.
- It should handle both the functional and quality requirements.
- It should realize all of the use cases, scenarios and hide implementation details.

- **Following are the types of software architecture.**

1. Business Architecture
2. Application Architecture
3. Information Architecture
4. Information Technology Architecture

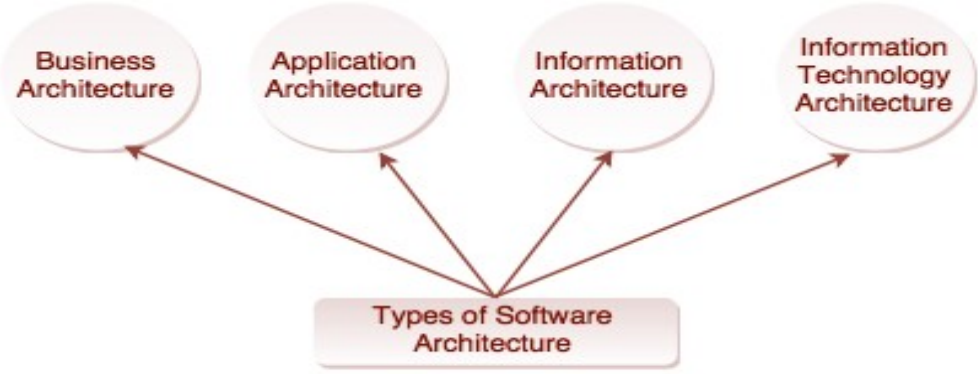


Fig. Types of Software Architecture

1. Business Architecture

- Business architecture defines the strategy of business, governance, organization and key business processes within an enterprise.
- This type of architecture focuses on the analysis and design of business processes.
- Focuses on defining the structure of the organization, including its business processes, goals, strategies, and stakeholders.
- It ensures that the IT systems align with the business objectives.

2. Application Architecture

- Application architecture serves as the blueprint for individual application system, their interactions and relationships to the business processes of the organization.
- Defines the structure and behavior of software applications, including how they interact with each other and with users.
- It focuses on the design of individual applications or systems.

3. Information Architecture

- Information Architecture defines the logical and physical data assets and data management resources.
- Focuses on the structure, organization, and management of data within an organization.
- It ensures that data is stored, processed, and accessed efficiently.

4. Information Technology Architecture

- IT architecture defines the hardware and software building blocks that make up the overall information system of the organization.
- Defines the overall structure of the IT infrastructure, including hardware, software, networks, and platforms.
- It ensures that the technology supports the organization's needs.

Software Architecture Design Process

- A good system should be designed with proper consideration for the user, IT infrastructure and business goals.
- Identifying important quality attributes for these areas helps to increase the reliability and scalability.
- A good design is focused on the user experience which supports user empowerment is flexible and configurable.

Following are the steps to compose the basic architecture design process.

A. Recognize the Problem

B. Estimate the Architecture Design

C. Modify the Architecture Design

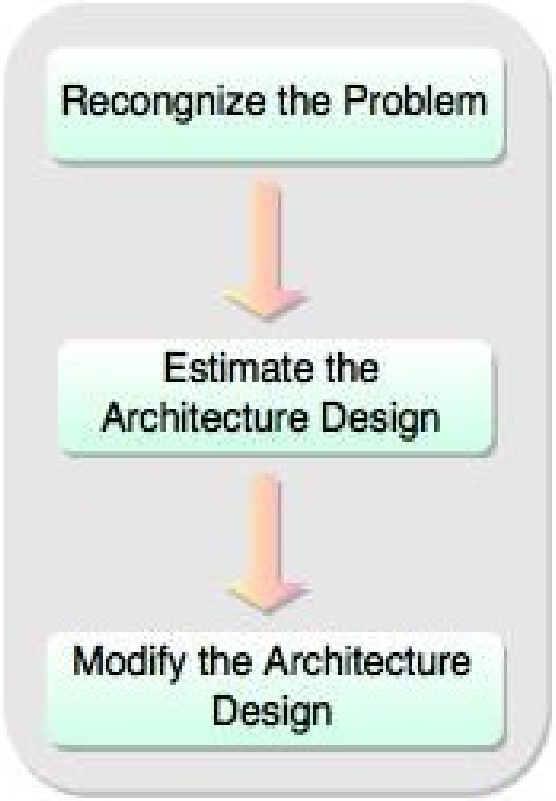


Fig. Software Architecture Design Process

A. Recognize the Problem

- Recognizing the problem is the most critical step of software architectural design process, because it affects the quality of the design that follows.
- Without recognizing the problem, it is not possible to clear the doubts or to create an effective solution.

B. Estimate the Architecture Design

- In this step, it evaluates the architecture correspondence to quality attribute requirements.
- If all the quality attributes are estimated as per the required standard then architectural design process is finished.

C. Transform the Architecture Design

- If all the quality attributes are not estimated as per the required standard then the third phase of software architecture design is entered that is transformation of architecture design.
- This phase is performed after an estimation of the architectural design.
- Transformation of architecture design is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.