UNIT II

SOFTWARE DESIGN AND UML DIAGRAMS

Design Principles (Modularity, Reusability, Abstraction),UML Diagrams: Use Case, Class, Activity, Sequence, Introduction to Design Patterns (Singleton, Factory, MVC),Building Simple System Architecture (Layered & Client-Server)

Design Principles:

Software is a program or set of programs containing instructions that provide the desired functionality. Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.

Software design is primarily about managing complexity. Software systems are often very complex and have many moving parts. Most systems must support dozens of features simultaneously. Each feature by itself might not seem very complicated. However, when faced with the task of creating one coherent structure that supports all of the required functionality at once, things become complicated very quickly. The primary objective of software design is to make and keep software systems well organized, thus enhancing our ability to understand, explain, modify, and fix them.

Based on this view of software design, disorganization is the antithesis of good software design. If created or modified without planning, software systems quickly become incomprehensible, tangled messes that don't work right and are impossible to fix. Even if a system starts out with a good design, we must consistently strive to preserve the integrity of its design throughout its lifetime by carefully considering all changes we make to it. Based on these principles, the important goals of software design are:

- Software that works
- Software that is easy to read and understand
- Software that is easy to debug and maintain
- Software that is easy to extend and holds up well under changes
- Software that is reusable in other projects

Key Principles of Software Engineering

1. Modularity: Breaking the software into smaller, reusable components that can be developed and tested independently.

2. Abstraction: Hiding the implementation details of a component and exposing only the necessary functionality to other parts of the software.

3. Encapsulation: Wrapping up the data and functions of an object into a single unit, and protecting the internal state of an object from external modifications.

4. Reusability: Creating components that can be used in multiple projects, which can save time and resources.

Maintenance: Regularly updating and improving the software to fix bugs, add new features, and address security vulnerabilities.

5. Testing: Verifying that the software meets its requirements and is free of bugs.

6. Design Patterns: Solving recurring problems in software design by providing templates for solving them.

7. Agile methodologies: Using iterative and incremental development processes that focus on customer satisfaction, rapid delivery, and flexibility.

8. Continuous Integration & Deployment: Continuously integrating the code changes and deploying them into the production environment.

Abstraction :

Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level details. It hides complex implementation details and showing only the essential features or functions to the user, simplifying interaction

Procedural Abstraction

- Sequence of instructions that have a specific and limited function. e Instructions are given in a named sequence
- Each instruction has a limited function
- The name of a procedural abstraction implies these functions, but specific details are suppressed.

An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)

- Data Abstraction
  - This is a named collection of data that describes a data object.
  - Data abstraction includes a set of attributes that describe an object.

The data abstraction for the door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door.**

- Control Abstraction
  - A program control mechanism without specifying internal details, e.g., semaphore

Advantages of abstraction:

- It separates design from implementation, which is easy to understand and manage.
- It helps in problem understanding and software maintenance.
- It reduces the complexity of modern computer programming for software users and engineers.
- It helps in program organization that can be generalized for recovering common problems and therefore it promotes software reuse.
- It also promotes scalability and helps in making early design decisions.

REUSABILITY

Reusability is a fundamental software design principle that focuses on creating software components in such a way that they can be used again in different applications or in different parts of the same application with little or no modification. Reusability aims to reduce duplication of code by designing components (such as functions, classes, modules, or libraries) that perform specific tasks and can be easily integrated into multiple systems. Instead of writing new code every time, developers reuse existing, well-tested components.

Characteristics of Reusability:

- ❖ Modularity — Software is divided into independent modules, each responsible for a single functionality.
- ❖ Loose Coupling — Components have minimal dependency on each other, making them easier to reuse.
- ❖ High Cohesion — Each component focuses on one well-defined task.
- ❖ Standard Interfaces — Clear and consistent interfaces allow components to be reused across different systems.
- ❖ Generality — Reusable components are designed to handle a variety of situations, not just one specific case.

Advantages of Reusability:

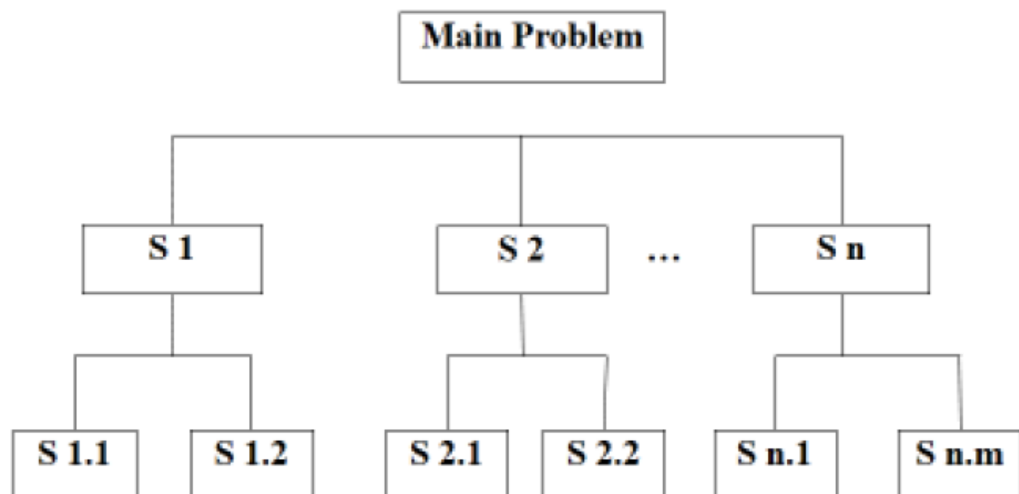
- ❖ Reduced Development Time — Existing components can be reused, saving coding effort.
- ❖ Improved Reliability — Reused components are usually well-tested and stable.
- ❖ Lower Maintenance Cost — Fixes or updates can be made in one place and reused everywhere.
- ❖ Consistency — Common functionality behaves the same across applications.
- ❖ Better Productivity — Developers can focus on new features rather than rewriting code.

Design Strategies:

- ➢ **Top-down and Bottom-up**
    - o A hierarchical organization helps in taking design decisions and performing design activity.
    - o A design activity varies with design techniques, such as structured design, Jackson structured design, object-oriented design, etc.
    - o The design techniques are based on design strategies that reflect the quality of design.
    - o Top-down and bottom-up are the most popular design strategies used in the industry.
    - o In the top-down strategy, the system is viewed as a single "black-box" program with a high-level interface with the external environment. It starts with a general level of specification and moves to a specific level of specifications.
    - o • In the bottom-up strategy, specific levels of details are designed and further these are to design the final system.

Design Strategies: Top-down strategy

- It starts with the global view defined at a high level of abstraction of the overall system. The system is refined and decomposed into the next lower-level subsystems.
- A top-down strategy (also referred to as stepwise refinement) is essentially partitioning a system to elaborate on its subsystems.
- Each subsystem is again decomposed into the specific level of detail to identify the concrete level of the subsystems.
- The process of elaboration is continued until we reach at the concrete level of detail. At this level, design decisions are taken easily and subsystems can easily be developed and managed.



Advantages of Top-Down Approach

- The main advantage of the top-down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.
- Simplifies complex problem-solving by breaking down the system into smaller sub-problems.
- Enhances clarity and understanding with a high-level overview.

Disadvantages of Top-Down Approach

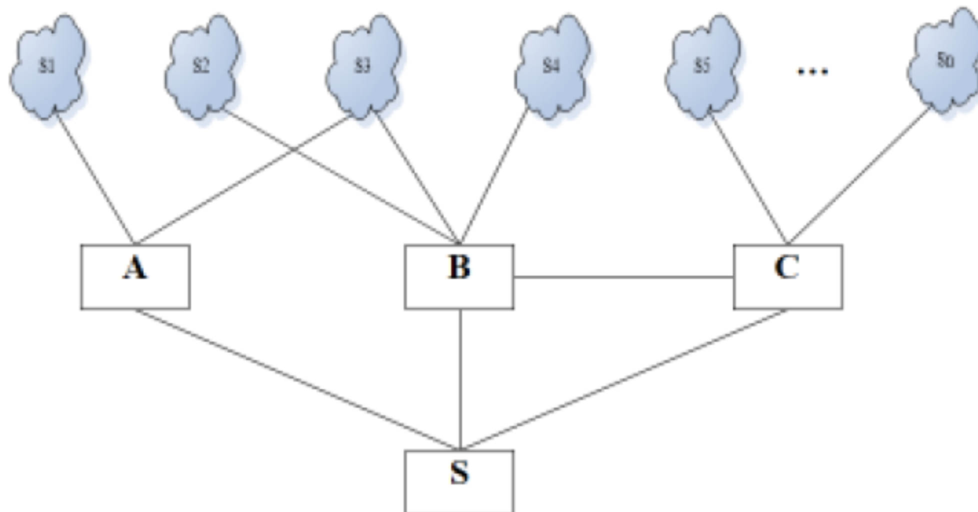
- Project and system boundaries tend to be application specification-oriented. Thus, it is more likely that the advantages of component reuse will be missed.
- The system is likely to miss, the benefits of a well-structured, simple architecture.

Design Strategies: Bottom-up strategy

- A bottom-up strategy (also referred to as layers of abstraction) is piecing together the subsystems to form the whole system.
- It starts with the lower-level subsystems at the bottom level, i.e., the individual elements in the system.

- ➢ The systems are then combined together to form the upper level of abstraction, i.e., subsystems. In turn, subsystems are again put together to design the next level of the system.
- ➢ This process of layering the abstraction levels is continued until the top level of the system is reached.
- ➢ Each level of abstraction performs services to its upper level of the system.

Design Strategies: Bottom-up strategy



Advantages of Bottom-up Approach

- ❖ The economics can result when general solutions can be reused.
- ❖ It can be used to hide the low-level details of implementation and be merged with the top-down technique.
- ❖ Simplifies the integration process by ensuring that low-level components are thoroughly tested and validated before being combined into higher-level modules.

Disadvantages of Bottom-up Approach

- ❖ It is not so closely related to the structure of the problem.
- ❖ High-quality bottom-up solutions are very hard to construct.
- ❖ It leads to the proliferation of'potentially useful' functions rather than the most appropriate ones.

Modularity

- ❖ Modularization is the process of breaking a system into pieces called modules so that these can be easily managed and implemented.
- ❖ A module is a part of a software system that can be separately implemented and a change in a module has a minimal effect on other modules.
- ❖ A module can be a function, procedure, program, subroutine, class, package, framework, library files, templates, components, etc..

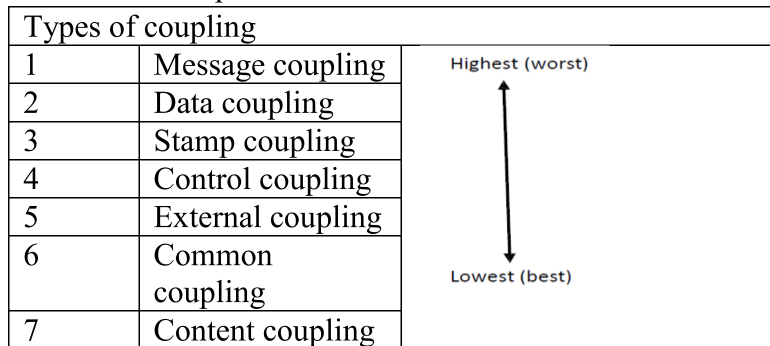
- A modular system consists of various modules linked via interfaces. An interface is a kind of link or relationship that combines two or more modules together. Modularity is the measurement of modularization of a system into pieces.

A module has the following characteristics:

- Modularity measures the independency of the parts of a system and enhances separation of concerns.
- Modularity enhances quality factors, such as portability, extensibility, compatibility, scalability, etc. A module contains data structures, input/output statements, instructions, and processing logic. A module can be called into another module.
- A module can be reused within other modules. A modular system can be easily developed, maintained, and debugged.
- Modularity reduces design complexities in a system through distributed software architectures. Modularity uses abstraction, which helps in defining a subsystem.
- Modularity improves design clarity and understandability.
- A modular design focuses on minimizing the interconnections between modules.
- In a modular system design, several independent and executable modules are composed together to construct an executable application program.
- The programming language support, interfaces, and the information-hiding principles ensure modular system design.
- The most common criterions are functional independency; levels of abstraction; information hiding; functional diagrams, such as DFD, modular programming languages, coupling, and cohesion.
- An effective modular system has low coupling and high cohesion.

- Coupling
  - It is the strength of interconnection between modules. It is the measure of the degree of interdependency between modules.
  - Modules are either loosely coupled or strongly coupled.
  - In strong coupling, two modules are dependent on each other. Strong coupling can be observed in assembly language programs where change in one part or data requires changes in other parts of the system.
  - Also, these are difficult to reuse, test, and release for the operation.
  - In loose coupling, there are weak interconnections between modules.
  - Interdependency between modules increases as coupling increases.
  - Interconnection between modules can be measured by the number of function calls, number of parameters passed, return values, data types, sharing of data files or data items, etc.
  - The strength of interconnection between modules is influenced by the level of complexity of the interfaces, type of connection, and the type of communication.
  - The complexity of an interface is the measure of the type of parameters, number of parameters, common sharing of data or code communicated by modules.

- The connection between modules is established by relating one module to another through parts of the system or through certain data value.
- Communication is the data passed, type of data, and the control information passed to another module.



Types of coupling		
1	Message coupling	Highest (worst)
2	Data coupling	
3	Stamp coupling	
4	Control coupling	
5	External coupling	
6	Common coupling	Lowest (best)
7	Content coupling	

Types of coupling

Message Coupling: Message coupling is the lowest (i.e. best) type of coupling which exists between modules. For example, in C++ objects communicate with each other by sending a message through parameters in the function call.
Data Coupling: Data coupling exists between modules when data are passed as parameters in the argument list of the function call. Each datum is a primary data item (e.g., integer, character, float, etc.) that can be used between modules.

Stamp Coupling: It occurs between modules when data are passed by parameters using complex data structures, which may use parts or the entire data structure by other modules. For example, structures in C, records in Pascal, etc.

Control Coupling: It exists when one module controls the flow of another by passing control information such as flag set or switch statements. For example, a flag variable decides what function or module is to be executed next.

External Coupling: External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Common Coupling: Common coupling is when two modules share common data (e.g., a global variable). In C language, external data items are accessed by all modules in the program. If there is any change in the shared resource, it influences all the modules using it.

Content Coupling : It is the highest coupling (worst). Content coupling exists between two modules when one module refers or shares its internal working with another module. Accessing local data items or instructions of another module is an example of content coupling.

Cohesion:

- Cohesion of a single module is the degree to which the elements of a single module are functionally related to achieve an objective.

- Module cohesion represents how tightly bound the internal elements of the module are to one another.
- High cohesion is often characterized by more understandability, modifiability, and maintainability of the modules in a system.
- Low cohesive modules are highly undesirable and should be modified or replaced to meet the objectives of modular design.
- The important goal of designers is to maximize cohesion and minimize coupling.
- Cohesion between the elements of a module is measured in terms of the strength of the hiding of the elements within the module itself.
- A functionally independent module has higher cohesion as compared to dependent modules.

Levels of cohesion:

1. Functional cohesion — Strongest (best)
2. Sequential cohesion
3. Communicational cohesion
4. Procedural cohesion
5. Temporal cohesion
6. Logical cohesion
7. Coincidental cohesion — Weakest (worst)

Functional Cohesion: In a functionally cohesive module, all the elements of the module perform a single function. For example, “log” computes the logarithm of a number and “printf” prints the results.

Sequential Cohesion: Sequential cohesion exists when the output from one element of a module becomes the input for some other element.. For example, “withdraw money” and “update balance” both are bound together to withdraw money from an account.

Communicational Cohesion: In communicational cohesion, all the elements of a module operate on the same input or output data. For example, “print and punch the output file” can be communicational cohesion. The binding of elements in a module is higher than procedural cohesion.

Procedural Cohesion: Procedural cohesion contains the elements which belong to a common procedural unit. The functions are executed in a certain order. For example, “entering, reading, and verifying the ATM password” are bound in an ordered manner for the procedurally cohesive module “enter password.”

Temporal Cohesion: Sometimes, a module performs several functions in a sequence but their execution is related to a certain time. For example, “a database trigger is activated on executing a certain procedure.”

Logical Cohesion: Logical cohesion exists when logicaly-related elements of a module are placed together. All the parts communicate with each other by passing control information such as flag variable, using some shared source code, etc.

Coincidental Cohesion: It occurs when the elements within a given module have no meaningful relationship to each other.