SOFTWARE ENGINEERING

UNIT --3

Author
DSALGO

Software Design

- It is conceptualizing and defining the architecture, components, and interfaces of a software system.
- Translating user needs, business goals, and technical constraints into a structured blueprint.
- Guiding the development and implementation of the software to meet specified requirements.

General Tasks in Software Design/steps involved to design stage of s/w

- **Requirements Analysis:** Understanding and analyzing stakeholder needs and objectives.
- **Architectural Design**: Creating the overall structure and organization of the system.
- **User Interface Design:** Designing intuitive user interfaces for usability.
- **Data Design**: Designing the data model and database structure.
- **Testing and Validation Design**: Planning tests to ensure the system meets requirements and standards.

Principles in software design

- **Modularity**: Break the system into manageable modules for reusability and maintainability.
- **Encapsulation**: Hide internal complexities within modules to reduce dependencies.
- **Abstraction**: Simplify complexity by exposing only relevant information.
- **Coupling and Cohesion:** Minimize dependencies between modules and ensure each has a clear purpose.
- **Simplicity:** Keep designs straightforward and easy to understand to improve maintainability.

Characteristics of Good software design

- **User-Centricity**: Prioritizing user needs and experiences to ensure the software is intuitive and meets user requirements effectively.
- **Modularity**: Breaking down the system into manageable modules to promote reusability, maintainability, and scalability.
- **Reliability**: Ensuring the software operates stably and consistently, minimizing errors, failures, and downtime.
- **Security**: Incorporating robust security mechanisms to protect data, prevent unauthorized access, and mitigate vulnerabilities.
- **Maintainability**: Designing the software in a way that facilitates easy maintenance, debugging, and troubleshooting over its lifecycle.

Architectural design

- Architectural design is a critical phase in software development
- where the overall structure and organization of a software system are defined.
- This phase involves making high-level decisions about system's components, modules, interactions, deployment architecture.

key aspects of architectural design

- Break the system into manageable components/modules.
- Define interfaces and communication between components/modules.
- Choose appropriate architectural styles and patterns.
- Design for scalability, performance, and reliability.
- Address security requirements & implement security features.

Low-level design (LLD)

- (LLD) is the detailed design phase of s/w development.
- It specifies how individual components will be implemented.
- It includes designing algorithms, data structures, interfaces, and interactions.
- LLD follows high-level architectural design.
- It provides a blueprint for developers to follow during implementation.

Types of low level design

1.Modularization:

- Modularization involves breaking down the system into smaller, manageable modules or components based on functionality and responsibilities.
- Each module encapsulates a specific set of related functionalities, promoting reusability, maintainability, and scalability.
- Modularization helps organize the system's structure and facilitates parallel development by allowing teams to work on different modules simultaneously.
- **Prinicples:**
  - **Encapsulation**: Hide internal details, expose necessary interfaces.
  - **High Cohesion**: Modules focus on single tasks.
  - **Low Coupling**: Minimize inter-module dependencies.
- **Example**: Consider a library management system.
  - **Module 1**: "Book Management" deals with functionalities related to adding, updating, and deleting books.
  - **Module 2**: "User Management" handles user registration, login, and profile management.
  - **Module 3**: "Borrowing Management" is responsible for handling book borrowing and returning operations.

2.Structure Chart

- A structure chart is a graphical representation that depicts the hierarchical structure of modules within the system.
- It illustrates how modules are organized and interconnected, showing the dependencies and relationships between them.
- Each module is represented as a box, and the connections between modules indicate the flow of control or data between them.
- Structure charts provide a visual overview of the system's architecture, aiding in understanding and communication among team members.
- **Principles**:
  - **Hierarchy**: Modules arranged in levels.
  - **Clarity**: Easy to understand connections.
  - **Consistency**: Maintain uniformity.
- **Example**: Continuing with the library management system example:
  - **Level 1:** Main module - "Library Management System".
  - **Level 2**: Sub-modules:
  - Book Management, User Management ,Borrowing Management

3.Pseudocode

- Pseudocode is a high-level, informal description of the algorithms and logic used within individual modules or components.
- It uses a mix of natural language and simplified programming constructs to describe the steps and operations performed by the module.
- Pseudocode helps clarify the intended behavior of module without getting bogged down in specific syntax or implementation details.

- It serves as a bridge between design and implementation, guiding developers in writing actual code based on the outlined logic and algorithms.
- **Principles**:
  - **Clarity**: Clear and concise language.
  - **Readability**: Understandable by all team members.
  - **Abstraction**: Focus on high-level logic.
- **Examples**:

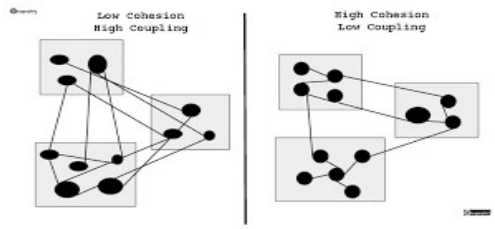
```
procedure searchBook(library, title):
for each book in library:
        if book.title == title:
        return book
return "Book not found"
```

Coupling

- Coupling in software design refers to the degree of interdependence between software modules or components.
- It describes how much one module relies on another module.
- There are different types of coupling, ranging from low to high, each influencing the design, maintenance, and flexibility of the software system.

Cohesion

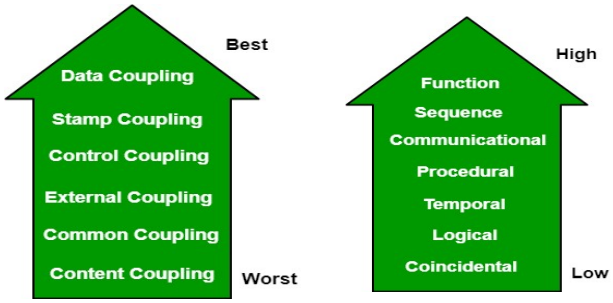
- Cohesion measures how closely elements within a module work together toward a shared goal.
- High cohesion means elements are strongly related and focused on a single task, while low cohesion indicates they are loosely related and perform diverse tasks.



"*High coupling and low cohesion hinder system flexibility, while low coupling and high cohesion enhance maintainability and scalability."*

Types of coupling

- **Data Coupling**:
  - Modules communicate by passing only data.
  - **Example**: A customer billing system where modules exchange data like customer information and purchase details.
- **Stamp Coupling:**



  - Entire data structures are passed between modules.
  - **Example**: Passing a complete record or object between modules for processing.
- **Control Coupling:**
  - Modules communicate by passing control information.
  - **Example**: A sort function that takes a comparison function as an argument to determine sorting order.
- **External Coupling:**
  - Modules depend on external entities like hardware or protocols.
  - **Example**: Modules relying on external file formats or device specifications.
- **Common Coupling:**
  - Modules share global data structures.
  - **Example**: Multiple modules accessing and modifying the same global variables.
- **Content Coupling:**
  - One module modifies the data or controls the flow of another module.
  - **Example**: Module A directly modifies data or controls the execution flow of Module B.

Types of cohesion

- **Functional Cohesion:**
  - Elements within a module are grouped together because they contribute to a single, well-defined task or function.
  - **Example**: A module containing functions that collectively implement a specific feature, such as user authentication or data validation.
- **Sequential Cohesion:**
  - Elements within a module are arranged in a sequential order, where the output of one element serves as the input to the next.
  - **Example**: A module containing functions that process data in a step-by-step manner, with each function building upon the output of the previous one.
- **Communicational Cohesion:**
  - Elements within a module are grouped together because they operate on the same data or share a common input/output.
  - **Example**: A module containing functions that manipulate the same data structure or communicate through shared variables.
- **Procedural Cohesion:**
  - Elements within a module are grouped together because they are related to a specific procedure or process.
  - **Example**: A module containing functions that collectively perform a specific task, such as processing an order.
- **Temporal Cohesion:**
  - Elements within a module are related by the timing of their execution.
  - **Example**: A module containing functions that must execute in a specific sequence, such as initialization followed by cleanup tasks.
- **Logical Cohesion:**
  - Elements within a module perform similar tasks, but the relationship is not based on functionality.
  - **Example**: A module containing functions related to arithmetic operations but not necessarily related in terms of purpose.
- **Coincidental Cohesion:**
  - Elements within a module are grouped arbitrarily, without any logical relationship.
  - **Example**: A utility module containing unrelated functions like sorting algorithms and file I/O operations.

Function-Oriented Design (FOD)

- It is an approach to software design where the system is decomposed into a set of interacting units or modules, with each module having a clearly defined function or purpose.

- Emphasizing functions or procedures over data or objects.
- Encapsulating related functionality within modules and defining clear interfaces for interaction.
- Designing modules to perform specific tasks, orchestrating interactions carefully for desired system behavior.
- Promoting modularity, encapsulation, and reusability for easier management and maintenance of complex software systems.

Object-Oriented Design (OOD)

- (OOD) is an approach to software design that emphasizes modeling real-world entities as objects with attributes and behaviors
- Modeling real-world entities as objects with attributes and behaviors.
- Organizing objects into classes, serving as blueprints for creating individual objects.
- Encapsulating related data and behavior within classes for modularity and reusability.
- Establishing relationships b/w classes, like inheritance & composition.
- Using abstraction and polymorphism to hide implementation details and enable flexibility.

Differences b/w fod vs ood

Factors	FOD	OOD
Focus	Functions or procedures.	Real-world entities as objects.
Unit of Organization	Function or procedure.	Object or class.
Modularity	Decomposing into function-based units.	Encapsulating data and behavior within classes
Data Handling	Functions operate on data.	Data encapsulated within objects
Inheritance and Polymorphism	Doesn't emphasize these concepts; focuses on function-based decomposition.	Supports inheritance and polymorphism for flexibility

Fundamental issues in s/w design

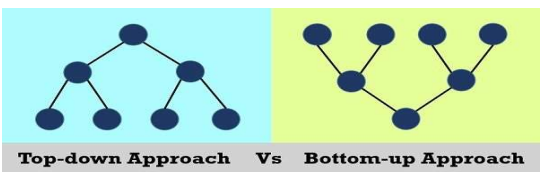
- Clarify stakeholder requirements.
- Break system into cohesive modules with minimal coupling.
- Hide complexity and promote reusability with clear interfaces.
- Design adaptable architectures for future changes.
- Plan for growth and optimize system performance.
- Prioritize intuitive interfaces for user satisfaction.

Top-Down Approach

- Starts with overarching view.
- Breaks down into smaller components.
- Emphasizes structured decomposition.
- May overlook smaller details initially.

Bottom-Up Approach

- Starts with individual components.
- Integrates incrementally to build system.
- Supports early testing and validation.



- May lack clear architectural vision initially.

Software matrices

- Software matrices, or software metrics, are quantitative measures
- used to assess various aspects of software development, quality, and performance.
- These metrics provide objective data to evaluate and improve the software development process
- Examples: size, complexity, quality, effort, functionality, risk, and process metrics.

Different types of S/W Matrices

- **Size Metrics:** Measure the size of software artifacts, like lines of code or number of modules.
- **Complexity Metrics:**Measure the complexity of software systems, such as cyclomatic complexity, Halstead complexity measures, and McCabe's complexity.
- **Quality Metrics**: Assess various aspects of software quality, including defect density and code coverage.
- **Effort Metrics:** Estimate the effort required for software development tasks.
- **Functionality Metrics**: Measure the functionality provided by a software application.
- **Risk Metrics**: Evaluate and prioritize project risks.
- **Process Metrics:** Measure the efficiency and effectiveness of software development processes.

Software Measurements:

- Specific instances of data collected during software development.
- Serve as raw data for calculating metrics or monitoring the development process.
- **Examples**: lines of code, defect counts, execution time, productivity rates, and test coverage percentages.

Lines of Code (LOC)

- Lines of Code (LOC) is a measure of the number of lines of source code in a software project
- providing a simple method for estimating project size.
- This count includes all code lines, including comments & blank lines.
- However, different programming languages may define lines differently. While commonly used, LOC may not directly reflect software complexity or functionality.
- It's often used with other metrics for a fuller picture.
- Limitations include ignoring coding style differences and being influenced by factors like code reuse and third-party libraries.

```
void selSort(int x[], int n) {
 //Below function sorts an array in ascending order
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
    if (x[j] < x[min])
    min = j;
    temp = x[i];
    x[i] = x[min];
    x[min] = temp;
  }
}
```

Halstead software Science

- Halstead metrics quantify software complexity and size using distinct operators and operands.
- **Purpose**: Assess complexity, effort, and quality of software.
- Aid software development, maintenance, and quality assurance.
- **Program Length (N):** N=N1 +N2
- **Program Vocabulary (n):** n=n1 +n2
- **Volume (V):** V=N×log2 (n)
- **Potential Minimum Volume:** V* = (2 + n2*) * log2 (2 + n2*)
- **Program level (L):** L=V*/V
- **Difficulty (D) : D=** (n1/2) * (N2/n2)
- **Effort :** E=V/L=D*V

  - Where
  - N1 is the total number of occurrences operators.
  - N2 is the total number of occurrences operands.
  - n1 is the total number of distinct operators.
  - n2 is the total number of distinct operands.
- **Advantages**:
  - Objective measurement aiding decision-making.
  - Early issue detection.
  - Effort estimation for better planning.
  - Quality assessment and improvement.
- **Disadvantages**:
  - Influence of coding style on metrics.
  - Language dependency.
  - Limited scope, overlooks design and usability.

Que 3.30. What do you understand by token count ? Consider a program having :

i. Number of distinct operator : 12
ii. Number of operands : 5
iii. Total number of operator occurrences : 20
iv. Total number of operand occurrences : 15

Calculate the different Halstead software metrics for above programs.

AKTU 2013-14, Marks 10

Answer

Token count : Token count is defined as size of program *i.e.*, number of tokens, where a token is a lexical token (keyword, arithmetic operator, constants, symbol), as recognized by compiler.

Numerical :

$$\text{Program length} = N = N_1 + N_2 = 20 + 15$$
$$N = 35$$
$$\text{Program vocabulary} = \eta = \eta_1 + \eta_2$$
$$\eta = 12 + 05$$
$$\eta = 17$$
$$\text{Volume of program} = V = N^* \log_2 \eta$$
$$V = 25^* \log 17 = 25^* 4.08 = 102$$
$$\text{Potential volume of program} = V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$
$$= (2 + 12) \log(2 + 12)$$
$$= 14 \log 14 = 14 \times 3.80 = 53.2$$
$$\text{Program level} = L = V^*/V = 53.2/102 = 0.54$$

  - Complex interpretation

Que 3.31. For the following 'C' program estimate the Halstea length and volume measures. Compare Halstead length and volu measures of size with LOC measure.

```
// Program to calculate GCD of two number
int compute-gcd(x, y)
{
  int x, y;
  while (x != y)
      if (x > y) then x = x - y;
      else y = y - x;
  return x;
}
```

AKTU 2014-15, Marks 1

Operator	Occurrence	Operands	Occurrence
Compute-gcd	1	x	8
=	3	y	7
!	1		
while	1		
>	1		
–	2		
return	1		
if	1		
(	1		
)	1		
{ }	3		
;	4		
int	2		
else	1		
then	1		
$n_1 = 15$	$N_1 = 24$	$n_2 = 2$	$N_2 = 15$

The length, $N = N_1 + N_2 = 24 + 15 = 39$

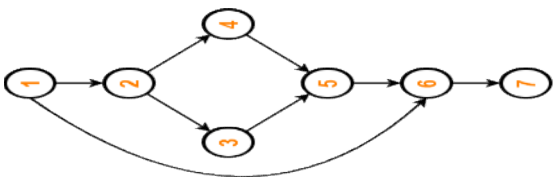
The volume, $V = N \log_2 (n_1 + n_2) = 39 \log_2 17 = 159.41$

Function points:

- Measure software size and complexity from a user's perspective.
- Aid in project estimation, resource allocation, and performance evaluation.
- Calculated based on five types of functionality: inputs, outputs, inquiries, files, and interfaces.
- **Advantages**:
  - Standardized, technology-independent measurement.
  - Accurate project estimation and resource allocation.
  - Facilitates benchmarking and comparison.
  - Enables better project planning.
- **Disadvantages**:
  - Subjective complexity assessment may introduce variability.
  - Requires expertise and training for accurate calculation.
  - May not capture all aspects of software complexity.
  - Limited applicability to certain projects or technologies.

Cyclomatic complexity:

- Measures program complexity by counting independent paths in its control flow.
- **Formula**: V(G)=E−N+2P where E is edges, N is nodes, and P is connected components.
- Helps assess maintainability and testability.
- Higher values indicate greater complexity.
- **Advantages**: quantifies complexity, identifies problematic areas.
- **Disadvantages**: may not capture all complexities, interpretation varies.



Control Flow Graph

- Cyclomatic Complexity = E – N + 2P= 8 – 7 + 2*1= 3