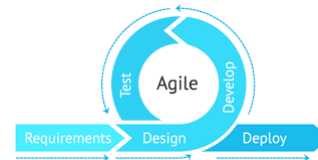


Software Engineering And Testing

Software Design

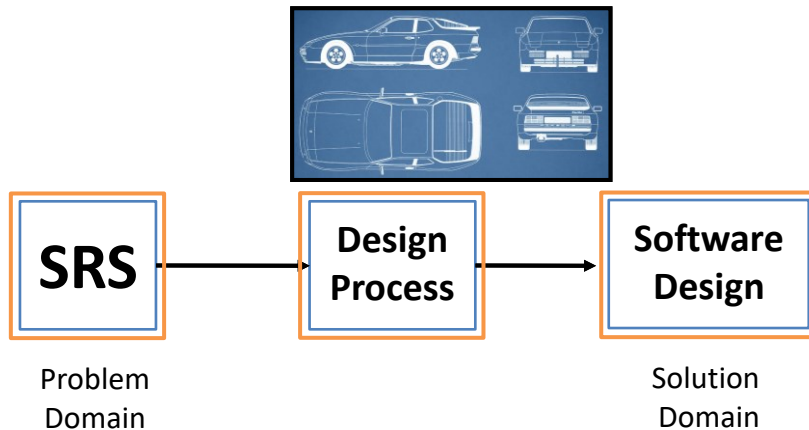


Outline

- Overview of the Design Process
- Characterize a Good Software Design
- Cohesion and Coupling
- Layered Arrangement of Modules
- Approaches to Software Design
- Function-Oriented Software Design
 - Overview of SA/SD Methodology
 - Structured Analysis

What is Design

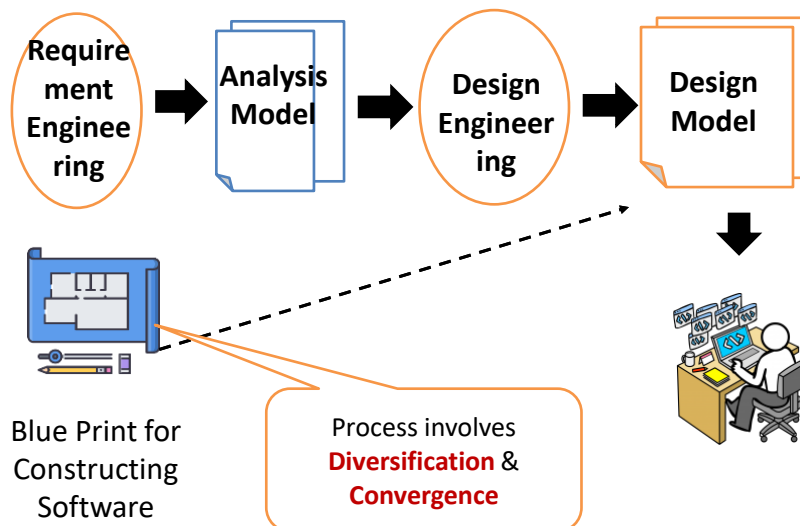
- A **meaningful representation** of something to be built
- It's a **process** by which **requirements** are **translated** into **blueprint** for constructing a software
- **Blueprint** gives us the **holistic view** (entire view) of a **software**



3

Software Design Process?

- Software **design** is the **most creative part** of the development process



4

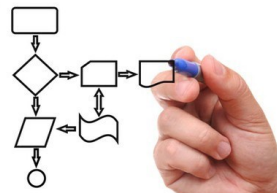
Software design work products

- For a design to be easily implemented in a conventional programming language, the following items must be designed during the design phase.
- **Different modules** required to implement the design solution.
- **Control relationship among** the identified **modules**. The relationship is also known as the call relationship or invocation relationship among modules.
- **Interface among** different **modules**. The interface among different modules identifies the exact data items exchanged among the modules.
- **Algorithms** required **to implement** each individual **module**.
- **Data structures of** the individual **modules**.

5

Characteristics of good Design

- The design must **implement all explicit requirements** available in requirement model.
- The design must **accommodate all implicit requirements** given by stakeholders.
- The design must be **readable & understandable**
- A good design solution should adequately address **resource, time** and **cost optimisation issues**.
- The good design should **provide complete picture of the software**, addressing the **data, functional** and **behavioral** domains.



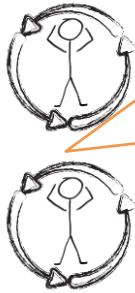
6

Cohesion & Coupling

A **good software design** implies **clean decomposition** of the **problem into modules**, and the **neat arrangement** of these **modules** in a **hierarchy**.

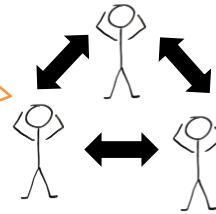


The primary **characteristics** of **neat module decomposition** are **high cohesion** and **low coupling**.



A **cohesive module** performs a single task, requiring little interaction with other components.

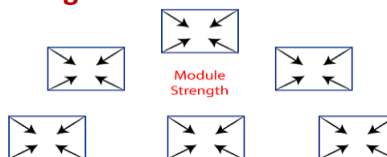
A **Coupling** is an indication of the relative interdependence among modules.



7

Cohesion

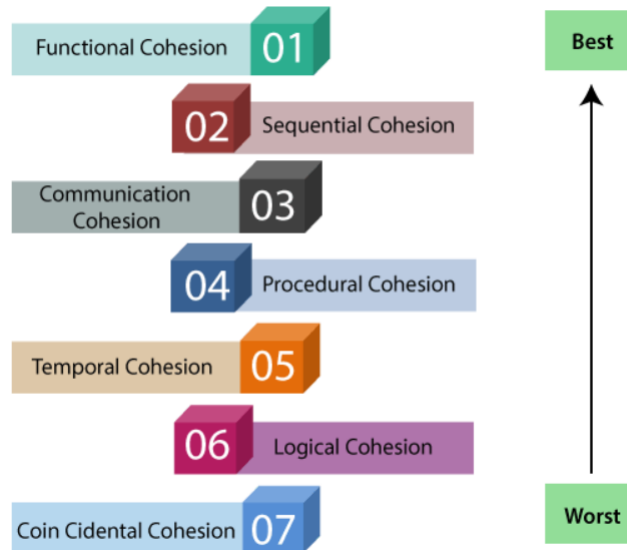
- Cohesion is an **indication** of the **relative functional strength** of a module.
- A **cohesive module** performs a **single task**, requiring **little interaction** with other components.
- Stated simply, a **cohesive module** should (ideally) **do just one thing**.
- A module having **high cohesion** and **low coupling** is said to be **functionally independent** of other modules.
- By the term functional independence, we mean that a **cohesive module performs a single task or function**.



Cohesion= Strength of relations within Modules

8

Classification of Cohesion



9

Classification of Cohesion

○ Coincidental Cohesion

- Coincidental cohesion occurs when the elements are not related to each other.
- Examples - Module for miscellaneous functions, customer record usage, displaying of customer records, calculation of total sales, and reading the transaction record, etc.

○ Logical cohesion

- A module is said to be logically cohesive, if all elements of the module perform similar operations.
- For Ex., error handling, data input, data output, etc.
- An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module. In this case, the module contains a random collection of functions.

10

Classification of Cohesion

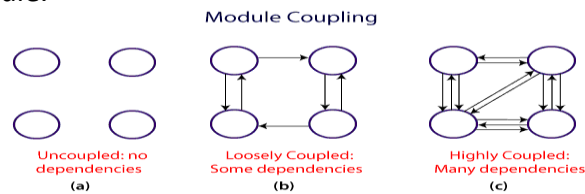
- **Temporal cohesion**
 - When a module contains functions that are related by the fact that all the functions must be executed in the same time span.
 - For Ex., the set of functions responsible for initialization, start-up, shutdown of some process, etc.
- **Procedural cohesion**
 - If the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective
 - For Ex., the algorithm for decoding a message.
- **Communicational cohesion**
 - If **all functions** of the module **refer to the same data structure**
 - For Ex., the set of functions defined on an array or a stack.¹¹

Classification of Cohesion

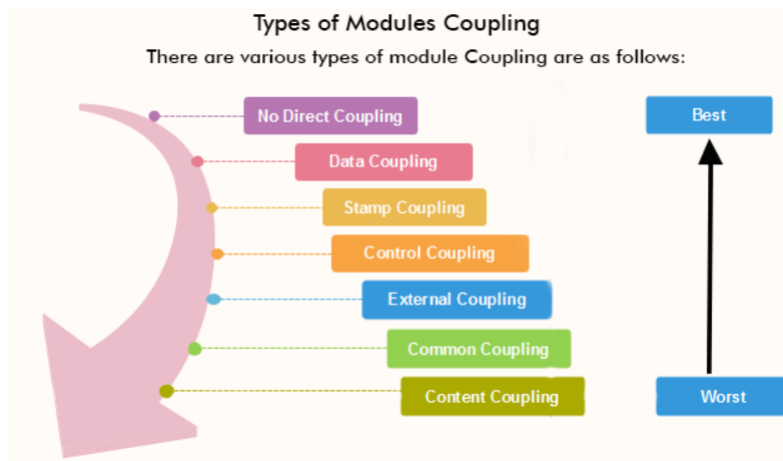
- **Sequential cohesion**
 - If the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.
 - For Ex., In a Transaction Processing System, the get-input, validate-input, sort-input functions are grouped into one module.
- **Functional cohesion**
 - If different elements of a module cooperate to achieve a single function.
 - For Ex., A module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.

Coupling

- **Coupling** between two modules is a **measure of the degree of interdependence** or interaction **between the two modules**.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- If **two modules interchange large amounts of data**, then they are **highly interdependent**.
- The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.



Classification of Coupling



Classification of Coupling

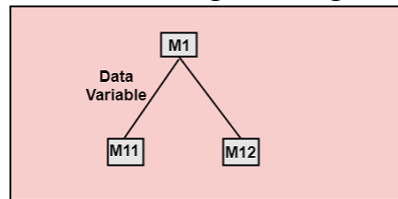
○ No direct coupling

- There is no direct coupling between M1 and M2
- In this case, modules are subordinates to different modules. Therefore, no direct coupling.



○ Data coupling

- Two modules are data coupled, if **they communicate through a parameter**.
- An example is an elementary (primal) data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.



15

Classification of Coupling

○ Stamp coupling

- This is a special case (or extension) of data coupling
- Two modules ("A" and "B") exhibit stamp coupling if **one passes** directly to the other a **composite data item** - such as a record (or structure), array, or (pointer to) a list or tree.
- This occurs when **ClassB** is **declared as a type** for an argument of an **operation of ClassA**

○ Control coupling

- If **data** from **one** module **is used to direct the order of instructions** execution **in another**.
- An example of control coupling is a flag set in one module and tested in another module

16

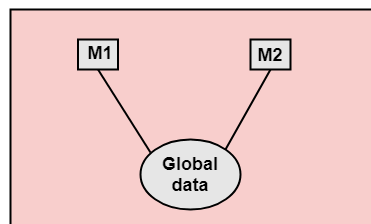
Classification of Coupling

- **Content coupling**
 - Content coupling occurs when **one component secretly modifies data** that is **internal to another component**.
 - This violates information hiding – a basic design concept.
 - Content coupling exists between two modules, if they share code e.g., a branch from one module into another module.
- **External coupling**
 - External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

17

Classification of Coupling

- **Common coupling**
 - Two modules are common coupled if they share information through some global data items.



18

Difference between Cohesion and Coupling

Cohesion	Coupling
Cohesion is the concept of intra-module.	Coupling is the concept of inter-module.
Cohesion represents the relationship within a module.	Coupling represents the relationships between modules.
Increasing cohesion is good for software.	Increasing coupling is avoided for software.
Cohesion represents the functional strength of modules.	Coupling represents the independence among modules.
Highly cohesive gives the best software.	Whereas loosely coupling gives the best software.
In cohesion, the module focuses on a single thing.	In coupling, modules are connected to the other modules.
Cohesion is created between the same module.	Coupling is created between two different modules.

Layered Arrangement of Modules

- The **control hierarchy** represents the organisation of program components in terms of their **call relationships**.
 - **control hierarchy is also called program structure.**
- **Most common notation:** a tree-like diagram called structure chart.
- An important characteristic feature of a good design solution is layering of the modules.
- A layered design achieves control abstraction and is easier to understand and debug.

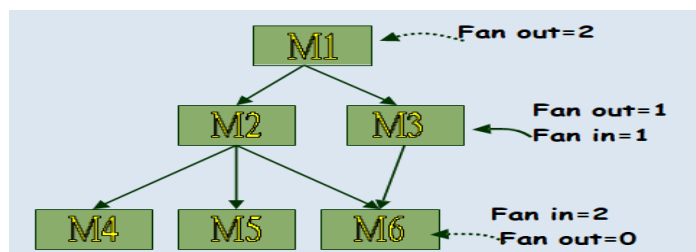
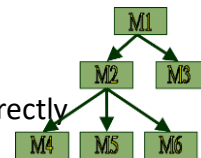
Characteristics of Module Structure

- **Superordinate and subordinate modules:**
 - **A module that controls another module:** said to be superordinate to the later module.
 - **Conversely, a module controlled by another module:** said to be subordinate to the later module.
- **Visibility:**
 - **A module A is said to be visible by another module B,** if A directly or indirectly calls B.
- **The layering principle requires / control abstraction :** modules at a layer can call only the modules immediately below it.

21

Characteristics of Module Structure

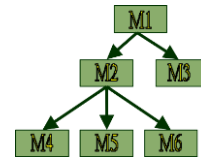
- **Depth:** number of levels of control
- **Width:** overall span of control.
- **Fan-out:** a measure of the number of modules directly controlled by given module.
- **Fan-in:** indicates how many modules directly invoke a given module.
 - High fan-in represents code reuse and is in general encouraged.



22

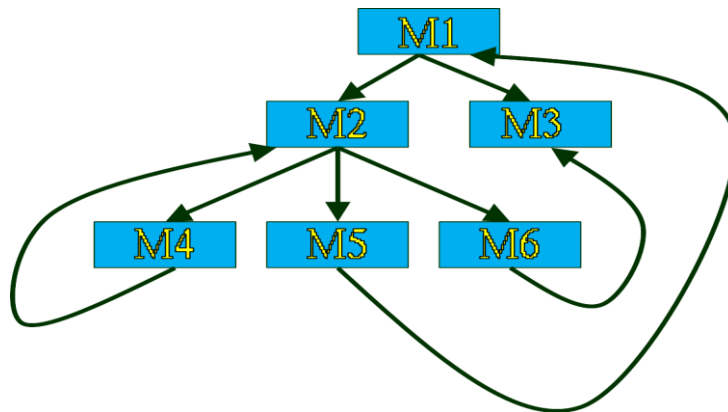
Goodness of Design

- **A design having modules:** with high fan-out numbers is not a good design.
 - a module having high fan-out lacks cohesion.
- **A module that invokes a large number of other modules:** likely to implement several different functions:
 - not likely to perform a single cohesive function



23

Bad Design



24

Design Approach

- **Two fundamentally different software design approaches:**
 - Function-oriented design
 - Object-oriented design
- **These two design approaches are radically different.**
 - However, are complementary rather than competing techniques.
 - Each technique is applicable at different stages of the design process.

25

Function-oriented Design

- **A system is looked upon as something that performs a set of functions.**
- **Starting at this high-level view of the system:** each function is successively refined into more detailed functions (**top-down decomposition**).
 - Functions are mapped to a module structure.
- Example: **The function create-new-library- member:**
 - creates the record for a new member,
 - assigns a unique membership number
 - prints a bill towards the membership

26

Function-oriented Design

- **The system state is centralized:** accessible to different functions, member-records:
 - available for reference and updating to several functions:
 - create-new-member
 - delete-member
 - update-member-record
- Example : **Create-library-member function consists of the following sub-functions:**
 - assign-membership-number
 - create-member-record
 - print-bill
 - **Split these into further subfunctions, etc.**

27

Object-Oriented Design

- **System is viewed as a collection of objects (i.e. entities).**
- **System state is decentralized among the objects:**
 - each object manages its own state information.
- Example : **Library Automation Software:**
 - each library member is a separate object with its own data and functions.
 - Functions defined for one object:
 - cannot directly refer to or change data of other objects.
- The object-oriented design paradigm makes extensive use of the principles of **abstraction** and **decomposition** .
- Objects decompose a system into functionally independent modules.
- Objects can also be considered as instances of **abstract data types (ADTs)**.

28

Object-Oriented Design

- **Objects have their own internal data:** defines their state.
- **Similar objects constitute a class.** each object is a member of some class.
- **Classes may inherit features :** from a super class.
- **Conceptually, objects communicate by message passing.**

29

Functional Programming Example

- Let's say you have this super simple form that calculates the sum of 2 numbers.

Number 1

Number 2

- With the functional programming approach.
- First function is to perform the calculations from the data supplied.

```
const x = document.querySelector('#number1').value;
const y = document.querySelector('#number2').value;
function sum(x, y) { return x + y;}
```

- Then, another function will output the result on console.

```
function displayResult(result){ console.log(result); }
```

30

Object oriented Programming Example

- Here we can create a class for our form in which we add the following properties in the constructor function:
- form: the form element
- number1 and number2: the 2 input fields
- calculateHandler event function, bind to the class to sum inputs and show on console

```
class SumForm { constructor(form, number1, number2) {  
    this.form = form;  
    this.number1 = number1;  
    this.number2 = number2;  
    this.form.addEventListener('submit',  
    this.calculateHandler.bind(this));  
}
```





31







Object oriented Programming Example







```
calculateHandler(event) {  
    event.preventDefault();  
    console.log(this.number1 + this.number2); }  
}
```

```
new SumForm(document.getElementById('sum-form'),  
document.getElementById('number1').value,  
document.getElementById('number2').value);
```

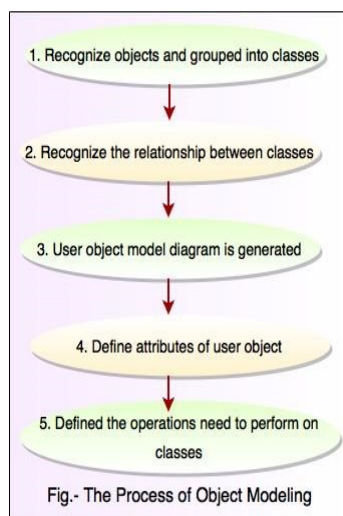
32

OOP	FP
OOP is based on the concept of objects	FP emphasizes an evaluation of functions
 <p>Basic elements are objects and methods</p>	 <p>Basic elements are variables and functions</p>
 <p>Data is mutable</p>	 <p>Data is immutable</p>

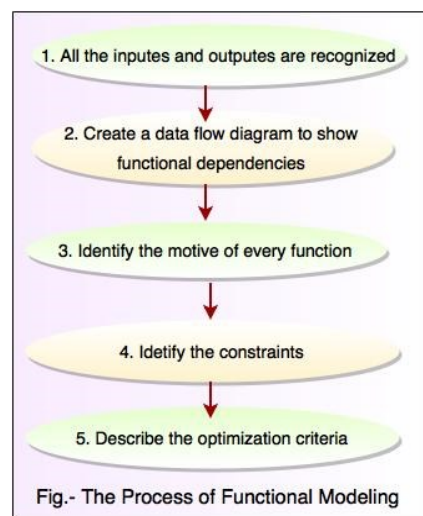
OOP	FP
 <p>Imperative programming model</p>	 <p>Declarative programming model</p>
 <p>Statements can be executed in particular order</p>	 <p>Statements can be executed in any order</p>
 <p>Uses loops for iteration</p>	 <p>Uses recursion for iteration</p>

OOP	FP
 Doesn't support parallel programming	 Supports parallel programming
 Imperative programming model	 Declarative programming model
 Is used in cases with many objects and only a few operations	 Is used incases with with few objects but a lot of operations

Following are the steps which shows the process of object modeling:



Following diagram shows the process of functional modeling



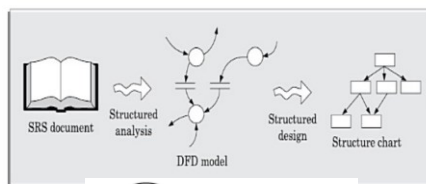
SA/SD Methodology

- **Structured analysis is a top-down decomposition technique:**
 - DFD (Data Flow Diagram) is the modelling technique used
 - Functional requirements are modelled and decomposed.
- **Why model functionalities?**
 - Functional requirements exploration and validation
 - Serves as the starting point for design.
- **Two distinct style of design:**
 - **Function-oriented or Procedural :** Top-down approach, Carried out using Structured analysis and structured design. Coded using languages such as C
 - **Object-oriented :** Bottom-up approach, Carried out using UML. Coded using languages such as Java, C++, C#

37

SA/SD Methodology

- **During Structured analysis:** High-level functions are successively decomposed: Into more detailed functions.
 - It includes DFD, Data Dictionary, ER Diagram, Process Specifications
- **During Structured design:** The detailed functions are mapped to a module structure.
 - It includes Structure charts, Pseudocode, Decision Tables and Hierarchy charts



Sample E-R Diagram

