UNIT 2

Contents:

Requirements Analysis and Specification: Requirements Gathering and Analysis, Software Requirement Specification (SRS): Characteristics of good SRS, Functional Requirements, Organization of SRS.

Software Design: Overview of the Design Process, How to Characterize a Design? Cohesion and Coupling, Approaches to Software Design.

Requirements Analysis and Specification

- All plan-driven life cycle models prescribe that before starting to develop a software, the exact requirements of the customer must be understood and documented.
- Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, and thereby alarmingly pushes up the development costs.
- A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases.

Overview of requirements analysis and specification:

- The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.
- The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed.
- The requirements specification document is usually called the software requirements specification (SRS) document.
- The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organize the requirements into a document called the Software Requirements Specification (SRS) document.

Who performs requirements analysis

- Requirements analysis and specification activity is usually carried out by a few experienced members of the development team.
- It normally requires them to spend some time at the customer site.
- The engineers who gather and analyze customer requirements and then write the requirements specification document are known as system analysts.

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis.
- Requirements specification.

Requirements gathering and analysis:

- The complete set of requirements are almost never available in the form of a single document from the customer.
- Complete requirements are rarely obtainable from any single customer representative.
- We can conceptually divide the requirements gathering and analysis activity into two separate tasks: *Requirements gathering and Requirements Analysis*

Requirements gathering.

- Requirements gathering is also popularly known as requirements elicitation.
- The primary objective of the requirements gathering task is to collect the requirements from the stakeholders.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- It is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents.
- Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.
- Important ways in which an experienced analyst gathers requirements:
  - ***Studying existing documentation:***
    - The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site.
    - Customers usually provide a statement of purpose (SoP) document to the developers.
  - ***Interview***:
    - Typically, there are many different categories of users of a software.
    - Each category of users typically requires a different set of features from the software.
    - Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.

      Refer to: Delphi method
  - ***Task analysis:***
    - The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities).
    - A service supported by software is also called a task.
    - The analyst tries to identify and understand the different tasks to be performed by the software.
    - For each identified task, the analyst tries to formulate the different steps necessary to realize the required functionality in consultation with the users.
  - ***Scenario analysis:***
    - A task can have many scenarios of operation.
    - The different scenarios of a task may take place when the task is invoked under different situations.

    - For different types of scenarios of a task, the behavior of the software can be different.
- ***Form analysis:***
    - Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.
    - In form analysis the existing forms and the formats of the notifications produced are analyzed to determine the data input to the system and the data that are output from the system.

Requirements analysis:

- After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements.
- During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:
    - **Anomaly:**
        - An anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible.
        - ***Example****: While gathering the requirements for a process control application, the following requirement was expressed by a certain stakeholder: "When the temperature becomes high, the heater should be switched off". Please note that words such as "high", "low", "good", "bad" etc. are indications of ambiguous requirements as these lack quantification and can be subjectively interpreted.*
    - **Inconsistency**:
        - Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
        - ***Example****: Consider the following two requirements that were collected from two different stakeholders in a process control application development project.*
            - *The furnace should be switched-off when the temperature of the furnace rises above 500°C.*
            - *When the temperature of the furnace rises above 500°C, the water shower should be switched-on and the furnace should remain on.*
    - **Incompleteness**:
        - An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software.
        - ***Example****: In a chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200°C then an alarm bell must be sounded. However, on an examination of all requirements, it was found that there is no provision for resetting the alarm bell after the temperature has been brought down in any of the requirements. This is clearly an incomplete requirement.*

Software Requirements Specification (SRS):

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document.
- The SRS document usually contains all the user requirements in a structured though an informal form.SRS document is probably the most important document and is the toughest to write.
- One reason for this difficulty is that the SRS document is expected to cater to the needs of a wide variety of audience.
- A well-formulated SRS document finds a variety of usage:
  - Forms an agreement between the customers and the developers.
  - Reduces future reworks.
  - Provides a basis for estimating costs and schedules
  - Provides a baseline for validation and verification
  - Facilitates future extensions

~~Users of SRS document:~~

- ***Users, customers, and marketing personnel:***
  These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs.
- ***Software developers:***
  The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- ***Test engineers:***
  The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working.
- ***User documentation writers:***
  The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- ***Project managers:***
  The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.
- ***Maintenance engineers:***
  The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system.

Characteristics of a Good SRS Document:

- IEEE Recommended Practice for Software Requirements Specifications describes the content and qualities of a good software requirements specification (SRS).
- Some of the identified desirable qualities of an SRS document are the following:
  - ***Concise***: The SRS document should be concise and at the same time unambiguous, consistent, and complete.

- ***Implementation-independent:***
  The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these.
- ***Traceable***:
  It should be possible to trace a specific requirement to the design elements that implement it and vice versa. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and vice versa.
- ***Modifiable***:
  Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured.
- ***Identification of response to undesired events:***
  The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.
- ***Verifiable***:
  All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation.

Categories of Customer requirements:

An SRS document should clearly document the following aspects of a software:

- Functional requirements
- Non-functional requirements
  - Design and implementation constraints
  - External interfaces required
  - Other non-functional requirements
- Goals of implementation.

Functional Requirements:

- The functional requirements capture the functionalities required by the users from the system.
- Consider a software as offering a set of functions $\{f_i\}$ to the user.
- These functions can be considered similar to a mathematical function $f : I \rightarrow O$, meaning that a function transforms an element (i_i) in the input domain (I) to a value (o_i) in the output (O).
- In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements.
- Each high-level function is an instance of use of the system (use case) by the user in some way.

- A high-level function is one using which the user can get some useful piece of work done.
- Each high-level requirement typically involves accepting some data from the user through a user interface, transforming it to the required response, and then displaying the system response in proper format.
- A high-level function transforms certain input data to output data.
- Except for very simple high- level functions, a function rarely reads all its required data in one go and rarely outputs all the results in one shot.
- A high-level function usually involves a series of interactions between the system and one or more users.
- Functional requirements form the basis for most design and test methodologies.
- Unless the functional requirements are properly identified and documented, the design and testing activities cannot be carried out satisfactorily.
- Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented.
- A function can be documented by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data.
- ***Refer to Withdraw cash from ATM example in text book.***

Non-functional Requirements:

- The non-functional requirements are non-negotiable obligations that must be supported by the software.
- The non-functional requirements capture those requirements of the customer that cannot be expressed as functions.
- Aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput.
- The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.
- ***Design and implementation constraints:***
    - Design and implementation constraints are an important category of non-functional requirements describing any items or issues that will limit the options available to the developers.
    - Some of the example constraints can be—corporate or regulatory policies that need to be honored; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations etc.
- ***External interfaces required:***
    - Examples of external interfaces are - hardware, software and communication interfaces, user interfaces, report formats, etc.
    - To specify the user interfaces, each interface between the software and the users must be described.
    - One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree

- ***Other non-functional requirements:***
  - This section contains a description of non- functional requirements that are neither design constraints nor are external interface requirements.
  - An important example is a performance requirement such as the number of transactions completed per unit time.

Goals of implementation:

- The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed.
- A goal, in contrast to the functional and nonfunctional requirements, is not checked by the customer for conformance at the time of acceptance testing.
- The goals of the implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc.

Organization of the SRS Document:

- The organization of an SRS document is prescribed by the IEEE 830 standard.
- IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it.
- Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged.
- The three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.
- The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics.

Various Sections of SRS:

Introduction

- ***Purpose***: This section should describe where the software would be deployed and how the software would be used.
- ***Project scope:*** This section should briefly describe the overall context within which the software is being developed.
- ***Environmental characteristics:*** This section should briefly outline the environment (hardware and other software) with which the software will interact.

Overall description of organization of SRS document

- ***Product perspective***: This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing system, or it is a new software.
- ***Product features***: This section should summarize the major ways in which the software would be used.
- ***User classes***: Various user classes that are expected to use this software are identified and described here.
- ***Operating environment***: This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.

- ***Design and implementation constraints:*** In this section, the different constraints on the design and implementation are discussed.
- ***User documentation***: This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

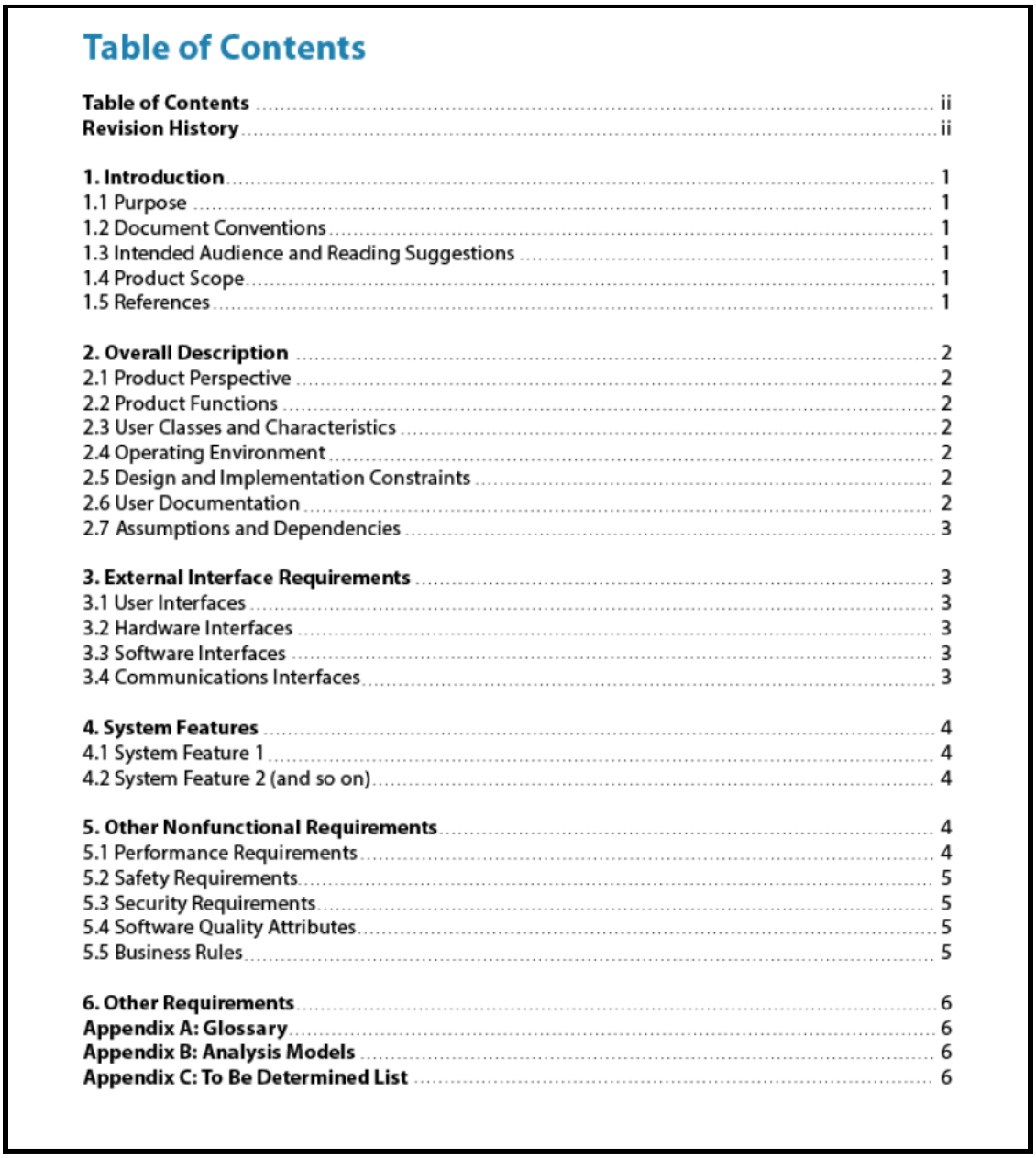


Table of Contents

IEEE format for SRS Document

External interface requirements

- ***User interfaces:*** This section should describe a high-level description of various interfaces and various principles to be followed.
- ***Hardware interfaces:*** This section should describe the interface between the software and the hardware components of the system.

- ***Software interfaces:*** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc.
- ***Communications interfaces:*** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc.

Other non-functional requirements for organization of SRS document

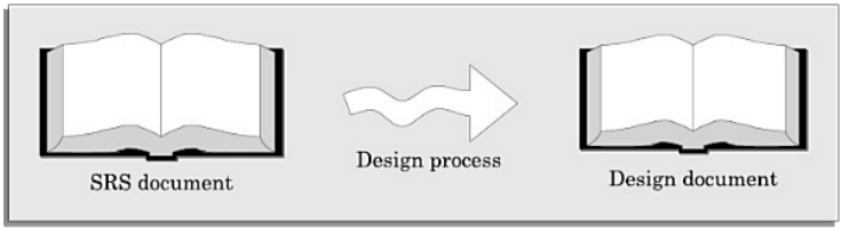
- ***Performance requirements***: Aspects such as number of transactions to be completed per second should be specified here.
- ***Safety requirements:*** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here.
- ***Security requirements:*** This section should specify any requirements regarding security or privacy requirements on data used or created by the software.

Self Study:

- ☐ ***Example Functional Requirements for Library Automation System in Text book.***

Software Design

- The activities carried out during the design phase (called as design process) transform the SRS document into the design document.
- The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.



The design process

Overview of the Design Process

Outcome of Design Process:

The following items are designed and documented during the design phase.

- ***Different modules required:*** The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs.
- ***Control relationships among modules:*** A control relationship between two modules essentially arises due to function calls across the two modules.
- ***Interfaces among different modules:*** The interfaces between two modules identifies the exact data items that are exchanged between the two modules.
- ***Data structures of the individual modules:*** Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

- ***Algorithms required to implement the individual modules:*** The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

Classification of Design Activities:

- A good software design requires iterating over a series of steps called the design activities.
- We can broadly classify them into two important stages.
  - ***Preliminary (or high-level) design:***
    - Through high-level design, a problem is decomposed into a set of modules.
    - The control relationships among the modules are identified, and also the interfaces among various modules are identified.
    - The outcome of high-level design is called the program structure or the software architecture.
    - High-level design is a crucial step in the overall design of a software.
  - ***Detailed design:***
    - Once the high-level design is complete, detailed design is undertaken.
    - During detailed design each module is examined carefully to design its data structures and the algorithms.
    - The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document.

How to characterize a good Design?

- There is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application.
- However, most researchers and software engineers agree on a few desirable characteristics.
  - ***Correctness***: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
  - ***Understandability***: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
  - ***Efficiency***: A good design solution should adequately address resource, time, and cost optimization issues.
  - ***Maintainability***: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Understandability of a Design: A Major Concern

- Understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.
- Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs.
- Therefore, a good design solution should be simple and easily understandable.

- understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.
- A design solution should have the following characteristics to be easily understandable:
    - It should assign consistent and meaningful names to various design components.
    - It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
- A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

Modularity

- A modular design is an effective decomposition of a problem.
- A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.
- Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.
- There are no quantitative metrics available yet to directly measure the modularity of a design.
- However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling.
- A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.

Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
- In a layered design solution, the modules are arranged in a hierarchy of layers.
- A module can only invoke functions of the modules in the layer immediately below it.
- A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

Cohesion and Coupling:

- Effective problem decomposition is an important characteristic of a good design.
- Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.
- Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

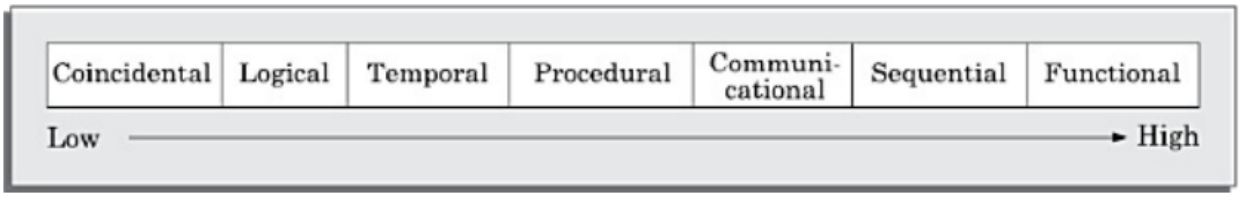
Cohesion

- When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion.
- If the functions of the module do very different things and do not cooperate with each other to perform a single piece of work, then the module has very poor cohesion.

- A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules.
- Functional independence is a key to any good design primarily due to the following advantages:
    - Error isolation
    - Scope of reuse
    - Understandability

Classifications of cohesiveness:

- Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
- The different modules of a design can possess different degrees of freedom.
- Cohesiveness increases from coincidental to functional cohesion.



- ***Coincidental cohesion:*** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.
- ***Logical cohesion:*** A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
- ***Temporal cohesion:*** When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
- ***Procedural cohesion:*** A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
- ***Communicational cohesion:*** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.
- ***Sequential cohesion:*** A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.
- ***Functional cohesion:*** A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task.

Coupling:

- Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:
    - If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
    - If the interactions occur through some shared data, then also we say that they are highly coupled.

- If two modules either do not interact or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Classification of Coupling:

- The coupling between two modules indicates the degree of interdependence between them.
- If two modules interchange large amounts of data, then they are highly interdependent or coupled.
- The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.
- The degree of coupling increases from data coupling to content coupling.

- ***Data coupling:*** Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc.
- ***Stamp coupling:*** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- ***Control coupling:*** Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another.
- ***Common coupling:*** Two modules are common coupled, if they share some global data items.
- ***Content coupling:*** Content coupling exists between two modules, if they share code.

Approaches to Software Design

- There are two fundamentally different approaches to software design that are in use today— function-oriented design, and object-oriented design.
- Though these two design approaches are radically different, they are complementary rather than competing techniques.
- The object oriented approach is a relatively newer technology and is still evolving.
- On the other hand, function-oriented designing is a mature technology and has a large following.

Function-oriented Design:

The following are the salient features of the function-oriented design approach:

- Top-down decomposition:
  - A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.

    - In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.
    - For example, consider a function create-new-library member.
    - This high-level function may be refined into the following subfunctions:
        - Assign-membership-number
        - Create-member-record
        - Print-bill
- ***Centralized system state:***
    - The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.
    - The system state is centralized and shared among different functions.
    - For example, in the library management system, several functions such as the following share data such as member-records for reference and updation
        - Create-new-member
        - Delete-member
        - update-member-record

Object-oriented Design:

- In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities).
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- 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).
- There are three important concepts associated with an ADT.
    - ***Data abstraction:*** The principle of data abstraction implies that how data is exactly stored is abstracted away.
    - ***Data structure:*** A data structure is constructed from a collection of primitive data items.
    - ***Data type:*** A type is a programming language terminology that refers to anything that can be instantiated.
- In object-orientation, classes are ADTs.
- There are three main advantages of using ADTs in programs:
    - The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding.
    - An ADT-based design displays high cohesion and low coupling.
    - Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

Self Study:

- ☐ ***Object oriented versus Function-oriented design approaches***