Unit-2
Requirement Analysis & Specification

❖ REQUIREMENT GATHERING & ANALYSIS

- Requirements of a customer play a key role in developing any software product.
- The task of gathering requirements and analyzing them is performed by a System analyst.
- Collecting all the information from the customer and then analyze the collected information to remove all ambiguities and inconsistencies from customer perception.
- Mainly two activities are concerned with this task.

Requirement gathering	Requirement analysis

Requirement gathering:

- It is usually the first part of any software product.
- This is the base for the whole development effort.
- The goal of the requirement gathering activity is → to collect all related information from the customer regarding the product to be developed.
- This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
- In this phase, meeting with customers, analyzing market demand and features of the product are mainly focused.
- So, activity of market research (for competitive analysis) is done.
- It involves interviewing the end-users and studying the existing documents to collect all possible information.

Requirement gathering activities are:

- o Studying the existing documents
- o Interview with end users or customers
- o Task analysis
- o Scenario analysis
- o Form analysis
- o Brainstorming
- o Questionnaires
- o Group discussion

Requirement analysis:

- The goal of the requirement analysis activity is → to clearly understand the exact requirements of the customers.
- **IEEE** defines requirement analysis as (1) the process of studying user needs and (2) The process of studying and refining system hardware or software requirements.
- Requirement analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.

Requirement analysis involves:

- *Eliciting requirements*: requirements are eliciting by communicating with customers and find their exact need.
- *Analyzing requirements*: requirements are then analyzed to make it complete, clear and unambiguous.
- *Requirements recording or storing*: all the requirements are recorded in form of use cases, process specifications, natural language documents etc.

System analyst should solve some of the following questions:

- What is the problem?
- What are the inputs and outputs?
- What is important to solve?
- What are the complexities?
- What are the solutions?

- Change in the environment or technical aspects may affect the requirement analysis process.
- System analyst identifies and resolves various requirements problems.
- For that, analyst has to identify and eliminate the problems of anomalies, inconsistencies and incompleteness. **Anomaly** is the ambiguity in the requirement, **Inconsistency** contradicts the requirements, and **Incompleteness** may overlook some requirements.
- Analyst detects above problems by discussing with end-users.
- Requirement analysis is necessary to develop the system that meets all the requirements of the end-user.
- Finally, make sure that requirements should be specific, measurable, timely, achievable and realistic.
- Output of this activity is $\rightarrow$ **SRS** (Software Requirements Specification).

- **Software requirements are the description of services which software will provide to end user.**
- **Requirement gathering and requirement analysis & specification collectively called 'Requirement Engineering'.**

❖ SOFTWARE REQUIREMENT SPECIFICATION (SRS)

- SRS is the output of requirement gathering and analysis activity.
- SRS is a document created by system analyst after the requirements are collected from various sources.
- SRS is a detailed description of the software that is to be developed. It describes the complete behavior the system.
- SRS describes 'what' the proposed system should do without describing 'how' the software will do (what part, not how).
- It is working as a reference document to the developer.
- It provides guideline for project development, so minimizes the time and efforts for software development.
- SRS is actually a contract between developer and end user. That helps to dissolve the disagreement.

- The SRS translates the ideas of the customers (input) into the formal documents (output).
- The SRS document is known as black-box specification, because:
  - In SRS, internal details of the system are not known (as SRS doesn't specify how the system will work).
  - Only its visible external (i.e., input/output) behavior is documented.
- SRS documents serves as contract between customer and developer, so it should be carefully written. (Sometimes SRS is also written by the customers also).
- The organization of SRS is done by the system analyst.

▪ **Benefits of SRS (Features of SRS):**

- SRS provides foundation for design work. Because it works as an input to the design phase.
- It enhances communication between customer and developer because user requirements are expressed in natural language.
- Developers can get the idea what exactly the customer wants.
- It enables project planning and helps in verification and validation process.
- Format of forms and rough screen prints can also be represented in SRS.
- High quality SRS reduces the development cost and time efforts.
- As it is working as an agreement between user and developer, we can get the partial satisfaction of the end user for the final product.
- SRS is also useful during the maintenance phase.

▪ **Contents of the SRS document:**

- An SRS should clearly document the following three things:

(i) Functional requirements of the system

- Functional requirements are those which are related to the technical functionality of the system.
- These are the services which the end users expect from the final product. And these are the services which a system provides to the end users.
- It clearly describes each of the function that the system needs to perform along with the input and output data set.

(ii) Non-functional requirements of the system

- The non-functional requirements describe the characteristics of the system that can't be expressed functionally. For example, portability, maintainability, reliability, usability, security, performance etc.
- Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system in particular conditions, rather than specific behaviors.
- Sometimes these requirements are also called quality attributes.

(iii) Constraints (restrictions) on the system

- That describes what the system should do or should not do. These are some general suggestions regarding development.
- A constraint can be classified as:
  - o Performance constraint
  - o Operating system constraint

- o Economic constraint
- o Life cycle constraint
- o Interface constraint

- **Characteristics of a good SRS: Concise**

  SRS should contain brief and concise information regarding the project; no more detailed description of the system should be there.

  Complete

  It should be complete regarding the project, so that can be completely understood by the analyst and developers as well as customers.

  Consistent

  An SRS should be consistent through the project development. Requirements may not be conflict at the later stage.

  Conceptual integrity

  SRS should clearly provide the concepts of the system, so that can be read easily.

  Structured

  SRS should be well structured to understand and to implement.

  Black box view

  SRS should have black box view means; there should not be much detailing of the project in it (only describe what part, not how).

  Verifiable

  It should be verifiable by the clients or the customers for whom the project is being made.

  Adaptable

  It should be adaptable in both sides from the clients as well as from the developers.

  Maintainable

  SRS should be maintainable so in future changes can be made easily.

  Portable

  It should be portable as if we can use the contents of it for the same types of developments.

  Unambiguous

  There should not be any alternates of SRS that creates ambiguity.

  Traceable

  Each of the requirements should be clear and refer to the future development.

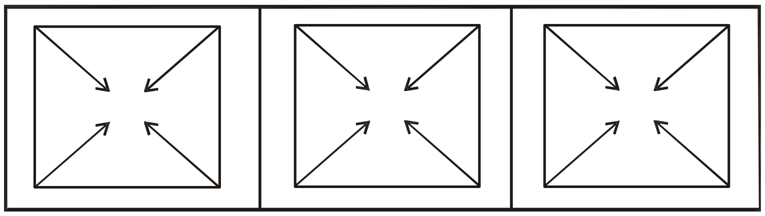
❖ COHESION AND COUPLING

- Modularity is clearly a desirable property of any software development.
- In software development, modularity is $\rightarrow$decomposition of a program into smaller programs (or modules).
- A system is considered modular if it consists of multiple modules so that each module can be implemented separately and debugged separately.
- Modular system provides advantages like:
  - Easy to understand the system.
  - System maintenance is easy.
  - Provide reusability.

- Modularity is successful because developers use prewritten code, which saves resources. Overall, modularity provides greater software development manageability.
- Cohesion and coupling are two modularization criteria that are often used together.
- Most researchers and developers are agreed that for good software design neat decomposition is highly needed, and the primary characteristic of neat decomposition is 'high cohesion and low coupling'.

▪ **Cohesion:**

- Cohesion is $\rightarrow$ a measure of functional strength of a module.
- Cohesion keeps the internal modules together, and represents the functional strength.
- Cohesion of a module represents how tightly bound the internal elements of a module are to one another.



Cohesion = strengths of relations within modules

Classification of cohesion:

Coincidental	Logical	Temporal	Procedural	Communicational	Sequential	Functional

Worst (Low) $\longrightarrow$ Best (High)

Coincidental cohesion

- It is the lowest cohesion. Coincidental cohesion occurs when there are no meaningful relationships between the elements.
- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.
- It is also called random or unplanned cohesion.

Logical cohesion

- A module is said to be logically cohesive if there is some logical relationships between the elements of module, and the elements perform functions that fall into same logical class.
- For example: the tasks of error handling, input and output of data.

Temporal cohesion

- Temporal cohesion is same as logical cohesion except that the elements are also related in time and they are executed together.
- A module is in temporal cohesion when a module contains functions that must be executed in the same time span.
- Example: modules that perform activities like initialization, cleanup, and start-up, shut down are usually having temporal cohesion.

Procedural cohesion

- A module has procedural cohesion when it contains elements that belong to common procedural unit.
- A module is said to have procedural cohesion, if the set of the modules are all part of a procedure (algorithm) in which certain sequence of steps are carried out to achieve an objective.

- Example: the algorithm for decoding a message

Communicational cohesion

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, for example the set of functions defined on an array or a stack.
- These modules may perform more than one function together.

Sequential cohesion

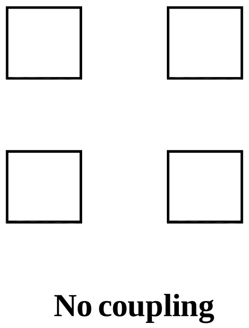
- When the output of one element in a module forms the input to another, we get sequential cohesion.
- Sequential cohesion does not provide any guideline how to combine these elements into modules.
- For example, in a TPS (transaction processing system), the get-input, validate-input, sort-input functions are grouped into one module.

Functional cohesion

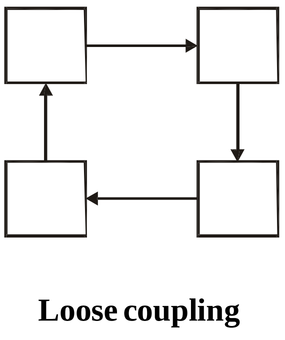
- Functional cohesion is the strongest cohesion.
- In it, all the elements of the module are related to perform a single task.
- All elements are achieving a single goal of a module.
- Functions like: compute square root and sort the array are examples of these modules.

▪ **Coupling:**

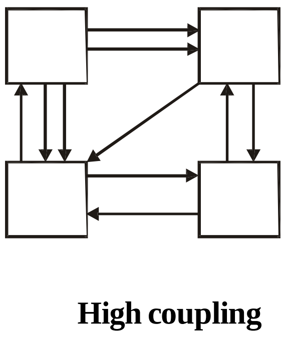
- Coupling between two modules is $\rightarrow$ a measure of the degree of interdependence or interaction between these two modules.
- Coupling refers to the number of connections between 'calling' and a 'called' module. There must be at least one connection between them.
- It refers to the strengths of relationship between modules in a system. It indicates how closely two modules interact and how they are interdependent.
- As modules become more interdependent, the coupling increases. And loose coupling minimize interdependency that is better for any system development.
- If two modules interchange large amount of data, then they are highly interdependent or we can say they are highly coupled.
- High coupling between modules makes the system difficult to understand and increase the development efforts. So low (OR loose) coupling is the best.



No coupling



Loose coupling



High coupling

Classification of coupling:

Five different types of coupling can occur between two modules.

Data	**Stamp**	**Control**	**Common**	**Content**

Best (Low) $\longrightarrow$ Worst (High)

Data coupling

- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between these two.
- For example $\rightarrow$ an int, a char, a float etc.
- It is lowest coupling and best for the software development.

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 instructions execution in another module.
- An example of control coupling is a flag set in one module and tested in another module.

Common coupling

- Two modules are common coupled, if they share data through some global data items. It means two or more modules are communicating using common data.

Content coupling

- It is the highest coupling and creates more problems in software development.
- Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.
- It is also known as 'pathological coupling'.

▪ **Functional independence:**

- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- So, that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- For good software design neat decomposition is highly needed, and the primary characteristic of neat decomposition is **'high cohesion and low coupling'.**

Intra dependency (Cohesion) between modules should be high and inter dependency (Coupling) should be low.

Need of functional independence:

- Functional independence is a good key to any software design process due to following reasons :

1. Error isolation:

- It reduces error propagation.

- The reason behind this is $\rightarrow$ if a module is functionally independent, its degree of interaction with the other modules is less.
- So, the error of one module can't affect another module.

2. **Scope of reuse:**

- Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal.
- Therefore, a cohesive module can be easily taken out and reused in a different program.

3. **Understandability:**

- Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Difference between functional and non-functional requirements:

Functional requirements	Non-functional requirements
These describe what the system should do.	These describe how the system should behave.
These describe features, functionality and usage of the system.	They describe various quality factors, attributes which affect the system's functionality.
Describe the actions with which the work is concerned.	Describe the experience of the user while doing the work.
Characterized by verbs.	Characterized by adjectives.
Ex: business requirements, SRS etc.	Ex: portability, quality, reliability, robustness, efficiency etc.

Difference between Cohesion & Coupling:

Cohesion	Coupling
Cohesion is the indication of the relationship within module.	Coupling is the indication of the relationships between modules.
Cohesion shows the module's relative functional strength.	Coupling shows the relative interdependence among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the single thing.	Coupling is a degree to which a component / module is connected to the other modules.
While designing you should go for high cohesion. i.e. a cohesive component/ module focus on a single task with little interaction with other modules of the system.	While designing you should go for low coupling i.e., dependency between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, class having all members visible with a package having default visibility.	Making private fields, private methods and nonpublic classes provides loose coupling.
Cohesion is Intra - Module Concept.	Coupling is Inter -Module Concept.