UNIT-II
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 as 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.
- The SRS document is the final outcome of the requirements analysis and specification phase.
- 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 requirements gathering and analysis activity is divided into two separate tasks:

- Requirements gathering
- 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.
- System analysts collect data pertaining to the product to be developed and analyses the collected data to conceptualize what exactly needs to be done.

The following are the important ways in which an experienced analyst gathers requirements:

1. 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 statement of purpose (Sop) document to the developers.

2. Interview: In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This procedure is repeated till the different users agree on the set of requirements.

3. Task analysis: In this context, 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.
Task Analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.

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. For example, the possible scenarios for the book issue task of a library automation software may be:

- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.
- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

For various identified tasks, the possible scenarios of execution are identified and the details of each scenario is identified in consultation with the users. For each of the identified scenarios, details regarding system response, the exact conditions under which the scenario occurs, etc. are determined in consultation with the user.

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. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis the exiting 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. For the different sets of data input to the system, how these input data would be used by the system to produce the corresponding output data is determined from the users.

Requirement 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.

The main purpose of the requirements analysis activity is to analyses the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly: It is an anomaly is an ambiguity i n a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

Egg: 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. If the threshold above which the temperature can be considered to be high is not specified, then it can be interpreted differently by different developers.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
Egg: 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.

The requirements expressed by the two stakeholders are clearly inconsistent.

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. An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.
Egg: In chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200C 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.

Users of SRS Document

The important categories of users of the SRS document and their needs for use are as follows:
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 it's 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. A proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.

Need and Uses of SRS Document

Forms an agreement between the customers and the developers: A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting.

Provides a basis for estimating costs and schedules: Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required, total cost required and work schedule to develop the software. The SRS document also serves as a basis for price negotiations with the customer.

Provides a baseline for validation and verification: The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.

Facilitates future extensions: The SRS document usually serves as a basis for planning future enhancements.

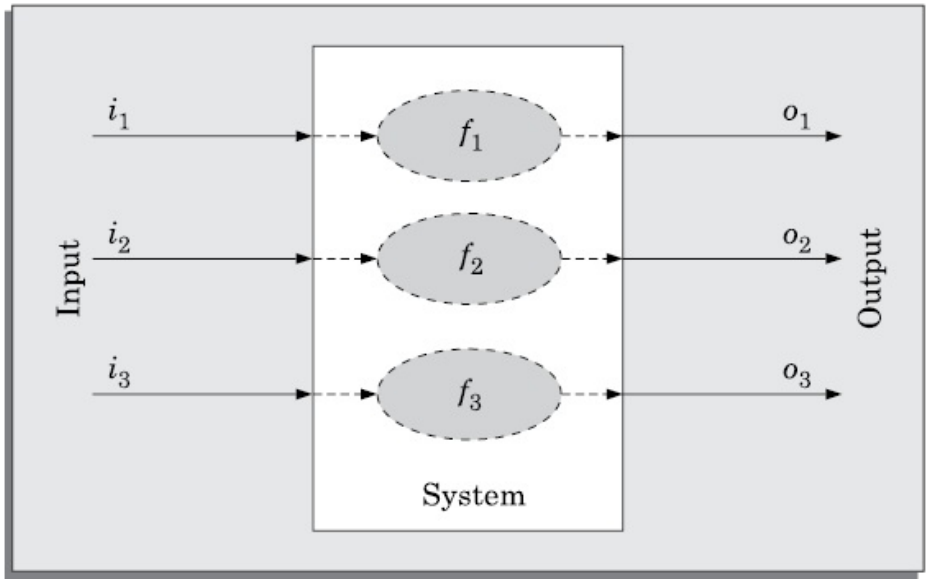
Characteristics of a Good SRS Document

IEEE Recommended Practice for Software Requirements Specifications[IEEE830] describes the content and qualities of a good software requirements specification (SRS).

The analyst should be aware of the following identified desirable qualities of an SRS document:

- **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. It should only specify what the system should do and refrain from stating how to do these.

The SRS document should describe the system to be developed as a black box, and should specify only the externally visible behavior of the system. For this reason, the SRS document is also called the *black-box* specification of the software being documented.



The black-box view of a system as performing a set of functions.

Traceable: It should be possible to trace a specific requirement to the design elements and code segments that implement it and *vice versa*.

Modifiable: An SRS document is often modified after the project completes to accommodate future enhancements and evolution. To cope up with the requirements changes, the SRS document should be easily modifiable. For this, an SRS document should be well-structured. A well-structured document is easy to understand and modify.

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.

Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. They are:

Over-specification: It occurs when the analyst tries to address the "how to" aspects in the SRS document. For example, in the library automation problem, one should not specify whether the library membership records need to be stored indexed on the member's first name or on the library member's identification (ID) number. Over-specification restricts the freedom of the designers in arriving at a good design solution.

Forward references: One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.

Wishful thinking: This type of problems concern description of aspects which would be difficult to implement.

Noise: The term noise refers to presence of material not directly relevant to the software development process. For example, in the register customer function, suppose the analyst writes that customer registration department is manned by clerks who report for work between 8am and 5pm, 7 days a week. This information can be called *noise* as it would hardly be of any use to the software developers and would unnecessarily clutter the SRS document, diverting the attention from the crucial points.

Important Categories of Customer/User Requirements

As per the IEEE 830 guidelines, the important categories of user requirements are the following.

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

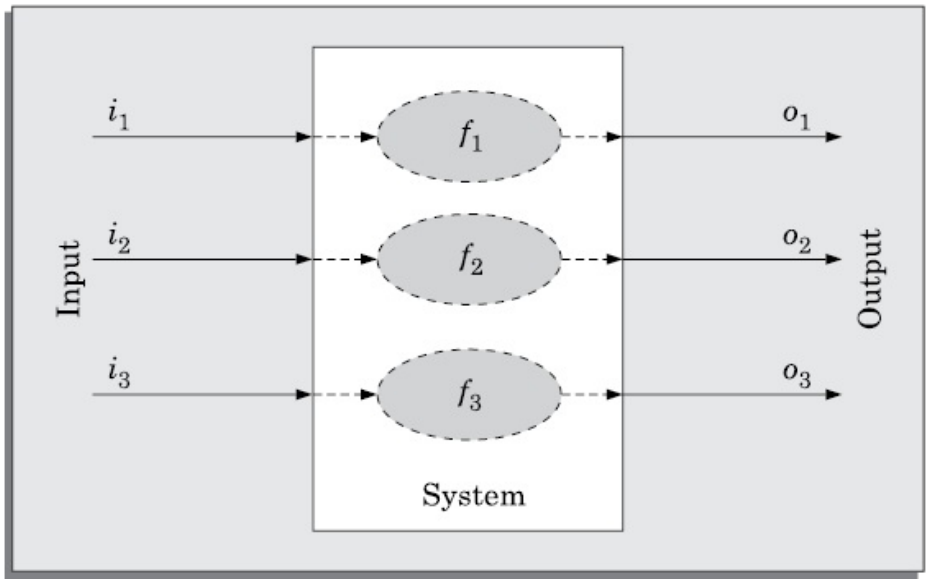
- Functional requirements
- Non-functional requirements
  - — Design and implementation constraints
  - — External interfaces required
  - — Other non-functional requirements
- Goals of implementation.

Functional Requirements

The functional requirements capture the functionalities required by the users from the system. The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set. Functions can be considered similar to a mathematical function f: I → *O*, meaning that a function transforms an

element (i*i*) in the input domain (I) to a value (o*i*) in the output (O). This functional view of a system is shown schematically in the following figure.

Non-



Functional Requirements

The non-functional requirements are non-negotiable obligations that must be supported by the software. Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, pportability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.).

The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections. The remaining non-functional requirements should be documented later in a section and these should include the performance and security requirements.

In the following subsections, we discuss the different categories of non- functional requirements that are described under three different sections:

Design and implementation constraints:

Design and implementation constraints are an important category of non-functional requirements describe 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 needs 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; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organization.

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. The description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on.

Other non-functional requirements:

This section contains a description of non- functional requirements that are neither design constraints and nor are external interface requirements. An important example is a performance requirement such as the number of transactions completed per unit time. Besides performance requirements, the other non-functional requirements to be described in this section may include reliability issues, accuracy of results, and security issues.

Goals of Implementation:

The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed. These are not binding on the developers, and they may take these suggestions into account if possible. For example, the developers may use these suggestions while choosing among different design solutions.

goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.

The goals of 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. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately. It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

Functional Requirements

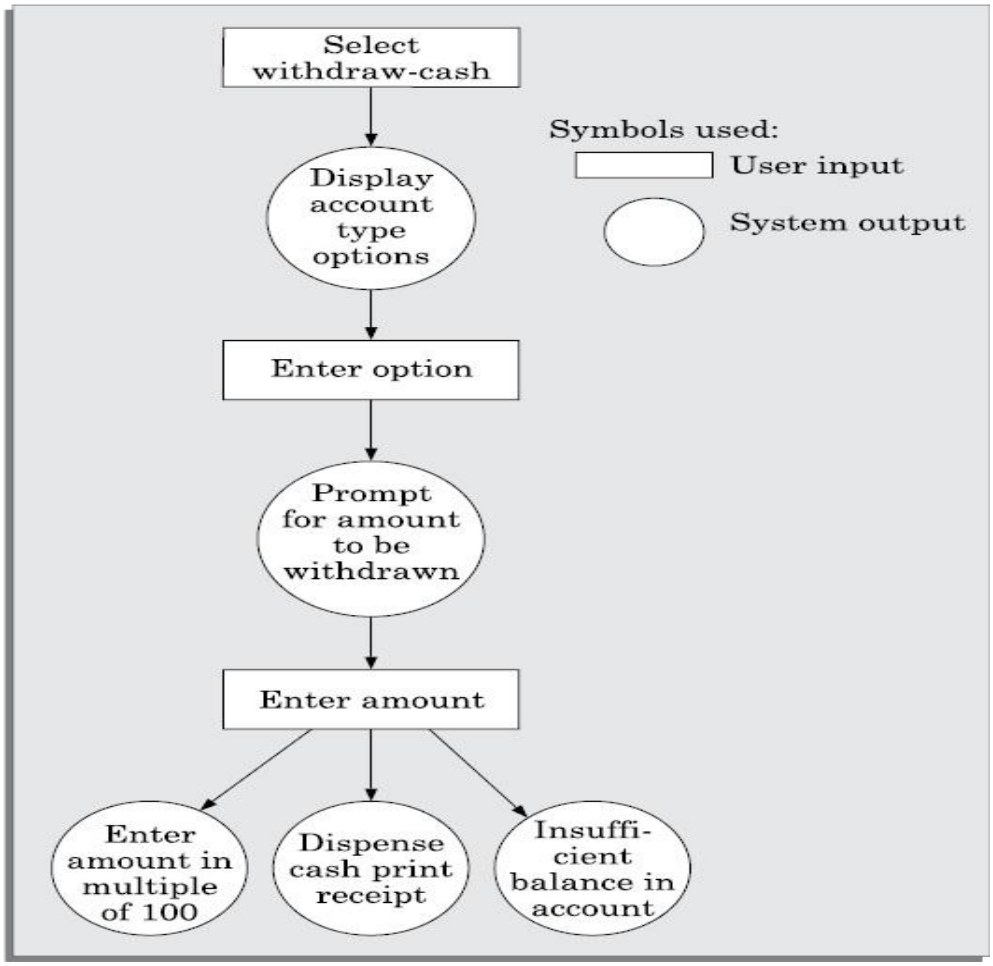
In order to document the functional requirements of a system, it is necessary to first learn to identify the high-level functions of the systems by reading the informal documentation of the gathered requirements.

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.For any given high-level function, there can be different interaction sequences or scenarios due to users selecting different options or entering different data items.

An example of the interactions that may occur in a single high-level requirement has been shown in the following Figure:



User and system interactions in high-level functional requirement.

In the above Figure, the user inputs have been represented by rectangles and the response produced by the system by circles. Observe that the rectangles and circles alternate in the execution of a single high-level function of the system, indicating a series of requests from the user and the corresponding responses from the system. In the same Figure, the different scenarios occur depending on the amount entered for withdrawal.

identifying Functional Requirements

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem.

Each high-level requirement characterizes a way of system usage (service invocation) by some user to perform some meaningful piece of work.

It is often useful to

- first identify the different types of users who might use the system and
- then try to identify the different services expected from the software by different types of users.

Documenting Functional Requirements

- 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.

Example 4.7 (Withdraw cash from ATM): An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (Sop). A Sop serves as a starting point for the analyst and he proceeds with the requirements gathering activity after a basic understanding of the Sop.

R.1: Withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R.1.1: Select withdraw amount option

Input: "Withdraw amount" option selected *Output:* User prompted to enter the account type

R.1.2: Select account type

Input: User selects option from any one of the followings—savings/checking/deposit.

Output: Prompt to enter amount

R.1.3: Get required amount

Input: Amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: The amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

Specification of large software: If there are large number of functional requirements we can organize the functional requirements by splitting the requirements into sections of related requirements.

- For example, the functional requirements of an academic institute automation software can be split into sections such as accounts, academics, inventory, publications, etc.
- When there are too many functional requirements, these should be properly arranged into sections.

- For example, the following can be sections in the trade house automation software:
  - Customer management
  - Account management
  - Purchase management
  - Vendor management
  - Inventory management

Level of details in specification:

- we would have to specify only the important input/output interactions in a functionality along with the processing required to generate the output from the input. However, if the interaction sequence is specified in too much detail, then it becomes an unnecessary constraint on the developers and restricts their choice in solution. On the other hand, if the interaction sequence is not sufficiently detailed, it may lead to ambiguities and result in improper implementation.