SOFTWARE ENGINEERING

Unit - 2

Requirement Analysis and Specification

Nature of Software:

Seven broad categories of computer software present continuing challenges for software engineers .which is given below:

1.System software: Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.

2. Real time software: These software are used to monitor, control and analyze real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forcast the weather.

3. Embedded software: This type of software is placed in "Read-Only- Memory (ROM)"of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software

4. Business software : This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software

5. Personal computer software : The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

6. Artificial intelligence software: Artificial Intelligence software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Examples are expert systems, artificial neural network,signal processing software etc.

7. Web applications: These Applications called "WebApps," this network-centric software category spans a wide area of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

UNIQUE NATURE OF WEB APPS

In the early days of the World Wide Web (1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications due to the development of HTML, JAVA, xml etc.

Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology." The following attributes are encountered in the vast majority of WebApps

Network intensiveness:- A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

Concurrency(Operation at the same time):- A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load:- The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.

Performance:- If a WebApp user must wait too long, he or she may decide to go elsewhere

Availability:- Although expectation of 100 percent availability is un reasonable, users of popular WebApps often demand access on a 24/7/365 basis

Data driven:- The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

Content sensitive:- The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

Continuous evolution:-Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

Immediacy:- Although immediacy—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks

Security:- Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes

Aesthetics:- An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design

SOFTWARE MYTHS

The development of software requires dedication and understanding on the developers' part. Many software problems arise due to myths that are formed during the initial stages of software development. **Software myths propagate false beliefs and confusion in the minds of management, users and developers.**

Managers, who own software development responsibility, are often under strain and pressure to maintain a software budget, time constraints, improved quality, and many other considerations. Common management myths are listed in Table

Management Myths

S.No	Myths	Reality
1.	The members of an organization can acquire all-the information, they require from a manual, which contains standards, procedures, and principles;	• Standards are often incomplete, inadaptable, and outdated. • Developers are often unaware of all the established standards. • Developers rarely follow all the known standards because not all the standards tend to decrease the delivery time of software while maintaining its quality
2.	If the project is behind schedule, increasing the number of programmers can reduce the time	• Adding more manpower to the project, which is already behind schedule, further

	gap.	delays the project. • New workers take longer to learn about the project as compared to those already working on the project.
3.	If the project is outsourced to a third party, the management can relax and let the other firm develop software for them.	• Outsourcing software to a third party does not help the organization, which is incompetent in managing and controlling the software project internally. The organization invariably suffers when it out sources the software project.

Users Myths

Users end to believe myths about the software because software managers and developers do not try to correct the false beliefs. These myths lead to false expectations and ultimately develop dissatisfaction among the users. Common user myths are listed in Table.

S.No	Myths	Reality
1.	Brief requirement stated in the initial process is enough to start development; detailed requirements can be added at the later stages.	• Starting development with incomplete and ambiguous requirements often lead to software failure. Instead, a complete and formal description of requirements is essential before starting development. • Adding requirements at a later stage often requires repeating the entire development process.
2.	Software is flexible; hence software requirement changes can be added during any phase of the development process.	• Incorporating change requests earlier in the development process costs lesser than those that occurs at later stages. This is because incorporating changes later may require redesigning and extra resources.

Developer Myths

In the early days of software development, programming was viewed as an art, but now software development has gradually become an engineering discipline. However, **developers** still believe in some myths-. Some of the common developer myths are listed in Table.

S.No	Myths	Reality
1.	Software development is considered complete when the code is delivered.	50% to 70% of all the efforts are expended after the software is delivered to the user.
2.	The success of a software project depends on the quality of the product produced.	The quality of programs is not the only factor that makes the project successful instead the documentation and software configuration also playa crucial role.
3.	Software engineering requires unnecessary documentation, which slows down the project.	Software engineering is about creating quality at every level of the software project. Proper documentation enhances quality which results in reducing the amount of rework.
5.	The only product that is delivered after the completion of a project is the working program(s).	The deliverables of a successful project includes not only the working program but also the documentation to guide the users for using the software.
5.	Software quality can be assessed only after the program is executed	The quality of software can be measured during any phase of development process by applying some quality assurance mechanism. One such mechanism is formal technical review that can be effectively used during each phase of development to uncover certain errors

Requirement gathering and Analysis

The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

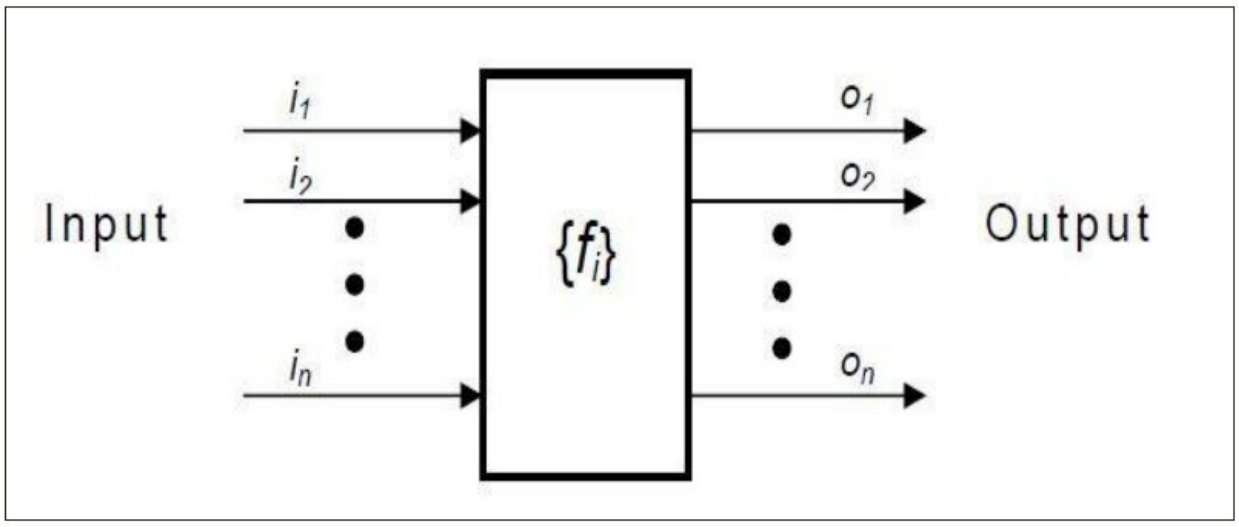
Requirements may be functional or non-functional

Functional requirements describes system services or functions

Non-functional requirements is a constraints on the system or on the development process

Functional

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in fig. Each function f of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.



Nonfunctional

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc

Identifying functional requirements from a problem description

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 by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Example: - Consider the case of the library system, where –

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details. Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions

Types of requirements

- User Requirements

Statements in natural language (NL) plus diagrams of the services the system provides and its operational . Written for customers

- System requirements

A structured document setting out detailed description of the system services. Written as contract between client and contractor

- Software specification

A detailed software description which can serve as a basis for a design or implementation . Written for developers.

Software Requirement Specification - [SRS]

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements(**A functional requirement defines a system or its component. A non-functional requirement defines the quality attribute of a software system)**. The SRS is developed based the agreement between customer

and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development. To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

Characteristics of an SRS

1. Correctness

- Each requirement accurately represents some desired feature in the final system Completeness
- All desired features/characteristics specified
- Hardest to satisfy
- Completeness and correctness strongly related

2. Completeness: The SRS is complete if, and only if, it includes the following elements:

a). All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

b). Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

c). Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

3. Consistency: The SRS is consistent if, and only if, no subset of individual requirements described in its conflict. There are three types of possible conflict in the SRS:

a). The specified characteristics of real-world objects may conflicts. For example,

i) The format of an output report may be described in one requirement as tabular but in another as textual.

ii) One condition may state that all lights shall be green while another states that all lights shall be blue.

b). There may be a reasonable or temporal conflict between the two specified actions. For example,

i) One requirement may determine that the program will add two inputs, and another may determine that the program will multiply them.

ii) One condition may state that "A" must always follow "B," while other requires that "A and B" co-occurs.

c) Two or more requirements may define the same real-world object but use different terms for that object. For example, a program's request for user input may be called a "prompt" in one requirement's and a "cue" in another. The use of standard terminology and descriptions promotes consistency

4. Unambiguousness: SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

5. Ranking for importance and stability: The SRS is ranked for importance and stability if each requirement in it has an identifier to indicate either the significance or stability of that particular requirement.

Typically, all requirements are not equally important. Some prerequisites may be essential, especially for life-critical applications, while others may be desirable

6. Modifiability: SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced

7. Verifiability: SRS is correct when the specified requirements can be verified with a cost effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

8. Traceability: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation

There are two types of Traceability:

a). Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

b). Forward Traceability: This depends upon each element in the SRS having a unique name or reference number.

9. Design Independence: There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

10. Testability: An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

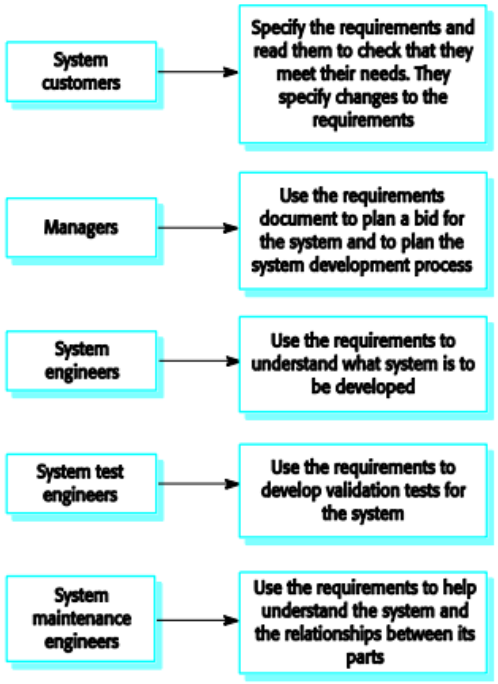
11. Understandable by the customer: An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

12. The right level of abstraction: If the SRS is written for the requirements stage, the details should be explained explicitly. Where as,for a feasibility study, fewer analysis can be used. Hence, the level of abstraction modifies according to the objective of the SRS.

The Software Requirements Specifications (SRS) Document

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of a requirements document



Purpose of SRS

• Communication between the Customer, Analyst, System Developers, Maintainers

• Firm foundation for the design phase

• Support system testing activities • Support project management and control

• Controlling the evolution of the system

IEEE Requirements Standard

• Defines a generic structure for a requirements document that must be instantiated for each specific system.

--Introduction.

– General description.

– Specific requirements.

– Appendices.

– Index.

1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms and Abbreviations

1.4 References 1.5 Overview

2. General description

2.1 Product perspective

2.2 Product function summary

2.3 User characteristics

2.4 General constraints

2.5 Assumptions and dependencies

3. Specific Requirements

- Functional requirements
- External interface requirements
- Performance requirements
- Design constraints
- Attributes eg. Security, availability, maintainability, transferability/conversion
- Other requirements
    - Appendices
    - Index

Suggested SRS Document Structure

- Preface – Should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version
- Introduction – This should describe the for the system. It should briefly describe its functions and explain how it will work with other. It should describe how it will with other systems. It should describe how the system fits into the overall business or strategic objectives of the organization commissioning the software
- **Glossary** – This should define the technical terms used in the document. Should not make assumptions about the experience or expertise of the reader • **User Requirements Definition** – The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified
- **System Architecture** – This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across modules. Architectural components that re reused should be highlighted
- **System Requirements Specification** – This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements e.g. interfaces to other systems may be defined

- **System Models** – This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models and data-flow models
- **System Evolution** – This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs etc
- **Appendices –** These should provide detailed, specific information which is related to the application which is being developed. E.g. Appendices that may include hardware and database descriptions.
- **Index –** Several indexes to the document may be included

DECISION TREE

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -

Consider Library Membership Automation Software (LMS) where it should support the following three options:

- New member
- Renewal
- Cancel membership

New member option-

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

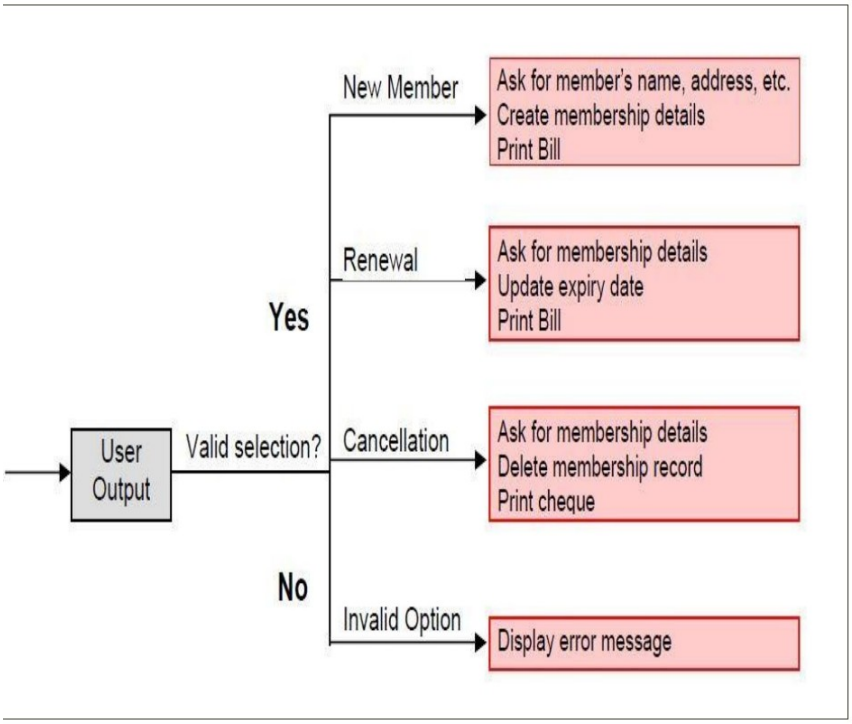
Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database

The graphical representation of the above example



DECISION TABLE

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied. A column in a table is called a rule. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -

Consider the previously discussed LMS example. The following decision table shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table

FORMAL SYSTEM SPECIFICATION

Formal Technique

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by the specification language

Formal Specification Language

A formal specification language consists of two sets syn and sem, and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if sat (syn, sem), then syn is said to be the specification of sem, and sem is said to be the specificand of syn

Syntactic Domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system

Semantic Domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc

Axiomatic Specification

In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post- conditions are essentially constraints on the results produced for the function execution to be considered successful

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Also find out other constraints on the input parameters and write it in the form of a predicate.
- Specify a predicate defining the conditions which must hold on the output of the function if it behaved properly.

- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Combine all of the above into pre and post conditions of the function

Example1: -

Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

f (x : real) : real

pre : $x \in R$

post : $\{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$