3.1 INTRODUCTION:

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design.

Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet.

Requirement analysis allows the software engineer (sometimes called analyst in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.

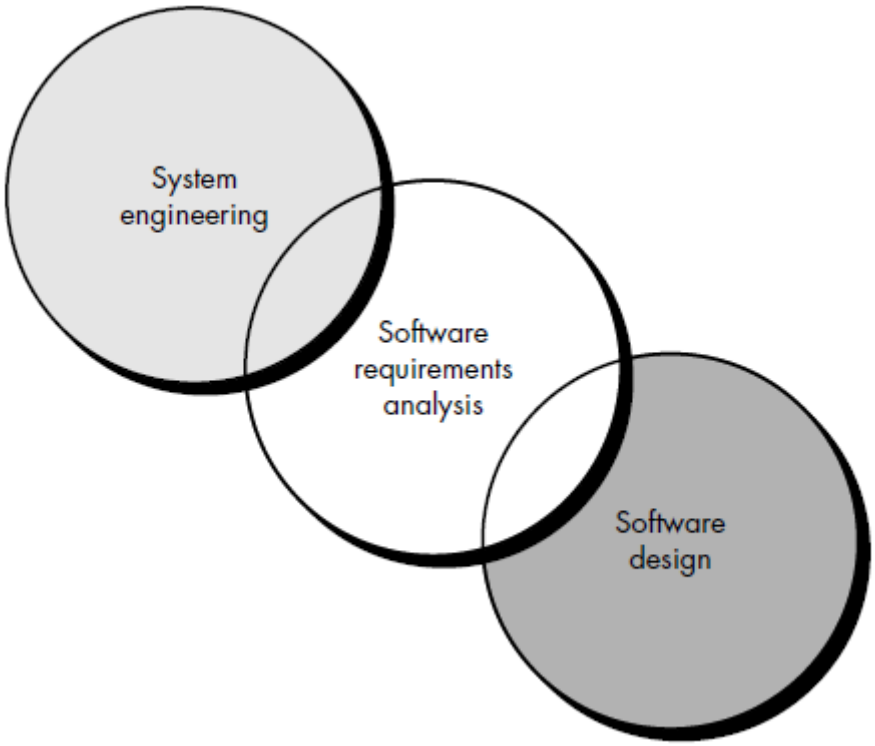
Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs.

Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Software requirements analysis may be divided into five areas of effort:
(1) problem recognition,
(2) evaluation and synthesis,
(3) modeling,
(4) specification, and
(5) review.

Initially, the analyst studies the System Specification (if one exists) and the Software Project Plan. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates.

Next, communication for analysis must be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users.



Analysis as a bridge between system engineering and software design

3.2 ***REQUIREMENT GATHERING TECHNIQUES*:**

1 Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview.

The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents.

Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with, but at the same time, both want it to be a success.

Yet, communication must be initiated. Gause and Weinberg suggest that the analyst start by asking context-free questions. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself.

The first set of context-free questions focuses on the customer, the overall goals, and the benefits. For example, the analyst might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

2 Facilitated Application Specification Techniques

Too often, customers and software engineers have an unconscious "us and them" mind-set. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach doesn't work very well. Misunderstandings abound, important information is omitted, and a successful working relationship is never established. It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called facilitated application specification techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.

Many different approaches to FAST have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

To better understand the flow of events as they occur in a typical FAST meeting, we present a brief scenario that outlines the sequence of events that lead up to the meeting, occur during the meeting, and follow the meeting.

3 Quality Function Deployments

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. Originally developed

in Japan and first used at the Kobe Shipyard of Mitsubishi Heavy Industries, Ltd., in the early 1970s, QFD "concentrates on maximizing customer satisfaction from the software engineering process." To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements:

Normal requirements: The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements: These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.
Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements: These features go beyond the customer's expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.
In actuality, QFD spans the entire engineering process However; many QFD concepts are applicable to the requirements elicitation activity.

FACT FINDING:

- Identification of what new system should be able to do.
- Specification of what the system should do as per user's requirements.
- Includes what the existing system does and what is the new one expected to do.
- Done by system or business analyst.
- Rapidly changing environment of organizations.
- Classifies data in 3 categories:
    - Functional Requirements
    - Non-Functional Requirements
    - Usability Requirements

Functional Requirements:

- Describes what a system is expected to do (Functionality).
- Describes the processes that system will carry out
- Details of the inputs into the system from paper forms and documents and other systems
- Details of the output expected from the system on screen display and as printouts on the paper

Non-Functional Requirements:

- Describes the quality parameters of the processing of functional requirements
- Performance criteria: Desired Response time for updating or retrieving data in/from the system.
- Ability of the system to handle with multi using at multi levels
- Security parameters: resistance and detection of attacks.

Usability Requirements:

- Describe the usability factors and facts between the system and users
- Ensures good match between the system and users performing tasks on the system
- Efficient Human-Computer interactions

Fact Finding Techniques:

1. Background Reading
2. Interviewing
3. Observation
4. Document Sampling
5. Questionnaires

1. Background Reading

- To have good understanding of the organization's business objectives
- Kind of documents to be looked for:
  - ❖ Company Reports
  - ❖ Organization Charts
  - ❖ Policy Manuals
  - ❖ Job Descriptions
  - ❖ Reports
  - ❖ Documentation of existing system

Advantages:

- Helps understanding the organization before meeting its work force
- Helps understanding the requirements of the system in the light of business objectives
- Documentation can provide information requirements of the current system.

Disadvantages:

- Difference between written policy and its application

2. Interviewing

- Most widely used technique
- Requires the most skills and sensitivity
- Structured meeting between analyst and staff
- Discussion of one or more areas of work of the staff
- Can be using fixed set of questions or informal questions
- Close and Open probes (Survey).

Advantages:
- Produces high Quality information.
- Provides greater depth of understanding of a person's work and **expectation.**

Disadvantages:
- Time consuming process
- Interviewee can provide conflicting information which becomes difficult to resolve later

3. Observation
- Watching people in their normal work flow carrying out their operations
- Analysts watch and note the type of the information the work is using and processing in the existing system
- Can be open ended or close ended

Advantages:
- Provides firsthand experience
- Real time data collection

Disadvantages:
- Most people don't like being observed and may behave differently
- Requires recursive training to have an analytical observation
- Logistics

4. Document Sampling
- Done in two (2) ways
- First, Collect copies of completed documents of the interviews and observations and carefully articulate.
- Second, Statistical analysis of the documents to find out patterns of data.

Advantages:
- Used for quantitative data.
- Used to find error rates in paper documents.

Disadvantages:
- Existing documents don't show what changes will be in future.

5. Questionnaires
- Effective fact finding instrument.
- Has series of questions to be answered.
- Multiple choices or Yes/No questions.
- Covers question ranging from Coding to Feedback.

Advantages:
- Economical way of gathering data.
- If well defined, results are effectively analyzed.

Disadvantages:
- Creating a good questionnaire is difficult.
- No follow-up or probing can be done with answers.

Analysis Principles:

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behavior of the software (as a consequence of external events) must be represented.
4. The models that depict information, function and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis process should move from essential information toward implementation detail.

In addition to these operational analysis principles, Davis suggests a set of guiding principles for requirements engineering:

• Understand the problem before you begin to create the analysis model. There is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!

• Develop prototypes that enable a user to understand how human/machine interaction will occur. Since the perception of the quality of software is often based on the perception of the "friendliness" of the interface, prototyping (and the iteration that results) are highly recommended.

• Record the origin of and the reason for every requirement. This is the first step in establishing traceability back to the customer.

• Use multiple views of requirements. Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.

• Rank requirements. Tight deadlines may preclude the implementation of every software requirement. If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.

• Work to eliminate ambiguity. Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity.

4GL Techniques:

The term fourth generation techniques (4GT) encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level. The tool then automatically generates source code based on the developer's specification. There is little debate that the higher the level at which software can be specified to a machine, the faster a program can be built. The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand.

4GT begins with a requirements gathering step. Ideally, the customer would describe requirements and these would be directly translated into an operational prototype. But this is unworkable. The customer may be unsure of what is required, may be ambiguous in specifying facts that are known, and may be unable or unwilling to specify information in a manner that a 4GT tool can consume. For this reason, the

customer/developer dialog described for other process models remains an essential part of the 4GT approach.

For small applications, it may be possible to move directly from the requirements gathering step to implementation using a nonprocedural fourth generation language (4GL) or a model composed of a network of graphical icons. However, for larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used. The use of 4GT without design (for large projects) will cause the same difficulties (poor quality, poor maintainability, poor customer acceptance) that have been encountered when developing software using conventional approaches.

There is some merit in the claims of both sides and it is possible to summarize the current state of 4GT approaches:

1. The use of 4GT is a viable approach for many different application areas. Coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problems.
2. Data collected from companies that use 4GT indicates that the time required producing software is greatly reduced for small and intermediate applications and that the amount of design and analysis for small applications is also reduced.
3. However, the use of 4GT for large software development efforts demands as much or more analysis, design, and testing (software engineering activities) to achieve substantial time savings that result from the elimination of coding.

3.3 EFFORT DISTRIBUTION:

A recommended distribution of effort across the definition and development phases is often referred to as the 40–20–40 rule. Forty percent of all effort is allocated to front-end analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is de-emphasized.

This effort distribution should be used as a guideline only. The characteristics of each project must dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2–3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Requirements analysis may comprise 10–25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15–20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30–40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

Software Requirement Specifications:

Software Requirements Specification (SRS) is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies at a particular point of time (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand each other's requirements from every perspective at a given point of time.

The Software Requirements Specification is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications (as well as other software engineering documentation).

The Introduction of the software requirements specification states the goals and objectives of the software, describing it in the context of the computer-based system. Actually, the Introduction may be nothing more than the software scope of the planning document.

The Information Description provides a detailed description of the problem that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A description of each function required to solve the problem is presented in the Functional Description. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements. The Behavioral Description section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

Validation Criteria is probably the most important and, ironically, the most often neglected section of the Software Requirements Specification. How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an implicit review of all other requirements. It is essential that time and attention be given to this section.

Finally, the specification includes a Bibliography and Appendix. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications.

Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the Software Requirements Specification may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a Preliminary User's Manual. The Preliminary User's Manual presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

3.4 WHAT ARE THE BENEFITS OR IMPORTANCE OF A GREAT SRS?

Establish the basis for agreement between the customers and the suppliers on what the software product is to do. The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.

Reduce the development effort. The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

Provide a basis for estimating costs and schedules. The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. [NOTE: Again, we use the SRS as the basis for our fixed price estimates]

Provide a baseline for validation and verification. Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured. [NOTE: We use the SRS to create the Test Plan].

Facilitate transfer. The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

Serve as a basis for enhancement. Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

3.6 WHAT ARE THE CHARACTERISTICS OF A GREAT SRS?

An SRS should be

a) Correct
b) Unambiguous
c) Complete
d) Consistent

e) Ranked for importance and/or stability
f) Verifiable
g) Modifiable
h) Traceable

Correct - This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "**Correct and Ever Correcting**." The discipline is keeping the specification up to date when you find things that are not correct.

Unambiguous - An SRS is unambiguous if, and only if, every requirement stated therein has only **one interpretation**. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

Complete - A simple judge of this is that is should be all that is needed by the software designers to create the software.

Consistent - The SRS should be consistent within itself and consistent to its reference documents. If you call an input "**Start** and **Stop**" in one place, don't call it "**Start/Stop**" in another.

Ranked for Importance - Very often a new system has requirements that are really marketing **wish lists**. Some may not be achievable. It is useful provide this information in the SRS.

Verifiable - Don't put in requirements like - "**It should provide the user a fast response**." Another of my favorites is - "**The system should never crash**." Instead, provide a quantitative requirement like: "**Every key stroke should provide a user response within 100 milliseconds**."

Modifiable - Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.

Traceable - Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to **connect** the requirements in the SRS to a higher level document. **Why do we need this requirement**?

Components of SRS

An SRS must address are:

1. Functional requirements
2. Performance requirements
3. Design constraints
4. External interface requirements

Conceptually, any SRS should have these components. Now we will discuss them one by one.

1. Functional Requirements

Functional requirements specify what output should be produced from the given inputs. So they basically describe the connectivity between the input and output of the system. For each functional requirement:

1. A detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs be specified:
2. All the operations to be performed on the input data obtain the output should be specified, and
3. Care must be taken not to specify any algorithms that are not parts of the system but that may be needed to implement the system.
4. It must clearly state what the system should do if system behaves abnormally when any invalid input is given or due to some error during computation. Specifically, it should specify the behavior of the system for invalid inputs and invalid outputs.

2. Performance Requirements (Speed Requirements)

This part of an SRS specifies the performance constraints on the software system. All the requirements related to the performance characteristics of the system must be clearly specified. Performance requirements are typically expressed as processed transaction s per second or response time from the system for a user event or screen refresh time or a combination of these. It is a good idea to pin down performance requirements for the most used or critical transactions, user events and screens.

3. Design Constraints

The client environment may restrict the designer to include some design constraints that must be followed. The various design constraints are standard compliance, resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Standard Compliance: It specifies the requirements for the standard the system must follow. The standards may include the report format and according procedures.

Hardware Limitations: The software needs some existing or predetermined hardware to operate, thus imposing restrictions on the design. Hardware limitations can include the types of machines to be used operating system availability memory space etc.

Fault Tolerance: Fault tolerance requirements can place a major constraint on how the system is to be designed. Fault tolerance requirements often make the system more complex and expensive, so they should be minimized.

Security: Currently security requirements have become essential and major for all types of systems. Security requirements place restriction s on the use of certain commands control access to database, provide different kinds of access, requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system.

4. External Interface Requirements

For each external interface requirements:

1. All the possible interactions of the software with people hardware and other software should be clearly specified,
2. The characteristics of each user interface of the software product should be specified and
3. The SRS should specify the logical characteristics of each interface between the software product and the hardware components for hardware interfacing.

3.6 SOFTWARE REQUIREMENT SPECIFICATION DOCUMENT:

I. Introduction
   - A. System reference
   - B. Overall description
   - C. Software project constraints

II. Information Description
   - A. Information content representation
   - B. Information flow representation
     1. Data flow
     2. Control flow

III. Functional Description
   - A. Functional partitioning
   - B. Functional description
     1. Processing narrative
     2. Restrictions/limitations
     3. Performance requirements
     4. Design constraints
     5. Supporting diagrams
   - C. Control Description
     1. Control specification
     2. Design constraints

IV. Behavioral Description
   - A. System states
   - B. Events and actions

V. Validation and Criteria
   - A. Performance bounds
   - B. Classes of tests
   - C. Expected software response
   - D. Special considerations

VI. Bibliography

VII. Appendix

A general structure of an SRS will be as follows:

1. Introduction
   - **(i)** Purpose of this document
   - **(ii)** Scope of this document
   - **(iii)** Overview
2. General description
3. Functional Requirements
4. Interface Requirements
5. Performance Requirements
6. Design Constraints
7. Non-Functional Attributes
8. Preliminary Schedule and Budget
9. Appendices

Software Requirement Specification (SRS) Format as name suggests, is complete specification and description of requirements of software that needs to be fulfilled for successful development of software system. These requirements can be functional as well as non-requirements depending upon type of requirement. The interaction between different customers and contractor is done because its necessary to fully understand needs of customers.

Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

1. Introduction :

(i) Purpose of this Document –
At first, main aim of why this document is necessary and what's purpose of document is explained and described.

(ii) Scope of this document –
In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.

(iii) Overview –
In this, description of product is explained. It's simply summary or overall review of product.

2. General description :

In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.

3. Functional Requirements :

In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order.

4. Interface Requirements :

In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.

5. Performance Requirements :

In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc.

6. Design Constraints :

In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc.

7. Non-Functional Attributes :

In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

8. Preliminary Schedule and Budget :

In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

9. Appendices :

In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.