- *Establishing traceability between related items, and*
- *Change control.*

5.7 Software Requirements Specifications (SRS)

Software Requirements Specifications (SRS) is basically an organization's understanding in writing of a customer or potential client's system requirements and dependencies *at a particular point in time* prior to any actual design or development takes place.

It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

5.7.1 SRS Document

The SRS document itself states in precise and explicit language those functions and capabilities a software system must provide, as well as states any required constraints by which the system must abide.

The SRS also functions as a blueprint for completing a project with as little cost growth as possible. The SRS is often referred to as the *"parent document"* because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.

It's important to note that an SRS contains functional and non-functional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be.

5.7.2 Goals of SRS Document

A well-designed, well-written SRS document accomplishes the following four major goals:

Feedback to Customer

- SRS document provides a feedback to the customer.
- It is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems.

- Therefore, the SRS should be written in natural language, in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables, and so on.

Problem Decomposition

- SRS document decomposes the problem into component parts.
- The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion.

Input to Design Specification

- SRS document serves as an input to the design specification.
- The SRS also serves as the parent document to subsequent documents, such as the software design specification and statement of work.
- Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.

Production Validation Check

- SRS document serves as a product validation check.
- The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

5.7.3 Qualities/Characteristics of a Good SRS Document

The following are the quality characteristics of a good SRS document:

Complete

- SRS should be complete.
- SRS defines precisely all the live situations that will be encountered and the system's capability to successfully address them.

Consistent

- SRS should be consistent.
- SRS capability functions and performance levels are compatible, and the required quality features (security, reliability, etc.) do not negate those capability functions.

Accurate

- SRS precisely defines the system's capability in a real-world environment, as well as how it interfaces and interacts with it.
- This aspect of requirements is a significant problem area for many SRSs.

Modifiable

- The logical, hierarchical structure of the SRS should facilitate any necessary modifications and that too with a greater ease.

Ranked

- Individual requirements of an SRS are hierarchically arranged according to stability, security, perceived ease/difficulty of implementation, or other parameter that helps in the design of that and subsequent documents.

Testable

- SRS must be stated in such a manner that unambiguous assessment criteria can be derived from the SRS itself.

Traceable

- Each requirement in SRS must be uniquely identified to a source (e.g. use case, government requirement, industry standard, etc.)

Unambiguous

- SRS must contain requirements statements that can be interpreted in one way only i.e., it should be unambiguous.
- This is another area that creates significant problems for SRS development because of the use of natural language.

Valid

- A valid SRS is one in which all parties and project participants can understand, analyze, accept, or approve it.
- This is one of the main reasons SRSs are written using natural language.

Verifiable

- A verifiable SRS is consistent from one level of abstraction to another.
- Most attributes of a specification are subjective and a conclusive assessment of quality requires a technical review by domain experts.
- Using indicators of strength and weakness provide some evidence that preferred attributes are or are not present.

5.7.4 Who Writes SRS Document?

Usually, systems architects and programmers write SRSs with little, if any, help from the technical communications organization. And when that assistance is provided, it's often limited to an edit of the final draft just prior to going out the door.

In short, a requirements-gathering team consisting solely of programmers, product marketers, systems analysts/architects, and a project manager runs the risk of creating a SRS document that may be too heavily loaded with technology-focused or marketing-focused issues.

With the increased complexity of the SRS, it is better to involve the technical writers in the process of writing SRS document. The presence of a technical writer on the team helps place at the core of the project those user or customer requirements that provide more of an overall balance to the design of the SRS, product, and documentation.

5.7.5 Benefits of Involving Technical Writers in SRS Writing

Having technical writers involved throughout the entire SRS development process can offer several benefits:

- *Technical writers are skilled information gatherers, ideal for eliciting and articulating customer requirements.* The presence of a technical writer on the requirements-gathering team helps balance the type and amount of information extracted from customers, which can help improve the SRS.
- *Technical writers can better assess and plan documentation projects and better meet customer document needs.* Working on SRSs provides technical writers with an opportunity for learning about customer needs firsthand—early in the product development process.
- *Technical writers know how to determine the questions that are of concern to the user or customer regarding ease of use and usability.* Technical writers can then take that knowledge and apply it not only to the specification and documentation development, but also to user interface development.
- *Technical writers involved early and often in the process, can become an information resource throughout the process, rather than an information gatherer at the end of the process.*

5.7.6 Uses of SRS Document

The following are few major uses of SRS documents:

- *Project managers base their plans and estimates of schedule, effort and resources on it.*
- *Development team needs it to develop product.*
- *The testing group needs it to generate test plans based on the described external behaviour.*
- *The maintenance and product support staff need it to understand what the software product is supposed to do.*
- *The publications group writes documentation, manuals, etc. from it.*
- *Customers rely on it to know what product they can expect.*
- *Training personnel can use it to help develop educational material for software product.*

5.7.7 Components of SRS Document

SRS development will be a collaborative effort for a particular project and ultimately results into a SRS Document. Several standards organizations including IEEE have identified the following nine components that must be addressed when designing and writing SRS:

1. Interfaces
2. Functional Capabilities
3. Performance Levels
4. Data Structures/Elements
5. Safety
6. Reliability
7. Security/Privacy
8. Quality
9. Constraints and Limitations

5.7.8 SRS Document Structure

The basic structure of a SRS document is shown in **Table 5.1.** This example is an adaptation and extension of the IEEE Standard 830-1998.

TABLE 5.1: Sample of a Basic SRS Outline

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document conventions
   - 1.3 Intended audience
   - 1.4 Additional information
   - 1.5 Contact information/SRS team members
   - 1.6 References
2. **Overall Description**
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User classes and characteristics
   - 2.4 Operating environment
   - 2.5 User environment
   - 2.6 Design/implementation constraints
   - 2.7 Assumptions and dependencies
3. **External Interface Requirements**
   - 3.1 User interfaces
   - 3.2 Hardware interfaces
   - 3.3 Software interfaces
   - 3.4 Communication protocols and interfaces
4. **System Features**
   - 4.1 System feature A
     - 4.1.1 Description and priority
     - 4.1.2 Action/result
     - 4.1.3 Functional requirements
   - 4.2 System feature B
5. **Other Non-functional Requirements**
   - 5.1 Performance requirements
   - 5.2 Safety requirements
   - 5.3 Security requirements
   - 5.4 Software quality attributes
   - 5.5 Project documentation
   - 5.6 User documentation
6. **Other Requirements**

Appendix A: Terminology/Glossary/Definitions list

Appendix B: To be determined

5.7.9 SRS Document Template

The easiest way of writing an SRS document is to use SRS template. Most of the software development organisations develop their own SRS templates, which can serve the purpose for all the software projects undertaken for development.

One such SRS Document Template structure is described in **Table 5.2.**

TABLE 5.2: SRS Document Template

Document Title
Author(s)
Affiliation
Address
Date
Document Version Control Information

1. **INTRODUCTION**

   1.1 **Purpose of this document**

   Describes the purpose of the document, and the intended audience.

   1.2 **Scope of this document**

   Describes the scope of this requirements definition effort. Introduces the requirements elicitation team, including users, customers, system engineers, and developers.

   This section also details any constraints that were placed upon the requirements elicitation process, such as schedules, costs, or the software engineering environment used to develop requirements.

   1.3 **Overview**

   Provides a brief overview of the product defined as a result of the requirements elicitation process.

   1.4 **Business Context**

   Provides an overview of the business organization sponsoring the development of this product. This overview should include the business's mission statement and its organizational objectives or goals.

2. **GENERAL DESCRIPTION**

   2.1 **Product Functions**

   Describes the general functionality of the product, which will be discussed in more detail below.

   2.2 **Similar System Information**

   Describes the relationship of this product with any other products. Specifies if this product is intended to be stand-alone, or else used as a component of a larger product. If the latter, this section discusses the relationship of this product to the larger product.

   2.3 **User Characteristics**

   Describes the features of the user community, including their expected expertise with software systems and the application domain.

   2.4 **User Problem Statement**

   This section describes the essential problem(s) currently confronted by the user community.

2.5 **User Objectives**
This section describes the set of objectives and requirements for the system from the user's perspective. It may include a "wish list" of desirable characteristics, along with more feasible solutions that are in line with the business objectives.

2.6 **General Constraints**
Lists general constraints placed upon the design team, including speed requirements, industry protocols, hardware platforms, and so forth.

3. **FUNCTIONAL REQUIREMENTS**
This section lists the functional requirements in ranked order. Functional requirements describe the possible effects of a software system, in other words, *what* the system must accomplish. Other kinds of requirements (such as interface requirements, performance requirements, or reliability requirements) describe *how* the system accomplishes its functional requirements. Each functional requirement should be specified in a format similar to the following:

1. **Short, imperative sentence stating highest ranked functional requirement.**
   1. *Description*
      A full description of the requirement.
   2. *Criticality*
      Describes how essential this requirement is to the overall system.
   3. *Technical issues*
      Describes any design or implementation issues involved in satisfying this requirement.
   4. *Cost and schedule*
      Describes the relative or absolute costs associated with this issue.
   5. *Risks*
      Describes the circumstances under which this requirement might not able to be satisfied, and what actions can be taken to reduce the probability of this occurrence.
   6. *Dependencies with other requirements*
      Describes interactions with other requirements.
   7. *... others as appropriate*
2. **<Name of second highest ranked requirement>**

And so forth...

4. **INTERFACE REQUIREMENTS**
This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared memory, data streams, and so forth.

4.1 **User Interfaces**
Describes how this product interfaces with the user.

4.1.1 **GUI**
Describes the graphical user interface if present. This section should include a set of screen dumps or mock-ups to illustrate user interface features.
If the system is menu-driven, a description of all menus and their components should be provided.

4.1.2 CLI

Describes the command-line interface (or command-user interface, CUI, if present). For each command, a description of all arguments and example values and invocations should be provided.

4.1.3 API

Describes the application programming interface, if present. For each public interface function, the name, arguments, return values, examples of invocation, and interactions with other functions should be provided.

4.1.4 Diagnostics or ROM

Describes how to obtain debugging information or other diagnostic data.

4.2 Hardware Interfaces

Describes interfaces to hardware devices.

4.3 Communications Interfaces

Describes network interfaces.

4.4 Software Interfaces

Describes any remaining software interfaces not included above.

5. PERFORMANCE REQUIREMENTS

Specifies speed and memory requirements.

6. DESIGN CONSTRAINTS

Specifies any constraints for the design team using this document.

6.1 Standards Compliance

6.2 Hardware Limitations

... Others as appropriate

. OTHER NON-FUNCTIONAL ATTRIBUTES

Specifies any other particular non-functional attributes required by the system. Examples are provided below.

7.1 Security

7.2 Binary Compatibility

7.3 Reliability

.4 Maintainability

.5 Portability

6 Extensibility

7 Reusability

Application Affinity/Compatibility

Resource Utilization

Serviceability

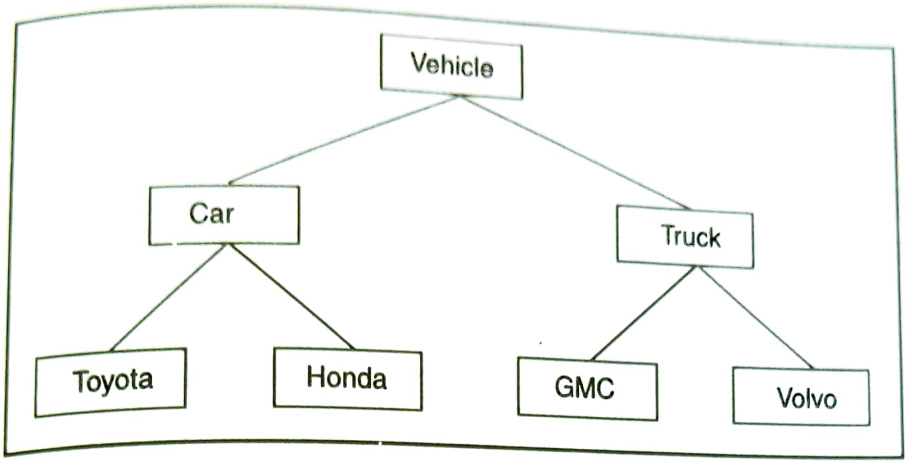
Others as appropriate

LIMINARY OBJECT-ORIENTED DOMAIN ANALYSIS

section presents a list of the fundamental objects that must be modelled within the system to its requirements. The purpose is to provide an alternative, "structural" view on the requirements above and how they might be satisfied in the system.

8.1 Inheritance Relationships

This section should contain a set of graphs that illustrate the primary inheritance hierarchy (is-kind-of) for the system. For example:



8.2 Class descriptions

This section presents a more detailed description of each class identified during the OO Domain Analysis.

Each class description should conform to the following structure:

8.2.1 *<Class name>*

8.2.1.1 *Abstract or Concrete*

Indicates whether this class is abstract or concrete.

8.2.1.2 *List of Superclasses*

Names all immediate superclasses.

8.2.1.3 *List of Subclasses*

Names all immediate subclasses.

8.2.1.4 *Purpose*

States the basic purpose of the class.

8.2.1.5 *Collaborations*

Names each class with which this class must interact in order to accomplish its purpose, and how.

8.2.1.6 *Attributes*

Lists each attribute (state variable) associated with each instance of this class, and indicates examples of possible values (or a range).

8.2.1.7 *Operations*

Lists each operation that can be invoked upon instances of this class. For each operation, the arguments (and their type), the return value (and its type), and any side effects of the operation should be specified.

8.2.1.8 *Constraints*

Lists any restrictions upon the general state or behaviour of instances of this class.

9. OPERATIONAL SCENARIOS

This section should describe a set of scenarios that illustrate, from the user's perspective, what will be experienced when utilizing the system under various situations.

10. PRELIMINARY SCHEDULE

This section provides an initial version of the project plan, including the major tasks to be accomplished, their interdependencies, and their tentative start/stop dates. The plan also includes information on hardware, software, and wetware resource requirements.

The project plan should be accompanied by one or more PERT or GANTT charts.

11. PRELIMINARY BUDGET

This section provides an initial budget for the project, itemized by cost factor.

12. APPENDICES

Specifies other useful information for understanding the requirements. All SRS documents should include at least the following two appendices:

12.1 Definitions, Acronyms, Abbreviations

Provides definitions of unfamiliar definitions, terms, and acronyms.

12.2 References

Provides complete citations to all documents and meetings referenced or used in the preparation of this document.

5.8 Specification Techniques