**UNIT-II**

**CHAPTER-II**

**Requirements Analysis and Specification**

 The requirements analysis and specification phase start 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 can be stated in a nutshell as follows.

The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

**REQUIREMENTS GATHERING AND ANALYSIS**

The complete set of requirements are almost never available in the form of a single document from the customer. In fact, it would be unrealistic to expect the customers to produce a comprehensive document containing a precise description of what they want. Further, the complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be systematically gathered by the analyst from several sources in bits and pieces.

These gathered requirements need to be analyzed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources. We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

* Requirements gathering
* Requirements analysis

**Requirements Gathering**

Requirements gathering activity is also popularly known as *requirements elicitation*. The primary objective of the requirement’s gathering task is to collect the requirements from the *stakeholders*.

Requirements gathering may sound like a simple task. However, in practice 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. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually. In this case, a working model of the system (that is, a manual system) exists. Availability of a working model is usually of great help in requirements gathering.

For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures. In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain oﬃce. In this case, the analyst would have to study the input and output forms and then understand how the outputs are produced from the input data. However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more diﬃcult. In the absence of a working system, much more imagination and creativity is required on the part of the system analyst.

Typically, even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site, the analysts normally interview the end-users and customer representatives,1 carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

**Requirements Gathering Activities**

**1.** **Studying the 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 analyst. Typically, these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, and the broad category of features required.

**2. Interview:** Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants.

**3. Task analysis:** The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. 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.

**4. 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 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.

**5. Form analysis:** Form analysis is an important and effective requirement 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 the input data would be used by the system to produce the corresponding output data is determined from the users.

**Requirements Analysis**

The main purpose of the requirements analysis activity is to analyze 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.

For carrying out requirements analysis effectively, the analyst first needs to develop a clear grasp of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

* What is the problem?
* Why is it important to solve the problem?
* What exactly are the data input to the system and what exactly are the data output by the system?
* What are the possible procedures that need to be followed to solve the problem?
* What are the likely complexities that might arise while solving the problem?

 If there are external software or hardware with which the developed software has\ to interface, then what should be the data interchange formats with the external systems?

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 in 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. The following are two examples of anomalous requirements:

**Example:**

* While gathering the requirements 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 may get interpreted differently by different developers.
* **Inconsistency:** Two requirements are said to be inconsistent, if one of the requirements contradicts the other. The following are two examples of inconsistent requirements:

**Example:**

* Consider the following partial 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 500C
* When the temperature of the furnace rises above 500C, 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. Often, incompleteness is caused by the inability of the customer to visualize the system that is to be developed and to anticipate all the features that would be required. 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.

**Example:**

* In a chemical plant automation software, suppose one of the requirements is that if the internal temperature of the reactor exceeds 200°C 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)**

The Main aim of requirements specification:

* + Systematically organize the requirements arrived during requirements analysis.
	+ Document requirements properly.
* The SRS document is useful in various contexts:
	+ Statement of user needs
	+ Contract document
	+ Reference document
	+ Definition for implementation
* 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. The SRS document usually contains all the user requirements in a structured though an informal form.
* Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write. One reason for this diﬃculty is that the SRS document is expected to cater to the needs of a wide variety of audience. In the following subsection, we discuss the different categories of users of an SRS document and their needs from it.

**Users of SRS Document**

Usually, a large number of different people need the SRS document for very different purposes. Some of 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. Remember that the customer may not be the user of the software, but may be some one employed or designated by the user. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.
* **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 its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
* **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 understand the functionalities supported by the system. A clear knowledge of the functionalities can help them to understand the design and code. Also, 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.

Many software engineers in a project consider the SRS document to be a reference document. However, it is often more appropriate to think of the SRS document as the documentation of a contract between the development team and the customer. In fact, the SRS document can be used to resolve any disagreements between the developers and the customers that may arise in the future. The SRS document can even be used as a legal document to settle disputes between the customers and the developers in a court of law. Once the customer agrees to the SRS document, the development team proceeds to develop the software and ensure that it conforms to all the requirements mentioned in the SRS document.

**Characteristics of a Good SRS Document**

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

* **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
* **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues. This view with which a requirement specification is written, has been shown in the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software is not discussed at all.
* **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*. Traceability is also important to verify the results of a phase with respect to the previous phase and to analyze the impact of changing a requirement on the design elements and the code.
* **Modifiable:** Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development. Also, 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. A requirement such as “the system should be user friendly” is not verifiable. On the other hand, the requirement—“When the name of a book is entered, the software should display whether the book is available for issue or it has been loaned out” is verifiable. Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

**Attributes of Bad SRS Documents**

SRS documents written by novices frequently suffer from a variety of problems the most damaging problems are incompleteness, ambiguity, and contradictions.

There are many other types of problems that a specification document might suffer from. By knowing these problems, one can try to avoid them while writing an SRS document. Some of the important categories of problems that many SRS documents suffer from are as follows:

* **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 8 am and 5 pm, 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.
* Several other “sins” of SRS documents can be listed and used to guard against writing a bad SRS document and is also used as a checklist to review an SRS document.

**Important Categories of Customer Requirements**

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. it is useful to consider a software as offering a set of functions {*fi*} to the user. These functions can be considered similar to a mathematical function *f* : *I* → *O*, meaning that a function transforms an element (*ii*) in the input domain (*I*) to a value (*oi*) in the output (*O*). This functional view of a system is shown schematically in Figure 4.1.

* Each function *f*i of the system can be considered as reading certain data *ii*, and then transforming a set of input data (*ii*) to the corresponding set of output data (*oi*). 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.
* Considering that the functional requirements are a crucial part of the SRS document, we discuss functional requirements how the functional requirements can be identified from a problem description. how the functional requirements can be documented effectively.

**Non-functional requirements**

* 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.
* 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.
* 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.
* One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree. The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.
* Other non-functional requirements: This section contains a description of non- functional requirements that neither are 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.

**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. The high-level functions would be split into smaller sub requirements.

Each high-level function is an instance of use of the system (use case) by the user in some way. A high-level function is one using which the user can get some useful piece of work done.

For example, how useful must a piece of work be performed by the system for it to be called ‘a useful piece of work’? Can the printing of the statements of the ATM transaction during withdrawal of money from an ATM be called a useful piece of work? Printing of ATM transaction should not be considered a high-level requirement, because the user does not specifically request for this activity. The receipt gets printed automatically as part of the withdraw money function. Usually, the user invokes (requests) the services of each high-level requirement.

* It may therefore be possible to treat print receipt as part of the withdraw money function rather than treating it as a high-level function. It is therefore required that for some of the high-level functions, we might have to debate whether we wish to consider it as a high-level function or not. However, it would become possible to identify most of the high-level functions without much diﬃculty after practicing the solution to a few exercise problems.
* 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. For example, in a library automation software, a high-level functional requirement might be search-book.
* This function involves accepting a book name or a set of key words from the user, running a matching algorithm on the book list, and finally outputting the matched books. The generated system response can be in several forms, e.g., display on the terminal, a print out, some data transferred to the other systems, etc. However, in degenerate cases, a high-level requirement may not involve any data input to the system or production of displayable results. For example, it may involve switch on a light, or starting a motor in an embedded application.
* Are high-level functions of a system similar to mathematical functions?
* 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.
* The different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behavior exhibited by the system for the same high-level function. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement can consist of several sub-requirements.



**Is it possible to determine all input and output data precisely?**

* In a requirements specification document, it is desirable to define the precise data input to the system and the precise data output by the system. Sometimes, the exact data items may be very diﬃcult to identify. This is especially the case, when no working model of the system to be developed exists. In such cases, the data in a high-level requirement should be described using high-level terms and it may be very diﬃcult to identify the exact components of this data accurately.
* Another aspect that must be kept in mind is that the data might be input to the system in stages at different points in execution. For example, consider the withdraw-cash function of an *automated teller machine* (ATM) of Figure 4.2. Since during the course of execution of the withdraw-cash function, the user would have to input the type of account, the amount to be withdrawn, it is very diﬃcult to form a single high-level name that would accurately describe both the input data. However, the input data for the subfunctions can be more accurately described

**How to Identify the 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.
* Remember that 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 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.
* The decision regarding which functionality of the system can be taken to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some subjectivity. For example, consider the issue-book function in a Library Automation System. Suppose, when a user invokes the issue-book function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the issue-book function? Many times, the choice is obvious. But sometimes it requires making non-trivial decisions.

**How to Document the Functional Requirements**

* Once all the high-level functional requirements have been identified and the requirements problems have been eliminated, these are documented. 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.
* We now illustrate the specification of the functional requirements through two examples. Let us first try to document the withdraw-cash function of an *automated teller machine* (ATM) system in the following. The *withdraw-cash* is a high level requirement. It has several sub-requirements corresponding to the different user interactions.
* These user interaction sequences may vary from one invocation from another depending on some conditions. These different interaction sequences capture the different *scenarios*. To accurately describe a functional requirement, we must document all the different scenarios that may occur.

**Example: Withdraw cash from ATM**

***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 following—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.

* In order to properly identify the high-level requirements, a lot of common sense and the ability to visualize various scenarios that might arise in the operation of a function are required. Please note that when any of the aspects of a requirement, such as the state, processing description, next function to be executed, etc. are obvious, we have omitted it.
* We have to make a trade-off between cluttering the document with trivial details versus missing out some important descriptions.
* **Specification of large software:** If there are large number of functional requirements (much larger than seen), should they just be written in a long-numbered list of requirements?
* A better way to organize the functional requirements in this case would be to split 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:** Even for experienced analysts, a common dilemma is in specifying too little or specifying too much. In practice, 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.
* **Traceability**

Traceability means that it would be possible to identify (trace) the specific design component which implements a given requirement, the code part that corresponds to a given design component, and test cases that test a given requirement. Thus, any given code component can be traced to the corresponding design component, and a design component can be traced to a specific requirement that it implements and *vice versa*.

Traceability analysis is an important concept and is frequently used during software development. For example, by doing a traceability analysis, we can tell whether all the requirements have been satisfactorily addressed in all phases. It can also be used to assess the impact of a requirements change. That is, traceability makes it easy to identify which parts of the design and code would be affected, when certain requirement change occurs. It can also be used to study the impact of a bug that is known to exist in a code part on various requirements, etc.

* To achieve traceability, it is necessary that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible for different documents to uniquely refer to specific requirements.
* An example scheme of numbering the functional requirements is shown in Examples 4.7 and 4.8, where the functional requirements have been numbered R.1, R.2, etc. and the sub requirements for the requirement R.1 have been numbered R.1.1, R.1.2, etc.

**Organization of the SRS Document**

* The organization of an SRS document as prescribed by the IEEE 830 standard [IEEE 830]. Please note that IEEE 830 standard has been intended to serve only as a guideline for organizing a requirements specification document into sections and allows the flexibility of tailoring it, as may be required for specific projects.
* Depending on the type of project being handled, some sections can be omitted, introduced, or interchanged as may be considered prudent by the analyst. However, organization of the SRS document to a large extent depends on the preferences of the system analyst himself, and he is often guided in this by the policies and standards being followed by the development company.
* Also, the organization of the document and the issues discussed in it to a large extent depend on the type of the product being developed. However, irrespective of the company’s principles and product type, the three basic issues that any SRS document should discuss are—functional requirements, non-functional requirements, and guidelines for system implementation.
* The introduction section should describe the context in which the system is being developed, and provide an overall description of the system, and the environmental characteristics. The introduction section may include the hardware that the system will run on, the devices that the system will interact with and the user skill-levels.
* Description of the user skill-level is important, since the command language design and the presentation styles of the various documents depend to a large extent on the types of the users it is targeted for. For example, if the skill-levels of the users is “novice”, it would mean that the user interface has to be very simple and rugged, whereas if the user-level is “advanced”, several short cut techniques and advanced features may be provided in the user interface.
* It is desirable to describe the formats for the input commands, input data, output reports, and if necessary, the modes of interaction. We have already discussed how the contents of the Sections on the functional requirements, the non-functional requirements, and the goals of implementation should be written. In the following subsections, we outline the important sections that an SRS document should contain as suggested by the IEEE 830 standard, for each section of the document, we also briefly discuss the aspects that should be discussed in it.
* **Introduction**

**Purpose:** This section should describe where the software would be deployed and how the software would be used.

**Project scope:** This section should briefly describe the overall context within which the software is being developed. For example, the parts of a problem that are being automated and the parts that would need to be automated during future evolution of the software.

**Environmental characteristics:** This section should briefly outline the environment (hardware and other software) with which the software will interact.

* **Overall description of organization of SRS document**

**Product perspective:** This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing system, or it is a new software. If the software being developed would be used as a component of a larger system, a simple schematic diagram can be given to show the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

**Product features:** This section should summarize the major ways in which the software would be used. Details should be provided in Section 3 of the document. So, only a brief summary should be presented here.

* **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities that they are expected to invoke, or their levels of expertise in using computers.
* **Operating environment:** This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.
* **Design and implementation constraints:** In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.
* **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.
* **Functional requirements**

This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.

1. User class 1

(a) Functional requirement 1.1

(b) Functional requirement 1.2

2. User class 2

 (a) Functional requirement 2.1

 (b) Functional requirement 2.2

* **External interface requirements**
* **User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides that are to be followed, screen layout constraints, standard push buttons (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, etc. The details of the user interface design should be documented in a separate user interface specification document.
* **Hardware interfaces:** This section should describe the interface between the software and the hardware components of the system. This section may include the description of the supported device types, the nature of the data and control interactions between the software and the hardware, and the communication protocols to be used.
* **Software interfaces:** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc. Identify the data items that would be input to the software and the data that would be output should be identified and the purpose of each should be described.
* **Communications interfaces:** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc. This section should define any pertinent message formatting to be used. It should also identify any communication standards that will be used, such as TCP sockets, FTP, HTTP, or SHTTP. Specify any communication security or encryption issues that may be relevant, and also the data transfer rates, and synchronisation mechanisms.
* **Other non-functional requirements for organization of SRS document**

This section should describe the non-functional requirements other than the design and implementation constraints and the external interface requirements that have been described

* **Performance requirements:** Aspects such as number of transactions to be completed per second should be specified here. Some performance requirements may be specific to individual functional requirements or features. These should also be specified here.
* **Safety requirements:** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here. For example, recovery after power failure, handling software and hardware failures, etc. may be documented here.
* **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software. Any user identity authentication requirements should be described here. It should also refer to any external policies or regulations concerning the security issues. Define any security or privacy certifications that must be satisfied.
* For software that have distinct modes of operation, in the functional requirements section, the different modes of operation can be listed and, in each mode, the specific functionalities that are available for invocation can be organized as follows.
* **Functional requirements**

1. Operation mode 1

* (a) Functional requirement 1.1
* (b) Functional requirement 1.2

2. Operation mode 2

* (a) Functional requirement 2.1

(b) Functional requirement 2.2