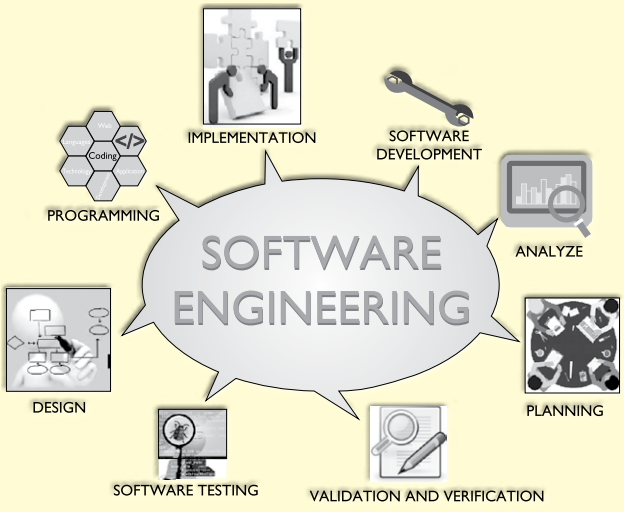
CHAPTER

4

REQUIREMENTS ANALYSIS AND SPECIFICATION



CHAPTER OUTLINE

- Requirements Gathering and Analysis
- Software Requirements Specification (SRS)
- Formal System Specification
- Axiomatic Specification
- Algebraic Specification
- Executable Specification and 4GL

Ours is a world where people don't know what they want and yet are willing to go through hell to get it.

—Don Marquis

LEARNING OBJECTIVES

- Requirements gathering and analysis techniques
- Requirements specification using IEEE 830 template
- Representation of complex logic
- Formal system specification

All plan-driven life cycle models prescribe that before starting to develop a software, the exact requirements of the customer must be understood and documented. In the past, many projects have suffered because the developers started to implement something without determining whether they were building what the customers exactly wanted. Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the later life cycle phases, and thereby alarmingly pushes up the development costs. This also sets the ground for customer dissatisfaction and bitter customer-developer disputes and protracted legal battles. No wonder that experienced developers consider the requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.

> Experienced developers take considerable time to understand the exact requirements of the customer and to meticulously document those. They know that without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.

For any type of software development project, availability of a good quality requirements document has been acknowledged to be a key factor in the successful completion of the project. A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases. When software is developed in a contract mode for some other organisation (that is, an outsourced project), the crucial role played by documentation of the precise requirements cannot be overstated. Even when an organisation develops a generic software product, the situation is not very different since some personnel from the organisation's own marketing department act as the customer. Therefore, for all types of software development projects, proper formulation of requirements and their effective documentation is vital. However, for very small software service projects, the agile methods advocate incremental development of the requirements.

An overview of requirements analysis and specification phase

The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.

The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed. The requirements specification document is usually called as the *software requirements specification* (SRS) document. The goal of the requirements analysis and specification phase 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.

Who carries out requirements analysis and specification?

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather and analyse customer requirements and then write the requirements specification document are known as *system analysts* in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the *software requirements specification* (SRS) document.

The SRS document is the final outcome of the requirements analysis and specification phase.

How is the SRS document validated?

Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

What are the main activities carried out during requirements analysis and specification phase?

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis
- Requirements specification

In the next section, we will discuss the requirements gathering and analysis activity and in the subsequent section we will discuss the requirements specification activity.

4.1 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 analysed 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

We discuss these two tasks in the following subsections.

4.1.1 Requirements Gathering

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

A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly is concerned with the software.

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.

Given that many customers are not computer savvy, they describe their requirements very vaguely. Good analysts share their experience and expertise with the customer and

[1] Note that the customer and the users of a software may, in general, be different. For example, the customer may be an organisation and the users may be a few select employees of the organisation.

give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete. In the following, we briefly discuss the important ways in which an experienced analyst gathers requirements:

1. Studying existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the 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. The library members would like to use the software to query availability of books and issue and return books. The librarians might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc. The accounts personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.

To systematise this method of requirements gathering, the Delphi technique can be followed. In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document. This procedure is repeated till the different users agree on the set of requirements.

3. Task analysis: 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 realise the required functionality in consultation with the users. For example, consider the issue book task. The steps may be—authenticate user, check the number of books issued to the customer and determine if the maximum number of books that this member can borrow has been reached, check whether the book has been reserved, post the book issue details in the member's record, and finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.

Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.

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

- Book is issued successfully to the member and the book issue slip is printed.
- The book is reserved, and hence cannot be issued to the member.

- The maximum number of books that can be issued to the member is already reached, and no more books can be issued to the member.

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

5. Form analysis: Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis, the exiting forms and the formats of the notifications produced are analysed 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.

CASE STUDY 4.1

Requirements gathering for office automation at the CSE department

The academic, inventory, and financial information at the CSE (Computer Science and Engineering) department of a certain institute was being carried out manually by two office clerks, a store keeper, and two attendants. The department has a student strength of 500 and a teacher strength of 30. The head of the department (HoD) wants to automate the office work. Considering the low budget that he has at his disposal, he entrusted the work to a team of student volunteers.

For requirements gathering, a member of the team who was responsible for requirements analysis and specification (analyst) was first briefed by the HoD about the specific activities to be automated. The HoD mentioned that three main aspects of the office work needs to be automated—stores-related activities, student grading activities, and student leave management activities. It was necessary for the analyst to meet the other categories of users. The HoD introduced the analyst (a student) to the office staff. The analyst first discussed with the two clerks regarding their specific responsibilities (tasks) that were required to be automated. For each task, they asked the clerks to brief them about the steps through which these are carried out. The analyst also enquired about the various scenarios that might arise for each task. The analyst collected all types of forms that were being used by the student and the staff of the department to submit various requests and to register various types of information with the office (e.g., student course registration, course grading) or requests for some specific service (e.g., issue of items from store). He also collected samples of various types of documents (outputs) the clerks were preparing. Some of these had specific printed forms that the clerks filled up manually, and others were entered using a spreadsheet, and then printed out on a laser printer. For each output form, the analyst consulted the clerks regarding how these different entries are generated from the input data.

The analyst met the storekeeper and enquired about the material issue procedures, store ledger entry procedures, and the procedures for raising indents on various vendors. He also collected copies of all the relevant forms that were being used by the storekeeper. The analyst also interviewed the student and faculty representatives. Since it was needed to automate the existing activities of a working office, the analyst could without much difficulty obtain the exact formats of the input data, output data, and the precise description of the existing office procedures.

4.1.2 Requirements Analysis

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness, since each stakeholder typically has only a partial and incomplete view of the software. Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.

The main purpose of the requirements analysis activity is to analyse 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?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.

Let us examine these different types of requirements problems in detail.

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 4.1 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.

EXAMPLE 4.2 In the case study 4.1, suppose one office clerk described the following requirement: during the final grade computation, if any student scores a sufficiently low grade in a semester, then his parents would need to be informed. This is clearly an ambiguous requirement as it lacks any well defined criterion as to what can be considered as a "sufficiently low grade".

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 4.3 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 500°C.
- When the temperature of the furnace rises above 500°C, the water shower should be switched- on and the furnace should remain on.

The requirements expressed by the two stakeholders are clearly inconsistent.

EXAMPLE 4.4 In the case study 4.1, suppose one of the clerks gave the following requirement—a student securing fail grades in three or more subjects must repeat the courses over an entire semester, and he cannot credit any other courses while repeating the courses. Suppose another clerk expressed the following requirement—there is no provision for any student to repeat a semester; the student should clear the subject by taking it as an extra subject in any later semester.

There is a clear inconsistency between the requirements given by the two stakeholders.

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 visualise 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. The following are two examples of incomplete requirements:

EXAMPLE 4.5 Suppose for the case study 4.1, one of the clerks expressed the following—If a student secures a *grade point average* (GPA) of less than 6, then the parents of the student must be intimated about the regrettable performance through a (postal) letter as well as through e-mail.

However, on an examination of all requirements, it was found that there is no provision by which either the postal or e-mail address of the parents of the students can be entered into the system. The feature that would allow entering the e-mail ids and postal addresses of the parents of the students was missing, thereby making the requirements incomplete.

EXAMPLE 4.6 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.

Can an analyst detect all the problems existing in the gathered requirements?

Many of the inconsistencies, anomalies, and incompleteness are detected effortlessly, while some others require a focused study of the specific requirements. A few problems in the requirements can, however, be very subtle and escape even the most experienced eyes. Many of these subtle anomalies and inconsistencies can be detected, if the requirements are specified and analysed using a formal method. Once a system has been formally specified, it can be systematically (and even automatically) analysed to remove all problems from the specification. We will discuss the basic concepts of formal system specification in Section 4.3. Though the use of formal techniques is not widespread, the current practice is to formally specify only the safety-critical[2] parts of a system.

4.2 SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organise 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.

4.2.1 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

[2] A safety-critical system is one whose improper working can result in financial loss, loss of property, or life.

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.

4.2.2 Why Spend Time and Resource to Develop an SRS Document?

A well-formulated SRS document finds a variety of usage other than the primary intended usage as a basis for starting the software development work. In the following subsection, we identify the important uses of a well-formulated SRS document:

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

Reduces future reworks: The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

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

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

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

Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

4.2.3 Characteristics of a Good SRS Document

The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. IEEE Recommended Practice for Software Requirements Specifications [IEEE, 1998] describes the content and qualities of a good software requirements specification (SRS). Some of the identified desirable qualities of an SRS document are the following:

- **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 behaviour of the system and not discuss the implementation issues. This view with which a requirement specification is written, has been shown in Figure 4.1. Observe that in Figure 4.1, 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.

> 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.

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 analyse 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

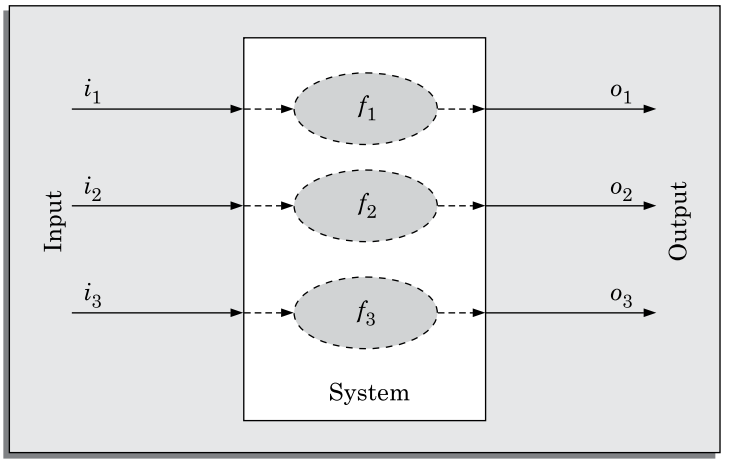


FIGURE 4.1 The black-box view of a system as performing a set of functions.

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. Having the description of a requirement scattered across many places in the SRS document may not be wrong—but it tends to make the requirement difficult to understand and also any modification to the requirement would become difficult as it would require changes to be made at large number of places in the document.

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.

4.2.4 Attributes of Bad SRS Documents

SRS documents written by novices frequently suffer from a variety of problems. As discussed earlier, 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.

4.2.5 Important Categories of Customer Requirements

A good SRS document, should properly categorize and organise the requirements into different sections [IEEE 830]. As per the IEEE 830 guidelines, the important categories of user requirements are the following:

In the following subsections, we briefly describe the different categories of 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. We have already pointed out in Chapter 2 that it is useful to consider a software as offering a set of functions $\{f_i\}$ to the user. These functions can be considered similar to a mathematical function $f : I \rightarrow O$, 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 in more detail in Section 4.2.6. Section 4.2.7 discusses how the functional requirements can be identified from a problem description. Finally, Section 4.2.8 discusses how the functional requirements can be documented effectively.

Non-functional requirements

The non-functional requirements are non-negotiable obligations that must be supported by the software. The non-functional requirements capture those requirements of the customer

that cannot be expressed as functions (i.e., accepting input data and producing output data). Non-functional requirements usually address aspects concerning external interfaces, user interfaces, maintainability, portability, usability, maximum number of concurrent users, timing, and throughput (transactions per second, etc.). The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum defined level in these requirements can be considered as a failure and make the software unacceptable by the customer.

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

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 honoured; hardware limitations; interfaces with other applications; specific technologies, tools, and databases to be used; specific communications protocols to be used; security considerations; design conventions or programming standards to be followed, etc. Consider an example of a constraint that can be included in this section—Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organisation.

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.

Goals of implementation

The 'goals of implementation' part of the SRS document offers some general suggestions regarding the software to be developed. These are not binding on the developers, and they

may take these suggestions into account if possible. For example, the developers may use these suggestions while choosing among different design solutions.

The goals of implementation section might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately. It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

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

How to classify the different types of requirements?

We should be clear regarding the aspects of the system requirement that are to be documented as the functional requirement, the ones to be documented as non-functional requirement, and the ones to be documented as the goals of implementation. Aspects which can be expressed as transformation of some input data to some output data (i.e., the functions of the system) should be documented as the functional requirement. Any other requirements whose compliance by the developed system can be verified by inspecting the system are documented as non- functional requirements. Aspects whose compliance by the developed system need not be verified but are merely included as suggestions to the developers are documented as goals of the implementation.

The difference between non-functional requirements and guidelines is the following. Non-functional requirements would be tested for compliance, before the developed product is accepted by the customer whereas guideline, on the other hand, are customer request that are desirable to be done, but would not be tested during product acceptance.

Functional requirements form the basis for most design and test methodologies. Therefore, unless the functional requirements are properly identified and documented, the design and testing activities cannot be carried out satisfactorily. We discuss how to document the functional requirements in the next section.

4.2.6 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 subrequirements. 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.

However, the above is not a very accurate definition of a high-level function. 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 practising 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?

We all know that a mathematical function transforms input data to output data. A high-level function transforms certain input data to output data. However, except for very simple high- level functions, a function rarely reads all its required data in one go and rarely outputs all the results in one shot. In fact, a high-level function usually involves a series of interactions between the system and one or more users. An example of the interactions that may occur in a single high-level requirement has been shown in Figure 4.2. In Figure 4.2, the user inputs have been represented by rectangles and the response produced by the system by circles. Observe that the rectangles and circles alternate in the execution of a single high-level function of the system, indicating a series of requests from the user and the corresponding responses from the system. Typically, there is some initial data input by the user. After accepting this, the system may display some response (called *system action*). Based on this, the user may input further data, and so on.

> 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.

In Figure 4.2, the different scenarios occur depending on the amount entered for withdrawal. The different scenarios are essentially different behaviour 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.

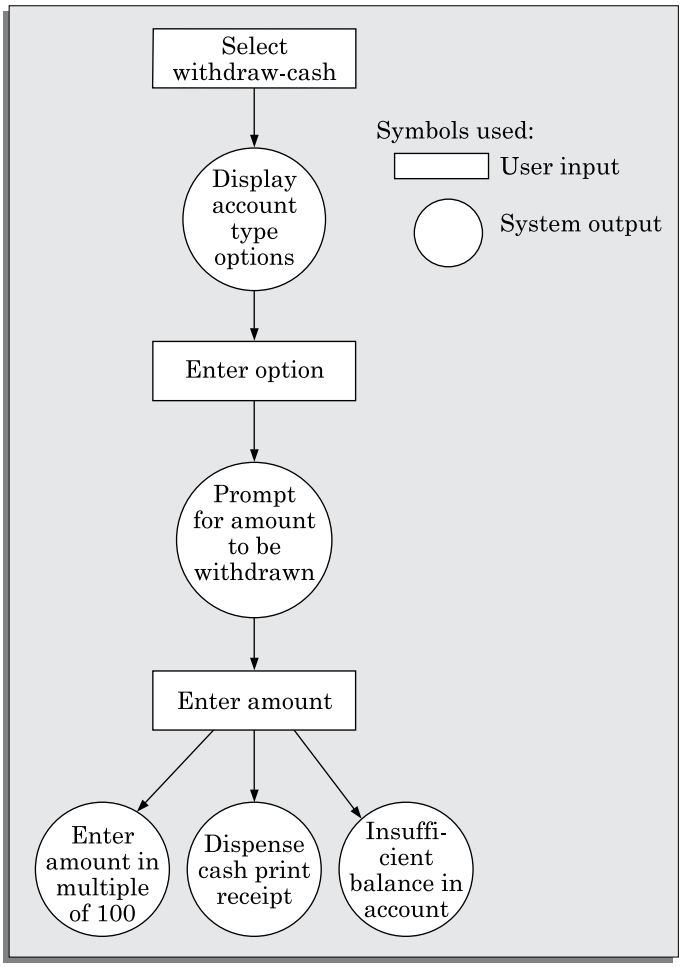


FIGURE 4.2 User and system interactions in high-level functional requirement.

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.

4.2.7 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.

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

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.

4.2.8 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 4.7 **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.

EXAMPLE 4.8 Search book availability in library

R.1: Search book

Description Once the user selects the search option, he would be asked to enter the keywords. The system would search the book in the book list based on the key words entered. After making the search, the system should output the details of all books whose title or author name match any of the key words entered. The book details to be displayed include: title, author name, publisher name, year of publication, ISBN number, catalog number, and the location in the library.

R.1.1: Select search option

Input: "Search" option

Output: User prompted to enter the key words

R.1.2: Search and display

Input: Key words

Output: Details of all books whose title or author name matches any of the key words entered by the user. The book details displayed would include—title of the book, author name, ISBN number, catalog number, year of publication, number of copies available, and the location in the library.

Processing: Search the book list based on the key words:

R.2: Renew book

Description: When the "renew" option is selected, the user is asked to enter his membership number and password. After password validation, the list of the books borrowed by him are displayed. The user can renew any of his borrowed books by indicating them. A requested book cannot be renewed if it is reserved by another user. In this case, an error message would be displayed.

R.2.1: Select renew option

State: The user has logged in and the main menu has been displayed.

Input: "Renew" option selection.

Output: Prompt message to the user to enter his membership number and password.

R.2.2: Login

State: The renew option has been selected.

Input: Membership number and password.

Output: List of the books borrowed by the user is displayed, and user is prompted to select the books to be renewed, if the password is valid. If the password is invalid, the user is asked to re-enter the password.

Processing: Password validation, search the books issued to the user from the borrower's list and display.

Next function: R.2.3 if password is valid and R.2.2 if password is invalid.

R.2.3: Renew selected books

Input: User choice for books to be renewed out of the books borrowed by him.

Output: Confirmation of the books successfully renewed and apology message for the books that could not be renewed.

Processing: Check if anyone has reserved any of the requested books. Renew the books selected by the user in the borrower's list, if no one has reserved those books.

In order to properly identify the high-level requirements, a lot of common sense and the ability to visualise 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 organise 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.

4.2.9 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 subrequirements for the requirement R.1 have been numbered R.1.1, R.1.2, etc.

4.2.10 Organisation of the SRS Document

In this section, we discuss the organisation 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, organisation 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 organisation 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 organisation 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 organisation 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 in Sections 2 and 4 respectively.

Performance requirements: Aspects such as number of transaction 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 organised 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

Specification of the behaviour may not be necessary for all systems. It is usually necessary for those systems in which the system behaviour depends on the state in which the system is, and the system transits among a set of states depending on some prespecified conditions and events. The behaviour of a system can be specified using either the *finite state machine* (FSM) formalism or any other alternate formalisms. The FSMs can used to specify the possible states (modes) of the system and the transition among these states due to occurrence of events.

EXAMPLE 4.9 **(Personal library software):** It is proposed to develop a software that would be used by individuals to manage their personal collection of books. The following is an informal description of the requirements of this software as worked out by the marketing department. Develop the functional and non-functional requirements for the software.

A person can have up to a few hundreds of books. The details of all the books such as name of the book, year of publication, date of purchase, price, and publisher would be entered by the owner. A book should be assigned a unique serial number by the computer. This number would be written by the owner using a pen on the inside page of the book. Only a registered friend can be lent a book. While registering a friend, the following data would have to be supplied—name of the friend, his address, land line number, and mobile number. Whenever a book issue request is given, the name of the friend to whom the book is to be issued and the unique id of the book is entered. At this, the various books outstanding against the borrower along with the date borrowed are displayed for information of the owner. If the owner wishes to go ahead with the issue of the book, then the date of issue, the title of the book, and the unique identification number of the book are stored. When a friend returns a book, the date of return is stored and the book is removed from his borrowing list. Upon query, the software should display the name, address, and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued. The software should allow the owner to update the details of a friend such as his address, phone, telephone number, etc. It should be possible for the owner to delete all the data pertaining to a friend who is no more active in using the library. The records should be

stored using a free (public domain) data base management system. The software should run on both Windows and UNIX machines.

Whenever the owner of the library software borrows a book from his friends, would enter the details regarding the title of the book, and the date borrowed and the friend from whom he borrowed it. Similarly, the return details of books would be entered. The software should be able to display all the books borrowed from various friends upon request by the owner.

It should be possible for anyone to query about the availability of a particular book through a web browser from any location. The owner should be able to query the total number of books in the personal library, and the total amount he has invested in his library. It should also be possible for him to view the number of books borrowed and returned by any (or all) friend(s) over any specified time.

Functional requirements

The software needs to support three categories of functionalities as described below:

1. Manage own books

1.1 Register book

Description: To register a book in the personal library, the details of a book, such as name, year of publication, date of purchase, price and publisher are entered. This is stored in the database and a unique serial number is generated.

Input: Book details

Output: Unique serial number

R.1.2: Issue book

Description: A friend can be issued book only if he is registered. The various books outstanding against him along with the date borrowed are first displayed.

R.1.2.1: Display outstanding books

Description: First a friend's name and the serial number of the book to be issued are entered. Then the books outstanding against the friend should be displayed.

Input: Friend name

Output: List of outstanding books along with the date on which each was borrowed.

R.1.2.2: Confirm issue book

If the owner confirms, then the book should be issued to him and the relevant records should be updated.

Input: Owner confirmation for book issue.

Output: Confirmation of book issue.

R.1.3: Query outstanding books

Description: Details of friends who have books outstanding against their name is displayed.

Input: User selection

Output: The display includes the name, address and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued.

R.1.4: Query book

Description: Any user should be able to query a particular book from anywhere using a web browser.

Input: Name of the book.

Output: Availability of the book and whether the book is issued out.

R.1.5: Return book

Description: Upon return of a book by a friend, the date of return is stored and the book is removed from the borrowing list of the concerned friend.

Input: Name of the book.

Output: Confirmation message.

2. Manage friend details

R.2.1: Register friend

Description: A friend must be registered before he can be issued books. After the registration data is entered correctly, the data should be stored and a confirmation message should be displayed.

Input: Friend details including name of the friend, address, land line number and mobile number.

Output: Confirmation of registration status.

R.2.2: Update friend details

Description: When a friend's registration information changes, the same must be updated in the computer.

R.2.2.1: Display current details

Input: Friend name.

Output: Currently stored details.

R2.2.2: Update friend details

Input: Changes needed.

Output: Updated details with confirmation of the changes.

R.3.3: Delete a friend record

Description: Delete records of inactive members.

Input: Friend name.

Output: Confirmation message.

3. Manage borrowed books

R.3.1: Register borrowed books

Description: The books borrowed by the user of the personal library are registered.

Input: Title of the book and the date borrowed.

Output: Confirmation of the registration status.

R.3.2: Deregister borrowed books
Description: A borrowed book is deregistered when it is returned.
Input: Book name.
Output: Confirmation of deregistration.

R.3.3: Display borrowed books
Description: The data about the books borrowed by the owner are displayed.
Input: User selection.
Output: List of books borrowed from other friends.

4. Manage statistics

R.4.1: Display book count
Description: The total number of books in the personal library should be displayed.
Input: User selection.
Output: Count of books.

R.4.2: Display amount invested
Description: The total amount invested in the personal library is displayed.
Input: User selection.
Output: Total amount invested.

R.4.3: Display number of transactions Description: The total numbers of books issued and returned over a specific period by one (or all) friend(s) is displayed.
Input: Start of period and end of period.
Output: Total number of books issued and total number of books returned.

Non-functional requirements

N.1: Database: A data base management system that is available free of cost in the public domain should be used.

N.2: Platform: Both Windows and UNIX versions of the software need to be developed.

N.3: Web-support: It should be possible to invoke the query book functionality from any place by using a web browser.

Observation: Since there are many functional requirements, the requirements have been organised into four sections: Manage own books, manage friends, manage borrowed books, and manage statistics. Now each section has less than 7 functional requirements. This would not only enhance the readability of the document, but would also help in design.

4.2.11 Techniques for Representing Complex Logic

A good SRS document should properly characterise the conditions under which different scenarios of interaction occur (see Section 4.2.5). That is, a high-level function might involve different steps to be undertaken as a consequence of some decisions made after each step. Sometimes the conditions can be complex and numerous and several alternative interaction

and processing sequences may exist depending on the outcome of the corresponding condition checking. A simple text description in such cases can be difficult to comprehend and analyse. In such situations, a decision tree or a decision table can be used to represent the logic and the processing involved. Also, when the decision making in a functional requirement has been represented as a decision table, it becomes easy to automatically or at least manually design test cases for it. However, use of decision trees or tables would be superfluous in cases where the number of alternatives are few, or the decision logic is straightforward. In such cases, a simple text description would suffice.

There are two main techniques available to analyse and represent complex processing logic—decision trees and decision tables. Once the decision-making logic is captured in the form of trees or tables, the test cases to validate these logic can be automatically obtained. It should, however, be noted that decision trees and decision tables have much broader applicability than just specifying complex processing logic in an SRS document. For instance, decision trees and decision tables find applications in information theory and switching theory.

Decision tree

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. Decision tables specify which variables are to be tested, and based on this what actions are to be taken depending upon the outcome of the decision-making logic, and the order in which decision making is performed.

The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the conditions. Instead of discussing how to draw a decision tree for a given processing logic, we shall explain through a simple example how to represent the processing logic in the form of a decision tree.

EXAMPLE 4.10 A library membership management software (LMS) should support the following three options—new member, renewal, and cancel membership. When the *new member* option is selected, the software should ask the member's name, address, and phone number. If proper information is entered, the software should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable. If the *renewal* option is chosen, the LMS should ask the member's name and his membership number and check whether he is a valid member. If the member details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership charge payable by the member should be printed. If the membership details entered are invalid, an error message should be displayed. If the *cancel membership* option is selected and the name of a valid member is entered, then the membership is cancelled, a choke for the balance amount due to the member is printed and his membership record is deleted. The decision tree representation for this problem is shown in Figure 4.3.

Observe from Figure 4.3 that the internal nodes represent conditions, the edges of the tree correspond to the outcome of the corresponding conditions. The leaf nodes represent the actions to be performed by the system. In the decision tree of Figure 4.3, first the user selection is checked. Based on whether the selection is valid, either further

condition checking is undertaken or an error message is displayed. Observe that the order of condition checking is explicitly represented.

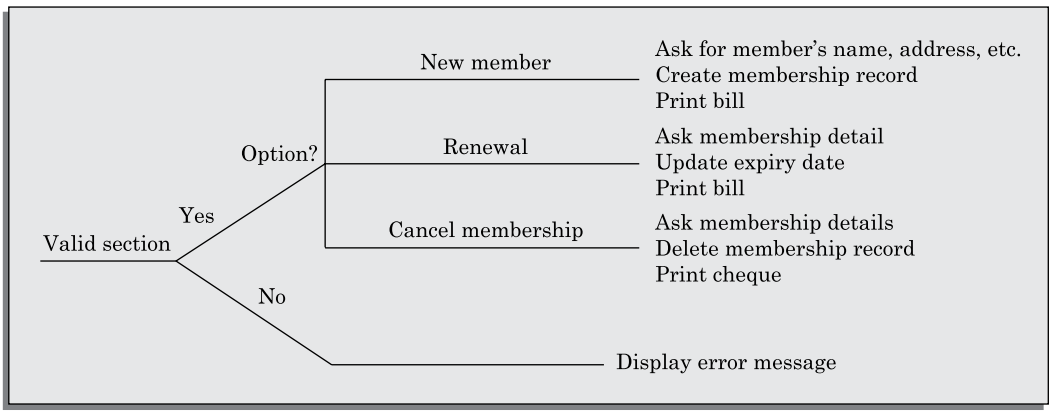


FIGURE 4.3 Decision Tree for LMS.

Decision table

A decision table shows the decision-making logic and the corresponding actions taken in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an evaluation test is satisfied. A column in the table is called a *rule*. A rule implies that if a certain condition combination is true, then the corresponding action is executed. The decision table for the LMS problem of Example 4.10 is as shown in Table 4.1.

TABLE 4.1 Decision Table for the LMS Problem

Conditions				
Valid selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				
Display error message	×			
Ask member's name, etc.		×		
Build customer record		×		
Generate bill		×	×	
Ask membership details			×	×
Update expiry date			×	
Print cheque				×
Delete record				×

Decision table *versus* decision tree

Even though both decision tables and decision trees can be used to represent complex program logic, they can be distinguishable on the following three considerations:

Readability: Decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit.

Explicit representation of the order of decision making: In contrast to the decision trees, the order of decision making is abstracted out in decision tables. A situation where decision tree is more useful is when multilevel decision making is required. Decision trees can more intuitively represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Representing complex decision logic: Decision trees become very complex to understand when the number of conditions and actions increase. It may even be to draw the tree on a single page. When very large number of decisions are involved, the decision table representation may be preferred.

4.3 FORMAL SYSTEM SPECIFICATION

In recent years, formal techniques[3] have emerged as a central issue in software engineering. This is not accidental; the importance of precise specification, modelling, and verification is recognised to be important in most engineering disciplines. Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. We say a system is correctly implemented when it satisfies its given specification. The specification of a system can be given either as a list of its desirable properties (property-oriented approach) or as an abstract model of the system (model-oriented approach). These two approaches are discussed here. Before discussing representative examples of these two types of formal specification techniques, we first discuss a few basic concepts in formal specification. We will first highlight some important concepts in formal methods, and examine the merits and demerits of using formal techniques.

4.3.1 What is a Formal Technique?

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by its specification language. More precisely, a formal specification language consists of two sets—syn and sem, and a relation sat between them. The set *syn* is called the *syntactic domain*, the set *sem* is called the *semantic domain*, and the relation *sat* is called the *satisfaction relation*. For a given specification *syn*, and model of the system *sem*, if *sat (syn, sem)*, then *syn* is said to be the *specification of sem*, and *sem* is said to be the *specificand* of *syn*.

The generally accepted paradigm for system development is through a hierarchy of abstractions. Each stage in this hierarchy is an implementation of its preceding stage and a specification of the succeeding stage. The different stages in this system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every

[3] Sections 4.3–4.5 can be omitted in a first level course on software engineering.

stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.

Syntactic domains

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

Semantic domains

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

Satisfaction relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined—those that *preserve* a system's behaviour and those that *preserve* a system's structure.

Model *versus* property-oriented methods

Formal methods are usually classified into two broad categories—the so-called *model-oriented* and the property-oriented approaches. In a *model-oriented* style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the *property-oriented* style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy. Let us consider a simple producer/consumer example. In a *property-oriented* style, we would probably start by listing the properties of the system like—the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. Two examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In *a model-oriented* style, we would start by defining the basic operations, p (produce) and c (consume). Then we can state that $S1 + p \Rightarrow S$, $S + c \Rightarrow S1$. Thus model-oriented approaches essentially specify a program by writing another, presumably simpler program. A few notable examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.

It is alleged that property-oriented approaches are more suitable for *requirements specification*, and that the model-oriented approaches are more suited to *system design*

specification. The reason for this distinction is the fact that property-oriented approaches specify a system behaviour not by what they say of the system but by what they do not say of the system. Thus, property-oriented specifications permit a large number of possible implementations. Furthermore, property-oriented approaches specify a system by a conjunction of axioms, thereby making it easier to alter/augment specifications at a later stage. On the other hand, model-oriented methods do not support logical conjunctions and disjunctions, and thus even minor changes to a specification may lead to overhauling an entire specification. Since the initial customer requirements undergo several changes as the development proceeds, the property-oriented style is generally preferred for requirements specification. Later in this chapter, we have discussed two property-oriented specification techniques.

4.3.2 Operational Semantics

Informally, the *operational semantics* of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a *single run* of the system and how the runs are grouped together to describe the *behaviour* of the system. In the following subsection we discuss some of the commonly used operational semantics.

Linear semantics: In this approach, a *run* of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the atomic actions. For example, a concurrent activity $a \mid\mid b$ is represented by the set of sequential activities $a; b$ and $b; a$. This is a simple but rather unnatural representation of concurrency. The behaviour of a system in this model consists of the set of all its runs. To make this model more realistic, usually *justice* and *fairness* restrictions are imposed on computations to exclude the unwanted interleavings.

Branching semantics: In this approach, the behaviour of a system is represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

Maximally parallel semantics: In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial order semantics: Under this view, the semantics ascribed to a system is a *structure of states* satisfying a partial order relation among the states (events). The partial order represents a *precedence ordering* among events, and constrains some events to occur only after some other events have occurred; while the occurrence of other events (called *concurrent events*) is considered to be incomparable. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

Merits and limitations of formal methods

In addition to facilitating precise formulation of specifications, formal methods possess several positive features, some of which are discussed as follows:

- Formal specifications encourage rigour. It is often the case that the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification. It is widely acknowledged that it is cost-effective to spend more efforts at the specification stage, otherwise, many flaws would go unnoticed only to be detected at the later stages of software development that would lead to iterative changes to occur in the development life cycle. According to an estimate, for large and complex systems like distributed real-time systems 80 per cent of project costs and most of the cost overruns result from the iterative changes required in a system development process due to inappropriate formulation of requirements specification. Thus, the additional effort required to construct a rigorous specification is well worth the trouble.
- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications. Informal specifications may be useful in understanding a system and its documentation, but they cannot serve as a basis of verification. Even carefully written specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to unverified programs.
- Formal methods have well-defined semantics. Therefore, ambiguity in specifications is automatically avoided when one formally specifies a system.
- The mathematical basis of the formal methods makes it possible for automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a "toy" working model of a system that can provide immediate feedback on the behaviour of the specified system, and is especially useful in checking the completeness of specifications.

It is clear that formal methods provide mathematically sound frameworks within which large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal methods suffer from several shortcomings, some of which are as following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check *absolute* correctness of systems using theorem proving techniques.
- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

It has been pointed out by several researchers that formal specifications neither replace nor make the informal descriptions obsolete but complement them. In fact, the comprehensibility of formal specifications is greatly enhanced when the specifications are accompanied by an informal description. What is suggested is the use of formal techniques as a broad guideline for the use of the informal techniques. An interesting example of such an approach is reported by Jones in 1980 [Jones, 1980]. In this approach, the use of a formal method identifies the necessary verification steps that need to be carried out, but it is legitimate to apply informal reasoning in presentation of correctness arguments and transformations. Any doubt or query relating to an informal argument is to be resolved by formal proofs.

In the following two sections, we discuss the axiomatic and algebraic specification styles. Both these techniques can be classified as the property-oriented specification techniques.

4.4 AXIOMATIC SPECIFICATION

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

How to develop an axiomatic specifications?

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

- Establish the range of input values over which the function should behave correctly. Establish the constraints on the input parameters as a predicate.
- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Combine all of the above into pre- and post-conditions of the function.

We now illustrate how simple abstract data types can be algebraically specified through two simple examples.

EXAMPLE 4.11 Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f(x : real) : real$$

$$pre : x \in R$$

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

EXAMPLE 4.12 Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$$search(X : intArray, key : integer) : integer$$
$$pre : \exists i \in [Xfirst...Xlast], X[i] = key$$
$$post : \{(X'[search(X, key)] = key) \wedge (X = X')\}$$

Please note that we have followed the convention that if a function changes any of its input parameters, and if that parameter is named X, then we refer to it after the function completes execution as X'. One practical application of the axiomatic specification is in program documentation. Engineers developing code for a function specify the function by noting down the pre- and post-conditions of the function in the function header. Another application of the axiomatic specifications is in proving program properties by composing the pre- and post-conditions of a number of functions.

4.5 ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980, 1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Essentially, algebraic specifications define a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g., {I, +, -, *, /}. In contrast, alphabetic strings S together with operations of concatenation and length $\{S, I, \text{con}, \text{len}\}$, is not a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in a heterogeneous algebra is called a *sort* of the algebra. To define a heterogeneous algebra, besides defining the sorts, we need to specify the involved operations, their signatures, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedures by using *equations*. An algebraic specification is usually presented in four sections.

Types section: In this section, the sorts (or the data types) being used is specified.

Exception section: This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section: This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the *signature* of the operator. For example, PUSH takes a stack and an element as its input and returns a new stack that has been created.

Equations section: This section gives a set of *rewrite rules* (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

By convention each equation is implicitly universally quantified over all possible values of the variables. This means that the equation holds for all possible values of the variable. Names not mentioned in the syntax section such *r* or *e* are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

Basic construction operators: These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, 'create' and 'append' are basic construction operators in Example 4.13.

Extra construction operators: These are the construction operators other than the basic construction operators. For example, the operator 'remove' in Example 4.13 is an extra construction operator, because even without using 'remove' it is possible to generate all values of the type being specified.

Basic inspection operators: These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let *S* be the set of operators whose range is not the data type being specified—these are the inspection operators. The set of the basic operators S1 is a subset of *S*, such that each operator from *S*-*S*1 can be expressed in terms of the operators from *S*1.

Extra inspection operators: These are the inspection operators that are not basic inspectors.

A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in Example 4.13, create is a constructor because point appears on the right hand side of the expression and point is the data type being specified. But, xcoord is an inspection operator since it does not modify the point type.

A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor (basic and extra) and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are $m1$ basic constructors, $m2$ extra constructors, $n1$ basic inspectors, and $n2$ extra inspectors, we should have $m1 \times (m2 + n1) + n2$ axioms. However, it should be clearly noted that these $m1 \times (m2 + n1) + n2$ axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

While developing the rewrite rules, different persons can come up with different sets of equations. However, while developing the equations one has to be careful that the equations should be able to handle all meaningful composition of operators, and they should have the unique termination and finite termination properties. These two properties of the rewrite rules are discussed later in this section.

EXAMPLE 4.13 Let us specify a data type point supporting the operations create, xcoord, ycoord, isequal; where the operations have their usual meaning.

```
Types:
    defines point
    uses boolean, integer
Syntax:
```

1. $create : integer \times integer \rightarrow point$
2. $xcoord : point \rightarrow integer$
3. $ycoord : point \rightarrow integer$
4. $isequal : point \times point \rightarrow boolean$

```
Equations:
```

1. $xcoord(create(x, y)) = x$
2. $ycoord(create(x, y)) = y$
3. $isequal(create(x1, y1), create(x2, y2)) = ((x1 = x2) and (y1 = y2))$

In this example, we have only one basic constructor (create), and three basic inspectors (xcoord, ycoord, and isequal). Therefore, we have only 3 equations.

The rewrite rules let you determine the meaning of any sequence of calls on the point type. Consider the following expression: *isequal (create (xcoord (create(2, 3)), 5),create (ycoord (create(2, 3)), 5))*. By applying the rewrite rule 1, you can simplify the given expression as *isequal (create (2, 5), create (ycoord (create(2, 3)), 5))*. By using rewrite rule 2, you can further simplify this as *isequal (create (2, 5), create (3, 5))*. This is false by rewrite rule 3.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

Completeness: This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. When the equations are not complete, at some step during the reduction process, we might not be able to reduce the expression arrived at that step by using any of the equations. There is no simple procedure to ensure that an algebraic specification is complete.

Finite termination property: This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.

Unique termination property: This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked—Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique termination property is a very difficult problem.

EXAMPLE 4.14 Let us specify a FIFO queue supporting the operations create, append, remove, first, and isempty; where the operations have their usual meaning.

```
Types:
          defines queue
          uses boolean, element
```

```
Exception:
          underflow, novalue
```

Syntax:

1. *create* : $\phi \rightarrow$ *queue*
2. *append* : *queue* $\times$ *element* $\rightarrow$ *queue*
3. *remove* : *queue* $\rightarrow$ *queue* + {*underflow*}
4. *first* : *queue* $\rightarrow$ *element* + {*novalue*}
5. *isempty* : *queue* $\rightarrow$ *boolean*

Equations:

1. *isempty*(*create*()) = *true*
2. *isempty*(*append*(*q*, *e*)) = *false*
3. *first*(*create*()) = *novalue*
4. *first*(*append*(*q*, *e*)) = *if isempty*(*q*) *then e else first*(*q*)
5. *remove*(*create*()) = *underflow*
6. *remove*(*append*(*q*, *e*)) = *if isempty*(*q*) *then create*() *else append*(*remove*(*q*), *e*)

In this example, we have two basic construction operators (create and append). We have one extra construction operator (remove). We have considered remove to be an extra construction operator because all values of the queue can be realised, even without having the remove operator. We have two basic inspectors (first and `isempty`). Therefore, we have 2 $\times$ 3 = 6 equations.

4.5.1 Auxiliary Functions

Sometimes while specifying a system, one needs to introduce extra functions not part of the system to define the meaning of some interface procedures. These are called *auxiliary functions*. In the following, we discuss an example where it becomes necessary to use an auxiliary function to be able to specify a system.

EXAMPLE 4.15 Let us specify a bounded FIFO queue having a maximum size of MaxSize and supporting the operations create, append, remove, first, and `isempty`; where the operations have their usual meaning.

```
Types:
       defines  queue
       uses  boolean,  element,  integer

Exception:
       underflow,  novalue,  overflow
```

Syntax:

1. *create* : $\phi \rightarrow$ *queue*
2. *append* : *queue* $\times$ *element* $\rightarrow$ *queue* + {*overflow*}
3. *size* : *queue* $\rightarrow$ *integer*
4. *remove* : *queue* $\rightarrow$ *queue* + {*underflow*}
5. *first* : *queue* $\rightarrow$ *element* + {*novalue*}
6. *isempty* : *queue* $\rightarrow$ *boolean*

Equations:

1. *first*(*create*()) = *novalue*
2. *first*(*append*(*q*,*e*)) = *if size*(*q*)= *MaxSize then overflow else if isempty*(*q*) *then e else first*(*q*)
3. *remove*(*create*()) = *underflow*
4. *remove*(*append*(*q*, *e*)) = *if isempty*(*q*) *then create*() *else*
   .
   if size(*q*) = *MaxSize then overflow else append*(*remove*(*q*), *e*)
5. *size*(*create*()) = 0
6. *size*(*append*(*q*, *e*)) = *if size*(*q*) = *MaxSize then overflow else size*(*q*) + 1
7. *isempty*(*q*) = *if* (*size*(*q*) = 0) *then true else false*

In this example, we have used the auxiliary function `size` to enable us to specify that during appending an element, overflow might occur if the queue size exceeds `MaxSize`. However, after we have introduced the auxiliary function `size`, we find that the operator `isempty` can no longer be considered as a basic inspector because `isempty` can be expressed in terms of `size`. Therefore, we have removed the axioms for the operator `isempty` used in Example 4.15, and have instead used an axiom to express `isempty` in terms of `size`. We have added two axioms to express size in terms of the basic construction operators (`create` and `append`).

4.5.2 Structured Specification

Developing algebraic specifications is time consuming. Therefore efforts have been made to devise ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

Incremental specification: The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.

Specification instantiation: This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Pros and Cons of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be analysed. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to integrate with typical programming languages. Also, algebraic specifications are hard to understand.

4.6 EXECUTABLE SPECIFICATION AND 4GL

When the specification of a system is expressed formally or is described by using a programming language, then it becomes possible to directly execute the specification without having to design and write code for implementation. However, executable

specifications are usually slow and inefficient, 4GLs[4] (4th Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of large granularity commonality across data processing applications which have been identified and mapped to program code. 4GLs get their power from software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in 3GLs results in up to 50 per cent lower memory usage and also the program execution time can reduce up to tenfolds.

SUMMARY

- Substantial time and effort must be spent in developing a good quality SRS document before starting the design activity. Any improper specification turns out to be very expensive adversely affects all subsequent phases of development.
- The requirements analysis and specification phase consists of two important activities—requirements gathering and analysis, and requirements specification.
- The aim of requirements analysis is to clearly understand the exact user requirements and to remove any inconsistencies, anomalies, and incompleteness in these requirements.
- During the requirements specification activity, the requirements are systematically organised into an SRS document.
- Formally specifying the requirements has many advantages. But, a major shortcoming of the formal specification techniques is that they are hard to use. However, it is possible that formal techniques will become more usable in future with the development of suitable front-ends. We discussed the axiomatic and algebraic techniques as example formal specification techniques to give an idea of some of the issues involved in formal specification.

EXERCISES

MULTIPLE CHOICE QUESTIONS

For each of the following questions only one of the options is correct. Choose the correct option:

1. Who among the following is a stakeholder in a software development project?
(a) A shareholder of the organisation developing the software
(b) Anyone who is interested in the software
(c) Anyone who is a source of requirements for the software
(d) Anyone who might be affected by the software

[4] Programming languages are generally classified into four generations. The first generation (1GL) programming languages consist of machine language programs. The second generation (2GL) started when the assembly language was introduced. All procedural languages are classified as 3GLs. In procedural languages, in order to solve a problem, you would have to precisely write down "how" the required result can be obtained. This requires writing the exact procedures or the algorithmic steps that need to be followed to arrive at the result. In contrast, using a 4GL only the "what" parts have to be specified.

2. A software requirements specification (SRS) document should avoid discussing which one of the following:
   (a) Functional requirements
   (b) Non-functional requirements
   (c) Design specification
   (d) Constraints on the implementation
3. Which one of the following is not a goal of requirements analysis?
   (a) Weed out ambiguities in the requirements
   (b) Weed out inconsistencies in the requirements
   (c) Weed out non-functional requirements
   (d) Weed out incompleteness in the requirements
4. Consider the following requirement for a word processor software: "The software should provide facility to import an existing image available as a jpeg file into the document being created." Which one of the following types of requirements is this?
   (a) Functional requirement
   (b) Non-functional requirement
   (c) Constraint on the implementation
   (d) Goal of implementation
5. Which one of the following assertions is FALSE about functional requirements?
   (a) A functional requirement is also called a behavioral requirement
   (b) A functional requirement may consist of several scenarios of operation
   (c) A functional requirement is a statement of how a software product must map program inputs to program outputs
   (d) Some of the functional requirements specify the system performance
6. Consider the following partial description of the IIT security software. "A camera at the main gate should detect an incoming vehicle, and the gate should be opened for registered vehicles." Which one of the following types of requirements is this?
   (a) Functional requirement
   (b) Non-functional requirement
   (c) Design requirement
   (d) Design constraint
7. Consider the following partial requirement for a software: "A web browser should be used as the front end." It can be considered to be which one of the following types of requirements:
   (a) Functional requirement
   (b) Non-functional requirement
   (c) Constraint
   (d) External interface

TRUE OR FALSE

State whether the following statements are **TRUE** or **FALSE**. Give reasons for your answer.

1. Applications developed using 4GLs would normally be more efficient and run faster compared to applications developed using 3GL.

2. A formal specification cannot be ambiguous.
3. A formal specification cannot be incomplete.
4. A formal specification cannot be inconsistent.
5. The system test plan can be prepared immediately after the completion of the requirements specification phase.
6. The SRS document is a formal specification of a system.
7. User interface issues of a system are usually its functional requirements.
8. The SRS document is written using the customer's terminology of various data and procedures in the problem, rather than the development team's terminology.
9. A precise specification cannot be incomplete.
10. If a requirement specification is precise, then it would automatically imply that it is an unambiguous requirements specification.

REVIEW QUESTIONS

1. What is the difference between requirements analysis and specification? What are the important activities that are carried out during requirements analysis and specification phase? What is the final outcome of the requirements analysis and specification phase?
2. What are the goals of the requirements analysis and specification phase? How are the requirements analysis and specification activities carried out and who carries out these activities?
3. Discuss the important ways in which a good SRS document can be useful to various stakeholders.
4. What is the difference between the functional and the non-functional requirements of a system? Identify at least two functional requirements of a bank automated teller machine (ATM) system. Also identify one non-functional requirement for an ATM system.
5. What are the four types of non-functional requirements that have been suggested by IEEE 830 standards document? Give one example of each of these categories of requirements.
6. What do you understand by requirements gathering? Name and explain the different requirements gathering techniques that are normally deployed by an analyst.
7. What are the different types of requirements problems that an analyst usually anticipates and rectifies in the gathered requirements before starting to write the SRS document? Give at least one example of each type of requirements problems.
8. Explain the likely consequences of starting a large project development effort without accurately understanding and documenting the customer requirements.
9. Suppose you have been appointed as the analyst for a large software development project. Discuss the aspects of the software product you would document in the software requirements specification (SRS) document? What would be the organisation of your SRS document? How would you validate your SRS document?

10. Make a checklist of various types of errors that might exist in an SRS document, so that this checklist can be used to review an SRS document.

11. Write down the important users of the SRS document for a project, the specific ways in which they use the document, and their specific expectations from the document, if any.

12. What do you understand by the problems of over specification, forward reference, and noise in an SRS document? Explain each of these with suitable examples.

13. What is the difference between functional and non-functional requirements? Give one example of each type of requirement for a library automation software.

14. List five desirable characteristics of a good software requirements specification (SRS) document.

15. Suppose you are trying to gather the requirements for a software that needs to be developed to automate the book-keeping activities of a supermarket. Identify the main tasks that you would undertake as the analyst to satisfactorily gather the requirements.

16. How are the abstraction and decomposition principles used in the development of a good software requirements specification?

17. Suppose the analyst of a large product development effort has prepared the SRS document in the form of a narrative essay of the system to be developed. Based on this document, the product development activity gets underway. Explain the problems that such a requirements specification document may create while developing the software.

18. Discuss the relative advantages of formal and informal requirements specifications.

19. Why is the SRS document also known as the black-box specification of a system?

20. Who are the different category of users of the SRS document? In what ways is the SRS document useful to them?

21. Give an example of an inconsistent functional requirement. Explain why you think that the requirement is inconsistent.

22. What do you understand by traceability in the context of software requirements specification? How is traceability achieved? Identify at least two important benefits of having traceability among development artifacts.

23. (a) What are the important differences between a model-oriented specification method and a property-oriented specification method?
(b) Compare the relative advantages of property-oriented specification methods over model-oriented specification methods.
(c) Name at least one representative popular property-oriented specification technique, and one representative model-oriented specification technique.

24. Consider the following requirement for a software to be developed for controlling a chemical plant. The chemical plant has a number of emergency conditions. When any of the emergency conditions occurs, some prespecified actions should be taken. The different emergency conditions and the corresponding actions that need to be taken are as follows:

(a) If the temperature of the chemical plant exceeds T_1°C, then the water shower should be turned ON and the heater should be turned OFF.
(b) If the temperature of the chemical tank falls below T_2°C, then the heater should be turned ON and the water shower should be turned OFF.
(c) If the pressure of the chemical plant is above P_1, then the valve v_1 should be OPENED.
(d) If the chemical concentration of the tank rises above M , and the temperature of the tank is more than T_3°C, then the water shower should be turned ON.
(e) If the pressure rises above P_3 and the temperature rises above T_1°C, then the water shower should be turned ON, valves v_1 and v_2 are OPENED and the alarm bells sounded.

Write the requirements of this chemical plant software in the form of a decision table.

25. Draw a decision tree to represent the processing logic of the chemical plant controller described in question 24.

26. Represent the decision making involved in the operation of the following wash-machine by means of a decision table:

The machine waits for the `start` switch to be pressed. After the user presses the `start` switch, the machine fills the wash tub with either hot or cold water depending upon the setting of the `HotWash` switch. The water filling continues until the high level is sensed. The machine starts the agitation motor and continues agitating the wash tub until either the preset timer expires or the user presses the `stop` switch. After the agitation stops, the machine waits for the user to press the `startDrying` switch. After the user presses the `startDrying` switch, the machine starts the hot air blower and continues blowing hot air into the drying chamber until either the user presses the `stop` switch or the preset timer expires.

27. Represent the processing logic of the following problem in the form of a decision table: A Library Membership Automation System needs to support three functions: **add new-member, renew-membership, cancel-membership**. If the user requests for any function other than these three, then an error message is flashed. When an **add new-member** request is made, a new member record is created and a bill for the annual membership fee for the new member is generated. If a membership renewal request is made, then the expiry date of the concerned membership record is updated and a bill towards the annual membership fee is generated. If a membership cancellation request is made, then the concerned membership record is deleted and a cheque for the balance amount due to the member is printed.

28. What do you understand by pre- and post-conditions of a function? Write the pre- and post-conditions to axiomatically specify the following functions:

(a) A function takes two floating point numbers representing the sides of a rectangle as input and returns the area of the corresponding rectangle as output.
(b) A function accepts three integers in the range of –100 and +100 and determines the largest of the three integers.
(c) A function takes an array of integers as input and finds the minimum value.

(d) A function named square-array creates a 10 element array where each all elements of the array, the value of any array element is square of its index.

(e) A function sort takes an integer array as its argument and sorts the input array in ascending order.

29. Using the algebraic specification method, formally specify a **string** supporting the following operations:

- **append:** append a given string to another string
- **add:** add a character to a string
- **create:** create a new null string
- **isequal:** checks whether two strings are equal or not
- **isempty:** checks whether the string is null

30. Using the algebraic specification method, formally specify an **array** of generic type elem. Assume that array supports the following operations:

- **create:** takes the array bounds as parameters and initializes the values of the array to undefined.
- **assign:** creates a new array, where a particular element has been assigned a value.
- **eval:** reveals the value of a specified element.
- **first:** returns the first bound of the array.
- **last:** returns the last bound of the array.

31. What do you understand by an executable specification language? How is it different from a traditional procedural programming language? Give an example of an executable specification language.

32. What is a fourth generation programming technique? What are its advantages and disadvantages vis-a-vis a third generation technique?

33. What are auxiliary functions in algebraic specifications? Why are these needed?

34. What do you understand by incremental development of algebraic specifications? What is the advantage of incremental development of algebraic specifications?

35. (a) Algebraically specify an abstract data type that stores a **set** of elements and supports the following operations. Assume that the ADT element has already been specified and you can use it:

- **new:** creates a null set.
- **add:** takes a set and an element and returns the set with the additional elements stored.
- **size:** takes a set as argument and returns the number of elements in the set.
- **remove:** takes a set and an element as its argument and returns the set with the element removed.
- **contains:** takes a set and an element as its argument and returns the Boolean value true if the element belongs to the set and returns false if the element does not belong to the set.
- **equals:** takes two sets as arguments and returns true if they contain identical elements and returns false otherwise.

(b) Using the specification you have developed for the ADT set, reduce the following expression by applying the rewrite rules: equals (add(5, (add(6, new()), add(6, (add(5, new())))). Show the details of every reduction.

36. Algebraically specify a data type `Point`, that supports the following operations: `create`, `xcoord`, `ycoord`, `move`, `movex`, `movey`. The informal meanings of these operations are the following—`create` takes two integers as its arguments and creates an instance of point type that has the two integers as its x and y coordinate values respectively, `xcoord` and `ycoord` return the x and y-coordinates of a given point, `move` takes a point and two integer values as its argument and sets the x and y-coordinates of point to the specified values, `movex` takes a point and an integer value as its argument and sets the x-coordinate of the point to the given integer value. Similarly, `movey` takes a point and an integer value and sets the y-coordinate of point to the given integer value.

Reduce the following expression, clearly showing each step and mentioning the reduction rule used.

```
xcoord(movex(create(20,100), ycoord(create(10,50)))
```

37. Write a formal algebraic specification of the sort **symbol-table** whose operations are informally defined as follows:

- **create:** bring a symbol table into existence.
- **enter:** enter a symbol table and its type into the table.
- **lookup:** return the type associated with a name in the table.
- **delete:** remove a name, type pair from the table, given a name as a parameter.
- **replace:** replace the type associated with a given name with the type specified as its parameter.

The enter operation fails if the name is already present in the table. The lookup, delete, and replace operations fail if the name is not available in the table.

38. Algebraically specify the data type **queue** which supports the following operations:

- **create:** creates an empty queue.
- **append:** takes a queue and an item as its arguments and returns a queue with the item added at the end of the queue.
- **remove:** takes a queue as its argument and returns a queue with the first element of the original queue removed.
- **inspect:** takes a queue as its argument and returns the value of the first item in the queue.
- **isempty:** takes a queue as its argument and returns true if the queue contains no elements, and returns false if it contains one or more elements.

You can assume that the data type item has previously been specified and that you can reuse this specification.

39. Define the finite termination and unique termination properties of algebraic specifications. Why is it necessary for an algebraic specification to satisfy these properties?

40. If the prototyping model is being used in a development effort, is it necessary to develop a requirements specification document?

41. Express the decision making involved in the following withdraw cash function of a bank ATM using a decision table.

To withdraw cash, first a valid customer identification is required. For this, the customer is prompted to insert his ATM card in the card checking machine. If his card is found to be invalid, the card is ejected out along with an appropriate message displayed. If the card is verified to be a valid card, the customer is prompted to type his password. If the password is invalid, an error message is shown and the customer is prompted to enter his password again. If the customer enters incorrect password consecutively for three times, then his card is seized and he is asked to contact the bank manager. On the other hand, if the customer enters his password correctly, then he is considered to have validly identified himself and is prompted to enter the amount he needs to withdraw. If he enters an amount that is not a multiple of ₹100, he is prompted to enter the amount again. After he enters an amount that is a multiple of ₹100, the cash is dispensed if sufficient amount is available in his account and his card is ejected; otherwise his card is ejected out without any cash being dispensed along with a message display regarding insufficient fund position in his account.

42. Identify the functional and non-functional requirements in the following problem description and document them.

A cosmopolitan clock software is to be developed that displays up to 6 clocks with the names of the city and their local times. The clocks should be aesthetically designed. The software should allow the user to change name of any city and change the time readings of any clock by typing *c* (for configure) on any clock. The user should also be able to toggle between a digital clock and an analog clock display by typing either *d* (for digital) or *a* (for analog) on a clock display. After the stand-alone implementation works, a web-version should be developed that can be downloaded on a browser as an applet and run. The clock should use only the idle cycles on the computer it runs.

43. What do you understand by inconsistencies, anomalies, and incompletenesses in an SRS document? Identify the inconsistencies, anomalies, and incompletenesses in the following requirements of an academic activity automation software of an educational institute:

"The semester performance of each student is computed as the average academic performance for the semester. The guardians of all students having poor performance record in the semester are mailed a letter informing about the poor performance of the ward and intimating that repetition of poor performance in the subsequent semester can lead to expulsion. The extracurricular activities of a student are also graded and taken into consideration for determination of the semester performance."

44. Identify any inconsistencies, anomalies, and incompleteness that are present in the following requirements that were gathered by interviewing the clerks of the CSE department for developing an academic automation software (AAS): "The CGPA of each student is computed as the average performance for the semester. The parents of all students having poor performance are mailed a letter informing about the poor performance of their ward and with a request to convey a warning to the student that the poor performance should not be repeated."

45. Represent the decision making involved in the following functional requirement of a library automation system: `Issue Item`: An item when submitted at the counter along with the library identity card, first it is determined if the member has exceeded his quota. If he has exceeded his quota, then no items can be issued to him. If the requested item is a journal, then it is issued for two days only. If it is a book, then it is checked whether it is a reference book. Reference books cannot be issued out. If it is not a reference book, it is determined if anyone has reserved it. Reserved books cannot be issued out. If the book issue request of the member meets all the mentioned criteria, then the book is issued to the member for one month, appropriate entry is made in the member's account and an issue slip is printed.

46. Suppose you wish to develop a word processing software that would have features similar to Microsoft Word. Develop the SRS document for this word processing software.

47. Identify at least three important reasons as to why the customer requirements may change after the requirements phase is complete and the SRS document has been signed off.

48. Suppose you are entrusted with the work of gathering the requirements for the hostel automation software that you have undertaken for an academic institute. Briefly discuss how you would carry out the requirements gathering and analysis activities.