PART 2

Perhaps the major problem that we face in developing large and complex software systems is that of requirements engineering. Requirements engineering is concerned with establishing what the system should do, its desired and essential emergent properties, and the constraints on system operation and the software development processes. You can therefore think of requirements engineering as the communications process between the software customers and users and the software developers.

Requirements engineering is not simply a technical process. The system requirements are influenced by users' likes, dislikes and prejudices, and by political and organisational issues. These are fundamental human characteristics, and new technologies, such as use-cases, scenarios and formal methods, don't help us much in resolving these thorny problems.

The chapters in this section fall into two classes—in Chapters 6 and 7 I introduce the basics of requirements engineering, and in Chapters 8 to 10 I describe models and techniques that are used in the requirements engineering process. More specifically:

1. The topic of Chapter 6 is software requirements and requirements documents. I discuss what is meant by a requirement, different types of requirements and how these requirements are organised into a requirements specification document. I introduce the second running case study—a library system—in this chapter.

2. In Chapter 7, I focus on the activities in the requirements engineering process. I discuss how feasibility studies should always be part of requirements engineering, techniques for requirements elicitation and analysis, and requirements validation. Because requirements inevitably change, I also cover the important topic of requirements management.

3. Chapter 8 describes types of system models that may be developed in the requirements engineering process. These provide a more detailed description for system developers. The emphasis here is on object-oriented modelling but I also include a description of data-flow diagrams. I find these are intuitive and helpful, especially for giving you an end-to-end picture of how information is processed by a system.

4. The emphasis in Chapters 9 and 10 is on critical systems specification. In Chapter 9, I discuss the specification of emergent dependability properties. I describe risk-driven approaches and specific issues of safety, reliability and security specification. In Chapter 10, I introduce formal specification techniques. Formal methods have had less impact than was once predicted but they are being increasingly used in the specification of safety and mission-critical systems. I cover both algebraic and model-based approaches in this chapter.

6 Software requirements

Objectives

The objectives of this chapter are to introduce software system requirements and to explain different ways of expressing software requirements. When you have read the chapter, you will:

- understand the concepts of user requirements and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand how requirements may be organised in a software requirements document.

Contents

6.1 Functional and non-functional requirements
6.2 User requirements
6.3 System requirements
6.4 Interface specification
6.5 The software requirements document

The requirements for a system are the descriptions of the services provided by the system and its operational constraints. These requirements reflect the needs of customers for a system that helps solve some problem such as controlling a device, placing an order or finding information. The process of finding out, analysing, documenting and checking these services and constraints is called *requirements engineering* (RE). In this chapter, I concentrate on the requirements themselves and how to describe them. I introduced the requirements engineering process in Chapter 4 and I discuss the RE process in more detail in Chapter 7.

The term *requirement* is not used in the software industry in a consistent way. In some cases, a requirement is simply a high-level, abstract statement of a service that the system should provide or a constraint on the system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis, 1993) explains why these differences exist:

> If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. *User requirements* are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate.
2. *System requirements* set out the system's functions, services and operational constraints in detail. The system requirements document (sometimes called a functional specification) should be precise. It should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different levels of system specification are useful because they communicate information about the system to different types of readers. Figure 6.1 illustrates the distinction between user and system requirements. This example from a library system shows how a user requirement may be expanded into several system requirements. You can see from Figure 6.1 that the user requirement is more abstract, and the system requirements add detail, explaining the services and functions that should be provided by the system to be developed.

Figure 6.1 User and system requirements

User requirement definition

1. LIBSYS shall keep track of all data required by copyright licensing agencies in the UK and elsewhere.

System requirements specification

1.1 On making a request for a document from LIBSYS, the requestor shall be presented with a form that records details of the user and the request made.
1.2 LIBSYS request forms shall be stored on the system for five years from the date of the request.
1.3 All LIBSYS request forms must be indexed by user, by the name of the material requested and by the supplier of the request.
1.4 LIBSYS shall maintain a log of all requests that have been made to the system.
1.5 For material where authors' lending rights apply, loan details shall be sent monthly to copyright licensing agencies that have registered with LIBSYS.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 6.2 shows the types of readers for the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

6.1 Functional and non-functional requirements

Software system requirements are often classified as functional requirements, non-functional requirements or domain requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

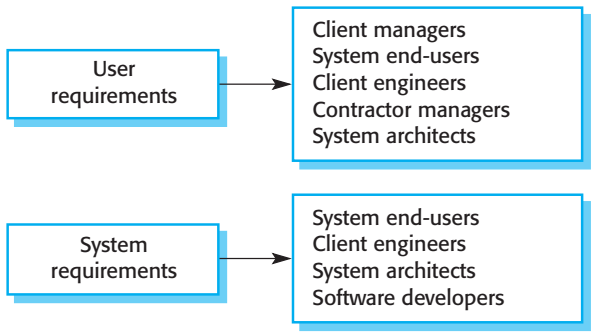


Figure 6.2 Readers of different types of specification

3. *Domain requirements* These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements

In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, say, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

6.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software and the general approach taken by the organisation when writing requirements. When expressed as user requirements, the requirements are usually described in a fairly abstract way. However, functional system requirements describe the system function in detail, its inputs and outputs, exceptions, and so on.

Functional requirements for a software system may be expressed in a number of ways. For example, here are examples of functional requirements for a university library system called LIBSYS, used by students and faculty to order books and documents from other libraries.

1. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. Every order shall be allocated a unique identifier (ORDER_ID), which the user shall be able to copy to the account's permanent storage area.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document, and they

illustrate that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

The LIBSYS system is a single interface to a range of article databases. It allows users to download copies of published articles in magazines, newspapers and scientific journals. I give a more detailed description of the requirements for the system on which LIBSYS is based in my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998).

Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement to simplify its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

Consider the second example requirement for the library system that refers to 'appropriate viewers' provided by the system. The library system can deliver documents in a range of formats; the intention of this requirement is that viewers for all of these formats should be available. However, the requirement is worded ambiguously; it does not make clear that viewers for each document format should be provided. A developer under schedule pressure might simply provide a text viewer and claim that the requirement had been met.

In principle, the functional requirements specification of a system should be both complete and consistent. *Completeness* means that all services required by the user should be defined. *Consistency* means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.

One reason for this is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that different system stakeholders (see Chapter 7) have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or, sometimes, after development is complete and the system is delivered to the customer.

6.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system such as the capabilities of I/O devices and the data representations used in system interfaces.

Non-functional requirements are rarely associated with individual system features. Rather, these requirements specify or constrain the emergent properties of the system, as discussed in Chapter 2. Therefore, they may specify system performance, security, availability, and other emergent properties. This means that they are often

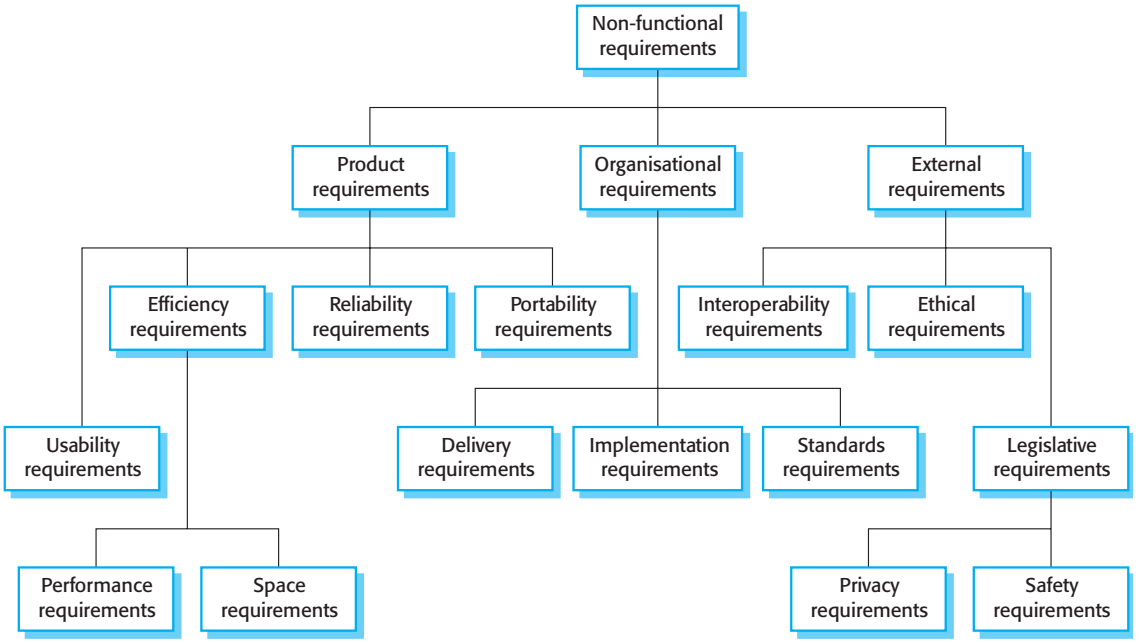


Figure 6.3 Types of non-functional requirements

more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if a real-time control system fails to meet its performance requirements, the control functions will not operate correctly.

Non-functional requirements are not just concerned with the software system to be developed. Some non-functional requirements may constrain the process that should be used to develop the system. Examples of process requirements include a specification of the quality standards that should be used in the process, a specification that the design must be produced with a particular CASE toolset and a description of the process that should be followed.

Non-functional requirements arise through user needs, because of budget constraints, because of organisational policies, because of the need for interoperability with other software or hardware systems, or because of external factors such as safety regulations or privacy legislation. Figure 6.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements) or from external sources.

Product requirement
4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.

Organisational requirement
9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

External requirement
7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Figure 6.4 Examples of non-functional requirements

The types of non-functional requirements are:

1. *Product requirements* These requirements specify product behaviour. Examples include performance requirements on how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; portability requirements; and usability requirements.
2. *Organisational requirements* These requirements are derived from policies and procedures in the customer's and developer's organisation. Examples include process standards that must be used; implementation requirements such as the programming language or design method used; and delivery requirements that specify when the product and its documentation are to be delivered.
3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include interoperability requirements that define how the system interacts with systems in other organisations; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements. Ethical requirements are requirements placed on a system to ensure that it will be acceptable to its users and the general public.

Figure 6.4 shows examples of product, organisational and external requirements taken from the library system LIBSYS whose user requirements were discussed in Section 6.1.1. The product requirement restricts the freedom of the LIBSYS designers in the implementation of the system user interface. It says nothing about the functionality of LIBSYS and clearly identifies a system constraint rather than a function. This requirement has been included because it simplifies the problem of ensuring the system works with different browsers.

The organisational requirement specifies that the system must be developed according to a company standard process defined as XYZCo-SP-STAN-95. The external requirement is derived from the need for the system to conform to privacy legislation. It specifies that library staff should not be allowed access to data, such as the addresses of system users, which they do not need to do their job.

Figure 6.5 System goals and verifiable requirements

A system goal
The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.

A verifiable non-functional requirement
Experienced controllers shall be able to use all the system functions after a total of two hours' training. After this training, the average number of errors made by experienced users shall not exceed two per day.

A common problem with non-functional requirements is that they can be difficult to verify. Users or customers often state these requirements as general goals such as ease of use, the ability of the system to recover from failure or rapid user response. These vague goals cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. As an illustration of this problem, consider Figure 6.5. This shows a system goal relating to the usability of a traffic control system and is typical of how a user might express usability requirements. I have rewritten it to show how the goal can be expressed as a 'testable' non-functional requirement. While it is impossible to objectively verify the system goal, you can design system tests to count the errors made by controllers using a system simulator.

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 6.6 shows a number of possible metrics that you can use to specify non-functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, however, customers for a system may find it practically impossible to translate their goals into quantitative requirements. For some goals, such as maintainability, there are no metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the required reliability (say) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying quantitative non-functional requirements may be very high, and the customers paying for the system may not think these costs are justified.

Therefore, requirements documents often include statements of goals mixed with requirements. These goals may be useful to developers because they give indications of customer priorities. However, you should always tell customers that they are open to misinterpretation and cannot be objectively verified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, it may be a requirement that the maximum memory used by a system should be no more than 4 Mbytes. Memory constraints are common for embedded systems where space or weight is limited and the number of ROM chips storing the system software must be minimised. Another requirement might be that the system should be written using Ada, a programming

Figure 6.6 Metrics for specifying non-functional requirements

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target-dependent statements Number of target systems

language for critical, real-time software development. However, it may not be possible to compile an Ada program with the required functionality into less that 4 Mbytes. There therefore has to be a trade-off between these requirements: an alternative development language or increased memory added to the system.

It is helpful if you can differentiate functional and non-functional requirements in the requirements document. In practice, this is difficult to do. If the non-functional requirements are stated separately from the functional requirements, it is sometimes difficult to see the relationships between them. If they are stated with the functional requirements, you may find it difficult to separate functional and non-functional considerations and to identify requirements that relate to the system as a whole. However, you should explicitly highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Non-functional requirements such as safety and security requirements are particularly important for critical systems. I therefore discuss dependability requirements in more detail in Chapter 9, which covers critical systems specification.

6.1.3 Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They usually include specialised domain terminology or reference to domain concepts. They may be new functional require-

The deceleration of the train shall be computed as:

$$D_{train} = D_{control} + D_{gradient}$$

where $D_{gradient}$ is 9.81 ms^2 * compensated gradient/alpha and where the values of 9.81 ms^2/alpha are known for different types of train.

Figure 6.7 A domain requirement from a train protection system

ments in their own right, constrain existing functional requirements or set out how particular computations must be carried out. Because these requirements are specialised, software engineers often find it difficult to understand how they are related to other system requirements.

Domain requirements are important because they often reflect fundamentals of the application domain. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily. The LIBSYS system includes a number of domain requirements:

1. There shall be a standard user interface to all databases that shall be based on the Z39.50 standard.
2. Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manual forwarding to the user or routed to a network printer.

The first requirement is a design constraint. It specifies that the user interface to the database must be implemented according to a specific library standard. The developers therefore have to find out about that standard before starting the interface design. The second requirement has been introduced because of copyright laws that apply to material used in libraries. It specifies that the system must include an automatic delete-on-print facility for some classes of document. This means that users of the library system cannot have their own electronic copy of the document.

To illustrate domain requirements that specify how a computation is carried out, consider Figure 6.7, taken from the requirements specification for an automated train protection system. This system automatically stops a train if it goes through a red signal. This requirement states how the train deceleration is computed by the system. It uses domain-specific terminology. To understand it, you need some understanding of the operation of railway systems and train characteristics.

The requirement for the train system illustrates a major problem with domain requirements. They are written in the language of the application domain (mathematical equations in this case), and it is often difficult for software engineers to understand them. Domain experts may leave information out of a requirement simply because it is so obvious to them. However, it may not be obvious to the developers of the system, and they may therefore implement the requirement in the wrong way.

6.2 User requirements

The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge. They should only specify the external behaviour of the system and should avoid, as far as possible, system design characteristics. Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing the system implementation. You should write user requirements in simple language, with simple tables and forms and intuitive diagrams.

However, various problems can arise when requirements are written in natural language sentences in a text document:

1. *Lack of clarity* It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
2. *Requirements confusion* Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.
3. *Requirements amalgamation* Several different requirements may be expressed together as a single requirement.

As an illustration of some of these problems, consider one of the requirements for the library shown in Figure 6.8.

This requirement includes both conceptual and detailed information. It expresses the concept that there should be an accounting system as an inherent part of LIBSYS. However, it also includes the detail that the accounting system should support discounts for regular LIBSYS users. This detail would have been better left to the system requirements specification.

It is good practice to separate user requirements from more detailed system requirements in a requirements document. Otherwise, non-technical readers of the user requirements may be overwhelmed by details that are really only relevant for technicians. Figure 6.9 illustrates this confusion. This example is taken from an actual requirements document for a CASE tool for editing software design models. The user may specify that a grid should be displayed so that entities may be accurately positioned in a diagram.

The first sentence mixes up three kinds of requirements.

1. A conceptual, functional requirement states that the editing system should provide a grid. It presents a rationale for this.
2. A non-functional requirement giving detailed information about the grid units (centimetres or inches).

Figure 6.8 A requirement for a user accounting system in LIBSYS

4.5 LIBSYS shall provide a financial accounting system that maintains records of all payments made by users of the system. System managers may configure this system so that regular users may receive discounted rates.

Figure 6.9 A user requirement for an editor grid

2.6 Grid facilities To assist in the positioning of entities on a diagram, the user may turn on a grid in either centimetres or inches, via an option on the control panel. Initially, the grid is off. The grid may be turned on and off at any time during an editing session and can be toggled between inches and centimetres at any time. A grid option will be provided on the reduce-to-fit view but the number of grid lines shown will be reduced to avoid filling the smaller diagram with grid lines.

3. A non-functional user interface requirement that defines how the grid is switched on and off by the user.

The requirement in Figure 6.9 also gives some but not all initialisation information. It defines that the grid is initially off. However, it does not define its units when turned on. It provides some detailed information—namely, that the user may toggle between units—but not the spacing between grid lines.

User requirements that include too much information constrain the freedom of the system developer to provide innovative solutions to user problems and are difficult to understand. The user requirement should simply focus on the key facilities to be provided. I have rewritten the editor grid requirement (Figure 6.10) to focus only on the essential system features.

Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and is particularly useful when requirements are changed. For example, the rationale in Figure 6.10 recognises that an active grid where positioned objects automatically 'snap' to a grid line can be useful. However, this has been deliberately rejected in favour of manual positioning. If a change to this is proposed at some later stage, it will be clear that the use of a passive grid was deliberate rather than an implementation decision.

To minimise misunderstandings when writing user requirements, I recommend that you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardising the format makes omissions less likely and requirements easier to check. The format I use shows the initial requirement in boldface, including a statement of rationale with each user requirement and a reference to the more detailed system requirement specification. You may also

Figure 6.10
A definition of an editor grid facility

2.6.1 Grid facilities

The editor shall provide a grid facility where a matrix of horizontal and vertical lines provide a background to the editor window. This grid shall be a passive grid where the alignment of entities is the user's responsibility.

Rationale: A grid helps the user to create a tidy diagram with well-spaced entities. Although an active grid, where entities 'snap-to' grid lines can be useful, the positioning is imprecise. The user is the best person to decide where entities should be positioned.

Specification: ECLIPSE/WS/Tools/DE/FS Section 5.6

Source: Ray Wilson, Glasgow Office

include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.

2. Use language consistently. You should always distinguish between mandatory and desirable requirements. *Mandatory requirements* are requirements that the system must support and are usually written using 'shall'. *Desirable requirements* are not essential and are written using 'should'.
3. Use text highlighting (bold, italic or colour) to pick out key parts of the requirement.
4. Avoid, as far as possible, the use of computer jargon. Inevitably, however, detailed technical terms will creep into the user requirements.

The Robertsons (Robertson and Robertson, 1999), in their book that covers the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, supporting materials, and so on. This extends the format that I have used in Figure 6.10, and it can be used for both user and system requirements.

6.3 System requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may

be used as part of the contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system.

Ideally, the system requirements should simply describe the external behaviour of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is impossible, in practice, to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organised according to the different sub-systems that make up the system. As I discuss in Chapter 7 and Chapter 18, this architectural definition is essential if you want to reuse software components when implementing the system.
2. In most cases, systems must interoperate with other existing systems. These constrain the design, and these constraints impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability, discussed in Chapter 20) may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified be used.

Natural language is often used to write system requirements specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand:

1. Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstandings because of the ambiguity of natural language. Jackson (Jackson, 1995) gives an excellent example of this when he discusses signs displayed by an escalator. These said ‘Shoes must be worn’ and ‘Dogs must be carried’. I leave it to you to work out the conflicting interpretations of these phrases.
2. A natural language requirements specification is overflexible. You can say the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct.
3. There is no easy way to modularise natural language requirements. It may be difficult to find all related requirements. To discover the consequence of a change, you may have to look at every requirement rather than at just a group of related requirements.

Because of these problems, requirements specifications written in natural language are prone to misunderstandings. These are often not discovered until later phases of the software process and may then be very expensive to resolve.

Figure 6.11 Notations for requirements specification

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT (Ross, 1977) (Schoman and Ross, 1977). Now, use-case descriptions (Jacobsen, et al., 1993) and sequence diagrams are commonly used (Stevens and Pooley, 1999).
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

It is essential to write user requirements in a language that non-specialists can understand. However, you can write system requirements in more specialised notations (Figure 6.11). These include stylised, structured natural language, graphical models of the requirements such as use-cases to formal mathematical specifications. In this chapter, I discuss how structured natural language supplemented by simple graphical models may be used to write system requirements. I discuss graphical system modelling in Chapter 8 and formal system specification in Chapter 10.

6.3.1 Structured language specifications

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification. Structured language notations limit the terminology that can be used and use templates to specify system requirements. They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification.

An early project that used structured natural language for specifying system requirements is described by Heninger (Heninger, 1980). Special-purpose forms were designed to describe the input, output and functions of an aircraft software system. The system requirements were specified using these forms.

Figure 6.12 System requirements specification using a standard form

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop
Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.	
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side effects	None

To use a form-based approach to specifying system requirements, you must define one or more standard forms or templates to express the requirements. The specification may be structured around the objects manipulated by the system, the functions performed by the system or the events processed by the system. An example of such a form-based specification is shown in Figure 6.12. I have taken this example from the insulin pump system that was introduced in Chapter 3.

The insulin pump bases its computations of the user's insulin requirement on the rate of change of blood sugar levels. These rates of change computed using the current and previous readings. You can download a complete version of the specification for the insulin pump from the book's web pages.

When a standard form is used for specifying functional requirements, the following information should be included:

1. Description of the function or entity being specified
2. Description of its inputs and where these come from

3. Description of its outputs and where these go to
4. Indication of what other entities are used (the *requires* part)
5. Description of the action to be taken
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called
7. Description of the side effects (if any) of the operation.

Using formatted specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organised more effectively. However, it is difficult to write requirements in an unambiguous way, particularly when complex computations are required. You can see this in the description shown in Figure 6.12, where it isn't made clear what happens if the pre-condition is not satisfied.

To address this problem, you can add extra information to natural language requirements using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. Figure 6.13 is a revised description of the computation of the insulin dose.

Graphical models are most useful when you need to show how state changes (see Chapter 8) or where you need to describe a sequence of actions. Figure 6.14 illustrates the sequence of actions when a user wishes to withdraw cash from an automated teller machine (ATM).

You should read a sequence diagram from top to bottom to see the order of the actions that take place. In Figure 6.14, there are three basic sub-sequences:

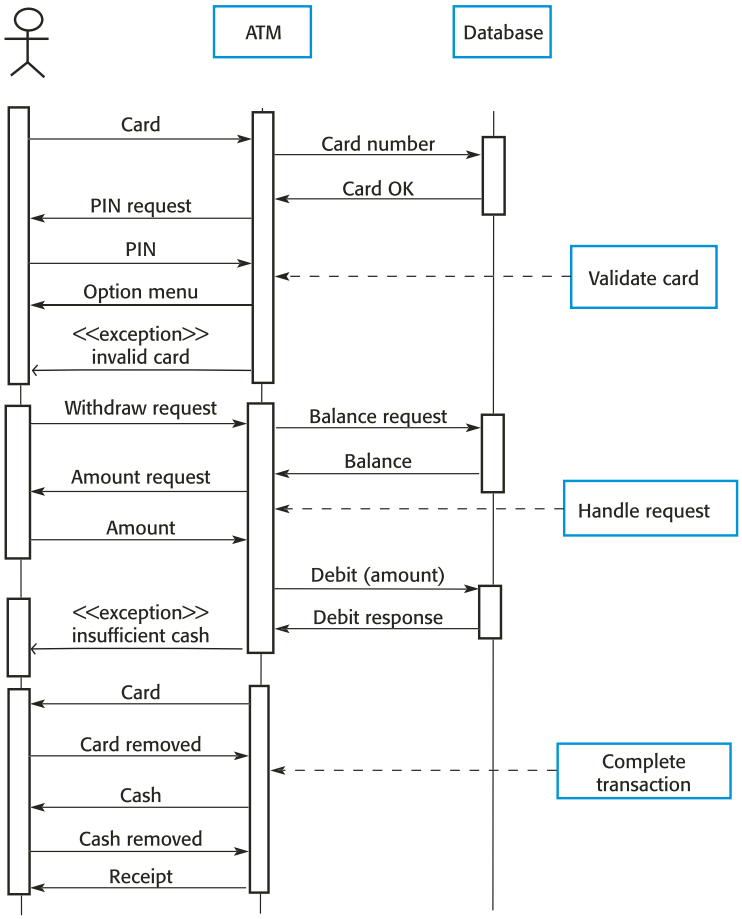
1. *Validate card* The user's card is validated by checking the card number and user's PIN.
2. *Handle request* The user's request is handled by the system. For a withdrawal, the database must be queried to check the user's balance and to debit the amount withdrawn. Notice the exception here if the requestor does not have enough money in their account.
3. *Complete transaction* The user's card is returned and, when it is removed, the cash and receipt are delivered.

You will see sequence diagrams again in Chapter 8, which covers system models, and in Chapter 14, which covers object-oriented design.

Figure 6.13 Tabular specification of computation

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ((r2 – r1) > (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 6.14 Sequence diagram of ATM withdrawal



6.4 Interface specification

Almost all software systems must operate with existing systems that have already been implemented and installed in an environment. If the new system and the existing systems must work together, the interfaces of existing systems have to be precisely specified. These specifications should be defined early in the process and included (perhaps as an appendix) in the requirements document.

There are three types of interface that may have to be defined:

1. *Procedural interfaces* where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs).
2. *Data structures* that are passed from one sub-system to another. Graphical data models (described in Chapter 8) are the best notations for this type of description. If necessary, program descriptions in Java or C++ can be generated automatically from these descriptions.
3. *Representations of data* (such as the ordering of bits) that have been established for an existing sub-system. These interfaces are most common in embedded, real-time system. Some programming languages such as Ada (although not Java) support this level of specification. However, the best way to describe these is probably to use a diagram of the structure with annotations explaining the function of each group of bits.

Formal notations, discussed in Chapter 10, allow interfaces to be defined in an unambiguous way, but their specialised nature means that they are not understandable without special training. They are rarely used in practice for interface specification although, in my view, they are ideally suited for this purpose. A programming language such as Java can be used to describe the syntax of the interface. However, this has to be supplemented by further description explaining the semantics of each of the defined operations.

Figure 6.15 is an example of a procedural interface definition defined in Java. In this case, the interface is the procedural interface offered by a print server. This manages a queue of requests to print files on different printers. Users may examine the queue associated with a printer and may remove their print jobs from that queue. They may also switch jobs from one printer to another. The specification in Figure 6.15 is an abstract model of the print server that does not reveal any interface details. The functionality of the interface operations can be defined using structured natural language or tabular description.

Figure 6.15 The Java PDL description of a print server interface

```
interface PrintServer {

// defines an abstract printer server
// requires: interface Printer, interface PrintDoc
// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

   void initialize ( Printer p ) ;
   void print ( Printer p, PrintDoc d ) ;
   void displayPrintQueue ( Printer p ) ;
   void cancelPrintJob (Printer p, PrintDoc d) ;
   void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

6.5 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. In some cases, the user and system requirements may be integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

The requirements document has a diverse set of users, ranging from the senior management of the organisation that is paying for the system to the engineers responsible for developing the software. Figure 6.16, taken from my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998) illustrates possible users of the document and how they use it.

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. When the system will be developed by an external contractor, critical system specifications need to be precise and very detailed. When there is more flexibility in the requirements and where an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities resolved during development of the system.

A number of large organisations, such as the US Department of Defense and the IEEE, have defined standards for requirements documents. Davis (Davis, 1993) discusses some of these standards and compares their contents. The most widely known

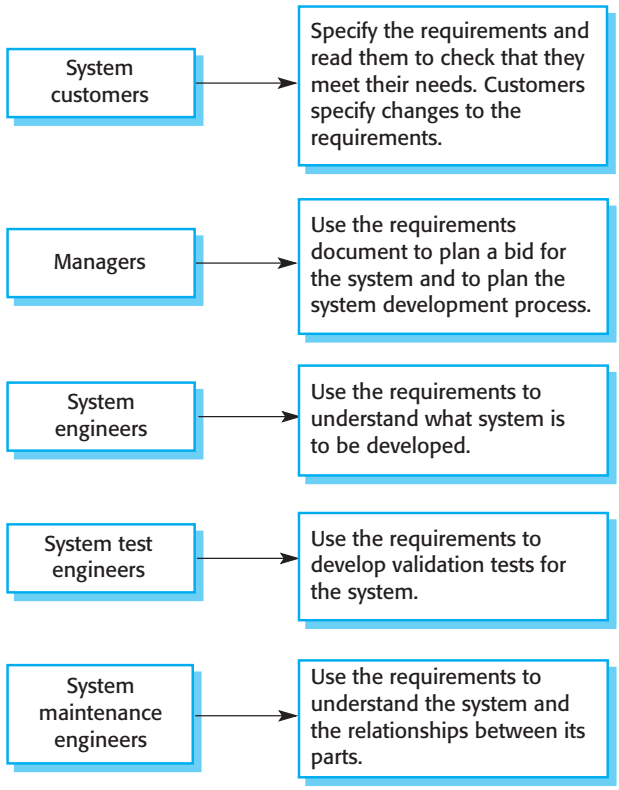


Figure 6.16 Users of a requirements document

standard is IEEE/ANSI 830-1998 (IEEE, 1998). This IEEE standard suggests the following structure for requirements documents:

1. **Introduction**
   - 1.1 Purpose of the requirements document
   - 1.2 Scope of the product
   - 1.3 Definitions, acronyms and abbreviations
   - 1.4 References
   - 1.5 Overview of the remainder of the document
2. **General description**
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 General constraints
   - 2.5 Assumptions and dependencies
3. **Specific requirements** cover functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because

of the wide variability in organisational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. **Appendices**
5. **Index**

Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organisational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organisation. Figure 6.17 illustrates a possible organisation for a requirements document that is based on the IEEE standard. However, I have extended this to include information about predicted system evolution. This was first proposed by Heninger (Heninger, 1980) and, as I have discussed, helps the maintainers of the system and may allow designers to include support for future system features.

Of course, the information that is included in a requirements document must depend on the type of software being developed and the approach to development that is used. If an evolutionary approach is adopted for a software product (say), the requirements document will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, non-functional system requirements. In this case, the designers and programmers use their judgement to decide how to meet the outline user requirements for the system.

By contrast, when the software is part of a large system engineering project that includes interacting hardware and software systems, it is often essential to define the requirements to a fine level of detail. This means that the requirements documents are likely to be very long and should include most if not all of the chapters shown in Figure 6.17. For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need.

Requirements documents are essential when an outside contractor is developing the software system. However, agile development methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort that is largely wasted. Rather than a formal document, approaches such as extreme programming (Beck, 1999) propose that user requirements should be collected incrementally and written on cards. The user then prioritises requirements for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I would argue that it is still useful to write a short supporting document that defines the business and dependability requirements for the system. It is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

Figure 6.17 The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organisation commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements, e.g. interfaces to other systems may be defined.
System models	This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models, data-flow models and semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, specific information which is related to the application which is being developed. Examples of appendices that may be included are hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organisation of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out. Domain requirements are functional requirements that are derived from characteristics of the application domain.
- Non-functional requirements constrain the system being developed and the development process that should be used. They may be product requirements, organisational requirements or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- User requirements are intended for use by people involved in using and procuring the system. They should be written using in natural language, with tables and diagrams that are easily understood.
- System requirements are intended to communicate, in a precise way, the functions that the system must provide. To reduce ambiguity, they may be written in a structured form of natural language supplemented by tables and system models.
- The software requirements document is the agreed statement of the system requirements. It should be organised so that both system customers and software developers can use it.
- The IEEE standard for requirements documents is a useful starting point for more specific requirements specification standards.

FURTHER READING

Software Requirements, 2nd ed. This book, designed for writers and users of requirements, discusses good requirements engineering practice. (K. M. Weigers, 2003, Microsoft Press.)

Mastering the Requirements Process. A well-written, easy-to-read book that is based on a particular method (VOLERE) but which also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 1999, Addison-Wesley.)

Requirements Engineering: Processes and Techniques. This book covers all aspects of the requirements engineering process and discusses specific requirements specification techniques. (G. Kotonya and I. Sommerville, 1999, John Wiley & Sons.)

Software Requirements Engineering. This collection of papers on requirements engineering includes several relevant articles such as 'Recommended Practice for Software Requirements Specification', a discussion of the IEEE standard for requirements documents. (R. H. Thayer and M. Dorfman (eds.), 1997, IEEE Computer Society Press.)

EXERCISES

6.1 Identify and briefly describe four types of requirements that may be defined for a computer-based system

6.2 Discuss the problems of using natural language for defining user and system requirements, and show, using small examples, how structuring natural language into forms can help avoid some of these difficulties.

6.3 Discover ambiguities or omissions in the following statement of requirements for part of a ticket-issuing system.

An automated ticket-issuing system sells rail tickets. Users select their destination and input a credit card and a personal identification number. The rail ticket is issued and their credit card account charged. When the user presses the start button, a menu display of potential destinations is activated, along with a message to the user to select a destination. Once a destination has been selected, users are requested to input their credit card. Its validity is checked and the user is then requested to input a personal identifier. When the credit transaction has been validated, the ticket is issued.

6.4 Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in some appropriate way.

6.5 Draw a sequence diagram showing the actions performed in the ticket-issuing system. You may make any reasonable assumptions about the system. Pay particular attention to specifying user errors.

6.6 Using the technique suggested here, where natural language is presented in a standard way, write plausible user requirements for the following functions:

- The cash-dispensing function in a bank ATM
- The spelling-check and correcting function in a word processor
- An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader and then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

6.7 Describe four types of non-functional requirements that may be placed on a system. Give examples of each of these types of requirement.

6.8 Write a set of non-functional requirements for the ticket-issuing system, setting out its expected reliability and its response time.

6.9 Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

6.10 You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. You have also a responsibility of confidentiality to your previous employer.