PART TWO

Requirements

5 Software requirements

Objectives

The objective of this chapter is to introduce software system requirements and to explain different ways of expressing these requirements. When you have read the chapter you will:

- understand the concepts of user requirements and system requirements and why these requirements may be expressed using different notations;
- understand the differences between functional and non-functional requirements;
- understand two techniques for describing system requirements, namely structured natural language and programming-language based descriptions;
- understand how requirements may be organised in a software requirements document.

Contents

The problems that software engineers have to solve are often immensely complex. Understanding the nature of the problems can be very difficult, especially if the system is new. Consequently, it is difficult to establish exactly what the system should do. The descriptions of the services and constraints are the *requirements* for the system and the process of finding out, analysing, documenting and checking these services and constraints is called *requirements engineering*. In this chapter, I concentrate on the requirements themselves and how to describe them. The requirements engineering process was introduced in Chapter 3 and I discuss it in more detail in Chapter 6.

The term *requirement* is not used throughout the software industry in a consistent way. In some cases, a requirement is seen as 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, mathematically formal definition of a system function. 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 pre-defined. 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 make this separation 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. As well as these two levels of detail, a more detailed description (a software design specification) may be produced to bridge the requirements engineering and design activities. User requirements, system requirements and software design specification 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 services and constraints in detail. The system requirements document, which is sometimes called a functional specification, should be precise. It may serve as a contract between the system buyer and software developer.
3. *A software design specification* is an abstract description of the software design which is a basis for more detailed design and implementation. This specification adds further detail to the system requirements specification.

Figure 5.1 User and system requirements

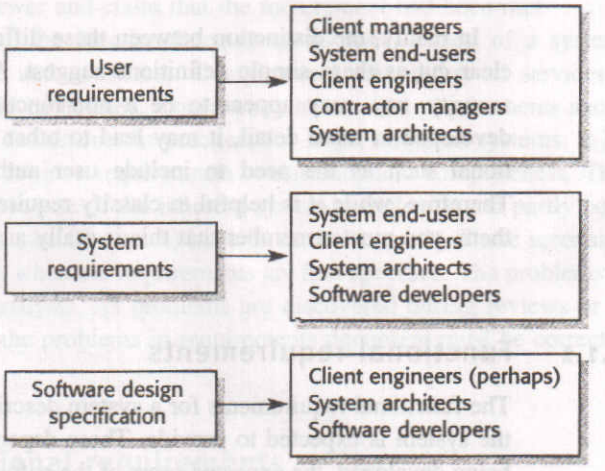
User requirement definition

1. The software must provide a means of representing and accessing external files created by other tools.

System requirements specification

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Figure 5.2 Readers of different types of specification



Different levels of system specification are useful because they communicate information about the system to different types of reader. Figure 5.1 illustrates the distinction between user and system requirements. It shows how a user requirement may be expanded into several system requirements.

The user requirements should be written for client and contractor managers who do not have a detailed technical knowledge of the system (Figure 5.2). The system requirements specification should be targeted at senior technical staff and project managers. Again, it will be used by staff from both the client and the contractor. System end-users may read both of these documents. Finally, the software design specification is an implementation-oriented document. It should be written for the software engineers who will develop the system.

5.1 Functional and non-functional requirements

Software system requirements are often classified as functional or non-functional requirements or as 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, standards, etc.
3. *Domain requirements* These are requirements that come from the application domain of the system and that reflect characteristics of that domain. They may be functional or non-functional requirements.

In reality, the distinction between these different types of requirement 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, it may lead to other requirements that are clearly functional such as the need to include user authorisation facilities in the system. Therefore, while it is helpful to classify requirements in this way when discussing them, you must remember that this is really an artificial distinction.

5.1.1 Functional requirements

The functional requirements for a system describe the functionality or services that the system is expected to provide. These depend on the type of software which is being developed, the expected users of the software and the type of system which is being developed. When expressed as user requirements, they are usually described in a fairly general way but functional system requirements describe the system function in detail, its inputs and outputs, exceptions, etc.

Functional requirements for a software system may be expressed in a number of different ways. Here are a number of functional requirements for a university library system (Kotonya and Sommerville, 1998) for 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 are functional user requirements which define specific facilities which must be provided by the system. These have been taken from the user requirements document for the system and they illustrate that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Many of the problems of software engineering stem from imprecision in the requirements specification. 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 in the above requirements list which refers to 'appropriate viewers' provided by the system. The library system can deliver documents in a range of formats and the intention of this requirement is that viewers for all of these formats should be available. However, it is worded ambiguously as 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. The reason for this is partly because of the inherent system complexity and partly because different viewpoints (see Chapter 6) have inconsistent needs. These inconsistencies may not be obvious when the requirements are first specified. The problems only emerge after deeper analysis. As problems are discovered during reviews or in later lifecycle phases, the problems in requirements document must be corrected.

5.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are those requirements which 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.

Many non-functional requirements relate to the system as a whole rather than to individual system features. This means that they are often more critical than individual functional requirements. While failure to meet an individual functional requirement may degrade the system, failure to meet a non-functional system requirement may make the whole system unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for

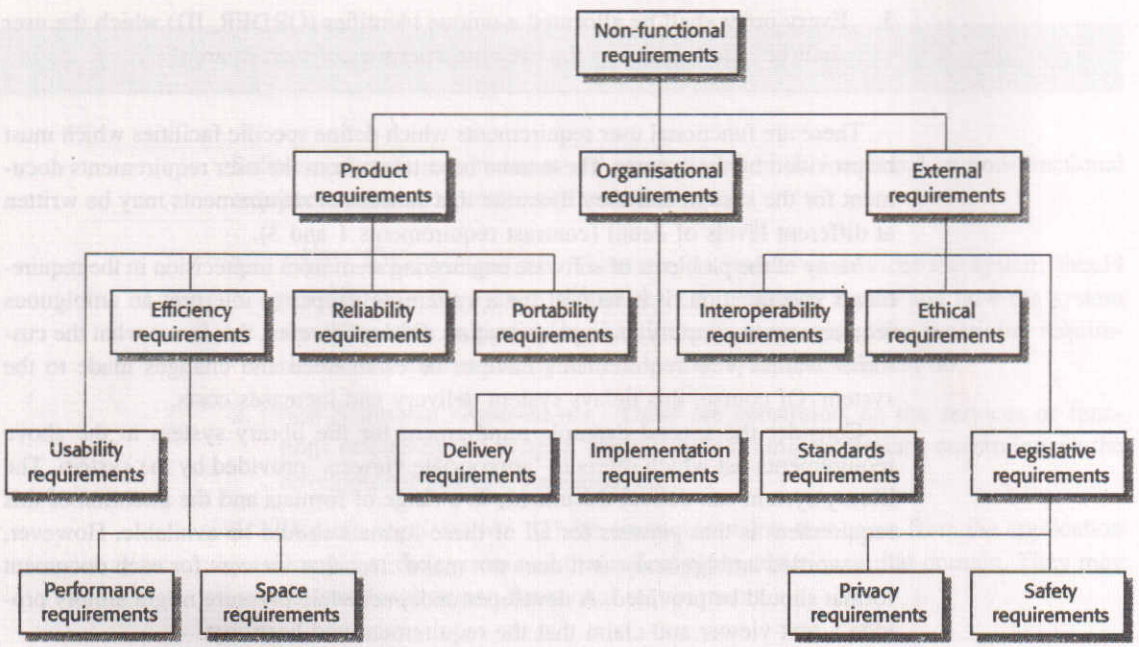


Figure 5.3 Types of non-functional requirement

operation; if a real-time control system fails to meet its performance requirements, the control functions will not operate correctly.

However, non-functional requirements are not always concerned with the software system to be developed. Some non-functional requirements may constrain the process which may be used to develop the system. Examples of process requirements include a specification of the quality standards which must be used in the process, a specification that the design must be produced with a specified CASE toolset and a description of the process which 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, privacy legislation, etc. Figure 5.3 is a classification of the different types of non-functional requirements that may arise.

I have classified the different types of non-functional requirements shown in Figure 5.3 according to their derivation:

1. *Product requirements* These are requirements that 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 are derived from policies and procedures in the customer's and developer's organisation. Examples include process

standards which must be used; implementation requirements such as the programming language or design method used; and delivery requirements which specify when the product and its documentation are to be delivered.

3. *External requirements* This broad heading covers all requirements which are derived from factors external to the system and its development process. These include interoperability requirements which define how the system interacts with systems in other organisations; legislative requirements which 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 5.4 shows examples of product, organisational and external requirements. The product requirement relates to a programming support environment for the Ada language (an APSE). This restricts the freedom of the APSE designers in their choice of symbols used in the APSE user interface. It says nothing about the functionality of the APSE and clearly identifies a system constraint rather than a function. 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 system operators should not have access to any data that they do not need.

A common problem with non-functional requirements is that they are sometimes difficult to verify. They may be written to reflect general goals of the customer such as ease of use, the ability of the system to recover from failure or rapid user response. These requirements 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 5.5. This shows a system goal relating to the usability of the system and how this can be expressed in a verifiable way as a non-functional requirement. This non-functional requirement can be verified by testing so you can check if the system has met the customer's goal.

Ideally, non-functional requirements should be expressed quantitatively using metrics that can be objectively tested. Figure 5.6 shows a number of possible metrics

Figure 5.4 Examples of non-functional requirements

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 5.5 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.

Figure 5.6 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

that may be used to specify non-functional system properties. Measurements can be made during system testing to determine whether or not the system meets these requirements.

In practice, quantitative requirements specification is often difficult. Customers may not be able to translate their goals into quantitative requirements; for some goals, such as maintainability goals, there are no metrics that can be used; the cost of objectively verifying quantitative non-functional requirements may be very high. Therefore, requirements documents will often include statements of goals mixed with requirements. These goals may be useful to developers because they give some clues to customer priorities. However, customers should be told that they are open to misinterpretation and cannot be objectively verified.

Non-functional requirements often conflict and interact with other system functional requirements. For example, it may be a requirement that the maximum store

occupied by a system should be 4 Mbytes because the entire system has to be fitted into read-only memory and installed on a spacecraft. A further requirement might be that the system should be written using Ada, a programming language that was designed for critical real-time software development. However, it may not be possible to compile an Ada program with the required functionality into less than 4 Mbytes. Some trade-off between these requirements must be made. An alternative development language may be used or increased memory added to the system.

In principle, functional and non-functional requirements should be differentiated in a requirements document. In practice, this is difficult. If the non-functional requirements are stated separately from the functional requirements, it is sometimes difficult to see the relationships between them. If stated with the functional requirements, it may be difficult to separate functional and non-functional considerations and to identify requirements which relate to the system as a whole. An appropriate balance must be found which depends on the type of system being specified. However, requirements that are clearly related to emergent system properties should be explicitly highlighted. This may be done either by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

5.1.3 Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements or set out how particular computations must be carried out. 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.

To illustrate domain requirements, consider the following requirements from the library system:

1. There shall be a standard user interface to all databases which 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 manually forwarding to the user or routed to a network printer.

The first of these requirements is a constraint on a system functional requirement. It specifies that the user interface to the database must be implemented according to a specific library standard. The second requirement has been introduced because of copyright laws which apply to material used in libraries. It specifies that the

Figure 5.7 A domain requirement from a train protection system

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.

system must include an automatic delete-on-print facility for some classes of document.

To illustrate domain requirements that specify how a computation is carried out, consider Figure 5.7 which has been taken from the specification of 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 the major problem with domain requirements. They are expressed using language which is specific to the application domain 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 an unsatisfactory way.

5.2 User requirements

The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge. They should only specify the external behaviour of the system and should avoid, as far as possible, system design characteristics. Consequently, the user requirements should not be defined using an implementation model. The user requirements must be written using natural language, forms and simple intuitive diagrams.

However, various problems can arise when requirements are written in natural language:

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.

Figure 5.8
A requirement on a database for a programming environment

4.A.5 The database shall support the generation and control of configuration objects; that is, objects which are themselves groupings of other objects in the database. The configuration control facilities shall allow access to the objects in a version group by the use of an incomplete name.

Figure 5.9 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.

As an illustration of some of these problems, consider one of the requirements for an Ada programming environment shown in Figure 5.8.

This requirement includes both conceptual and detailed information. It expresses the concept that there should be configuration control facilities provided as an inherent part of the APSE. However, it also includes the detail that the configuration control facilities should allow access to the objects in a version group without specifying their complete name. 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 which are really only relevant for technicians. Figure 5.9 illustrates this confusion. This example is taken from the 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 different kinds of requirement.

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

The requirement in Figure 5.9 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.

Figure 5.10
A definition of an editor grid facility

2.6 Grid facilities

2.6.1 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

Figure 5.11 User requirements for node creation

3.5.1 Adding nodes to a design

3.5.1.1 **The editor shall provide a facility for users to add nodes of a specified type to their design.**

3.5.1.2 The sequence of actions to add a node should be as follows:

1. The user should select the type of node to be added.
2. The user should move the cursor to the approximate node position in the diagram and indicate that the node symbol should be added at that point.
3. The user should then drag the node symbol to its final position.
   Rationale: The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

Specification: ECLIPSE/WS/Tools/DE/FS. Section 3.5.1

When user requirements include too much information, it constrains the freedom of the system developer to provide innovative solutions to user problems and makes the requirements difficult to understand. The user requirement should simply focus on the key facilities to be provided. This is illustrated in Figure 5.10 where I have rewritten the requirement for the editor grid to focus only on the essential system features.

The rationale associated with the requirements is important. It helps the system developers and maintainers to understand why the requirement has been included and to assess the impact of requirements change. For example, in Figure 5.10, the rationale recognises that an active grid where positioned objects automatically 'snap' to a grid line can be useful. However, it deliberately rejects this option in favour of manual positioning. If a change to this is proposed at some later stage, it is then clear that the use of a passive grid was deliberate rather than an implementation decision.

A further example of a more specific user requirement, which also defines part of the editing system, is shown in Figure 5.11. This is a more detailed specification of a function. In this case, the definition includes a list of user actions. This is sometimes necessary so that all functions can be provided in a consistent way. Implementation details should not be included in this additional information.

Therefore, the definition does not set out how the cursor and the symbol are moved, or how the type is selected.

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

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardising the format means that omissions are less likely and makes requirements easier to check. The format I use includes emboldening the initial requirement, including a statement of rationale with each user requirement and a reference to the more detailed system requirement specification.
2. Use language consistently. In particular, distinguish between mandatory and desirable requirements. It is usual practice to define mandatory requirements using 'shall' and desirable requirements using 'should' as can be seen in Figure 5.11. Therefore, it is mandatory that the system includes a facility to add nodes to a design. It is desirable that the sequence of actions should be as specified but this is not absolutely essential if there are good reasons why it can't be done in that way.
3. Use text highlighting (bold and italic) to pick out key parts of the requirement.
4. Avoid, as far as possible, the use of computer jargon. However, it will inevitably be the case that detailed technical terms which are used in the application domain of the system will be included in the user requirements.

5.3 System requirements

System requirements are more detailed descriptions of the user requirements. They may serve as the basis for a contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system. They are used by software engineers as the starting point for the system design.

The system requirements specification may include different models of the system such as an object model or a data-flow model. I describe different models that may be used in the system requirements specification in Chapter 7.

In principle, the system requirements should state what the system should do and not how it should be implemented. However, at the level of detail required to specify the system completely, it is virtually impossible to exclude all design information. There are several reasons for this:

1. An initial architecture of the system may be defined to help structure the requirements specification. The system requirements are organised according to the different sub-systems which make up the system.

2. In most cases, systems must interoperate with other existing systems. These constrain the design and these constraints generate requirements for the new system.
3. The use of a specific design (such as N-version programming to achieve reliability, discussed in Chapter 18) may be an external system requirement.

Natural language is often used to write system requirements specifications. However, as well as the problems identified in section 5.2, further problems with natural language can arise when it is used for more detailed specification:

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 (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 over-flexible. 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 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.

There are various alternatives to the use of natural language which add structure to the specification and which help reduce ambiguity. These are shown in Figure 5.12.

Other approaches, such as specialised requirements languages (Teichrow and Hershey, 1977; Alford, 1977; Bell *et al.*, 1977; Alford, 1985), have also been developed but these are now rarely used. Davis (1990) summarises and compares some of these different approaches to requirements specification. In this chapter, I focus on the first two of these approaches, namely structured natural language and the use of design description languages.

5.3.1 Structured language specifications

Structured natural language is a restricted form of natural language for writing system requirements. The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree

Figure 5.12 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.
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). More recently, use-case descriptions (Jacobsen *et al.*, 1993) have been used. I discuss these in the following chapter.
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 them as a system contract. I discuss formal specification in Chapter 9.

of uniformity is imposed on the specification. Structured language notations may limit the terminology used and may use templates to specify system requirements. They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification.

A project which used structured natural language for specifying system requirements is described by 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.

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 5.13. This is a more detailed definition of the Add node function for the design-editing system defined in Figure 5.11.

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

1. a description of the function or entity being specified;
2. a description of its inputs and where these come from;
3. a description of its outputs and where these go to;
4. an indication of what other entities are used (the *requires* part);

Figure 5.13 System requirements specification using a standard form

ECLIPSE/Workstation/Tools/DE/FS/3.5.1

Function Add node

Description Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.

Inputs Node type, Node position, Design identifier.

Source Node type and Node position are input by the user, Design identifier from the database.

Outputs Design identifier.

Destination The design database. The design is committed to the database on completion of the operation.

Requires Design graph rooted at input design identifier.

Pre-condition The design is open and displayed on the user's screen.

Post-condition The design is unchanged apart from the addition of a node of the specified type at the given position.

Side-effects None

Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1

5. 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;
6. a description of the side-effects (if any) of the operation.

Using formatted specifications removes some of the problems of natural language specification in that there is less variability in the specification and requirements are partitioned more effectively. However, some ambiguity may remain in the specification. Alternative methods which use more structured notations such as a PDL (described below) go some way towards tackling the problem of specification ambiguity. However, non-specialists usually find them harder to understand.

5.3.2 Requirements specification using a PDL (program description language)

To counter the inherent ambiguities in natural language specification, it is possible to describe requirements operationally using a program description language or PDL. A PDL is a language derived from a programming language like Java or Ada. It may contain additional, more abstract, constructs to increase its expressive power. The advantage of using a PDL is that it may be checked syntactically and semantically by software tools. Requirements omissions and inconsistencies may be inferred from the results of these checks.

Figure 5.14 A PDL description of ATM operation

```
class ATM {
  // declarations here
  public static void main (String args[]) throws InvalidCard {
      try {
          thisCard.read () ; // may throw InvalidCard exception
          pin = KeyPad.readPin () ; attempts = 1 ;
          while ( !thisCard.pin.equals (pin) & attempts < 4 )
              {   pin = KeyPad.readPin () ; attempts = attempts + 1 ;
              }
              if (!thisCard.pin.equals (pin))
                  throw new InvalidCard ("Bad PIN") ;
          thisBalance = thisCard.getBalance () ;
          do { Screen.prompt (" Please select a service ") ;
              service = Screen.touchKey () ;
              switch (service) {
                  case Services.withdrawalWithReceipt:
                          receiptRequired = true ;
                  case Services.withdrawalNoReceipt:
                          amount = KeyPad.readAmount () ;
                          if (amount > thisBalance)
                          {   Screen. printmsg ("Balance insufficient") ;
                              break ;
                          }
                          Dispenser.deliver (amount) ;
                          newbalance = thisBalance – amount ;
                          if (receiptRequired )
                              Receipt.print (amount, newBalance) ;
                          break ;
                  // other service descriptions here
                  default: break ;
              }
          }
          while (service != Services.quit ) ;
          thisCard.returnToUser ("Please take your card") ;
      }
      catch (InvalidCard e )
      {   Screen.printmsg ("Invalid card or PIN") ;
      }
      // other exception handling here
  } //main ()
} //ATM
```

PDLs result in very detailed specifications and, sometimes, they are too close to the implementation for inclusion in a requirements document. However, I recommend their use in two situations:

1. When an operation is specified as a sequence of simpler actions and the order of execution is important. Descriptions of such sequences in natural language are sometimes confusing, particularly if nested conditionals and loops are involved. This is illustrated in Figure 5.14 which specifies part of the specification of an ATM. I have used Java as the PDL but have deliberately left out

parts of the description to save space. A complete Java description of the ATM can be downloaded from the book's web pages.

2. When hardware and software interfaces have to be specified. In many cases, the interfaces between sub-systems are defined in the system requirements specification. Using a PDL allows interface objects and types to be specified.

If the reader of the system requirements is familiar with the PDL used, specifying the requirements in this way can make them less ambiguous and easier to understand. If the PDL is based on the implementation language, there is a natural transition from requirements to design. The possibility of misinterpretation is reduced. Specifiers need not be trained in another description language.

There are disadvantages to this approach to requirements specification:

1. The language used to write the specification may not be sufficiently expressive to describe the system functionality.
2. The notation is only understandable to people who have some programming language knowledge.
3. The requirement may be taken as a design specification design rather than a model to help the user understand the system.

An effective way to use this approach to specification is to combine it with the use of structured natural language. A forms-based approach may be used to specify the overall system. A PDL may then be used to define control sequences or interfaces in more detail.

5.3.3 Interface specification

The vast majority of software systems must operate with other systems which 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 must 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 which may have to be defined:

1. Procedural interfaces where existing sub-systems offer a range of services which are accessed by calling interface procedures.
2. Data structures which are passed from one sub-system to another. A Java-based PDL may be used for this with the data structure being described using a class definition with attributes representing fields of the structure. However, I think entity-relationship diagrams (described in Chapter 7) are better for this type of description.
3. Representations of data (such as the ordering of bits) which have been established for an existing sub-system. Java does not support such detailed representation specification so I don't recommend the use of a Java-based PDL for this.

Figure 5.15 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
```

Formal notations, discussed in Chapter 9, 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. Less formal, PDL interface descriptions are a compromise between comprehensibility and precision but are usually more precise than natural language interface specification.

Figure 5.15 is an example of a definition of the first of these interface types. 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 5.15 is an abstract model of the print server without revealing any interface details. The functionality of the interface operations can be defined using structured natural language (Figure 5.10), using a Java-based PDL (Figure 5.14), or by using a formal notation as discussed in Chapter 9.

5.4 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what is required of the system developers. 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 as separate documents.

The requirements document has a diverse set of users ranging from the senior management of the organisation who are paying for the system to the engineers responsible for developing the software. Figure 5.16, taken from (Kotonya and Sommerville, 1998), illustrates possible users of the document and how they use it.

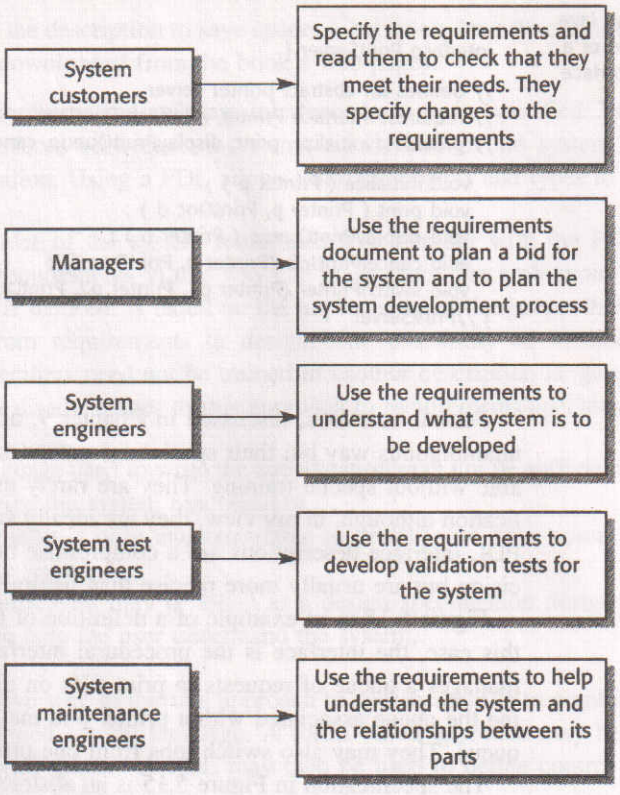


Figure 5.16 Users of a requirements document

Heninger (1980) suggests that there are six requirements which a software requirements document should satisfy:

- It should specify only external system behaviour.
- It should specify constraints on the implementation.
- It should be easy to change.
- It should serve as a reference tool for system maintainers.
- It should record forethought about the life cycle of the system.
- It should characterise acceptable responses to undesired events.

Although more than 20 years old, this is good advice. However, it is sometimes difficult to specify systems in terms of what they will do (their external behaviour). Inevitably, because of constraints from existing systems, the system design is constrained and this must be reflected in the requirements document. Other advice, such as the need to record forethought about the system life cycle, is widely accepted but not widely followed when writing requirements documents.

A number of different large organisations such as the US Department of Defense and the IEEE have defined standards for requirements documents. Davis (1993) discusses some of these standards and compares their contents. The most widely known standard is the IEEE/ANSI 830-1993 standard (IEEE, 1993). Thayer and Dorfman (1997), in their excellent collection of articles on requirements engineering, include

a full specification of this standard. 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** covering 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. However, it can be tailored and adapted to define a standard which is geared to the needs of a particular organisation. Figure 5.17 illustrates a possible organisation for a requirements document which is based on the IEEE standard. However, I have extended this to include information about predicted system evolution as recommended by Heninger.

Of course, the information which is included in a requirements document must depend on the type of software being developed and the approach to development which is used. If an evolutionary approach is adopted for a software product (say), the requirements document will leave out many of the detailed chapters suggested above. 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 which 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 they should include most of the chapters shown in Figure 5.17. 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.

Figure 5.17 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 are product requirements which constrain the system being developed, process requirements which apply to the development process, and external requirements. They often relate to the emergent properties of the system so 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 natural language, tables and diagrams so that they are understandable.
- System requirements are intended to communicate, in a precise way, the functions which the system must provide. To reduce ambiguity, they may be written in a structured language of some kind. This may be a structured form of natural language, a language based on a high-level programming language or a special language for requirements specification.
- The software requirements document is the agreed statement of the system requirements. It should be organised so that it can be used by both system customers and software developers.

FURTHER READING

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, 1998 Wiley.)

Software Requirements Engineering. This is a collection of papers on requirements engineering that includes several relevant articles, including 'Recommended practice for software requirements specification'. This is a discussion of the IEEE standard for requirements documents. (R. H. Thayer and M. Dorfman (eds.), 1997, IEEE Computer Society Press.)

EXERCISES

5.1 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.

5.2 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 with its cost. 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.

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

5.4 Write system requirements for the above system using a Java-based notation. You may make any reasonable assumptions about the system. Pay particular attention to specifying user errors.

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

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

5.6 Describe three different types of non-functional requirement which may be placed on a system. Give examples of each of these types of requirement.

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

5.8 What are the requirements for a programming language so that it is suitable for defining interface specifications? Comment on the suitability of C, Java and Ada for this purpose.

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

5.10 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.