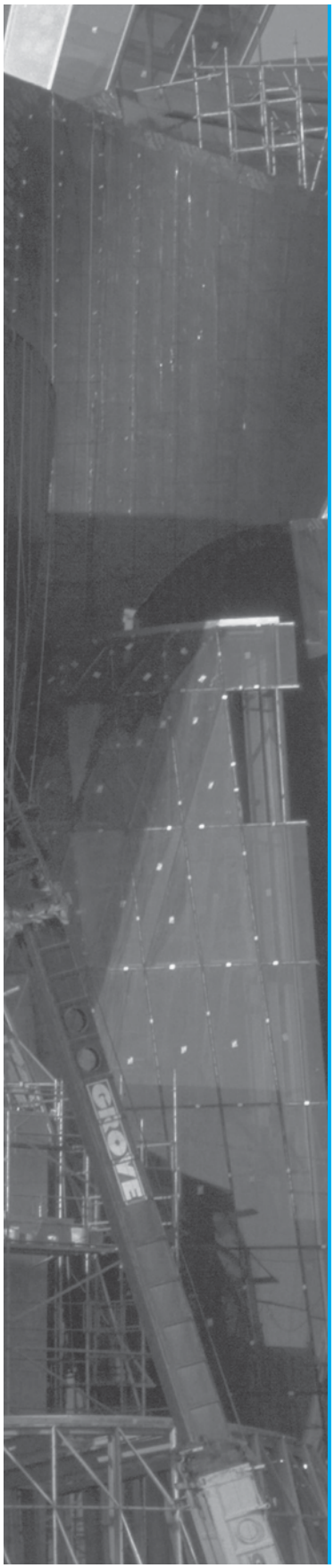

4

Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to discuss the processes involved in discovering and documenting these requirements. When you have read the chapter you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and nonfunctional software requirements;
- understand how requirements may be organized in a software requirements document;
- understand the principal requirements engineering activities of elicitation, analysis and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

The requirements for a system are the descriptions of what the system should do—the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term 'requirement' is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, 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 predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization'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 to system users and the constraints under which it must operate.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) 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 requirements are useful because they communicate information about the system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4.1 User and system requirements

You need to write requirements at different levels of detail because different readers use them in different ways. Figure 4.2 shows possible readers of 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.

In this chapter, I present a 'traditional' view of requirements rather than requirements in agile processes. For most large systems, it is still the case that there is a clearly identifiable requirements engineering phase before the implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, there are usually subsequent changes to the requirements and user requirements may be expanded into more detailed system requirements. However, the agile approach of concurrently eliciting the requirements as the system is developed is rarely used for large systems development.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional requirements or non-functional 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

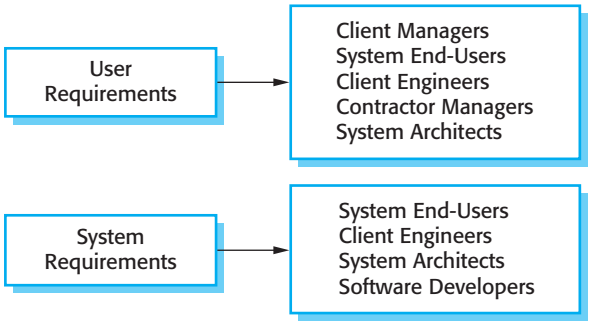

Figure 4.2 Readers of different types of requirements specification

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 constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, 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.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

4.1.1 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 organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional

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.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. They often cannot tell whether or not a domain requirement has been missed out or conflicts with other requirements.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

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 in a way that simplifies 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.

For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic.

The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer.

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 complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders 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 after the system has been delivered to the customer.

4.1.2 Non-functional requirements

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

Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often 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 an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or

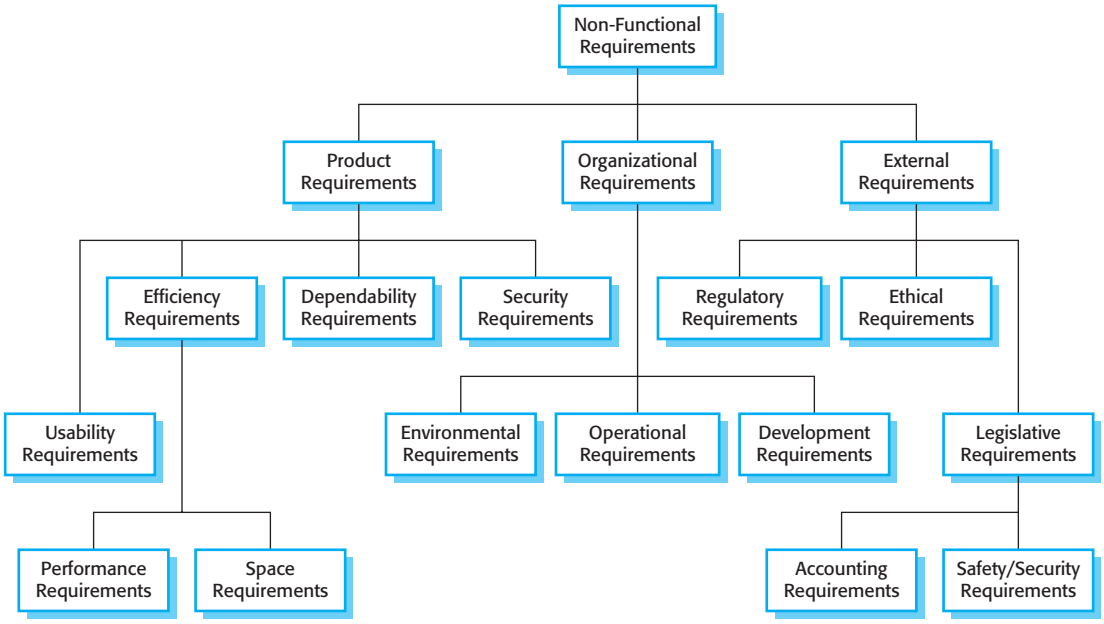

Figure 4.3 Types of non-functional requirement

hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 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:

1. *Product requirements* These requirements specify or constrain the behavior of the software. 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, security requirements, and usability requirements.
2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.
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 regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

PRODUCT REQUIREMENT
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of non-functional requirements in the MHC-PMS

Figure 4.4 shows examples of product, organizational, and external requirements taken from the MHC-PMS whose user requirements were introduced in Section 4.1.1. The product requirement is an availability requirement that defines when the system has to be available and the allowed down time each day. It says nothing about the functionality of MHC-PMS and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in healthcare systems and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

> *The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.*

I have rewritten this to show how the goal could be expressed as a 'testable' non-functional requirement. It is impossible to objectively verify the system goal, but in the description below you can at least include software instrumentation to count the errors made by users when they are testing the system.

> *Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM 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

Figure 4.5 Metrics for specifying non-functional requirements

when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable 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 measurable, non-functional requirements can be very high and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the authentication requirement in Figure 4.4 obviously requires a card reader to be installed with each computer attached to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' laptops. These are not normally equipped with card readers so, in these circumstances, some alternative authentication method may have to be allowed.

It is difficult, in practice, to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. 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.

Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these requirements in Chapter 12, where I describe specific techniques for specifying dependability and security requirements.

4.2 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an 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. Sometimes, the user and system requirements are 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.

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 is largely wasted. Rather than a formal document, approaches such as Extreme Programming (Beck, 1999) collect user requirements incrementally and write these on cards as user stories. The user then prioritizes 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 think 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.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.6, taken from my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998) shows possible users of the document and how they use it.

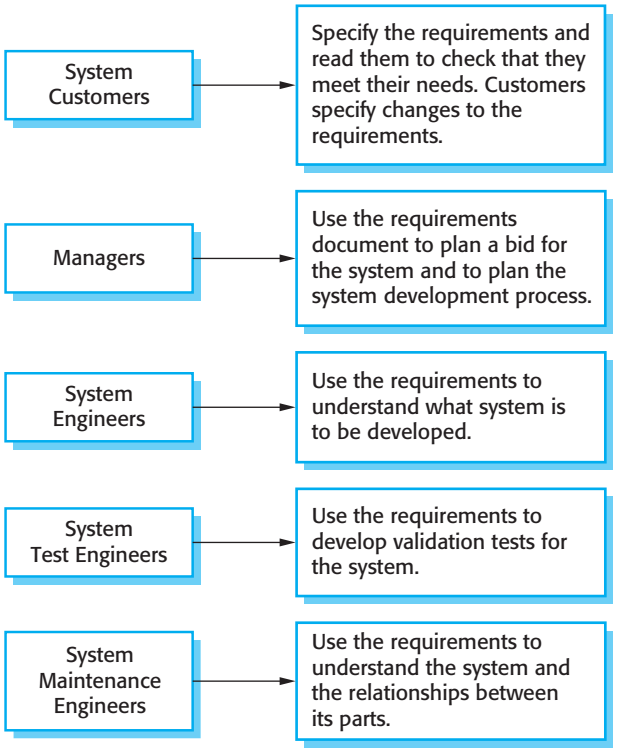

Figure 4.6 Users of a requirements document

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. Critical systems need to have detailed requirements because safety and security have to be analyzed in detail. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities can be resolved during development of the system.

Figure 4.7 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE, 1998). This standard is a generic standard that can be adapted to specific uses. In this case, I have extended the standard to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Naturally, the information that is included in a requirements document depends on the type of software being developed and the approach to development that is to be used. If an evolutionary approach is adopted for a software product (say), the

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 the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization 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	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that 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. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization 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, and so on.

Figure 4.7 The structure of a requirements document

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 judgment to decide how to meet the outline user requirements for the system.

However, when the software is part of a large system project that includes interacting hardware and software systems, it is usually necessary to define the requirements

Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence and structuring natural language requirements can be difficult.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

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 4.7. 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.

4.3 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is difficult to achieve as stakeholders interpret the requirements in different ways and there are often inherent conflicts and inconsistencies in the requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

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 detailed specification of the whole system.

Ideally, the system requirements should simply describe the external behavior 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 practically impossible 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 organized according to

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
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 now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Figure 4.8 Ways of writing a system requirements specification

the different sub-systems that make up the system. As I discuss in Chapters 6 and 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 existing systems, which constrain the design and 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 13) may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Figure 4.8 summarizes the possible notations that could be used for writing system requirements.

Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts, described in Chapter 5, show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances. I explain this approach to writing specifications in Chapter 12.

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.9 Example requirements for the insulin pump software system

4.3.1 Natural language specification

Natural language has been used to write requirements for software since the beginning of software engineering. It is expressive, intuitive, and universal. It is also potentially vague, ambiguous, and its meaning depends on the background of the reader. As a result, there have been many proposals for alternative ways to write requirements. However, none of these have been widely adopted and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use expresses the requirement in a single sentence. I associate a statement of rationale with each user requirement to explain why the requirement has been proposed. The rationale may also 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 to 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 color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy for words like 'architecture' and 'module' to be misunderstood. You should, therefore, avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included. It is particularly useful when requirements are changed as it may help decide what changes would be undesirable.

Figure 4.9 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. You can download the complete insulin pump requirements specification from the book's web pages.

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.
Requirements	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.

Figure 4.10 A structured specification of a requirement for an insulin pump

4.3.2 Structured 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. This approach maintains most of the expressiveness and understandability of natural language but ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson, 1999), in their book on 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 is similar to the approach used in the example of a structured specification shown in Figure 4.10.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. 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 a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.10.

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 4.11 Tabular specification of computation for an insulin pump

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. Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
5. A 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. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organized more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by 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. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.11 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

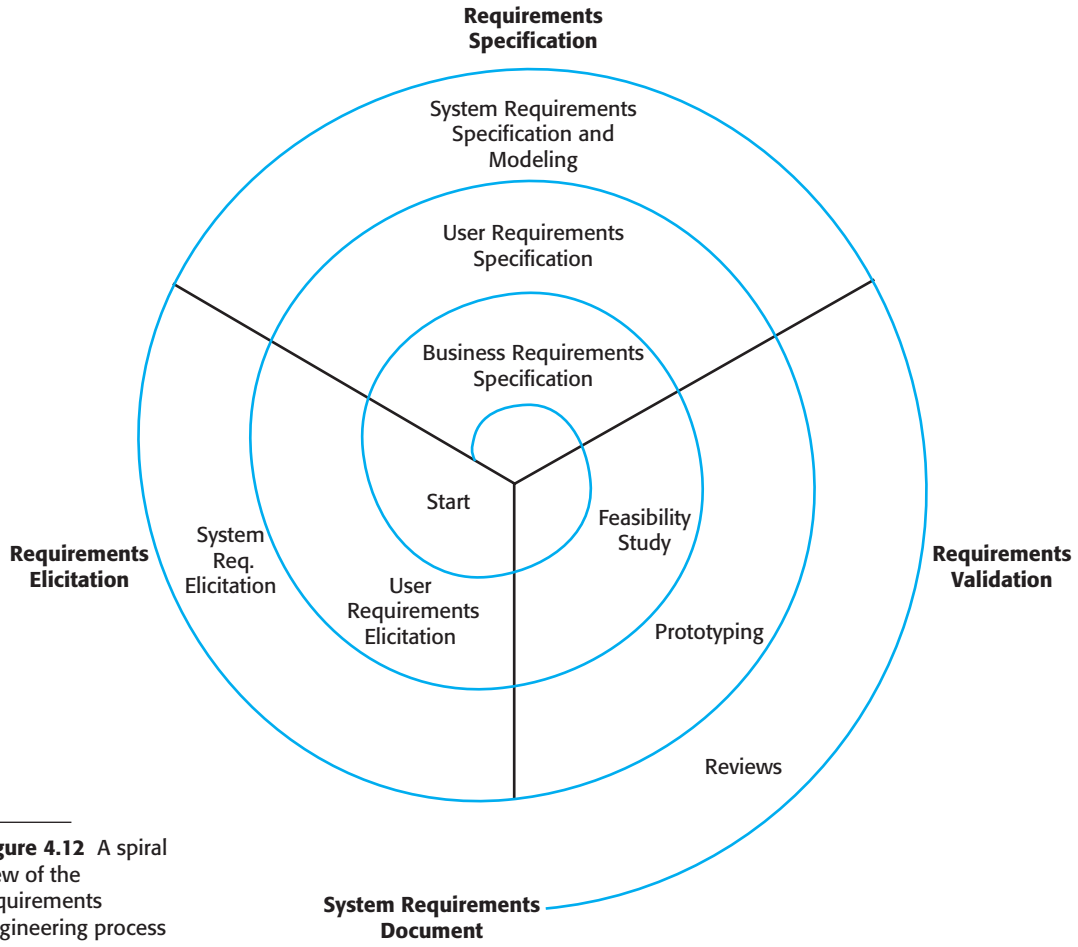

Figure 4.12 A spiral view of the requirements engineering process

4.4 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering processes may include four high-level activities. These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.6. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.12 shows this interleaving. The activities are organized as an iterative process around a spiral, with the output being a system requirements document. The amount of time and effort devoted to each activity in each iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and

Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: a) does the system contribute to the overall objectives of the organization? b) can the system be implemented within schedule and budget using current technology? and c) can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html

non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

Some people consider requirements engineering to be the process of applying a structured analysis method, such as object-oriented analysis (Larman, 2002). This involves analyzing the system and developing a set of graphical system models, such as use case models, which then serve as a system specification. The set of models describes the behavior of the system and is annotated with additional information describing, for example, the system's required performance or reliability.

Although structured methods have a role to play in the requirements engineering process, there is much more to requirements engineering than is covered by these methods. Requirements elicitation, in particular, is a human-centered activity and people dislike the constraints imposed on it by rigid system models.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; modifications are made to the system's hardware, software, and organizational environment. The process of managing these changing requirements is called requirements management, which I cover in Section 4.7.

4.5 Requirements elicitation and analysis

After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

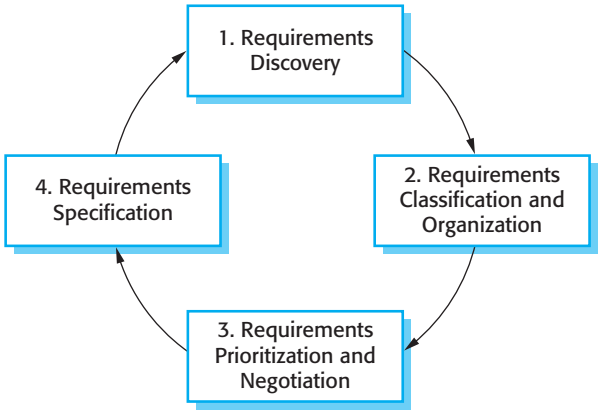

Figure 4.13 The requirements elicitation and analysis process

Requirements elicitation and analysis may involve a variety of different kinds of people in an organization. A system stakeholder is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include end-users who will interact with the system and anyone else in an organization who will be affected by it. Other system stakeholders might be engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

A process model of the elicitation and analysis process is shown in Figure 4.13. Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

The process activities are:

1. *Requirements discovery* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.
2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.
3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements specification* The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced, as discussed in Section 4.3.

Figure 4.13 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document is complete.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Inevitably, different stakeholders have different views on the importance and priority of requirements and, sometimes, these views are conflicting. During the process, you should organize regular stakeholder negotiations so that compromises can be reached. It is impossible to completely satisfy every stakeholder but if some stakeholders feel that their views have not been properly considered then they may deliberately attempt to undermine the RE process.

At the requirements specification stage, the requirements that have been elicited so far are documented in such a way that they can be used to help with requirements discovery. At this stage, an early version of the system requirements document may be produced with missing sections and incomplete requirements. Alternatively, the requirements may be documented in a completely different way (e.g., in a spreadsheet or on cards). Writing requirements on cards can be very effective as these are easy for stakeholders to handle, change, and organize.

Viewpoints

A viewpoint is way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, etc. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

4.5.1 Requirements discovery

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information. Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems. You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.

Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system. For example, system stakeholders for the mental healthcare patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

In addition to system stakeholders, we have already seen that requirements may also come from the application domain and from other systems that interact with the system being specified. All of these must be considered during the requirements elicitation process.

These different requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints with each viewpoint showing a subset of the

requirements for the system. Different viewpoints on a problem see the problem in different ways. However, their perspectives are not completely independent but usually overlap so that they have common requirements. You can use these viewpoints to structure both the discovery and the documentation of the system requirements.

4.5.2 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions but these usually lead on to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work so are usually happy to get involved in interviews. However, interviews are not so helpful in understanding the requirements from the application domain.

It can be difficult to elicit domain knowledge through interviews for two reasons:

1. All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.
2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are also not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures

rarely match the reality of decision making in an organization but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.
2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews supplements other information about the system from documentation describing business processes or existing systems, user observations, etc. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information and so it should be used in conjunction with other requirements elicitation techniques.

4.5.3 Scenarios

People usually find it easier to relate to real-life examples rather than abstract descriptions. They can understand and criticize a scenario of how they might interact with a software system. Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.

Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system. The stories used in extreme programming, discussed in Chapter 3, are a type of requirements scenario.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expects when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Figure 4.14 Scenario for collecting medical history in MHC-PMS

Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios. Scenarios may be written as text, supplemented by diagrams, screen shots, etc. Alternatively, a more structured approach such as event scenarios or use cases may be used.

As an example of a simple text scenario, consider how the MHC-PMS may be used to enter data for a new patient (Figure 4.14). When a new patient attends a clinic, a new record is created by a medical receptionist and personal information (name, age, etc.) is added to it. A nurse then interviews the patient and collects medical history. The patient then has an initial consultation with a doctor who makes a diagnosis and, if appropriate, recommends a course of treatment. The scenario shows what happens when medical history is collected.

4.5.4 Use cases

Use cases are a requirements discovery technique that were first introduced in the Objectory method (Jacobson et al., 1993). They have now become a fundamental feature of the unified modeling language. In their simplest form, a use case identifies

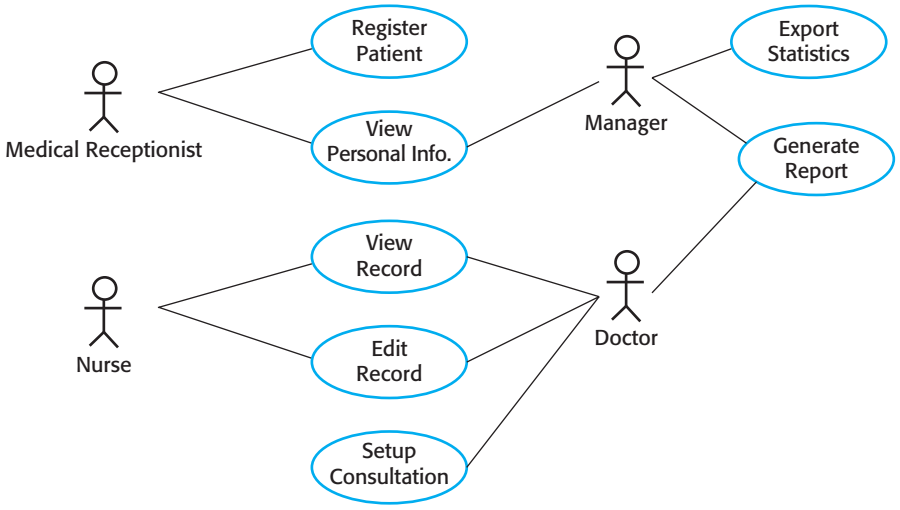

Figure 4.15 Use cases for the MHC-PMS

the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the patient information system.

There is no hard and fast distinction between scenarios and use cases. Some people consider that each use case is a single scenario; others, as suggested by Stevens and Pooley (2006), encapsulate a set of scenarios in a single use case. Each scenario is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. You can, in practice, use them in either way.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

> *Setup consultation allows two or more doctors, working in different offices, to view the same record at the same time. One doctor initiates the consultation by choosing the people involved from a drop-down menu of doctors who are online. The patient record is then displayed on their screens but only the initiating doctor can edit the record. In addition, a text chat window is created to help*

coordinate actions. It is assumed that a phone conference for voice communication will be separately set up.

Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system. Each type of interaction can be represented as a use case. However, because they focus on interactions with the system, they are not as effective for eliciting constraints or high-level business and nonfunctional requirements or for discovering domain requirements.

The UML is a de facto standard for object-oriented modeling, so use cases and use case–based elicitation are now widely used for requirements elicitation. I discuss use cases further in Chapter 5 and show how they are used alongside other system models to document a system design.

4.5.5 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational context and software system requirements may be derived or constrained by that context. Satisfying these social and organizational requirements is often critical for the success of the system. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how the social and organizational context affects the practical operation of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but which are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a work group may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (1987) pioneered the use of ethnography to study office work. She found that the actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of

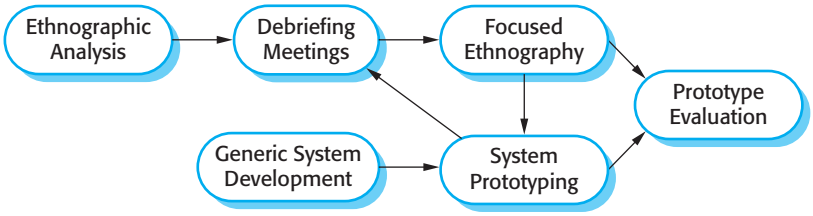

Figure 4.16 Ethnography and prototyping for requirements analysis

integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville, 1999; Viller and Sommerville, 2000) and documenting patterns of interaction in cooperative systems (Martin et al., 2001; Martin et al., 2002; Martin and Sommerville, 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. They deliberately put the aircraft on conflicting paths for a short time to help manage the airspace. Their control strategy is designed to ensure that these aircrafts are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.
2. Requirements that are derived from cooperation and awareness of other people's activities. For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircrafts that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with prototyping (Figure 4.16). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al., 1993).

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements. They cannot always identify new features that should be added to a system. Ethnography is not, therefore, a complete approach to elicitation on its own and it should be used to complement other approaches, such as use case analysis.

Requirements reviews

A requirements review is a process where a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html

4.6 Requirements validation

Requirements validation is the process of checking that requirements actually define the system that the customer really wants. It overlaps with analysis as it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed. Furthermore the system must then be re-tested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

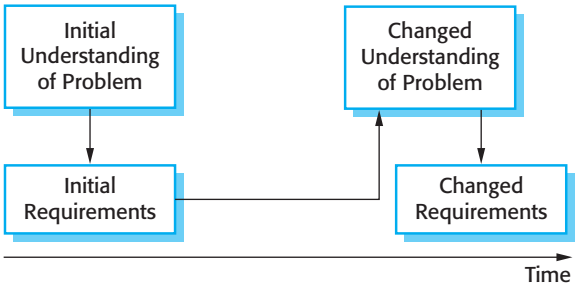

Figure 4.17 Requirements evolution

There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users. As a result, you rarely find all requirements problems during the requirements validation process. It is inevitable that there will be further requirements changes to correct omissions and misunderstandings after the requirements document has been agreed upon.

4.7 Requirements management

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing (Figure 4.17). The system requirements must then also evolve to reflect this changed problem view.

Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html

Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done. Once end-users have experience of a system, they will discover new needs and priorities. There are several reasons why change is inevitable:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements management is the process of understanding and controlling changes to system requirements. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements. The formal process of requirements management should start as soon as a draft version of the requirements document is available. However, you should start planning how to manage changing requirements during the requirements elicitation process.

4.7.1 Requirements management planning

Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

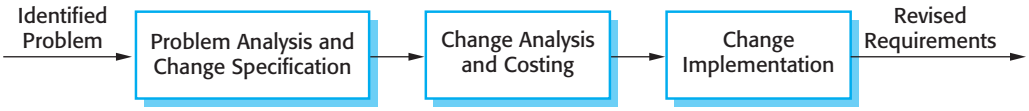

Figure 4.18 Requirements change management

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. *Change management* The process of change management (Figure 4.18) is simplified if active tool support is available.
3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, it may not be necessary to use specialized requirements management tools. The requirements management process may be supported using the facilities available in word processors, spreadsheets, and PC databases. However, for larger systems, more specialized tool support is required. I have included links to information about requirements management tools in the book's web pages.

4.7.2 Requirements change management

Requirements change management (Figure 4.18) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of

Requirements traceability

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html

using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. You should try to avoid this as it almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document or to add information to the requirements document that is inconsistent with the implementation.

Agile development processes, such as extreme programming, have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped.

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.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification, requirements validation, and requirements management.
- Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

FURTHER READING

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

'Integrated requirements engineering: A tutorial'. This is a tutorial paper that I wrote in which I discuss requirements engineering activities and how these can be adapted to fit with modern software engineering practice. (I. Sommerville, IEEE Software, 22(1), Jan–Feb 2005.) http://dx.doi.org/10.1109/MS.2005.13.

Mastering the Requirements Process, 2nd edition. 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, 2006, Addison-Wesley.)

'Research Directions in Requirements Engineering'. This is a good survey of requirements engineering research that highlights future research challenges in the area to address issues such as scale and agility. (B. H. C. Cheng and J. M. Atlee, Proc. Conf on Future of Software Engineering, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.17.

EXERCISES

4.1. Identify and briefly describe four types of requirement that may be defined for a computer-based system.

4.2. 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. 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.

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

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

4.5. Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

- An 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 ATM.
- The spelling-check and correcting function in a word processor.

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

4.7. Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.

4.8. Who should be involved in a requirements review? Draw a process model showing how a requirements review might be organized.

4.9. When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.

4.10. 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. However, you have also a responsibility of confidentiality to your previous employer.

REFERENCES

Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer*, **32** (10), 70–8.

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography.* London: Springer-Verlag.

Davis, A. M. (1993). *Software Requirements: Objects, Functions and States.* Englewood Cliffs, NJ: Prentice Hall.

IEEE. (1998). 'IEEE Recommended Practice for Software Requirements Specifications'. In *IEEE Software Engineering Standards Collection.* Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993). *Object-Oriented Software Engineering.* Wokingham: Addison-Wesley.

Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* Chichester, UK: John Wiley and Sons.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process.* Englewood Cliff, NJ: Prentice Hall.

Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. and Viller, S. (2001). 'Finding Patterns in the Fieldwork'. *Proc. ECSCW'01.* Bonn: Kluwer. 39–58.

Martin, D., Rouncefield, M. and Sommerville, I. (2002). 'Applying patterns of interaction to work (re)design: E-government and planning'. *Proc. ACM CHI'2002,* ACM Press. 235–42.

Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. *ACM Trans. on Computer-Human Interaction,* **11** (1), 59–89.

Robertson, S. and Robertson, J. (1999). *Mastering the Requirements Process.* Harlow, UK: Addison-Wesley.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. and Twidale, M. (1993). 'Integrating ethnography into the requirements engineering process'. *Proc. RE'93,* San Diego CA.: IEEE Computer Society Press. 165–73.

Stevens, P. and Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison Wesley.

Suchman, L. (1987). *Plans and Situated Actions.* Cambridge: Cambridge University Press.

Viller, S. and Sommerville, I. (1999). 'Coherence: An Approach to Representing Ethnographic Analyses in Systems Design'. *Human-Computer Interaction,* **14** (1 & 2), 9–41.

Viller, S. and Sommerville, I. (2000). 'Ethnographically informed analysis for software engineers'. *Int. J. of Human-Computer Studies,* **53** (1), 169–96.